Rejo Langage d'Objects Réactifs et d'Agents

Raul Acosta-Bermejo

▶ **To cite this version:**

Raul Acosta-Bermejo. Rejo Langage d'Objects Réactifs et d'Agents. domain_other. École Nationale Supérieure des Mines de Paris, 2003. Français. NNT : 2003ENMP1158 . pastel-00001355

HAL Id: pastel-00001355
https://pastel.hal.science/pastel-00001355

Submitted on 22 Aug 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE
présentée et soutenue publiquement par

Raúl Acosta-Bermejo

pour obtenir le grade de DOCTEUR
de l'École des Mines de Paris
Spécialité : ***Informatique temps-réel, Robotique, Automatique***

Rejo
Langage d'Objets Réactifs et d'Agents

Date de soutenance : Le 9 octobre 2003

Composition du jury :

Président	*Charles André*	
Rapporteurs	*Florence Maraninchi*	
	Marc Pouzet	
Examinateurs	*Fréderic Boussinot*	(directeur)
	Laurent Hazard	

Thèse préparée au
Centre de Mathématiques Appliquées de l'EMP
à l'Institut National de Recherche en Informatique et en Automatique

Remerciements

Le travail présenté dans cette thèse a été effectué au sein du Centre de Mathématiques Appliquées (CMA) qui se trouve dans l'Institut National de Recherche en Informatique et Automatique (INRIA). Je tiens à remercier ces organismes qui m'ont permis d'effectuer mon travail dans les meilleures conditions matérielle, scientifique et technique que l'on puisse espérer.

Je suis particulièrement reconnaissant envers Fréderic BOUSSINOT, maître de recherche au CMA, pour avoir accepté de diriger cette thèse mais aussi pour sa disponibilité et ses nombreux conseils.

J'exprime ma profonde gratitude à Charles André, professeur à l'Université de Nice, pour l'honneur qu'il me fait de présider mon jury de thèse. Je souhaite remercier Florence Maraninchi, professeur à l'ENSIMAG-INPG, et Marc Pouzet, Maître de Conférences à l'Université de Paris 6, pour avoir accepté la lourde charge d'être rapporteur et pour leurs précieuses remarques qui m'ont permis d'améliorer ce manuscrit. Je remercie aussi Laurent HAZARD, ingénieur à France Télécom R&D, d'avoir accepté de faire partie de mon jury et pour les nombreuses discussions que nous avons eues ensemble.

Mes remerciements vont naturellement à l'ensemble des gens que j'ai côtoyé durant ces trois années et qui m'ont aidé ou tout simplement rendu le déroulement de cette thèse agréable. En particulier je voudrais remercier toutes les personnes étrangères que j'ai côtoyé car ils m'ont permis de mieux connaître leurs différentes cultures. Je citerai entre autres :

- Les copains de l'INRIA : Alexander SAMARIN, Ana MATOS, Christian BRUNETTE, Damien CIABRINI, Dimitri PETOUKHOV, Dimitru POTOP-BUTUCARU (alias Jacky), Eric VECCHI, Fabrice PEIX, Jean-Ferdinand SUSINI, Olivier TARDIEU, Olivier PARRA, Pascal ZIMMER, Yannis BRES, Nicolas MAGAU.
- Les chercheurs : Annie RESSOUCHE, Davide SANGIORGI, Gerard BOUDOL, Ilaria CASTELLANI, Ives ROUCHALEAU, Marc BORDIER, Manuel SERRANO, Nadia MAIZI, Remi DRAI, Robert de SIMONE, Valerie ROY.
- Les secrétaires : Catherine JUNKER, Dominique MICOLLIER et Evelyne Largeteau.
- Les copains mexicains : Gabriela MARTINEZ et Miguel RUIZ, Sonia LOZANO et Felipe LUNA, Carolina MEDINA et Victor RAMOS, Alejandra RAMIREZ et Gerardo HERMOSILLO, Lucero LOPEZ, Ricardo MARTINEZ, Oscar VIVEROS.
- Mes amis français : Edith CASTILLON et Louis GIBERT, Susan et Yves GARNIER, Marie-Cecile LAFONT.

Je tiens à remercier tout particulièrement ma femme, Maribel RESENDIZ, qui m'a beaucoup aidé tout au long de ces quatre années. Merci pour ta patience et pour me rappeler que l'on peut s'amuser autrement qu'en programmant. Je t'aime très fort.

Je remercie très chaleureusement ma maman, pas seulement parce qu'elle m'a aide à gérer mes affaires au Mexique mais surtout parce qu'elle a été la seule qui a eu le courage de me rendre visite et me soutenir. Qu'est-ce que je t'aime.

Je voudrais aussi remercier à ma tante Magdalena BERMEJO qui m'a aussi beaucoup aidé dans les démarches administratives de l'IPN.

Traduction en espagnol pour elles :

> *Quiero agraderle a mi mamita querida de una manera especial, no solamente porque me ayudo en los tramites administrativos en México, si no porque tuvo el valor de venir a visitarme en Francia y por su apoyo moral. Te quiero mucho.*
> *Tambien quiero agraderle a mi tia Magdalena que me ayudo en los tramites administrativos del IPN.*

Enfin, je tiens à remercier le CONACyT, l'ESCOM et le CMA pour le soutient qu'ils m'ont donné pendant mes études.

Table des matières

Chapitre 1

Introduction

Ce chapitre présente les différents concepts utilisés lors de cette thèse. Le travail de la thèse a porté sur le développement d'un langage de programmation appelé Rejo et sur la construction d'une plate-forme, appelée ROS, qui exécute les applications construites avec le langage Rejo.

Le langage Rejo, comme la plupart des langages de programmation, est né du besoin d'améliorer un langage de programmation considéré comme incomplet (mal adapté, insuffisant) ou de difficile utilisation pour résoudre certains types de problèmes. En particulier Rejo est né de Junior qui est le dernier échelon dans l'évolution d'autres langages (SugarCubes, SL, Reactive-C, ESTEREL). Tous ces langages ont un ensemble de caractéristiques en commun qui permettent de les classifier dans les langages dits synchrones ou qui utilisent l'approche synchrone.

La section 1.1 présente l'approche synchrone. On commence par une description des origines du modèle et du modèle d'exécution pour ensuite passer à une description des diverses variantes en termes de style de programmation et en termes de modifications faites au modèle d'exécution de base.

Le langage Rejo a été utilisé pour programmer un type particulier de systèmes réactifs, les Systèmes d'Agents Mobiles (SAM). La construction d'un SAM, appelé ROS, a servi à tester le langage Rejo et à explorer son utilisation dans d'autres domaines. La section 1.2 présente les principales notions d'agents mobiles: définitions, classifications et exemples.

Finalement la section 1.3 présente les objectifs qui ont guidé la création du langage Rejo et de la plate-forme ROS.

1.1 L'approche synchrone

Selon D. Harel et A. Pnueli, les systèmes informatiques peuvent être classifiés comment étant réactifs, interactifs ou transformationnels, en fonction de leur degré d'interaction avec leur environnement [HAR 85]. Un système transformationnel effectue des calculs à partir des données fournies en entrée, pour produire des résultats en sortie avant de se terminer. L'interaction avec l'environnement se limite à l'acquisition des données et à la production de résultats, comme dans le cas d'un compilateur.
Contrairement à un système transformationnel, correspondant à un unique calcul de fonction, un système interactif ou un système réactif doit maintenir une interaction constante avec son environnement. Les systèmes interactifs et réactifs interagissent continuellement avec leur environnement, en produisant des résultats à chaque invocation. Ces résultats dépendent des données fournies par l'environnement lors de l'invocation, ainsi que de l'état interne du système. La différence entre ces deux types de systèmes réside dans l'entité qui contrôle l'interaction. Dans un système interactif, par exemple une base de données, la prise en compte des requêtes et la production des réponses se font à l'initiative du système, qui impose ainsi son propre rythme. Par contre, un système réactif doit être toujours en mesure de fournir une réponse immédiate quand l'environnement le sollicite. L'évolution d'un système réactif est donc une suite de réactions provoquées par l'environnement, chaque réaction

étant considérée comme instantanée par rapport à l'échelle de temps propre à l'environnement Les interfaces homme-machine et les programmes de contrôle de processus industriels sont des exemples typiques de systèmes réactifs.

En pratique, la plupart des systèmes réactifs sont implémentés par des systèmes interactifs suffisamment rapides pour prendre en compte tous les stimuli et y répondre à temps. Rares sont aussi les applications purement réactives : un grand nombre d'applications sont constituées d'un cœur réactif et de traitements transformationnels s'exécutant en parallèle.

Les systèmes réactifs introduits par Harel et Pnueli ont été développés dans plusieurs cadres qui ont donné naissance au Modèle Synchrone et à un ensemble de langages de programmation, dits synchrones, qui l'implémentent. Parmi ces langages, on trouve ESTEREL, Lustre et Signal. Le modèle synchrone permet d'exprimer à la fois des comportements de bas niveau ainsi que des comportements de haut niveau ; ceci a conduit à la création de plusieurs langages synchrones qui utilisent différents ***styles de programmation***. On trouve, par exemple, des *langages impératifs* (ESTEREL, ECL) qui sont bien adaptés à la construction de systèmes embarqués et de circuits ; des *langages fonctionnels* (Lustre, Signal) qui permettent de décrire de comportements de haut niveau en quelques lignes ; des *langages graphiques* (Argos, SyncCharts) qui facilitent l'interaction homme-machine; des *langages orientés objets* (Jester) qui profitent de la technologie OO.
Le modèle synchrone présente néanmoins quelques caractéristiques qui rendent difficile sont utilisation dans la construction de certains systèmes; plusieurs travaux ont été menés pour modifier le modèle et l'adapter, par exemple, à la construction de systèmes dynamiques. L'une des premières ***variantes du modèle*** qui fut créé est l'approche Réactive-Synchrone. L'évolution de l'approche réactive a démarré avec une première proposition en C appelée Reactive-C et une implémentation restreinte de l'approche synchrone appelée SL. L'approche réactive a également généré tout un ensemble de langages de programmation et de styles de programmation : Junior (programmation impérative), SugarCubes et Rejo (programmation OO), Senior (programmation fonctionnelle).

1.1.1 Description du modèle

L'approche synchrone a été créée en 1982 par plusieurs équipes de recherche françaises qui étudiaient les caractéristiques des systèmes de contrôle et les systèmes temps réel: systèmes avec des contraintes temporelles qui sont utilisées de plus en plus dans des contextes critiques où la propriété de déterminisme s'impose pour le débugage et la vérification des contraintes.

Ces systèmes ont longtemps été programmés soit au moyen de primitives de bas niveau, soit avec des langages de programmation du parallélisme (CSP, Ada). Cette façon de construire ces systèmes, basés sur l'approche dite asynchrone, est non-déterministe, et se prête mal aux preuves de propriétés.

Dans l'approche synchrone, on passe d'un système et de son environnement qui s'exécutent en temps continu à un système discret qui définit une horloge globale qui sert à échantillonner les changements qui s'effectuent dans l'environnement. La passage au monde discret se fait par des couples d'activation/réaction du système discret. A chaque activation, le système réagit aussi rapidement que possible à toute modification de son environnement (les entrées) et produit à son tour une modification dans l'environnement (les sorties). Les entrées et les sorties sont souvent modélisées par des signaux générés soit par l'environnement, soit par le système synchrone. Ces systèmes sont composés par des réactions qui s'exécutent en parallèle pour pouvoir réagir immédiatement à toutes les modifications de l'environnement.

L'approche synchrone repose sur une hypothèse fondamentale selon laquelle ***la durée logique des réactions est nulle***. En d'autres termes, le système réagit suffisamment vite par rapport aux modifications de l'environnement pour ne pas perdre d'événements. Dans ce cas, les sorties d'un système synchrone sont émises simultanément aux entrées. Cela a pour conséquence que l'environnement d'un tel système est parfaitement déterminé et stable lors d'une réaction, puisque de façon instantanée les entrées et les sorties sont connues.

Les formalismes synchrones qui ont été créés proposent différents langages de programmation pour exprimer les réactions d'un système à une succession d'activations dans un environnement d'exécution. Ces langages de programmation doivent s'assurer de l'absence d'incohérence dans ce qui est produit par les différentes réactions exécutées en parallèle. En effet, la puissance expressive des langages synchrones est souvent trop grande et

permet d'écrire des systèmes incohérents. Cette contrainte supplémentaire impose une analyse statique pour rejeter les programmes non conformes; en particulier, il faut s'assurer que si un signal est considéré comme étant absent dans une réaction, toutes les autres ont la même vision. Les problèmes que l'on rencontre au cours de cette analyse sont appelés *problèmes de causalité*

De plus, les sorties d'un système synchrone doivent être définies de manière unique en fonction des entrées à chaque réaction car ***les formalismes synchrones sont déterministes***. L'analyse statique des programmes doit donc aussi vérifier que le programme génère une seule sortie pour une même configuration d'entrée. Or, cette analyse ne peut se faire que par approximations, ce qui conduit à rejeter des programmes qui auraient pu être acceptés en utilisant un autre choix. Il est donc important de fixer le choix d'analyse car celui-ci a un impact direct sur la puissance expressive du langage.

La figure 1.1 illustre la représentation graphique d'un système synchrone composé par trois programmes qui s'exécutent en parallèle et qui réagissent tous de manière synchrone à chaque activation (instant) du système.

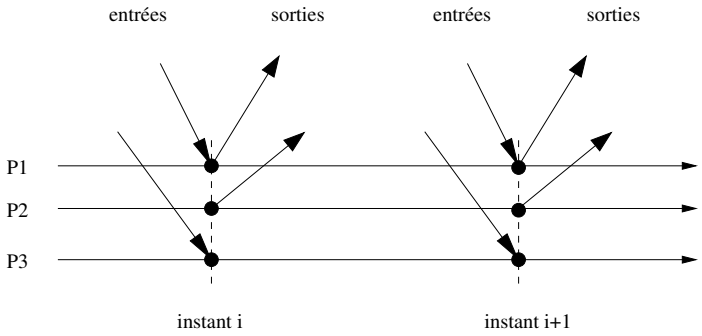

FIG. 1.1: L'Approche Synchrone

1.1.2 Langages Synchrones

Cette section présente un récapitulatif de la plupart de langages qui ont été créés en utilisant l'approche synchrone. L'idée de cette liste est de donner une vision générale du développement de ce domaine. Comme tous les langages présentés ici sont issus du modèle synchrone, il est normal de retrouver les même concepts, et on se concentrera sur les caractéristiques principales qui les distinguent.

Le lecteur intéressé en l'approche synchrone peut lire [HAL 93b] qui donne un aperçu de l'état de l'art dans ce domaine. Récemment il a été créé une page web [Web Synch] qui regroupe toute l'information dans le domaine.

La figure 1.2 montre l'évolution des différents langages synchrones au cours du temps. L'évolution des langages synchrones a été motivée par plusieurs raisons, comme la génération de code embarqué (Jester et ECL), la programmation graphique (Argos et SynChart) ou l'évolution dynamique de la structure des programmes au cours de l'exécution (Reactive-C). Les flèches de la figure indiquent la source d'inspiration des nouveaux langages. Les nouveaux langages synchrones sont des variantes ou des mélanges des formalismes d'origine. Dans le cas de SL, par exemple, il s'agit d'une variante d'Esterel qui comme Reactive-C interdit la réaction instantanée à l'absence pour éviter les problèmes de causalité.

Voici maintenant la description des principaux langages synchrones présentés dans la figure 1.2. Cette liste n'est nullement exhaustive. En particulier, il existe des applications comme Simulink et Ptolemy, qui intègrent seulement certaines caractéristiques des langages synchrones, et que l'on ne considérera pas ici.

ESTEREL [BER 87] est l'un des premiers langages synchrones conçu pour construire des systèmes réactifs de contrôle, par exemple des systèmes embarqués, des interfaces homme-machine et des protocoles de communication. Esterel a été développé depuis 1983 au CMA et à l'INRIA de Sophia Antipolis; il est actuellement commercialisé par Esterel Technologies et Simulog.

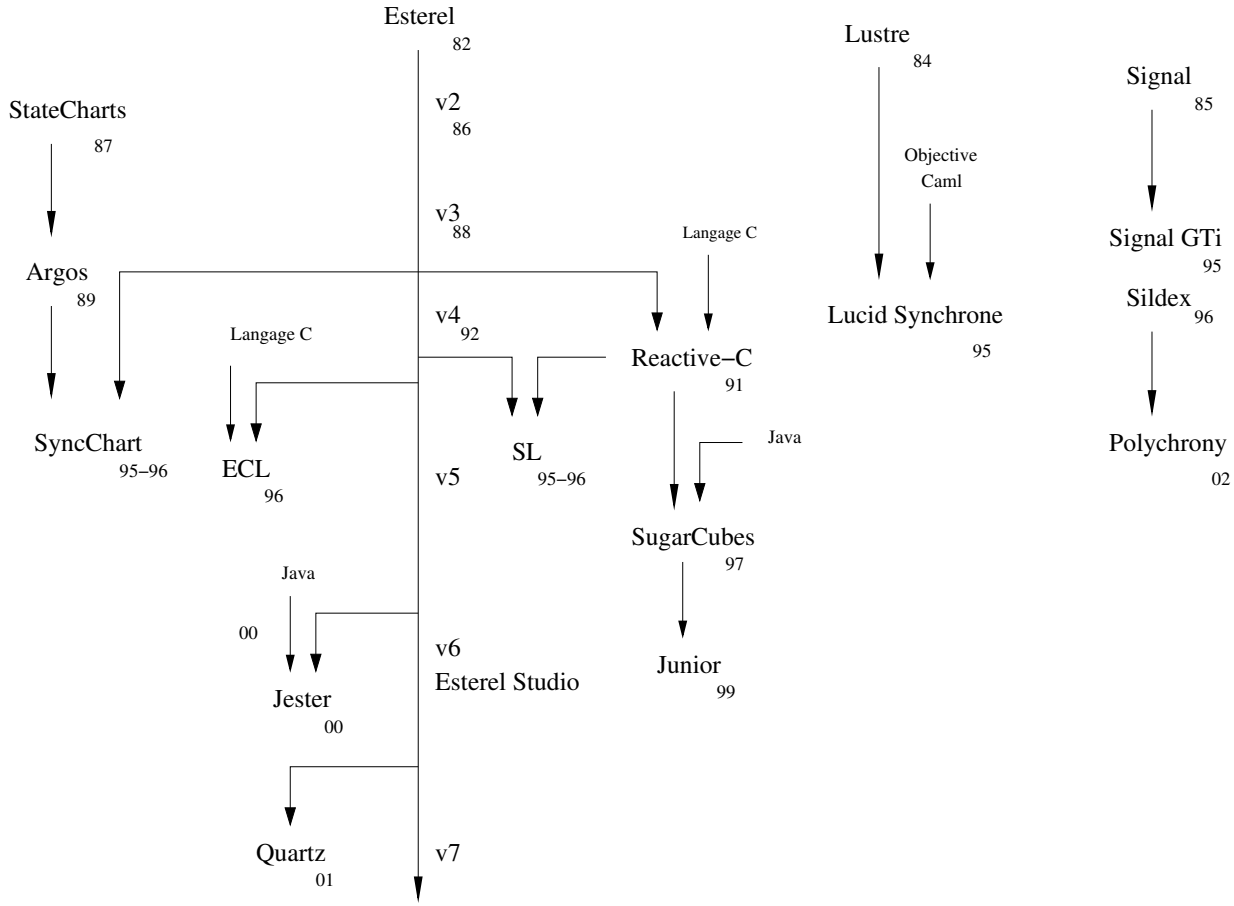

FIG. 1.2: Evolution de l'approche synchrone

Le langage Esterel est le langage synchrone le plus répandu; il est utilisé par plusieurs universités et entreprises. Parmi les principales écoles et entreprises on trouve l'Ecole d'Ingénieurs de Genève, l'Ecole Supérieur en Sciences Informatiques, l'Ecole de Mines de Paris, l'entreprise Dassault-Aviation, ThalesÒ, Intel et Texas Instruments.

Les caractéristiques d'Esterel sont les suivantes :

- Le langage a été défini rigoureusement par une sémantique formelle [BER 92].
- Il existe un ensemble très riche d'outils de programmation: un compilateur, un simulateur graphique, un système de vérification et des optimiseurs.
- Le nom Esterel est aussi le nom du compilateur qui peut générer du logiciel ou du matériel. Plusieurs versions du compilateur ont été bâties; la version actuelle est la V6.
- Lorsque le compilateur génère du logiciel, il peut produire du code C. Le code C est utilisé comme le noyau réactif d'un système plus large qui interagit avec le noyau en utilisant une interface bien définie, celle du noyau basée sur la notion d'événement.
- Lorsque le compilateur génère du matériel, il génère des diagrammes de circuits qui utilisent des *gates* (*netlists*).
- La vérification des programmes Esterel est faite avec les machines d'états finis générées par le compilateur. Une autre technique consiste à utiliser les BDD qui permettent de faire des réductions par bisimulation et de vérifier des propriétés de sûreté.

Le langage Esterel est composé par un ensemble très riche d'instructions dont la plupart sont en réalité des macros. Les instructions principales sont la génération et la détection de la présence d'un événement (`emit` et `present`), la préemption (`abort-when`), les boucles (`loop`), le parallélisme (||) et d'autres instructions plus générales (`trap`, `halt`).

Lustre [ACM 87, HAL 91, HAL 02] est un langage déclaratif qui sert à programmer des systèmes réactifs. C'est un langage déclaratif car la description du système réactif est un ensemble d'équations qui sont toujours satisfaites par les variables du programme. Les variables en Lustre sont considérées comme des fonctions qui dépendent du temps. Elles ont une horloge qui définit la séquence d'instants où elles prennent leurs valeurs. Le langage a été inspiré par la famille des formalismes utilisés dans le contrôle de machines, par exemple les systèmes d'équations différentielles et les réseaux d'opérateurs synchrones (les bloc-diagrammes).

Le compilateur de Lustre produit soit du code en boucle simple, soit du code séquentiel (sous la forme d'un automate fini étendu) par la synthèse de la structure de contrôle du code. Plusieurs niveaux de détail de la structure de contrôle peuvent être choisis.

En ce qui concerne la vérification de programmes, il a été prouvé que Lustre peut être considéré comme une logique temporelle qui permet de: 1) exprimer de propriétés de sûreté, 2) décrire le programme et sa spécification dans le même langage, 3) construire un outil de vérification (*model-checker*) appelé Lurette, qui les vérifie.

Lurette considère un modèle fini du programme, similaire à celui des automates produit par le compilateur. Lurette n'est capable que de vérifier des propriétés qui dépendent du contrôle du programme. Lurette est capable de générer un nombre arbitraire de séquences d'entrées de longueur arbitraire, qui satisfont des suppositions d'un programme, tout en prouvant que les propriétés spécifiées sont satisfaites.

Il existe une version commerciale de Lustre, appelée SCADE (*Safety-Critical Application Development Environment*) qui a été utilisée dans plusieurs projets européens d'aviation (Airbus A340-600, A380, Eurocopter). SCADE fut développé par Aérospatiale et Verilog et a été récemment racheté par Esterel Technologies. SCADE peut facilement être interfacé avec le compilateur Lustre. SCADE dispose d'un éditeur graphique pour spécifier des systèmes en utilisant les notions de flot de données et de machine d'états finis. Les modèles spécifiés avec SCADE peuvent être simulés, vérifiés formellement et utilisés pour la génération de code. Les outils de simulation (*ModelBuild*) et vérification (*ModelVerify*) font partie de l'environnement ASDE (*Avionics Systems Development Environment*) développé dans le projet SAFEAIR.

Signal [GUE 91] est un langage synchrone de type déclaratif et flot de données. Les objets de base manipulés par ce langage sont les signaux. Un signal est une suite non bornée de valeurs typées à laquelle est associée une horloge qui détermine l'ensemble des instants où le signal est présent. Par exemple, un signal X dénote la séquence de données indexées par le temps t dans un domaine T. Des signaux d'un type particulier appelés *event* sont caractérisés seulement par leur horloge, c'est-à-dire leur présence (ils ont la valeur Booléenne *true* à chaque occurrence). Etant donné un signal X, son horloge est donnée par l'expression *event* X, qui donne l'événement présent simultanément à X.

Signal est construit autour d'un petit nombre d'opérateurs de base qui permettent de spécifier dans un style équationnel les relations entre les signaux. Chaque équation issue d'un programme Signal peut être vue comme un processus élémentaire. Ces processus Signal décrivent donc à la fois les relations fonctionnelles et temporelles entre les signaux. Ils peuvent communiquer, par l'intermédiaire de signaux constituant leurs ports d'entrée et de sortie, avec le monde extérieur ou avec d'autres processus. Enfin, la composition d'un ensemble de processus produit le programme Signal.

Le compilateur de Signal consiste principalement en un système formel capable de raisonner sur les horloges des signaux, la logique, et les graphes de dépendance. En particulier, le calcul d'horloges et le calcul de dépendances fournissent une synthèse de la synchronisation globale du programme à partir de la spécification des synchronisations locales (qui sont données par les équations Signal), ainsi qu'une synthèse de l'ordonnancement global des calculs spécifiés. Des contradictions et des inconsistances peuvent être détectées au cours de ces calculs. Si le compilateur détermine que les contraintes de synchronisation sont vérifiées et que le programme est contraint de façon à calculer une solution unique, alors un code exécutable en C ou en Fortran est produit.

Lucid Synchrone [CAS 87] est un langage synchrone utilisé pour l'implémentation de systèmes réactifs. Il combine les caractéristiques de Lustre et du langage ML. Le nom provient de Lucid, un langage flot de données qui gère des streams. Les caractéristiques principales du langage sont:

- C'est un langage fonctionnel d'ordre supérieur avec un typage fort. Le langage utilise des séquences infinies (streams) qui sont des valeurs primitives. Les streams sont utilisés pour représenter les signaux d'entrées et sorties d'un système réactif. Les streams sont manipulés par des opérateurs synchrones de flot de données à la Lustre.
- Plusieurs analyses statiques peuvent être réalisées, par exemple l'inférence de types, le calcul d'horloge, les problèmes de causalité. En particulier le langage est basé sur la notion d'horloge qui permet de spécifier plusieurs débits d'exécution d'un programme. Le programme doit vérifier quelques règles sur les horloges pour assurer sa réactivité, c'est-à-dire que l'on puisse générer un système de transitions fini.
- Le langage est bâti sur Ocaml; les programmes sont un sous-ensemble de la syntaxe de Ocaml; les valeurs combinatoires sont importées d'Ocaml; les programmes sont traduits en code Ocaml.
- Un système de modules a été mis en place pour pouvoir utiliser les valeurs du langage hôte, Ocaml, ou d'autres modules synchrones.

StateCharts [HAR 87] a été créé par David Harel avec le but d'étendre le modèle de Machine d'Etats Finis (MEF) pour décrire des comportements complexes sans les limitations des MEF (manque de modularité et explosion dans le nombre d'états lors de la construction de systèmes complexes). Statechart est un langage de spécification qui permet de construire un modèle et de le valider. La construction d'un StateChart, comme la plupart des formalismes graphiques, est à base de boîtes et de flèches qui disposent d'une sémantique formelle [HAR 96]. Les boîtes représentent des états; il y a trois types d'états: l'état AND, l'état OR et l'état basique. Les flèches représentent les transitions entre les états; elles sont étiquetées pour spécifier les conditions sous lesquelles les transitions sont faites et pour spécifier les actions à faire lorsqu'une transition est empruntée.

Si on compare les StateCharts par rapport aux MEF, ils ajoutent trois éléments principaux: hiérarchie, orthogonalité et diffusion. La hiérarchie en StateCharts représente la possibilité de pouvoir imbriquer des états. Cette propriété est importante pour éviter l'explosion en nombre d'états et des transitions lors de la construction d'applications réelles. L'orthogonalité représente le parallélisme existant entre l'exécution de deux ou plus états qui se trouvent à l'intérieur d'un état AND. La diffusion est le mécanisme de communication utilisé entre les états orthogonaux. En résumé les StateCharts sont définis de la façon suivante:

$$StateCharts = MEF + Hiérarchie + Orthogonalité + Diffusion$$

D'autres travaux ont été menés autour des StateCharts. Le système Statemate les utilise pour représenter des comportements réactifs. Derek Coleman a créé les Objectcharts [DER 92], une extension aux StateCharts qui définit un environnement orienté objet. Bran Selic a utilisé les StateCharts pour implémenter un mécanisme de contrôle récursif [BRA 93].

Argos [MAR 89] est un langage impératif développé au laboratoire VERIMAG. Argonaute est l'environnement de programmation basé sur Argos qui offre un compilateur et plusieurs outils de vérification. Argos a été inspiré des StateCharts et il offre donc une syntaxe textuelle et graphique. Les principales différences avec StateCharts sont l'utilisation d'un véritable opérateur hiérarchique qui supprime: 1) les transitions entre les nivaux utilisées en StateCharts, et 2) la supposition d'un synchronisme strict.

SyncCharts [AND 95, AND 96] est un formalisme graphique dédié à la modélisation de systèmes réactifs. SyncCharts est le nom du modèle et un syncChart en est une instance. Plusieurs caractéristiques sont héritées de StateCharts et Argos. SyncCharts permet la spécification de comportements réactifs ainsi que la programmation synchrone d'applications. Un syncChart peut être traduit en un programme Esterel; cette traduction permet de profiter de l'environnement de développement de ce langage.

Les éléments graphiques des SyncCharts sont les états, la hiérarchisation d'états, la concurrence et les transitions. Le bloc graphique de base, appelé *star*, est un état et ses arcs; l'état définit un comportement invariant et les arcs définissent la façon dont on sort de l'état. L'interconnexion de stars est appelée *constellation* et la composition parallèle de constellations est appelée un *macrostate*(macro états).

Les principales caractéristiques de SyncCharts sont:

- *Hiérarchie*: l'encapsulation de macro états.
- *Concurrence*: l'exécution orthogonale de constellations.
- *Communication*: La communication est faite par la diffusion instantanée de signaux.
- *Préemption*: la sortie d'un état (un arc) en fonction d'un événement. Comme en Esterel, il y a la préemption forte (un arc terminé par une flèche) et la préemption faible (un arc terminé par un cercle).

SL [BOU 96f, REP 95] est une variante d'Esterel dans laquelle on restreint la puissance d'expression dans le but d'éliminer les problèmes de causalité. L'approche prise en SL consiste à interdire toute réaction instantanée à l'absence des signaux; un signal ne peut être considéré comme absent qu'à la fin de l'instant courant, ce qui entraîne qu'une éventuelle émission de ce même signal ne peut avoir lieu qu'à l'instant suivant. La préemption forte réalisée en Esterel par l'instruction `abort` est également interdite en SL car elle implique une réaction instantanée à l'absence. Seule une primitive `kill` de préemption faible (correspondant au `weak-abort` d'Esterel) est donc disponible en SL.

L'absence de préemption forte en SL constitue certainement la limitation fondamentale du langage. Sa force réside, par contre, dans la disparition des problèmes de causalité avec le gain en modularité que cela implique: il n'y a plus de risque de voir apparaître un problème de causalité en mettant en parallèle deux modules corrects qui n'en ont pas. Les principales caractéristiques de l'approche synchrone restent valables: un opérateur de parallélisme simple et puissant, une programmation déterministe qui rend plus facile le debuggage de programmes, et une programmation à base d'événements diffusés.

ECL [LAV 99] est un langage pour spécifier des systèmes embarqués. Le nom ECL (de l'anglais Esterel-C Language) est utilisé pour le langage et pour le compilateur. L'objectif du langage est de combiner les meilleures caractéristiques des deux langages, Esterel et C. Le langage utilisé est ANSI-C avec l'ajout d'instructions inspirées d'Esterel pour spécifier la réactivité du système. Le langage supporte le mélange de modules, de données et de code. Le code C est utilisé pour spécifier les structures qui n'ont pas d'équivalent en Esterel: les boucles instantanées, la définition de types de données et quelques structures de contrôle.

Le compilateur sépare le code source, générant du code C et du code Esterel, les deux codes sont ensuite compilés par les compilateurs correspondants. La partie réactive est la plus flexible car elle est traduite dans une machine d'états finis qui permet l'analyse des programmes pour ensuite choisir le type d'implémentation: soit en matériel soit en logiciel. L'analyse est basée sur l'estimation de la taille et de la vitesse du programme. La partie en C doit être implémentée en logiciel.

Jester [WEB Jester] est une extension à Java qui permet de spécifier, de vérifier et d'exécuter des programmes Esterel. L'objectif de Jester est faciliter la programmation de systèmes embarqués avec Esterel en utilisant Java et en particulier en utilisant la Spécification de Java Embarqué (*The Embedded Java Specification* [JavaEmbed]). Les programmes Jester mélangent du code Esterel et du code Java dans un seul fichier (*.jst). La syntaxe du code Esterel est similaire à celle qui a été formellement définie en Esterel mais avec l'esprit de Java (par exemple, la portée des instructions est donnée avec des accolades et pas par le mot `end` *`name`*). Le mélange du code Esterel et du code Java n'est pas si généralisé que l'on pourrait penser: le code Esterel doit se trouver dans une méthode "réactive" appelée `reaction` qui n'accepte que quelques instructions Java dans les instructions réactives. En général le code Java est utilisé par les méthodes réactives avec les instructions `run` et `exec` qui ont une sémantique bien claire en Esterel. Ce qu'apporte Jester à Esterel est la facilité de définir des types de données, la création de tâches (threads) exécutées de façon asynchrone avec l'instruction `exec`, et la réutilisation de code en utilisant l'héritage (avec beaucoup de restrictions pour éviter les problèmes).

Quartz [SCH 00, SCH 01] est une variante d'Esterel qui élimine les problèmes de causalité en exécutant les affectations et la génération d'événements à l'instant d'après. Quartz définit quelques instructions qui ne sont pas présentes en Esterel, par exemple l'exécution non-déterministe de deux programmes et l'exécution asynchrone de branches parallèles (une des branches peut être exécutée lors de plusieurs macro-step tandis que les autres en prennent un ou aucun). La modèle de Quartz a été vérifié dans le prouveur de théorèmes HOL (*Higher Order Logic*) qui a servi aussi pour donner une traduction correcte des programmes Quartz en circuits.

Reactive-C (ou RC) [BOU 91, BOU 96e] est un langage qui permet une programmation réactive en C. L'approche réactive introduit une notion d'instant correspondant aux activations des programmes. RC induit

un style d'écriture dans lequel on manipule explicitement les instants. Il définit un ensemble d'instructions réactives dont les plus importantes sont l'instruction Stop (qui termine la réaction courante), l'instruction Rif (extension réactive du if traditionnel) et l'instruction Merge (opérateur de parallélisme déterministe, réalisant un *interleaving* au cours des instants).

Les programmes sont structurés en procédures réactives dont les corps sont formés d'instructions réactives. L'appel d'une procédure provoque la réaction de son corps. Les procédures réactives peuvent être définies récursivement. RC doit surtout être considéré comme un assembleur réactif, c'est-à-dire un langage noyau utilisable pour implémenter des modèles ou langages de niveau plus élevés. Parmi les implémentations déjà réalisées on peut citer: le langage SL et les réseaux de processus réactifs. Le travail sur RC a débuté en 1988. Deux versions d'un compilateur ont été réalisées qui sont en fait des préprocesseurs, générant du C. La dernière version de RC accepte la syntaxe C++.

SugarCubes [BOU 98c, BOU 97] est un langage de programmation réactive qui propose une implémentation du modèle réactif synchrone au-dessus de Java. Par rapport à Reactive-C, SugarCubes propose une implémentation dans laquelle la notion d'événement et la gestion des instants sont intégrées dans le moteur d'exécution. La version actuelle de SugarCubes (la v3 et prochainement la v4) dispose d'un ensemble très riche de primitives réactives (43 instructions réactives) dont une est particulièrement importante: le Cube. Le Cube définit un modèle d'objet réactif dans lequel un comportement réactif est associé à un objet Java.

SugarCubes est un ensemble de classes Java qui implémente des événements, des comportements réactifs et la machine réactive qui les exécute. Les primitives réactives du langage sont implémentées par des objets Java instanciés à partir de ces classes. Un programme en SugarCubes est donc une collection d'objets ; le programme est compilé et exécuté comme n'importe quel programme Java. Il ne s'agit donc pas à proprement parler d'un langage à part entière avec une grammaire spécifique mais d'une API de programmation.

Les SugarCubes ont été utilisés pour implémenter d'autres formalismes, par exemple les Reactive Scripts et les Icobjs.

Junior [HAZ 99] est un langage de programmation réactive qui a été créé à partir de SugarCubes. Comme les SugarCubes, Junior propose aussi une implémentation de l'approche réactive en Java dans laquelle l'objectif principal a été de donner une sémantique formelle aux instructions réactives. La sémantique est donnée avec des règles de réécriture du type SOS (*Structural Operational Semantics*) [PLO 81] et plusieurs implémentations ont été créées (Rewrite, Replace, Simple, Storm, Glouton, etc.). Etant donné que Junior est une pièce centrale de la thèse, il est expliqué en détails dans les chapitres 2 et 3.

Junior a été développé par l'EMP-CMA, l'Inria, et France Telecom/R&D. Il est utilisé dans plusieurs projets: il a été utilisé pour implémenter d'autres langages (Rhum et Rejo), ainsi que quelques applications comme les Icobjs et la plate-forme PING (Projet IST).

1.2 Le paradigme d'agents mobiles

Grâce à l'arrivée des réseaux d'ordinateurs, il a été possible de construire un nouveau type d'applications : les applications réparties. Ces applications offrent plusieurs avantages mais aussi posent certaines difficultés dans leur exécution. Malgré les inconvénients, on les adopte parce que les avantages, comme la tolérance aux défaillances et la disponibilité de ressources offerte par la réplication, sont des caractéristiques dont on a besoin aujourd'hui. Pour faciliter la construction, plusieurs paradigmes ont été proposés. Le paradigme le plus utilisé est le paradigme client-serveur; cependant de nouveaux paradigmes sont apparus mieux adaptés à certains types d'applications. L'un de ces paradigmes est celui des Agents Mobiles qui offre les avantages suivants:

- Réduction de la bande passante utilisée.
- Réduction du temps de latence (*Latency*) et du temps total d'exécution.
- Continuité de service en cas de déconnexion.
- Equilibrage de charge (*load balancing*).

- Développement dynamique des applications.

Bien que ces avantages semblent moins importants avec l'augmentation de la bande passante disponible sur le réseau, le paradigme des agents mobiles paraît bien adapté à certaines applications, par exemple à la personnalisation des communications (dans les réseaux actifs).

Dans la suite, on présente les sujets d'études principaux autour des agents.

1.2.1 Etat de l'art

La plupart des applications réparties utilisent le paradigme client-serveur, dans lequel le client et le serveur communiquent en utilisant une technique *Message Passing* ou *Remote Procedure Calls* (RPC). RPC est normalement synchrone: le client suspend son exécution après avoir envoyé sa requête au serveur, et attend les résultats de l'appel. Dans le RPC, les données sont transférées entre le client et le serveur, dans les deux directions. Une autre approche appelée *Code on Demand* (COD) permet au client de télécharger du code dont il a besoin, mais qui ne se trouve pas localement, pour accéder aux ressources et aux données. Une autre architecture appelée *Evaluation Remote* permet, au lieu d'invoquer une procédure distante, que le client envoie son code au serveur, en demandant que le serveur l'exécute avec ses données et ses ressources pour retourner les résultats. Plus récemment, le concept de message actif a été introduit. Un message actif peut migrer d'un nœud à un autre, en transportant le code du programme pour que celui-ci puisse être exécuté par les nœuds. Une approche plus générale est l'*Agent Mobile* (AM) qui encapsule des données avec leurs opérations pour migrer d'un nœud à un autre. L'approche de AM s'inspire de la notion d'agent qui s'utilise dans le domaine de l'intelligence artificielle: un agent est un programme suffisamment autonome pour fonctionner de façon indépendante même si l'utilisateur ou l'application qui l'a lancé n'est plus disponible pour prendre des décisions en présence d'erreurs ou pour le guider dans son exécution.

1.2.2 Conception d'un SAM

La méthodologie orientée objet introduit la notion d'objet dans les langages de programmation. Un objet est formé d'un ensemble de variables et de leurs méthodes. Les objets sont des éléments passifs car ils n'ont pas de comportement et leur exécution ne dépend que de l'interaction entre les objets; donc un objet n'est pas responsable de son exécution. Un objet passif qui a la capacité de migrer est appelé *Mobile Code* (Code Mobile) par exemple, le *mobile code* est utilisé dans le contexte des programmes qui s'exécutent sur place, par exemple les Applets Java. Par ailleurs, il y a les objets actifs qui sont des objets qui ont un comportement associé et qui peuvent interagir avec des autres objets. Un exemple d'un objet actif est l'*Intelligent Agent* (IA, Agent Intelligent). Les IAs sont typiquement des entités statiques parce qu'ils n'ont pas la possibilité de migrer en prenant la décision eux-mêmes, et ainsi pouvoir faire une tâche spécifique comme l'administration des réseaux. Un AM est soit un programme, soit un objet actif qui peut migrer de manière autonome, d'une place à une autre pour continuer son exécution comme le comportement d'un usager. Une des principales différences entre l'IA et le AM est que les AMs sont des entités dynamiques qui peuvent migrer, tandis que l'IA ne peut pas migrer. Un AM n'a pas nécessairement d'intelligence. Nous pouvons imaginer la notion d'*Intelligent Mobile Agent* (IMA, Agent Mobile Intelligent) comme un hybride entre un AM et un IA. Un IMA peut être vu comme une entité autonome intelligente qui à la possibilité de migrer de manière autonome pour exécuter une tâche d'une façon répartie, comme un administrateur de réseaux répartis.

L'étude des agents mobiles touche, principalement, les sujets qui sont expliqués dans la suite et qui sont largement discutés en [FUG 98].

Mobilité

Un AM comme son nom l'indique doit avoir la facilité de migrer de manière autonome d'un site à un autre. Le support pour les agents mobiles est une exigence fondamentale pour une infrastructure d'agents. Il existe deux types de mobilité: 1) la *Strong Mobility* (mobilité forte) est la capacité d'un Système d'Agents Mobiles (SAM) à faire la migration du code et de l'état d'exécution d'un agent vers différents hôtes, 2) La *Weak Mobility* (mobilité

faible) est la capacité d'un SAM à transférer le code à travers différents hôtes; le code peut être accompagné par quelques initialisations des données mais l'état d'exécution n'est pas migré.

La mobilité forte est supportée par deux mécanismes: la migration et le clonage distant. Le mécanisme de migration suspend un agent, le transmet à la machine de destination, et puis le reprend. La migration peut être soit proactive soit réactive. Dans la migration proactive le temps et la destination de la migration sont déterminés par l'agent de manière autonome. Dans la migration réactive le mouvement est déclenché par un agent différent ou peut être provoqué par un usager en relation avec l'agent qui est migré. Le mécanisme de clonage distant crée une copie d'un agent dans un hôte distant. Le clonage distant diffère du mécanisme de migration car l'agent original n'est pas détaché de sa localisation actuelle. Comme dans la migration, le clonage distant peut être soit proactif soit réactif. Un mécanisme qui supporte la mobilité faible doit fournir la capacité de transférer le code sur le réseau. Un agent peut soit lire le code soit copier le code d'un autre hôte. Le code peut être migré soit comme un code *Stand-Alone* soit comme un Code Fragmenté. Un code *stand-alone* n'a pas besoin d'autres bibliothèques pour créer un nouvel agent dans le site destinataire. Par contre, le code fragmenté doit être lié dans le contexte d'exécution aux routines manquantes et éventuellement exécuté (lié dynamiquement).

Routage

Les agents mobiles, comme leur nom l'indique, doivent avoir la facilité de migrer de manière autonome d'un site à un autre pour continuer leur exécution; un agent mobile doit donc connaître l'adresse du nœud destinataire. Plusieurs approches ont été proposées pour résoudre ce problème. Une des ces approches est le transfert des adresses entre les agents. L'approche *Itinerary* consiste à avoir une liste des sites à visiter, à l'intérieur de l'agent, et une liste des services avec les sites qui les offrent. Cette dernière approche présente quelques désavantages; par exemple, la liste doit être mise à jour chaque fois qu'un service change de localisation ou qu'un service est ajouté ou éliminé. Une autre approche, appelée *forwarding*, consiste à renvoyer les agents systématiquement vers un autre site; l'avantage de cette approche est que la gestion du *forwarding* n'est pas faite par l'agent et elle est donc indépendante des modifications de la plate-forme.
Une autre nouvelle approche, appelée *Ticket* est celle proposée par Bradshaw [WHI 97] qui consiste à envoyer un agent vers un autre site avec quelques informations, par exemple, la QoS (*Quality Of Service*) et les permis nécessaires pour que l'agent soit exécuté. L'idée de cette approche est de choisir intelligemment la machine destinataire, par exemple on pourrait stocker le temps que l'agent va attendre pour qu'un site l'accueille, ce qui permettrait de ne pas perdre du temps en essayant de migrer vers un site déconnecté.

Communication

Pour effectuer leur travail, les agents ont besoin de communiquer entre eux. Actuellement, les systèmes utilisent différents mécanismes de communication selon leurs propriétés et leurs besoins. Une approche est le *Message Passing* qui permet aux agents de s'envoyer des messages asynchrones ou de créer une connexion vers chaque agent. *Method Invocation* est une autre approche pour faire la communication dans un système basé objets. Pour des agents qui ne sont pas sur le même site, le serveur peut fournir un service *Remote Method Invocation* (RMI). Cette approche permet à deux (ou plus) agents d'invoquer leurs opérations en utilisant leurs références. La communication peut être aussi implémentée en utilisant les segments de mémoire partagée (*Shared Data*) qui permet aux agents d'échanger directement leurs données. Une approche alternative pour la communication de groupe est le *Publish Subscribe*, dans laquelle l'agent s'enregistre lui-même en envoyant au manager d'événements une liste d'événements qu'il veut recevoir. Enfin, un autre modèle est la diffusion (*Broadcast*) événementielle.

Pour communiquer avec un agent, il est nécessaire de connaître sa localisation. Pour résoudre ce problème, le mécanisme de nommage peut être utilisé. Un agent peut trouver la localisation d'un autre agent en utilisant son adresse qui est codée dans le nom de l'agent.

Indépendamment du mécanisme de communication utilisé, il y a deux façons principales de faire communiquer les agents, à savoir la Communication Intra-groupe qui s'effectue entre agents qui appartiennent au même groupe, et la Communication Inter-groupe, c'est-à-dire la communication entre agents qui appartiennent à des groupes différents. Enfin, on peut distinguer la communication entre des agents qui appartiennent au même groupe et qui sont dans la même machine, communication locale intra-groupes, et la communication intra-groupes entre différentes machines, communication globale intra-groupes.

Nommage

Dans un système, les agents ont besoin d'un nom unique. Certains SAM utilisent le mécanisme de nommage pour trouver la localisation actuelle d'un agent en utilisant son nom. Ces systèmes changent le nom de l'agent quand il migre en utilisant le nom de l'hôte et le nom du port. Le problème avec ces systèmes est qu'ils doivent mettre à jour le nom d'un agent à chaque fois qu'il migre. Ces approches sont appelées *Location Dependent*. D'autres systèmes utilisent un *Domain Name Server* (DNS) pour résoudre le problème. Le nommage est global, indépendant de la localisation, et le nom n'a pas besoin de changer quand l'agent migre. Le Serveur de Noms peut être implémenté de façon centralisée ou répartie. Durant la migration, le serveur doit mettre à jour la correspondance entre la localisation actuelle d'un agent et son nom global.

Langage de Programmation

L'un des problèmes les plus importants à résoudre dans la construction d'un système d'Agents Mobiles est le langage de programmation à utiliser. Quelques caractéristiques que le langage de programmation doit réunir sont: portabilité, robustesse, sécurité, et efficacité.

Le système a besoin de portabilité puisqu'un agent doit pouvoir migrer sur des machines hétérogènes avec des systèmes d'exploitation différents; la portabilité d'un SAM est une propriété fondamentale. La robustesse est une autre propriété importante car le SAM doit fournir une plate-forme sûre à tous les agents qui arrivent. Le SAM doit garantir que le système fonctionnera malgré la présence d'un bug d'un agent. La sécurité d'un SAM consiste en 4 propriétés: confidentialité, intégrité, disponibilité et authentification. Finalement, l'efficacité signifie avoir le minimum de surcharge d'exécution pour assurer la portabilité, la robustesse et la sécurité.

Plusieurs systèmes utilisent des langages de scripts car ils permettent un prototypage rapide, un meilleur debuggage, et ils sont simples et dynamiques. Etant donné que les langages de scripts sont généralement faibles dans les aspects modularité, encapsulation et performance, quelques systèmes utilisent un langage orienté objets comme Java. Ces systèmes profitent des avantages des langages orientés objets pour construire des agents complexes tout en gardant le dynamisme.

Sécurité

Le terme de sécurité doit être entendu ici au sens plus large, englobant les notions d'authentification, d'autorisation, de protection, de confidentialité.

Le site d'accueil d'un agent doit se protéger contre la sur-utilisation des ressources et les actions malveillantes (intentionnelles ou non) des agents accueillis. Ceci suppose l'identification du propriétaire, du site de provenance, et la vérification du code. Des limitations sur la consommation des ressources (temps CPU, taille mémoire) peuvent également être fixées, et l'accès au système de fichiers restreint à une sous-arborescence, comme le ftp anonyme. Enfin, le système peut donner des permissions différentes aux agents dont le code provient d'un fichier local et aux agents dont le code est reçu via le réseau.

Inversement, l'agent doit avoir l'assurance que ni son code ni ses données ne seront modifiés par le système hôte. Ce problème est plus complexe que le précédent, puisque tout ce qui est accessible à l'agent l'est aussi au système d'exploitation dans lequel il s'exécute, et n'a pas reçu, à ce jour de solution satisfaisante.

1.2.3 Normes et SAM

Plusieurs groupes de recherche travaillent actuellement dans le domaine des SAM. Chaque approche proposée apporte certaines innovations par rapport aux autres, néanmoins toutes ont un ensemble de caractéristiques similaires. Parmi les groupes qui font de la recherche sur les SAM, quelques-uns travaillent sur la standardisation de l'AM, par exemple FIPA [WEB FIPA], AgentSociety [WEB AgSoc], AgentX, OMG MASIF [MASIF], W3C [WEB W3C], The Mobility Mailing List [WEB AgMail], UMBC AgentWeb [WEB AgWeb], etc.

Actuellement, la plupart des SAM utilisent une architecture modulaire qui parfois est structurée en couches. Ces couches commencent par définir un comportement individuel puis un comportement social et finalement elles définissent certaines normes. Chaque module implémente certains services par exemple: persistance, sécurité, nommage, communication, Agent Tracking, Agent Transport, Agent Management, etc.

Etant donnée la grande quantité de SAM qui ont été développés [1] et qui continue à croître, on finit cette section avec la présentation de la table 1.1 qui résume les caractéristiques de quelques SAM. Le lecteur intéressé peut trouver plus d'information dans les sites [WEB Agents].

Caractéristique	Description	SAM
Langage	Tcl/Tk: Java: C/C++: Orienté Objet: autres :	TACOMA, Ara, Agent Tcl. Ara,Aglets,Voyager,MARS,Concordia. TACOMA,Ara,Messangers(melange C) Telescript(C++),Obliq. Messangers(postcript).
Communication	Message-passing: Broadcast:	Agent Tcl, Aglets, MARS, Odyssey. RAMA, Rejo/ROS.
Protocole	TCP/IP: RMI: autres :	TACOMA, Odyssey, MARS. Odyssey, RAMA, Rejo/ROS. Aglets(ATP=AgentTransferProtocol)
Persistence		Concordia
Sécurité	Authentification:	Ara Concordia (Tamper)
Mobilité	Strong: Weak:	Ara,AgentTcl,Telescript,RAMA,Ajanta, Concordia,Rejo/ROS. Tacoma,Aglets,Obliq,Messengers,Voyager.
Routage	Itinerary: Ring:	Concordia RAMA

Tab. 1.1: Caractéristiques de SAM

Finalement on voudrait définir le terme d'agent mobile que l'on va utiliser, vue la grand variété des définitions qui ont été données [FRA 96]. La définition que l'on va utiliser est celle donnée par Z. Guessoum [GUE 96]: un agent est une entité active, autonome mais sociable, qui peut être intelligente et qui interagit avec un environnement dynamique, auquel elle doit s'adapter. En particulier on va s'intéresser aux agents mobiles: des agents qui sont capables de migrer entre deux sites.

1.3 Objectifs de la thèse et réalisations

Le modèle synchrone s'occupe fondamentalement de systèmes statiques dont le nombre de composantes parallèles et d'événements est connu à la compilation et ne varie pas au cours de l'exécution. Au contraire, le modèle réactif synchrone se propose de lever cette restriction en considérant des systèmes que l'on appellera dynamiques, et dont la structure pourra évoluer au cours de l'exécution. Le modèle réactif synchrone a été, dans un premier temps, implémenté en C (Reactive-C [BOU 91]) puis en Java (SugarCubes [BOU 98c] et Junior [HAZ 99]) et en Scheme (Senior [DEM 01]).

On cherche un langage avec les caractéristiques suivantes :

- Un langage de haut niveau orienté objet et son modèle d'objets réactifs associé. Le langage offert par Junior et SugarCubes est de bas niveau et sans mécanisme de modularité. Aucun des langages synchrones n'est orienté objet et la modularité est fortement limitée par les problèmes de causalité;
- Un langage qui intègre de façon transparente le code réactif et le code impératif. Dans le cas des langages synchrones comme Esterel le mélange est fortement restreint à cause du modèle; le code impératif est souvent codé à part dans un autre langage. Dans le cas de l'approche réactive synchrone, il y a eu

[1] Parmi les plus connus on peut citer : Agent Tcl, MARS, Aglets, KQML, Ara, Arcadia, Ajanta, Gossip, Concordia, BizBots, Java-To-Go, Grasshopper, Knowbots, Messengers, Mole, TACOMA, Telescript, Voyager, Obliq, Odyssey, Mubot.

quelques essais comme les *Reactive Scripts* [BOU 96a], mais dans ceux-ci l'intégration reste limitée et seul le cas interprété est considéré;

- Un langage qui puisse s'adapter aux nouveaux paradigmes de programmation, par exemple la migration dynamique de code (de comportements réactifs). Ceci est impossible dans les langages purement synchrones comme ESTEREL, et dans le cas des langages réactifs synchrones, il n'existe pas de primitive de migration ni d'environnement d'exécution nécessaire.

L'objectif donc est de créer un langage de programmation de haut niveau qui sert à construire des systèmes réactifs. On veux avoir un langage avec une syntaxe et une sémantique simple et claire, que l'on puisse utiliser pour construire des systèmes dynamiques pour faire, en particulier, la migration de code.

En particulier, on va chercher à rendre la programmation des moteurs réactifs comme celui de Junior plus simple. L'objectif est de cacher la programmation de bas niveau de Junior (comme celle de SugarCubes) tout en explorant l'implémentation de quelques aspects manquants: un mécanisme de modularité, un modèle d'objet, et un mécanisme de migration, entre autres choses.

Ce travail est la suite du travail que j'ai fait dans le stage du DEA Réseaux et Systèmes Répartis (voir mémoires [ACO 00a]. Le stage a consisté à améliorer une plate-forme d'agents mobiles appelée RAMA dont on va aussi poursuivre l'étude; ce qui nous intéresse sont les problèmes que pose ce type de systèmes en relation avec le modèle réactif synchrone.

Voici, pour terminer, une description abrégée de mon apport personnel :

- Analyse de la programmation et des implémentations de Junior. Ce travail s'est concentré sur trois aspects : 1) réalisation d'un workbench pour évaluer les différentes implémentations de l'approche réactive, 2) proposition et implémentation de plusieurs nouvelles instructions réactives, et 3) étude de nouvelles instructions réactives, par exemple, pour la QoS ou le *pattern-matching*.
- Conception et implémentation d'un nouveau langage de programmation réactive appelé Rejo. L'appendice A donne la syntaxe précise du langage et les chapitres 5 et 7 illustrent sa programmation. Le langage Rejo a été utilisé dans deux projets IST à FranceTelecom/R&D : la plate-forme PING et la simulation de vie artificielle [LIM 01]. J'ai également enseigné Rejo à l'ESSI (École Supérieure en Sciences Informatiques) dans l'apprentissage de la programmation réactive en Java.
- Implémentation de la plate-forme ROS pour l'exécution de programmes réactifs. Cette plate-forme facilite l'exécution des programmes Rejo mais aussi la construction de systèmes réactifs basés sur le paradigme d'agents mobiles. La plate-forme ROS utilise une architecture de micro-noyau extensible, deux programmes réactifs (un shell réactif Rsh et une interface graphique Ricobjs) ont été implémentés pour illustrer le dynamisme de l'approche.
- Réalisation d'une version expérimentale de Rejo et de Junior en C (RAUL et Cr respectivement).
- Publication de trois papiers [ACO 00b, ACO 02, ACO 03], qui décrivent le langage Rejo et la plate-forme ROS et réalisation d'une page web [WEB RejoRos] qui regroupe toute l'information autour de Rejo (papiers, logiciels, etc.).

1.4 Structure du document

Cette thèse contient les chapitres suivants :

Le ***chapitre 2*** décrit Junior : le modèle d'exécution, la sémantique formelle et la façon de programmer ; on verra aussi quelques exemples. Pour ceux qui ne connaissent pas le modèle synchrone ou les particularités du modèle Réactif-Synchrone, ce chapitre est indispensable.

Le ***chapitre 3*** décrit les diverses implémentations de Junior qui ont été faites. Ce chapitre contient des sous-sections pour : 1) analyser les performances des différentes implémentations, 2) étudier les aspects fins

de la programmation avec Junior, et 3) décrire des travaux similaires. La deuxième sous-section contient aussi les différentes expérimentations et propositions pour soit améliorer Junior, soit l'adapter au langage de programmation créé.

Le ***chapitre 4*** présente une série de tests de vitesse pour mesurer les performances des différentes implémentations de Junior, de SugarCubes et des implémentations du modèle réactif en C, comme Loft, Cr et Jrc. Ce workbench souligne les avantages et désavantages de chaque moteur réactif.

Le ***chapitre 5*** est consacré à la description du langage Rejo : modèle d'exécution, syntaxe et sémantique. Le langage est expliqué avec plusieurs mini-exemples pour chaque instruction mais aussi, à la fin, avec des exemples plus réels.

Le ***chapitre 6*** détaille l'implémentation du langage. On fait une analyse des qualités du compilateur et on étudie aussi les travaux similaires. En particulier dans ce chapitre on décrit une implémentation de Rejo en C dans laquelle quelques nouvelles idées sont expérimentées.

Le ***chapitre 7*** est dédié à la description du système d'agents mobiles construit, ROS. On donne l'architecture du système, l'API de programmation, la structure des agents et l'étude des travaux similaires. Pour illustrer l'utilisation de ROS, on présente l'implémentation de deux applications.

Finalement, le ***chapitre 8*** donne les conclusions des apports de ce travail et les perspectives.

Chapitre 2

Junior: modèle, programmation et sémantique

2.1 Introduction

JUNIOR est né des expériences réalisées avec la première implémentation du modèle réactif synchrone en Java, les SugarCubes. En particulier Junior est né avec l'objectif de donner une sémantique formelle au modèle réactif synchrone; la sémantique permet une meilleure compréhension du modèle et de valider les différentes implémentations.

Le modèle d'exécution de Junior est, grosso modo, le même que celui de SugarCubes. Ce modèle est décrit dans la section 2.2. Plusieurs implémentations de Junior ont été faites; pour standardiser leur programmation, il a été défini une interface de programmation (API). L'API de Junior et la façon de construire un programme en Junior sont données dans la section 2.3. La section 2.4 donne quelques exemples de programmes Junior. La section 2.5 présente la sémantique de Junior; cette section est un résumé du papier [BOU 00d]. Finalement la section 2.6 présente les conclusions.

Le langage Rejo proposé dans ce texte est construit au-dessus de Junior que l'on présente dans ce chapitre de façon succincte. Le lecteur intéressé par plus de détails sur Junior peut lire [HAZ 00a].

2.2 Modèle d'exécution

Le modèle d'exécution de Junior est celui de l'approche réactive synchrone. L'approche réactive a été conçue pour programmer des systèmes combinant les deux caractéristiques principales suivantes:

- Les systèmes considérés sont *cycliques*, c'est-à-dire qu'il ne finissent pas. En ce sens, ils différent des programmes traditionnels qui, après avoir reçu des données, s'exécutent pendant un certain temps, puis terminent en fournissant un résultat. Au contraire, un système réactif réagit de manière continue à l'évolution de son environnement.
- Ils doivent être suffisamment *rapides* pour suivre les changements de leur environnement. En fait, on considère que l'environnement détermine le rythme des activations d'un système réactif: un système réactif réagit à une activation et, suivant l'état de l'environnement, le transforme; puis, il attend une nouvelle activation et réagit à nouveau, et ainsi de suite indéfiniment.

L'approche réactive est caractérisée par 4 éléments :

- la notion d'instant;
- les comportements réactifs;
- la notion d'événement;
- la propriété de dynamisme.

Ces 4 éléments sont expliqués dans la suite.

La notion d'instant

Chaque activation du système réactif définit une étape dans son évolution. On appelle ces étapes ***instants***. Les instants définissent ainsi une base de temps abstraite (une sorte d'horloge virtuelle) pour le système. Il est important de noter que, à ce niveau, la durée effective des instants peut être quelconque.

L'évolution d'un système réactif au cours du temps est représentée sur la figure 2.1.

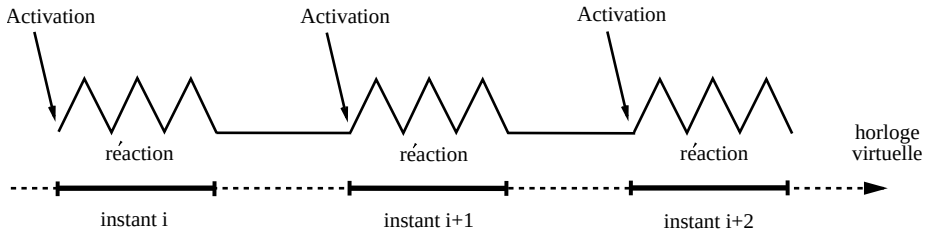

FIG. 2.1: Approche réactive synchrone

Les comportements réactifs

En junior les réactions effectuées par le système réactif en réponse aux activations sont des comportements réactifs spécifiques incarnés sous la forme d'instructions. Junior définit un ensemble d'instructions réactives (présentées dans la sous-section 2.3.3) qui peuvent être activées ou réinitialisées. Chacune des activations retourne l'une de valeurs suivantes:

TERM (pour terminée): signifie que l'instruction est totalement terminée. Il n'y a plus rien à exécuter à l'instant présent ni aux instants suivants. Ainsi, les activations suivantes d'une instruction terminée n'auront aucun effet et retourneront toujours TERM (voir figure 2.2).

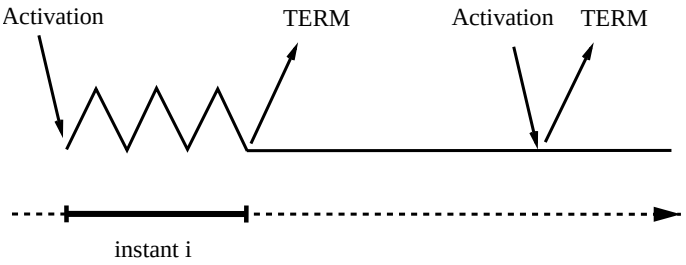

FIG. 2.2: Comportement qui rend TERM

STOP (pour stoppée): signifie que l'instruction est terminée pour l'instant courant, mais qu'il reste du code à exécuter à l'instant suivant (voir figure 2.3).

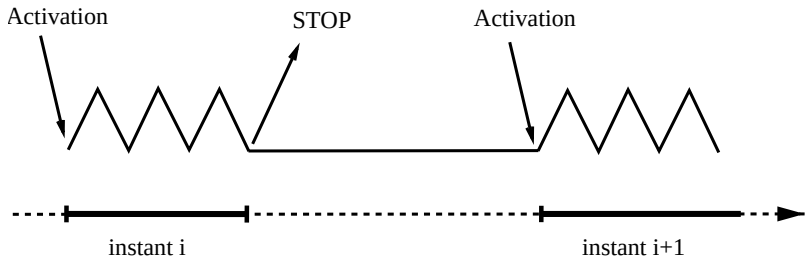

FIG. 2.3: Comportement qui rend STOP

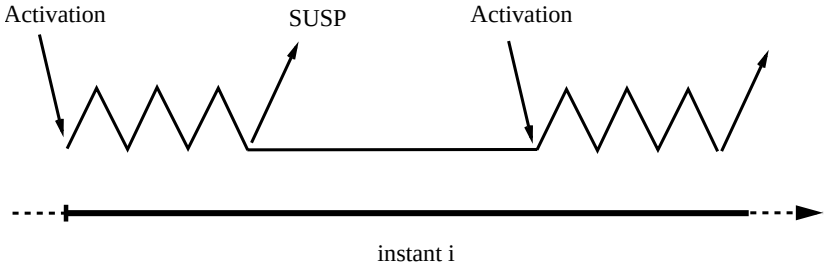

FIG. 2.4: Comportement qui rend SUSP

SUSP (pour suspendue): signifie que l'instruction n'a pas atteint un état stable et qu'en conséquence son exécution doit être poursuivie dans l'instant présent (voir figure 2.4). C'est le cas de l'attente d'un événement non généré; l'exécution est suspendue pour permettre aux autres instructions exécutées au cours du même instant de le générer éventuellement.

La notion d'événement

L'approche réactive propose une notion d'événement diffusé instantanément à tous les comportements réactifs, comme moyen de communication. Les événements sont des données qui appartiennent à l'environnement d'exécution du système. La notion d'instant permet de définir clairement les notions de présence, d'absence et de simultanéité d'événements auxquelles un programme peut réagir.

La sémantique des événements est la suivante:

- Les événements sont non-définis au début de chaque instant. Ce ne sont pas des données persistantes car ils perdent leur valeur (présent ou absent) à la fin de chaque instant.
- Les événements sont générés lors de l'exécution d'une instruction réactive appelée *generate*. Après avoir été généré, un événement devient présent pour l'instant courant. Générer un événement déjà présent n'a aucun effet.
- Les événements sont diffusés instantanément. Toutes les instructions réactives ont la même vision de chaque événement à chaque instant.
- On ne peut décider qu'un événement est absent durant un instant qu'à la fin de celui-ci. Ainsi la réaction à l'absence d'un événement est nécessairement repoussée à l'instant suivant.

Un programme réactif peut également manipuler des expressions booléennes événementielles basées sur la présence d'événements à un instant donné. Ces expressions événementielles seront présentées dans la sous-section 2.3.2.

La propriété de dynamisme

La propriété de dynamisme du modèle réactif porte sur deux aspects:

1. La génération d'événements : les programmes peuvent générer n'importe quel type d'événement. Les événements ne sont pas déclarés.
2. Le nombre d'instructions : la taille du programme réactif à exécuter n'est pas connue en avance (comme dans l'approche synchrone) et celui-ci peut même changer lors de son exécution.

Les quatre éléments que l'on vient de décrire sont gérés par ce que l'on appelle une machine réactive. Une représentation graphique de l'approche réactive synchrone est donnée dans la figure 2.5.

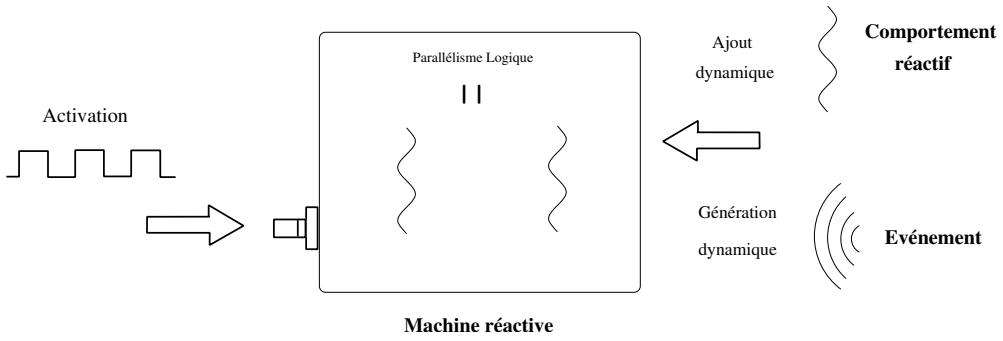

FIG. 2.5: Approche réactive synchrone

En conclusion l'Approche Réactive Synchrone peut être résumée comme :

$$\text{Approche Réactive Synchrone} = \text{Instant} + \text{Comportement Réactif} + \text{Evénement} + \text{Dynamisme}$$

Par rapport au modèle synchrone, le modèle réactif synchrone se différencie par :

- La durée d'un instant. Dans l'approche synchrone la durée d'un instant est nulle tandis que dans le modèle réactif elle est non-nulle. La répercution principale de cette différence est la possibilité de créer, dans l'approche synchrone, des programmes incohérents (problèmes de causalité). Les autres différences sont données dans les points suivants.
- Le dynamisme. Dans l'approche synchrone on construit des programmes statiques, c'est-à-dire des programmes qui ne changent pas de taille et dont le nombre d'événements est connu à l'avance.
- La préemption. Dans l'approche synchrone on peut faire de la préemption forte et faible. Dans le modèle réactif il y a que la préemption faible qui est possible.
- Le déterminisme. L'approche synchrone est strictement déterministe, elle rejette les programmes qui ont plus de deux solutions. L'approche réactive n'est pas obligée d'être déterministe, et à l'heure actuelle certaines implémentations sont déterministes et certaines non-déterministes.

2.3 Programmation et API

Pour programmer en Junior, les programmeurs doivent savoir utiliser 4 choses :

- les wrappers et les identifiers;
- les configurations événementielles;
- les instructions réactives;

- la machine réactive et son environnement.

Pour avoir une interface de programmation commune à toutes les implémentations de Junior qui ont été développées, il a été créé une interface de programmation appelée Jr qui spécifie le format (la signature en termes Java) de la plupart des 4 éléments mentionnés antérieurement.

Par ailleurs, l'interface Jr est une interface très primitive et il a été créé donc une extension appelée Jre qui rend la programmation plus facile. On préféra l'utilisation de l'interface Jre lors de la présentation des quelques exemples.

Les sous-sections suivantes expliquent chacun des ces éléments.

2.3.1 Les wrappers et les identifiers

Les wrappers en Junior ont été définis pour palier l'absence de pointeurs en Java qui sont nécessaires dans la construction d'un programme réactif. Cette absence se fait aussi sentir lorsqu'on veut migrer un programme réactif (les adresses des objets changent).

Un wrapper n'est pas autre chose qu'une méthode qui rend une valeur d'un type particulier. En termes Java, un wrapper est une interface qui ne spécifie qu'un objet, implémentant l'interface, contient une méthode `evaluate` qui reçoit comme paramètre l'environnement d'exécution et qui rend un type bien connu. Junior définit 5 types de wrappers: `IntegerWrapper`, `BooleanWrapper`, `IdentifierWrapper`, `ProgramWrapper` et `ObjectWrapper`.

Par exemple, voici le code complet de l'interface `BooleanWrapper`.

```
package junior;

public interface BooleanWrapper extends java.io.Serializable
{
   boolean evaluate(Environment env);
}
```

Etant donné que les wrapper ne sont que des interfaces, il faut créer des objets qui les implémentent. L'implémentation d'un wrapper dépend, en général, de ce que l'on veut faire. Par exemple, si on veut implémenter un `BooleanWrapper` qui rend vrai lorsque la condition `i.getValue()==5` est vraie, on construit la classe suivante.

```
package junior;

public class ImplemBooleanWrapper
{
  Integer i;

  ImplemBooleanWrapper(Integer ii){
   i=ii;
  }

  boolean evaluate(Environment env){
    return i.getValue()==5;
  }
}
```

Ceci veut dire que c'est le programmeur qui doit construire ses propres implémentations des wrappers. Pour rendre la tâche plus facile dans les cas où la valeur à rendre est constante et connue à la construction, Junior définit 5 objets qui implémentent chacune des interfaces: `ConstBooleanWrapper`, `ConstIntegerWrapper`, `ConstIdentifierWrapper`, `ConstProgramWrapper`, `ConstObjectWrapper`.

Identifiers

La représentation d'un événement en Java se fait en utilisant un type particulier. Si on choisissait, par exemple, le type `String` on serait limité dans le pouvoir expressif (et aussi dans certain cas dans l'efficacité du programme) des types d'événements que l'on pourrait générer. Pour ne pas avoir cette limitation et pouvoir générer des événements d'un type quelconque, Junior définit l'`Identifier`. La seule chose dont on a besoin pour généraliser la notion d'événement est d'avoir un mécanisme qui nous permet de savoir si deux événements sont égaux. L'Identifier est une interface qui spécifie que les objets qui l'implémentent doivent avoir deux méthodes:

- `boolean equals(Object object)`: Cette méthode est utilisée pour savoir si deux objets sont égaux.
- `int hashCode()`: Cette méthode permet de trouver un objet dans une table de hash. Ceci est nécessaire car les événements sont stockés dans une table de hash dans l'environnement.

Voici un exemple d'implémentation d'un `Identifier` en utilisant `String`.

```
import junior.*;

public class StringIdentifier implements Identifier
{
 public String id;

    public StringIdentifier(String id){ this.id = id; }
    public int hashCode(){ return id.hashCode(); }
    public String toString(){ return id; }

    public boolean equals(Object object)
    {
       if(object instanceof StringIdentifier)
         return id.equals(((StringIdentifier)object).id);
       return false;
    }
}
```

2.3.2 Les configurations événementielles

Comme on a vu dans la section 2.2, Junior définit une notion d'événement avec laquelle on peut décider si un événement est présent ou absent dans un instant. Cette dualité dans les valeurs prises par un événement à chaque instant, permet de définir les opérateurs de la logique booléenne classiques qui utilisent, au lieu de l'égalité des valeurs, la présence ou l'absence d'un événement. Cette logique booléenne basée sur la notion d'événement a été construite en Junior avec la définition de 4 opérateurs de type `Configuation`.

`Jr.Presence(IdentifierWrapper Iw)`

Teste si un événement est présent. L'événement est identifié par l'évaluation du *wrapper* Iw qui retourne un identificateur.

`Jr.And(Configuration C1, Configuration C2)`

Retourne la conjonction des configurations C1 et C2.

`Jr.Or(Configuration C1, Configuration C2)`

Retourne la disjonction des configurations C1 et C2.

`Jr.Not(Configuration C)`

Retourne la négation de la configuration C.

2.3.3 Les instructions réactives

Junior dispose de 16 instructions réactives qui sont utilisées pour construire ce que l'on appelle un ***programme réactif***. Un programme réactif peut être, donc, composé par une ou plusieurs instructions réactives. Cette section explique les seize instructions réactives et la section suivante explique comment combiner les instructions pour construire un programme réactif.

`Jr.Nothing()`

Ne fait rien et termine.

`Jr.Stop()`

Marque la fin de l'instant dans la branche d'exécution courante.

`Jr.Atom(Action A)`

Exécute une action atomique `A` implémentée en Java et termine immédiatement.

`Jr.Seq(Program First, Program Second)`

Exécute en séquence deux programmes `First` et `Second`. Lorsque `First` termine on passe immédiatement au programme `Second`.

`Jr.Loop(Program Body)`

Implémente une boucle infinie: le programme `Body` est immédiatement redémarré dès qu'il termine.

`Jr.Repeat(IntegerWrapper Iw, Program Body)`

Implémente une boucle finie: le programme `Body` est exécuté un certain nombre d'itérations. Le nombre d'itérations à réaliser est déterminé lors de la première activation de l'instruction Repeat par l'évaluation du wrapper `Iw` qui retourne une valeur entière.

`Jr.Par(Program Left, Program Right)`

Exécute en parallèle deux programmes `Left` et `Right`. Les programmes `Left` et `Right` sont exécutés à chaque instant dans un ordre quelconque. L'instruction Par stoppe pour l'instant courant lorsque les deux branches ont marqué la fin de l'instant. L'instruction termine quand les deux programmes terminent. Le tableau suivant résume les valeurs retournées par l'instruction Par après chaque activation.

Left/Right	TERM	STOP	SUSP
TERM	TERM	STOP	SUSP
STOP	STOP	STOP	SUSP
SUSP	SUSP	SUSP	SUSP

TAB. 2.1: Les codes de retour de l'instruction Par

`Jr.Generate(IdentifierWrapper Iw)`
`Jr.Generate(IdentifierWrapper Iw, ObjectWrapper Ow)`

Génère un événement dans l'environnement d'exécution. L'événement est identifié à *runtime* par l'évaluation du wrapper `Iw` qui retourne un identificateur. La génération de l'événement peut être valuée si on spécifie un deuxième paramètre. La valeur associée à l'événement est calculée à *runtime* par l'évaluation du wrapper `Ow` qui retourne un `Object`.

`Jr.Await(Configuration C)`

Attend la satisfaction de la configuration `C` pour terminer.

`Jr.When(Configuration C, Program Then, Program Else)`

Teste au cours d'un instant la configuration `C`. Si elle est satisfaite, le programme `Then` est exécuté, sinon le programme `Else` est exécuté. L'exécution du programme choisi est instantanée si la configuration peut être évaluée avant la fin d'instant, sinon l'exécution est repoussée à l'instant suivant.

`Jr.Until(Configuration C, Program Body, Program Handler)`

Préempte le programme `Body` lorsque la configuration `C` est satisfaite. La préemption est faible car elle est réalisée après avoir exécuté `Body` à l'instant ou `C` est satisfaite. Le programme `Handler` est exécuté si la préemption est effective. Son exécution est immédiate si la configuration est satisfaite avant la fin de l'instant et que le corps est stoppé, sinon `Handler` est exécuté à l'instant suivant.

`Jr.Control(IdentifierWrapper Iw, Program Body)`

Exécute le programme `Body` en fonction de la présence d'un événement. L'événement est identifié à *runtime* lors de la première activation de l'instruction par l'évaluation du wrapper `Iw` qui retourne un identificateur. Le programme n'est exécuté pour un instant que lorsque l'événement de contrôle est présent.

`Jr.Local(Identifier Id, Program Body)`

Déclare un événement, identifié par `Id`, local au programme `Body`, c'est-à-dire dont la portée est limitée au programme `Body`.

`Jr.If(BooleanWrapper Bw, Program Then, Program Else)`

Choisit d'exécuter le programme `Then` ou le programme `Else` en fonction de l'évaluation du wrapper `Bw`. Le wrapper `Bw` est évalué une seule fois lors de la première activation de l'instruction If. Si le wrapper rend vrai le programme `Then` est exécuté, sinon c'est le programme `Else`.

`Jr.Link(ObjectWrapper Ow, Program Body)`

Associe un objet Java au programme `Body`. L'objet associé est connu à *runtime* par l'évaluation du wrapper `Ow`.

`Jr.Freezable(IdentifierWrapper Iw, Program Body)`

Gèle le programme `Body` en fonction de la présence d'un événement. L'événement est identifié à *runtime* lors de la première activation de l'instruction par l'évaluation du wrapper `Iw` qui retourne un identificateur. A l'instant où l'événement est présent, `Body` est exécuté une dernière fois, à la manière de la préemption. Le programme gelé est stocké dans l'environnement en parallèle avec les autres instructions gelées par le même événement, s'il y en a.

2.3.4 La machine réactive

La machine réactive est la partie centrale d'un système réactif construit avec Junior. Elle contient et gère les éléments décrits dans la section 2.2. En particulier elle:

- Contient un programme réactif et gère l'ajout de nouveaux programmes réactifs en cours exécution. Les nouveaux programmes réactifs sont ajoutés en parallèle avec les anciens et sont exécutés à l'instant suivant de leur ajout.
- Contient un environnement d'exécution. La machine diffuse dans l'environnement les événements générés en dehors du programme et donne accès aux instructions gelées (voir l'instruction Freezable ci-dessous).
- Gère les instants de son programme.

Une machine finit l'instant courant lorsque toutes les instructions parallèles du programme qu'elle contient sont soit totalement terminées soit stoppées (c'est-à-dire qu'il n'y a plus d'instructions suspendues). Sinon, la machine active cycliquement le programme tant qu'il reste des instructions suspendues. A la fin de chaque activation, la machine teste si un nouvel événement a été généré durant l'activation; s'il n'y en a pas, alors la situation ne peut plus évoluer et la fin de l'instant courant est décidée. Dans ce cas, un drapeau est levé et une dernière activation du programme permet aux instructions suspendues de stopper.

Types de machines et leur API

Junior offre deux types de machines: la *safe*-machine et la *unsafe*-machine. Ces deux types sont créés en utilisant les primitives suivantes:

`Jr.Machine(Program P)`

Crée une *unsafe*-machine et y charge le programme `P`.

`Jr.SafeMachine(Program P)`

Crée une *safe*-machine et y charge le programme `P`.

La *safe*-machine est une machine qui a été conçue pour être exécutée dans un environnement multitâche, c'est-à-dire qu'elle supporte l'accès concurrent de 2 ou plus threads Java, sans perte d'information. En particulier, elle garantit la consistance de l'information lorsque 2 threads utilisent l'API de la machine réactive en même temps. L'API des deux machines est la suivante:

`react()`

Fait réagir la machine une fois, c'est-à-dire, exécute le programme un instant.

`add(Program P)`

Ajoute un programme `P` dans la machine. S'il s'agit d'une SafeMachine, on ajoute une copie du programme. Le programme `P` est ajouté à l'instant i et il commence à s'exécuter à l'instant $i+1$ pour éviter tout conflit.

`generate(Identifier E)`

Génère l'événement `E` dans la machine.

`getFrozen(Identifier E)`

Rend une copie des programmes gelés, grâce à l'instruction Freezable, associés à l'événement `E`.

2.3.5 Construction et exécution d'un programme Junior

Pour construire un programme réactif avec Junior il faut effectuer les étapes suivantes:

1. Créer une machine réactive M.
2. Créer un programme réactif P.
3. Ajouter le programme P dans la machine M.
4. Faire réagir la machine M un nombre quelconque de fois.

Les pas 2 et 3 peuvent être répétés au cours de l'exécution du pas 4.

Une application Junior peut être vue comme un système ***coopératif*** plongé dans une horloge logique; ceci veut dire que le programmeur, lors de la conception du système réactif, doit toujours vérifier que les différents composants lâchent le contrôle. Le programmeur peut passer la main aux autres composants explicitement en utilisant l'instruction Stop ou implicitement lorsqu'il utilise les instructions événementielles. La coopération est un aspect fondamental en Junior, le programmeur doit s'assurer qu'il rend la main, en particulier lorsqu'il programme les actions atomiques et les wrappers. La création du programme Java `for(;;);` dans un atome est une erreur qui cause l'arrêt total du système. Ce style de programmation et ses problèmes seront discutés plus en détail dans les chapitres suivants. Voyons maintenant deux exemples qui illustrent la plupart des notions vues jusqu'ici.

Un exemple concret

L'exemple suivant crée un programme qui exécute une action atomique à chaque instant. L'action atomique (ligne 9) consiste à imprimer la chaîne `atom`. Après la création du programme (lignes 7-13) et de la machine (ligne 16), le programme est chargé dans la machine (ligne 16) et puis activé dix instants (lignes 18, 19).

```
1  import junior.*;
2  import jre.*;
3
4  public class Exemple
5  {
6     public static void main(String[] args)
7     {
8       Program P = Jre.Loop(Jre.Seq(Jre.Atom(new Action(){
9                                        void execute(Environment env){
10                                          System.out.print("atom");
11                                       }}),
12                                       Jre.Stop()
13                                      )
14                           );
15
16       Machine M = Jre.Machine();
17       M.add(P);
18
19       for(int i=1; i<=10; i++)
20          M.react();
21    }
22 }
```

Listing 2.1: Exemple d'un programme Junior

Pour exécuter ce programme, on lance les commandes suivantes dans le cas d'un terminal UNIX :

```
$ javac -classpath .:$JUNIOR_PATH Example.java
$ java -classpath .:$JUNIOR_PATH Example
atom atom atom...
```

2.4 Exemples

Cette section présente quelques exemples des programmes Junior avec le but de montrer la puissance de Junior mais aussi les difficultés de sa programmation.

Exemple 1

Le programme Junior du listing 2.2 est composé de deux comportements réactifs mis en parallèle (le premier entre les ligne 39-44 et le deuxième entre les lignes 45-48). Le premier comportement exécute une action atomique à chaque instant (pour ne pas créer une classe anonyme, comme dans l'exemple 1, l'atome est défini dans une classe à part qui s'appelle `MonAtom1`) et lorsque l'événement `name` est généré, on arrête d'exécuter le comportement (à l'instant d'après, préemption faible). Le deuxième comportement laisse passer quatre instants, puis il génère un événement `NamE` et il exécute une action atomique (`MonAtom2()`). Les événements utilisés par l'instruction Generate et l'instruction Until (`name` et `NamE` respectivement) sont différents mais vus comme le même grâce à l'identifier défini par la classe `MonEvent` qui compare les chaînes (ligne 29) sans considérer si elles ont des lettres en majuscules ou minuscules. Le programme principal crée la machine et le programme, puis il exécute le programme tant qu'il n'a pas fini (lignes 52-58).

```
1  import junior.*;
2  import jre.*;
3
4  class MonAtom1 implements Action
5  {
6     public void execute(Environment env){
7         System.out.println("    *");
8     }
9  }
10
11 class MonAtom2 implements Action
12 {
13    public void execute(Environment env){
14        System.out.println(" Evénement généré ");
15    }
16 }
17
18 class MonEvent implements Identifier
19 {
20     public String id;
21
22     public MonEvent(String i){ id = i+"!kill"; }
23
24     public int hashCode(){ return id.hashCode(); }
25
26     public boolean equals(Object object)
27     {
28        if (object instanceof MonEvent)
29           return id.equalsIgnoreCase(((MonEvent)object).id);
30        return false;
31     }
32 }
33
34 public class Exemple1
35 {
36    public static void main(String[] args)
37    {
38      Program P =
39      Jre.Par(Jre.Until(new MonEvent("name"),
40                        Jre.Loop(Jre.Seq(Jre.Atom(new MonAtom1()),
41                                         Jre.Stop()
42                                        )
43                                 ),
44                        ),
45              Jre.Seq(Jre.Repeat(4,Jre.Stop()),Jre.Seq(
46                      Jre.Generate(new MonEvent("NamE")),
47                      Jre.Atom(new MonAtom2())))
```

```
48                          )
49        );
50
51
52        Machine M = Jre.Machine();
53        M.add(P);
54
55        int i=1;
56        do{
57            System.out.print((i++) + ":");
58        }while( !M.react() );
59      }
60  }
```

Listing 2.2: Exemple 1 d'un programme Junior

La sortie de ce programme est:

```
$ java -classpath .:$JUNIOR_PATH Example1
1:    *
2:    *
3:    *
4:    *
5:    *
 Evénement généré
$
```

Exemple 2

Cet exemple montre la propriété de dynamisme de Junior. On montre l'ajout de nouveaux programmes réactifs ainsi que la génération d'événements. Le programme principal crée une machine réactive et un programme (`makeProgram()`, ligne 41) qui se comporte comme la première branche de l'exemple précédent (exécute un atome à chaque instant, tant qu'il n'est pas préempté). Ensuite il charge le programme (ligne 41) et fait réagir la machine tant que son programme ne finit pas (ligne 57). Au cours des activations de la machine (au 2eme et 5eme instant), on ajoute deux nouveaux comportement réactifs identiques au premier (juste avec un paramètre différent) et on génère un événement (au 8ème instant) qui tue tous les comportements réactifs. En conclusion, on exécute trois comportements réactifs (qui commencent à des instants différents) qui impriment le caractère `*` (séparé par tab pour les distinguer, lignes 11-13) et qui sont tués à l'instant 9 par la génération de l'événement `NamE`.

```
1  import junior.*;
2  import jre.*;
3
4  class MonAtom1 implements Action
5  {
6      int t;
7
8      public MonAtom(int i){ t = i; }
9
10     public void execute(Environment env){
11       for(int i=1; i<=t; i++)
12         System.out.print("\t");
13       System.out.print("*");
14     }
15 }
16
17
18 class MonEvent implements Identifier
```

```
19  {
20    // même de l'exemple 1
21  }
22
23  public class Exemple2
24  {
25
26     public Program makeProg(int i)
27     {
28        return
29        Jre.Until(new MonEvent("name"),
30                  Jre.Loop(Jre.Seq(Jre.Atom(new MonAtom1(i)),
31                                   Jre.Stop()
32                                  )
33                           ),
34                  );
35     }
36
37     public static void main(String[] args)
38     {
39       Exemple2 e2 = new Exemple2();
40       Machine M = Jre.Machine();
41       M.add( e2.makeProg(1) );
42
43       int i=1;
44       do{
45          System.out.print("\n"+(i++) + ":");
46          switch(i){
47             case 2:
48                      M.add( e2.makeProg(2) );
49                      break;
50             case 5:
51                      M.add( e2.makeProg(3) );
52                      break;
53             case 8:
54                      M.generate( new MonEvent("NamE") );
55                      break;
56          }
57       }while( !M.react() );
58     }
59  }
```

Listing 2.3: Exemple 2 d'un programme Junior

La sortie de ce programme est :

```
$ java -classpath .:$JUNIOR_PATH Example2

1:    *
2:    *
3:    *    *
4:    *    *
5:    *    *
6:    *    *    *
7:    *    *    *
8:    *    *    *
9:    *    *    *
$
```

Maintenant, on va voir la description formelle du langage de programmation de Junior.

2.5 Sémantique

Dans la sous-section 2.3.3 on a donné la description du comportement de chaque instruction réactive. Cette description donnée en langage naturel s'avère utile pour donner une idée générale du comportement de chaque instruction réactive. Cependant, lorsqu'on commence à construire des programmes réactifs plus complexes, cette description n'est pas suffisante car elle ne donne pas la description exhaustive de toutes les interactions possibles entre les instructions réactives. Pour avoir une description précise du fonctionnement de Junior, sa sémantique formelle a été définie.

Il existe plusieurs façons de donner une sémantique formelle à un langage, dont les plus connues sont les suivantes :

1. *Sémantique Opérationnelle* [PLO 81, KAN 87]. Les langages de programmation sont définis par des règles de réduction qui décrivent comment l'état initial du programme est transformé pas à pas vers l'état final. Ce type de description est utile pour comprendre le comportement dynamique d'un langage, par exemple, pour construire un interprète. Les sémantiques opérationnelles font suite aux travaux de G. Plotkin sur les *structural operational semantics*.

2. *Sémantique Dénotationnelle* [STO 77, SCH 88]. Les langages de programmation sont définis par une fonction d'évaluation qui associe à un programme un objet mathématique. L'objet mathématique donne une signification à ce que fait un programme. Ce type de description est utile pour comprendre la logique interne du langage, par exemple, pour raisonner sur ses propriétés. Les sémantiques dénotationnelles font suite aux travaux de C. Strachey, D. Scott, J.W. de Bakker et M. Nivat, pour n'en citer que quelques uns.

3. *Sémantique Axiomatique* [HOR 73, MEY 93, FRA 92]. Les langages de programmation sont définis par des assertions qui décrivent comment tirer des conclusions sur les entrées et les sorties du programme. Ce type de description est utile pour comprendre les effet externes du langage, par exemple, pour vérifier un programme. Les sémantiques axiomatiques font suite aux travaux de R. Floyd et de C.A.R. Hoare.

Il serait erroné d'opposer ces sémantiques. Chacune possède son utilité. Ainsi, une sémantique opérationnelle claire est très utile pour concevoir une implémentation, les sémantiques axiomatiques offrent des systèmes de preuves élégants pour développer et vérifier les programmes, et les sémantiques dénotationnelles sont à l'origine des techniques les plus sophistiquées.

Etudier ces sémantiques séparément serait aussi une erreur. Ainsi, établir la correction d'une sémantique axiomatique requiert la connaissance à priori d'une sémantique opérationnelle, voire dénotationnelle. De même, l'argumentation de la correction d'une implémentation par rapport à une sémantique dénotationnelle requiert le passage par une sémantique opérationnelle et son lien avec la sémantique dénotationnelle considérée. Le problème se pose alors naturellement de distinguer parmi les différents types de sémantiques celles qui correspondent le plus étroitement aux autres.

Le domaine de la description formelle des langages de programmation est très vaste et bien d'autres types de sémantique ont été donnés: Sémantique Algébrique, Sémantique d'Actions. La sémantique formelle de Junior est une sémantique opérationnelle structurelle que l'on décrit plus en détail maintenant.

Sémantique Opérationnelle Structurelle

La sémantique opérationnelle spécifie un langage de programmation comme l'exécution de celui-ci dans une machine abstraite. La Sémantique Opérationnelle Structurelle (SOS), développée par Gordon Plotkin en 1981 [PLO 81], représente l'exécution par un système déductif qui exécute la machine abstraite dans un système d'inférence logique. Etant donné que la description sémantique est basée sur une logique déductive, les preuves des propriétés des programmes sont obtenues directement à partir des définitions du langage.

La SOS est définie par des règles d'inférence constituées par une conclusion qui se vérifie à partir d'un ensemble de prémisses probablement sous le contrôle de quelques conditions. La forme générale de la règle d'inférence est une fraction qui a comme numérateur les prémisses et comme dénominateur la conclusion; si la condition est présente elle est donnée à droite:

$$\frac{premisse1 \quad premisse2 \quad ... \quad premisse3}{conclusion}$$

Si le nombre de prémisse est nul, la ligne est omise et on parle d'axiome. Cette syntaxe a évolué à partir d'une syntaxe de logique appelée déduction naturelle.

Plusieurs variantes de la SOS ont été proposées, par exemple la sémantique *big-step* ou *macro-step* qui, par opposition à la sémantique *small-step* ou *micro-step*, cherche à capturer le comportement décrit entre deux états considérés comme l'état initial et l'état final. Un exemple de sémantique *macro-step* est celle qui fut développée par Gilles Kahn, nommée sémantique naturelle. Les autres sémantiques opérationnelles sont la sémantique de réduction [FEL 87], la sémantique moduler [MOS 99].

La sémantique formelle de Junior est décrite dans [BOU 00d] et elle est basée sur l'utilisation de règles de réécriture du type SOS. Le format de réécriture utilisé est le suivant:

$$t, E \xrightarrow{\alpha} t', E'$$

qui signifie que l'instruction t exécutée dans l'environnement E se transforme (on peut dire, se réécrit) en t' avec l'environnement E', et retourne α comme drapeau de terminaison.

La sémantique de Junior est composée de 45 règles de réécriture de type *micro-step* qui définissent les instructions réactives du langage Junior et de 3 règles *micro-step*, plus une règle de réécriture de type *macro-step*, qui définissent la notion d'instant. La notion d'instant du modèle réactif est divisée en micro-instants ou micro-réactions; les micro-réactions sont implémentées à l'aide de trois drapeaux de terminaison:

- TERM qui signifie que l'exécution est terminée pour l'instant courant et qu'il ne reste rien à faire au prochain instant.
- STOP signifie que l'exécution est terminée pour l'instant courant mais qu'il reste quelque chose à faire dans le prochain instant.
- SUSP signifie que l'exécution n'est pas terminée pour l'instant courant et qu'elle doit être reprise dans le même instant.

La sémantique repose sur un nombre réduit de règles que l'on a regroupées dans les figures 2.6 et 2.7. Celles-ci fournissent une définition extrêmement compacte de Junior à la base de ses implémentations. L'interprétation de la syntaxe utilisée est la suivante :

- Les instructions réactives sont représentées par leur syntaxe abstraite, et non leur syntaxe concrète; par exemple l'instruction Stop qui a la syntaxe concrète `Jr.Stop()` est représentée par *Stop*.
- Lorsque les instructions utilisent des paramètres qui sont des programmes réactifs, on utilise des lettres minuscules (en général t et u). Par exemple, l'instruction réactive Loop qui a la syntaxe concrète `Jr.Loop(Program t)` est dénoté par $Loop(t)$. Si une instruction réactive, par exemple t, est réécrite en un nouveau terme, on le dénote par t'.
- Les événements sont dénotés par des lettres majuscules: S pour les événements simples et C pour les configurations (des conditions événementielles formées avec des opérateurs And et Or). Par exemple, l'instruction réactive Await qui a la syntaxe concrète `Jr.Await(Config c)` est dénotée par $Await(C)$.
- Les instructions événementielles utilisent 3 prédicats : $sat(C, E)$ pour tester si la configuration C est satisfaite dans l'environnement E, $unsat(C, E)$ lorsque la configuration C n'est pas satisfaite, et $unknown(C, E)$ lorsque on peut pas encore savoir son état.

Nothing

$$Nothing, E \xrightarrow{TERM} Nothing, E$$

Atoms

$$Atom(a), E \xrightarrow{TERM} Nothing, E$$

Sequence

$$\frac{t,E \xrightarrow{TERM} t',E' \qquad u,E' \xrightarrow{\alpha} u',E''}{Seq(t,u),E \xrightarrow{\alpha} u',E''}$$

$$\frac{t,E \xrightarrow{\alpha} t',E' \qquad \alpha \neq TERM}{Seq(t,u),E \xrightarrow{\alpha} Seq(t',u),E'}$$

Loop

$$\frac{t,E \xrightarrow{\alpha} t',E' \qquad \alpha \neq TERM}{Loop(t),E \xrightarrow{\alpha} Seq(t',Loop(t)),E'}$$

$$\frac{t,E \xrightarrow{TERM} t',E' \qquad Loop(t),E' \xrightarrow{\alpha} u,E''}{Loop(t),E \xrightarrow{\alpha} u,E''}$$

Repeat

$$\frac{n \leq 0}{Repeat(n,t),E \xrightarrow{TERM} Nothing,E}$$

$$\frac{n>0 \qquad Seq(t,Repeat(n-1,t)),E \xrightarrow{\alpha} u,E'}{Repeat(n,t),E \xrightarrow{\alpha} u,E'}$$

Control

$$\frac{sat(S,E) \qquad t,E \xrightarrow{\alpha} t',E'}{Control(S,t),E \xrightarrow{\alpha} Control(S,t'),E}$$

$$\frac{unsat(S,E)}{Control(S,t),E \xrightarrow{STOP} Control(S,t),E}$$

$$\frac{unknown(S,E)}{Control(S,t),E \xrightarrow{SUSP} Control(S,t),E}$$

When

$$\frac{sat(C,E) \qquad eoi(E)=false \qquad t,E \xrightarrow{\alpha} t',E'}{When(C,t,u),E \xrightarrow{\alpha} t',E'}$$

$$\frac{sat(C,E) \qquad eoi(E)=true}{When(C,t,u),E \xrightarrow{STOP} t,E}$$

$$\frac{unsat(C,E) \qquad eoi(E)=false \qquad u,E \xrightarrow{\alpha} u',E'}{When(C,t,u),E \xrightarrow{\alpha} u',E'}$$

$$\frac{unsat(C,E) \qquad eoi(E)=true}{When(C,t,u),E \xrightarrow{STOP} u,E}$$

$$\frac{unknown(C,E)}{When(C,t,u),E \xrightarrow{SUSP} When(C,t,u),E}$$

Stop

$$Stop, E \xrightarrow{STOP} Nothing, E$$

Contexte d'exécution

$$\frac{Instant(t),Fresh \xrightarrow{\alpha} Instant(t'),E}{ExecContext(t) \xRightarrow{b} ExecContext(t')}$$

Await

$$\frac{sat(C,E) \qquad eoi(E)=false}{Await(C),E \xrightarrow{TERM} Nothing,E}$$

$$\frac{sat(C,E) \qquad eoi(E)=true}{Await(C),E \xrightarrow{STOP} Nothing,E}$$

$$\frac{unsat(C,E)}{Await(C),E \xrightarrow{STOP} Await(C),E}$$

$$\frac{unknown(C,E)}{Await(C),E \xrightarrow{SUSP} Await(C),E}$$

Until

$$\frac{t,E \xrightarrow{\alpha} t',E' \qquad \alpha \neq STOP}{Until(C,t,u),E \xrightarrow{\alpha} Until(C,t',u),E'}$$

$$\frac{t,E \xrightarrow{STOP} t',E' \qquad Until*(C,t',u),E' \xrightarrow{\alpha} v,E''}{Until(C,t,u),E \xrightarrow{\alpha} v,E''}$$

$$\frac{sat(C,E) \qquad eoi(E)=false \qquad u,E \xrightarrow{\alpha} u',E'}{Until*(C,t,u),E \xrightarrow{\alpha} u',E'}$$

$$\frac{sat(C,E) \qquad eoi(E)=true}{Until*(C,t,u),E \xrightarrow{STOP} u,E}$$

$$\frac{unsat(C,E) \qquad eoi(E)=true}{Until*(C,t,u),E \xrightarrow{STOP} Until(C,t,u),E}$$

$$\frac{unknown(C,E)}{Until*(C,t,u),E \xrightarrow{SUSP} Until*(C,t,u),E}$$

Local

$$\frac{t,E-S \xrightarrow{SUSP} t',E' \qquad S \notin E'}{Local-(S,t),E \xrightarrow{SUSP} Local-(S,t'),E'/E[S]}$$

$$\frac{t,E-S \xrightarrow{SUSP} t',E' \qquad S \in E'}{Local-(S,t),E \xrightarrow{SUSP} Local+(S,t'),E'/E[S]}$$

$$\frac{t,E+S \xrightarrow{SUSP} t',E'}{Local+(S,t),E \xrightarrow{SUSP} Local+(S,t'),E'/E[S]}$$

$$\frac{t,E-S \xrightarrow{\alpha} t',E' \qquad \alpha=TERM \quad or \quad \alpha=STOP}{Local-(S,t),E \xrightarrow{\alpha} Local-(S,t'),E'/E[S]}$$

$$\frac{t,E+S \xrightarrow{\alpha} t',E' \qquad \alpha=TERM \quad or \quad \alpha=STOP}{Local+(S,t),E \xrightarrow{alpha} Local-(S,t'),E'/E[S]}$$

FIG. 2.6: Sémantique de Junior 1ère partie (règles de réécriture)

Par

$$\frac{t,E\xrightarrow{\alpha}t',E'}{Par_{SUSP,SUSP}(t,u),E\xrightarrow{SUSP}Par_{\alpha,SUSP}(t',u),E'[move{=}true]}$$

$$\frac{u,E\xrightarrow{\beta}u',E'}{Par_{SUSP,SUSP}(t,u),E\xrightarrow{SUSP}Par_{SUSP,\beta}(t,u'),E'[move{=}true]}$$

$$\frac{t,E\xrightarrow{\alpha}t',E' \qquad u,E'\xrightarrow{\beta}u',E''}{Par_{SUSP,SUSP}(t,u),E\xrightarrow{\gamma(\alpha,\beta)}Par_{\delta_1(\alpha,\beta),\delta_2(\alpha,\beta)}(t',u'),E''}$$

$$\frac{u,E\xrightarrow{\beta}u',E' \qquad t,E'\xrightarrow{\alpha}t',E''}{Par_{SUSP,SUSP}(t,u),E\xrightarrow{\gamma(\alpha,\beta)}Par_{\delta_1(\alpha,\beta),\delta_2(\alpha,\beta)}(t',u'),E''}$$

$$\frac{\beta{\neq}SUSP \qquad t,E\xrightarrow{\alpha}t',E'}{Par_{SUSP,\beta}(t,u),E\xrightarrow{\gamma(\alpha,\beta)}Par_{\delta_1(\alpha,\beta),\delta_2(\alpha,\beta)}(t',u),E'}$$

$$\frac{\alpha{\neq}SUSP \qquad u,E\xrightarrow{\beta}u',E'}{Par_{\alpha,SUSP}(t,u),E\xrightarrow{\gamma(\alpha,\beta)}Par_{\delta_1(\alpha,\beta),\delta_2(\alpha,\beta)}(t,u'),E'}$$

Freezable

$$\frac{t,E\xrightarrow{\alpha}t',E' \qquad \alpha{\neq}STOP}{Freezable(S,t),E\xrightarrow{\alpha}Freezable(S,t'),E'}$$

$$\frac{t,E\xrightarrow{STOP}t',E' \qquad Freezable{*}(S,t'),E'\xrightarrow{\alpha}v,E''}{Freezable(S,t),E\xrightarrow{\alpha}v,E''}$$

$$\frac{S{\in}E \qquad eoi(E){=}false}{Freezable{*}(S,t),E\xrightarrow{TERM}Nothing,E\{S{:=}Par(t,E\{S\})\}}$$

$$\frac{S{\in}E \qquad eoi(E){=}true}{Freezable{*}(S,t),E\xrightarrow{STOP}Nothing,E\{S{:=}Par(t,E\{S\})\}}$$

$$\frac{S{\notin}E \qquad eoi(E){=}true}{Freezable{*}(S,t),E\xrightarrow{STOP}Freezable(S,t),E}$$

$$\frac{S{\notin}E \qquad eoi(E){=}false}{Freezable{*}(S,t),E\xrightarrow{SUSP}Freezable{*}(S,t),E}$$

Instant

$$\frac{t,E\xrightarrow{\alpha}t',E' \qquad \alpha{\neq}SUSP}{Instant(t)\xrightarrow{\alpha}Instant(t'),E'}$$

$$\frac{t,E\xrightarrow{SUSP}t',E' \qquad move(E'){=}false \qquad Instant(t'),E'[eoi{=}true]\xrightarrow{\alpha}u,E''}{Instant(t)\xrightarrow{\alpha}u,E''}$$

$$\frac{t,E\xrightarrow{SUSP}t',E' \qquad move(E'){=}true \qquad Instant(t'),E'[move{=}false]\xrightarrow{\alpha}u,E''}{Instant(t)\xrightarrow{\alpha}u,E''}$$

Generate

$$Generate(S),E\xrightarrow{TERM}Nothing,(E+S)[move=true]$$

FIG. 2.7: Sémantique de Junior 2ème partie (règles de réécriture)

- les variables *eoi* et *move* de l'environnement E sont testées par les prédicats $eoi(E)$ et $move(E)$ respectivement et modifiées par $E[eoi = true]$ et $E[move = false]$.

Ce système de règles de réécriture est déterministe (à l'exception de l'instruction Par) et lorsqu'on exécute un programme réactif il n'existe donc qu'une seule règle à appliquer. Par exemple si l'on exécute le programme:

```
Jr.Seq(Jr.Atom(new MyAction()),Jr.Stop());
```

on procède de la façon suivante (on utilisant la syntaxe abstraite) :

1. les deux règles de la séquence disent qu'il faut exécuter d'abord la branche gauche qui est un atome :
$$Atom(a), E \stackrel{TERM}{\longrightarrow} Nothing, E$$
2. la première règle de la séquence dit qu'il faut exécuter tout de suite la branche droite si la gauche finit, ce qui est le cas :
$$Stop, E \stackrel{STOP}{\longrightarrow} Nothing, E$$
3. la même règle de la séquence dit que maintenant on se comporte comme la branche gauche, mais on rend STOP ce qui veut dire que le premier instant est fini.
$$Seq(Atom(MyAction), Stop), E \stackrel{STOP}{\longrightarrow} Nothing, E$$
4. A l'instant d'après, on exécute Nothing qui rend TERM et on finit l'exécution
$$Nothing, E \stackrel{TERM}{\longrightarrow} Nothing, E$$

Le système de règles de réécriture de Junior décrit l'exécution d'un programme Junior par des transformations (réécritures) des instructions réactives. Ces transformations ne doivent être réalisées que par le moteur réactif pour éviter les problèmes liés aux modifications concurrentes du programme. L'ajout dynamique de nouveaux comportements réactifs, qui implique une modification du programme réactif, est une opération réalisée entre deux instant pour ne pas perturber l'instant courant. L'ajout dynamique n'est donc pas décrit par les règles de réécriture et il n'existe aucune instruction réactive qui fasse des modifications au programme réactif en cours d'exécution. Dans le chapitre suivant on va proposer deux nouvelles instructions qui permettent l'ajout dynamique de comportements réactifs intra-instant.

2.6 Conclusions

Ce chapitre a présenté Junior qui est une implémentation en Java du Modèle Réactif Synchrone. La construction d'un système réactif en Junior consiste en la création d'une machine réactive et d'un programme composé d'instructions réactives; l'exécution du système est une suite d'activations/réactions du système. Les réactions du système sont décrites par les instructions réactives qui décrivent des comportements réagissant à la présence ou absence d'événements diffusés dans l'environnement d'exécution. Junior est un langage dynamique qui permet la création et l'exécution de nouveaux comportements réactifs à *run-time* ainsi que la génération d'événements. Un des principaux atouts de Junior est l'instruction réactive de parallélisme logique qui a une sémantique précise et qui n'utilise pas les threads Java. Junior est un langage de bas niveau implémenté par un ensemble de classes Java; sa programmation est standardisée par l'API Jr.

La section 2.5 est une description générale de la sémantique de Junior. Cette section est un résumé du papier [BOU 00d].

Chapitre 3

Junior: implémentations et expérimentations

Dans ce chapitre on présente les principales implémentations de Junior réalisées à nos jours. L'objectif de donner une sémantique formelle à Junior est de valider les implémentations faites du modèle réactif synchrone qui nous intéresse.

On commence dans la section 3.1 avec une description des différentes implémentations. Cette description se centre sur l'algorithme utilisé et les détails de l'implémentation qui caractérisent le comportement de l'implémentation (avantages et inconvénients). La section 3.3 présente les travaux similaires, c'est-à-dire d'autres implémentations du modèle réactif et des systèmes qui ont des caractéristiques semblables. Finalement dans la section 3.5 on présente les conclusions sous la forme d'une liste d'avantages et de désavantages de Junior.

3.1 Implémentations

Le modèle d'exécution de Junior a été implémenté, principalement, de 6 manières différentes. Chaque implémentation a un nom qui reflète l'essence de l'algorithme utilisé. Les noms des versions créées sont Rewrite, Replace, Simple, Storm, Rvm, et Glouton. Ces implémentations sont décrites dans la suite.

Rewrite

Rewrite est la première implémentation qui a été faite de Junior. Dès le départ, l'objectif de Rewrite fut d'avoir une implémentation dont on est sûr qu'elle n'est pas buggée, c'est-à-dire elle respecte bien la sémantique formelle donnée dans la section 2.5. Pour accomplir son objectif, Rewrite implémente les règles de réécriture le plus fidèlement possible. Par fidèlement on entend ici que n'importe qui (avec un minimum de connaissance) peut vérifier facilement la validité de l'implémentation.

La table 3.1 montre les règles de réécriture de l'instruction Repeat ainsi que leur implémentation en Java . Chaque instruction en Junior est implémentée par une classe; la création d'un programme est donc l'instanciation de ces classes. Dans le cas du Repeat, on a deux classes: une qui fixe la valeur N (que l'on ne montre pas), et une autre qui implémente les règles. La classe `RepeatStatic` implémente les deux règles de réécriture; par exemple les lignes 11 et 12 implémentent la première règle de réécriture dans laquelle la flèche est codée comme la création d'un objet `MicroState` qui contient le code de retour `TERM` et la nouvelle instruction `Nothing`.

L'implémentation de Rewrite est clairement divisée en deux ensembles, celui qui implémente les règles de réécriture (24 classes) et celui qui implémente les interfaces, le moteur d'exécution et les événements (27 classes).

La notion d'événement est implémentée à l'aide d'une table de hash. La première fois qu'un événement est généré ou attendu, il est stocké dans la table de hash: la clé utilisée est l'identificateur de l'événement (un objet

$$\frac{n \leq 0}{Repeat(N,B), E \xrightarrow{TERM} Nothing, E}$$

$$\frac{N>0 \quad Seq(B, Repeat(N-1,B)), E \xrightarrow{\alpha} u, E'}{Repeat(N,B), E \xrightarrow{\alpha} u, E'}$$

```
1  public class RepeatStatic extends UnaryInstruction
2  {
3    final public long N;
4
5    public RepeatStatic(long N, Program B){
6        super(B);
7        this.N = N;
8    }
9
10   public MicroState rewrite(EnvironmentImpl env){
11       if (N <= 0)
12           return new MicroState(TERM,new Nothing());
13       return new Seq(B.copy(),
14             new RepeatStatic(N-1,B)).rewrite(env);
15   }
16 }
```

TAB. 3.1: Règles de réécriture vs Implémentation

qui implémente l'interface Identifier) et la donnée est un objet qui contient l'information associée à l'événement (les valeurs associées à l'événement de l'instant courant et précédent ainsi que le numéro de l'instant où il a été généré).

Cette implémentation des événements privilégie un usage fréquent des événements car la structure de donnée est créée une seule fois; par contre, si les événements sont utilisés rarement, la mémoire est gaspillée et il faudrait nettoyer la table de hash (ce qui n'est pas fait et qui prendrait aussi du temps d'exécution). En termes de temps d'accès, l'implémentation est efficace grâce à l'algorithme de recherche de la table de hash. Cependant son utilisation n'est pas optimale : les instructions événementielles cherchent toujours dans la table de hash les événements même lorsqu'ils ont été déjà fixés. Cette façon d'implémenter les événements (créés à *run-time*) est due à la propriété de dynamisme de Junior qui n'est pas présente dans les implémentations du modèle synchrone où les événements sont déclarés et optimisés pour l'accès. En Junior on pourrait accélérer l'accès aux événements par un mécanisme similaire. Ceci est à l'étude et pour l'instant le programmeur doit analyser quelle est la meilleure solution pour son application. Dans la sous-section 3.4.1 on présentera une autre façon d'optimiser l'accès aux événements.

La plupart des avantages et des désavantages de Rewrite sont dus à ce que l'implémentation dérive directement de la sémantique donnée sous la forme de règles de réécriture. Les principales caractéristiques de Rewrite sont:

- Une sémantique exécutable. Grâce au fait que l'implémentation dérive de la sémantique, le programmeur peut avoir une très grande confiance dans l'implémentation.
- Efficacité très faible :
  - Rewrite crée beaucoup d'objets. Si on analyse le code de Rewrite on trouvera que le nombre d'objets créés à chaque fois que l'on exécute une instruction réactive (la méthode `rewrite`) se trouve entre zéro et six, avec une moyenne de 2 objets. A ce nombre, il faut ajouter le nombre d'objets nécessaires pour implémenter un événements (3); au total on a $3N$ où N est le nombre d'événements.
  - Implémentation inefficace de la méthode rewrite. A chaque fois que cette méthode est invoquée, elle passe l'environnement d'exécution et retourne un nouvel objet (`MicroState`) contenant le statut de l'exécution (TERM, STOP ou SUSP). Cette façon d'activer les programmes a un grand impact sur l'espace mémoire (la pile et le tas) lorsqu'on exécute un grand nombre d'instructions.
  - L'efficacité du programme dépend de l'ordre dans lequel on écrit les instructions. Si on ne fait pas attention lors de l'écriture d'un programme, on peut avoir des cas exceptionnels, comme la cascade inverse, ou l'efficacité est très basse.
  - Quelques instructions sont intrinsèquement inefficaces. Ces instructions sont: Loop, Repeat, Local et Freezable. Loop et Repeat créent une copie de leur corps à chaque itération. Local et Freezable créent aussi plusieurs objets lorsque l'événement de contrôle est présent.

En conclusion, Rewrite est une implémentation très simple à comprendre, vérifier, modifier et étendre. Par contre, elle très inefficace car toutes les instructions créées sont utilisées une seule fois et puis jetées (réécriture des termes). L'algorithme a une mauvaise complexité temporelle et spatiale car l'arbre d'exécution est reconstruit à chaque activation.

Replace

L'idée de Replace consiste à reprendre l'implémentation de Rewrite et à éliminer la création inutile d'objets. Pour cela, Replace optimise, principalement, 2 choses :

1. Les instructions réactives sont réutilisées le plus possible avant d'être détruites, en particulier elles sont réutilisées : 1) tant qu'elles n'ont pas fini, et 2) lorsqu'elles sont sous le contrôle d'une instruction Loop ou Repeat. Pour réutiliser les instructions réactives, elles sont implémentées avec un état interne qui est réinitialisé, par exemple, lorsque l'instruction est finie et est sous le contrôle d'une instruction Loop.
2. Le code de retour décrit dans les règles de réécriture est implémenté à l'aide d'une variable globale et non une structure (`MicroState`) contenant le code de retour et la nouvelle instruction; cette optimisation élimine la recréation, souvent inutile, de la structure ainsi que l'utilisation des paramètres dans la méthode de réécriture `rewrite()`.
3. On ne crée des instructions Nothing que lorsque cela est nécessaire. Les instructions qui se réécrivent en Nothing (Atome, Stop, Repeat, Await et Freezable) se comportent comme Nothing (elles rendent TERM) lorsqu'elles ont fini et qu'elles n'ont pas été réinitialisées.

Après l'élimination d'objets inutiles, il reste d'autres problèmes :

- L'utilisation d'une structure d'arbre pour représenter les programmes réactifs. La structure d'arbre pose des problèmes à la construction et à l'exécution du programme car dans les deux cas on utilise des fonctions récursives pour parcourir l'arbre. Si on compile un programme réactif qui génère un arbre de hauteur considérable, c'est la pile du compilateur qui explose. Si on essaie d'exécuter un programme qui a été chargé dans la machine par petits bouts mais qu'à la fin on obtient aussi un arbre de profondeur considérable, c'est la pile de l'application qui explose lorsqu'on fait l'appel récursif de la méthode `rewrite()`.
- L'activation inutile d'instructions. Ce problème se présente sous plusieurs formes qui peuvent se combiner pour donner des performances très pauvres :
  - On exécute un arbre en forme de peigne formé par des instructions Par ou Seq dans lequel (dans le pire cas) il y a juste un composant à exécuter qui est dans la feuille la plus profonde. On active ainsi des instructions qui n'ont plus rien à faire mais qui restent dans l'arbre, près de la racine, pour garder la cohésion du programme.

    Ce cas se présente le plus souvent avec les instructions Par qui se trouvent à la racine de l'arbre d'exécution de la machine.

    Ce problème ne se présente pas en Rewrite car l'arbre est reconstruit à chaque activation, sans les instructions inutiles; c'est le principe même de réécriture d'un programme en un autre.
  - Replace (et aussi Rewrite) implémente l'attente active d'un événement. Autrement dit, Replace parcourt toutes les branches de l'arbre qui contiennent des instructions événementielles même si l'événement attendu n'est jamais généré. Ce parcours est fait au moins une fois même si aucun événement n'est généré. Le pire cas de cette situation est un programme qui contient seulement des instructions événementielles et dans lequel les événements attendus ne sont pas générés dans l'instant.
  - L'algorithme implémenté pour faire converger l'instant n'est nullement efficace. L'exemple plus clair de ce problème est celui connu sous le nom de cascade inverse. Voici la structure du programme:

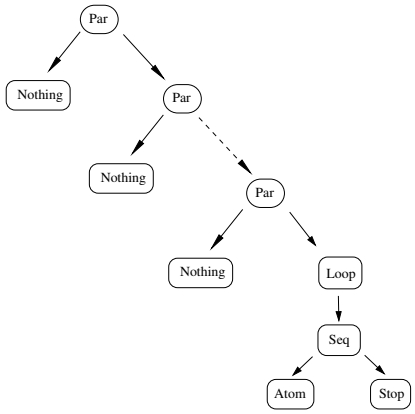

FIG. 3.1: Arbre en peigne

```
Jr.Seq(Jre.Generate(N),
       Jr.Par(Jre.Await(1),Jr.Par(
                Jr.Seq(Jre.Await(2), Jre.Generate(1)),Jr.Par(
                Jr.Seq(Jre.Await(3), Jre.Generate(2)),
                ...
                Jr.Seq(Jre.Await(N),Jre.Generate(N-1))
               ))...)
)
```

Ce programme parcourt $(N + (N-1) + ... + 2 + 1$ fois les branches, ce qui donne une complexité $O(n^2)$ dans le pire cas.

Simple

Cette version a été créée par L. Hazard avec l'objectif d'avoir une implémentation de Junior qui traite un grand nombre d'événements et de composants en parallèle de façon efficace. Pour cela Simple implémente un algorithme complètement différent de celui de Replace; l'algorithme se base sur l'utilisation de files d'attente pour éviter le problème de l'attente active de Replace. Le résultat est une implémentation très efficace mais aussi très complexe (son nom est une ironie), difficile à comprendre, à modifier et surtout pour laquelle il est difficile de prouver qu'elle respecte la sémantique (du moins formellement).

Voici une description abrégée de l'algorithme:

- Un programme en Simple est au début un arbre d'instructions comme en Rewrite et Replace.
- L'exécution du programme se fait en deux phases:
  - La première phase consiste à activer le programme, comme en Rewrite, et à le transformer au fur et à mesure que l'on le parcourt. La transformation principale consiste à éliminer les instructions Par et à mettre les branches obtenues dans des files d'attente.
  - La deuxième phase consiste à traiter les branches qui sont dans les files d'attente.

    Simple ne détruit pas complètement l'arbre, en particulier il garde les instructions de séquence qui sont parfois utilisées pour remonter l'arbre ou plutôt les morceaux de branches qui restent. Cette exécution remontante pose des problèmes pour vérifier et formaliser Simple; l'exécution remontante n'est en effet pas une exécution structurelle.

Le gain en vitesse de Simple est dû, principalement, à deux choses : 1) Il n'y a plus d'instructions Par et 2) on n'exécute que le strict nécessaire grâce aux files d'attentes. Parmi les optimisations également faites en Simple, on trouve le nettoyage de la table d'événements qui réduit l'espace mémoire utilisé.

Storm

Cette version a été créée par J.F. Susini [SUS 01] avec l'objectif de concilier efficacité et sémantique formelle. Pour garantir la sémantique formelle de Junior, l'implémentation de Storm dérive de celle de Replace. Pour avoir une version plus rapide que Replace, Storm emprunte le concept de file d'attente de Simple.

Les objectifs particuliers de Storm sont les suivants :

1. Eviter le parcours inutile des branches dans l'arbre d'exécution qui sont dans un état suspendu et que l'on exécute parce qu'il a été généré un événement qui n'est pas celui qu'attend la branche.
2. Eviter que la taille de l'arbre soit trop grande car la pile explose lors de l'appel de la méthode `rewrite`. Ce problème empêche l'exécution de programmes de grande taille.

Voici une description abrégée de l'algorithme :

- On utilise la même implémentation des événements que Rewrite et Replace (une table de hash qui contient des cellules qui représentent les événements définis de façon générique par l'interface `Identifier`).
- On conserve l'exécution structurelle de Replace, c'est-à-dire que l'on continue à utiliser un arbre d'exécution qui est réutilisé par les instructions Loop et Repeat grâce à la méthode `reset()`.
- La plupart des instructions sont implémentées de la même façon qu'en Replace. Les changements les plus importants sont :

  1. La définition du code WAIT.
     Pour ne pas activer inutilement les branches suspendues sur un événement qui n'a pas encore été généré, les instructions événementielles ont un nouveau comportement : elles s'enregistrent dans des files d'attentes associées à l'événement attendu et rendent le code WAIT. Le code WAIT sert ainsi à distinguer les instructions qui sont vraiment suspendues (par exemple celles qui attendent la fin de l'instant) de celles qui attendent un événement. Le code WAIT est traité, tout particulièrement, par l'instruction Par qui ne réactive que les instructions suspendues.
  2. La définition de la méthode `zap()`.
     Cette méthode, exécutée par l'instruction Generate, libère un chemin dans l'arbre d'exécution pour réactiver les branches qui ont rendu WAIT. Le chemin est codé dans les instructions Par.
  3. Le balancement des instructions Par.
     Le balancement réduit la taille de l'arbre ainsi que l'activation inutile de branches. Il est important de mentionner que le balancement n'est fait qu'en plus haut niveau, dans les instructions Par insérées par la machine réactive et pas dans les instructions Par imbriquées à l'intérieur d'autres instructions réactives.

Storm est en général un bon algorithme, cependant il reste quelques problèmes à régler: garder le statut WAIT inter-instant, balancer toujours l'arbre et pas seulement lorsqu'on ajoute un nouveau comportement et l'utiliser dans les instructions Par imbriquées. Une nouvelle version, appelée ThunderBall, réglera ces problèmes.

Rvm

Cette version de Junior a été créée par D. Petoukhov avec l'objectif d'avoir une version qui tourne dans un système embarqué comme la JavaCard. L'algorithme d'exécution fonctionne sur la base d'un bytecode qui est programmé à l'aide d'un langage assembleur. Le langage assembleur est donc un langage assembleur réactif de bas niveau qui peut être utilisé pour programmer d'autres formalismes, par exemple des programmes Junior et SL. Il existe deux implémentations de la Rvm, une en Java et une autre en C.

Voici une description abrégée de l'algorithme:

- L'allocation de mémoire est une opération extrêmement chère dans les systèmes de temps réel, dans la JavaCard par exemple. En effet, la création des objets est une des opérations les plus coûteuses et la désallocation de mémoire est tout simplement impossible. La Rvm n'utilise pas donc d'allocation dynamique de mémoire; les instructions réactives et les événements ne peuvent pas être libérés. Pour être dynamique, la Rvm utilise son propre gestionnaire de mémoire qui est plutôt un gestionnaire d'objets (gérés par une file d'objets libres).

- Les instructions réactives sont constituées par un opcode (28 optcodes ont été définis), un pointeur sur la prochaine instruction et des paramètres qui dépendent de l'instruction. Les instructions sont stockées dans une zone de mémoire finie qui est allouée statiquement au démarrage de la machine. Comme les instructions ne sont pas modifiées (elles sont des instructions sans état), l'implémentation d'une instruction réactive est définie par plusieurs optcodes.

- Les instructions de composition ont une implémentation particulière :

  La séquence La prochaine instruction à exécuter est soit la prochaine instruction en mémoire, soit un père. L'instruction de séquence n'existe donc pas, et est implémentée par la suite des instructions dans la mémoire.

  Le parallélisme L'instruction Par est implémentée par plusieurs files de nœuds qui pointent sur les branches. L'exécution d'un programme réactif consiste à activer les branches qui sont dans la file d'instructions prêtes à s'exécuter, et à gérer les valeurs de retour pour savoir ce qu'il faut faire après : 1) re-exécuter les branches suspendues; plusieurs cas sont analysés pour savoir s'il faut donner la main à une instruction père ou pas, 2) exécuter les branches stoppées à l'instant d'après; il existe une file pour ces instructions et 3) ne plus exécuter les branches terminées; sauf lorsqu'elles sont dans une boucle et dans ce cas il y a une autre file.

- La Rvm emprunte à Simple, comme Storm, la gestion des événements avec des files d'attente pour ne pas implémenter des attentes actives. Les dernières versions de la Rvm utilisent ce mécanisme de gestion de telle sorte que l'information n'est pas perdue inter-instant comme en Storm.

- Pour faciliter la programmation, il a été défini un assembleur réactif. Chaque ligne dans le code source représente une instruction réactive; l'opcode est représenté par un nom symbolique et les pointeurs sont implémentés comme des numéros de ligne du fichier qui contient le code source. Lorsque le programme est chargé en mémoire, les numéros sont remplacés par des vraies adresse mémoire (pour ce changement un algorithme de deux phases est utilisé).

- L'exécution de programmes Junior est faite à l'aide d'un module de chargement qui traduit un programme Junior en un programme pour la Rvm. En général, la traduction est directe avec quelques particularités : la séquence est éliminée, et l'instruction Local est éliminée et implémentée comme un renommage d'événements.

Glouton

Cette version a été créée par L.Mandel [MAN 02] avec l'objectif de formaliser Simple. La sémantique de Glouton est donnée avec des règles de réécriture qui décrivent l'opération principale de l'algorithme: le déplacement des bouts de code qui sont dans des files d'attente vers les files d'exécution de l'instant courant et de l'instant d'après. Comme Simple, Glouton n'exécute que le strict nécessaire, c'est-à-dire que les déplacements ne sont faits que lorsque la présence, l'absence ou la configuration événementielle associée à une file est satisfaite.

Les différences principales par rapport à Simple sont les suivantes:

1. une notion de groupe est utilisée pour : 1)implémenter la séquence, 2) implémenter les boucles, et 3) gérer la dépendance d'un programme réactif sur un événement.

2. une sémantique différente de l'instruction de préemption. La sémantique de l'instruction de préemption est celle de l'instruction Kill du langage SL, c'est-à-dire une instruction qui ne finit pas instantanément en cas de préemption effective.

3. L'instruction Control n'existe pas en tant qu'instruction. L'instruction est traduite en transformant les configurations dans chaque instruction événementielle.

Pour finir cette section, on va expliquer l'exécution d'un programme Junior dans trois des implémentations que l'on vient de présenter : Replace, Simple et Storm. L'explication que l'on fera de Replace est également valable pour Rewrite et celle de Simple est également valable pour Glouton. Pour faciliter l'explication on va écrire le programme Junior dans un pseudo-code où les instructions de séquence sont omises et les instructions de parallélisme sont notées avec le caractère |||. Voici le programme Junior considéré :

```
1       Jr.Await("A")
    ||
2       Jr.Await("B")
3       Jr.Generate("A")
    ||
4       Jr.Stop()
5       Jr.Generate("B")
```

Ce programme Junior, indépendamment de l'implémentation, s'exécute en deux instants : 1) dans le premier instant les deux premières branches stoppent leur exécution car les événements qu'elles attendent ne sont pas générés; la troisième branche se stoppe explicitement, 2) dans le deuxième instant la troisième branche génère l'événement `"B"` qui permet l'exécution de la deuxième branche qui à son tour, avec la génération de l'événement `A`, déclenche l'exécution de la première branche. Le programme finit donc au deuxième instant avec la génération des événements `"A"` et `"B"`.

Voici la suite des pas réalisés par chaque implementation (attente désigne la mise dans la file d'attente de l'événement attendu par l'instruction) :

Rewrite/Replace

instant 1 : 1, 2, 4, 1, 2
instant 2 : 1, 2, 5, 2, 3, 1

Simple/Glouton

instant 1 : 1(attente), 2(attente), 4
instant 2 : 5, 2, 3, 1

Storm

instant 1 : 1(attente), 2(attente), 4
instant 2 : 1(attente), 2(attente), 5, 2, 3, 1

Remarquez que lorsque l'implémentation repasse plusieurs fois sur la même instruction cela signifie qu'elle fait de l'attente active. C'est le cas en Rewrite/Replace au cours du même instant, par exemple pour l'instruction 1. C'est le cas de Storm entre les deux instants. Par contre en Simple/Glouton il n'y a jamais d'attente active. Les différences entre les implémentations par rapport à l'attente active seront considérées plus en détail dans le chapitre 4.

3.2 Analyse de Junior

Dans cette section on va présenter une analyse des instructions réactives de Junior, et les propositions que l'on a faites pour pallier l'absence de quelques comportements. Junior, par rapport à Reactive-C et SugarCubes, est censé définir un modèle de programmation bien établi et simple à utiliser; malheureusement ce n'est pas toujours le cas car certaines instructions ont une sémantique complexe. En plus, dans certain cas, il manque des comportements qui rendent le code moins lisible et compact. Voici les instructions en question:

L'instruction Control La particularité de l'instruction Control est la restriction dans le type d'expressions événementielles que l'on peut utiliser. En effet, l'instruction Control n'accepte pas de configurations. Cette limitation n'est pas complètement justifiable car la seule limitation valide est le contrôle d'un comportement réactif lorsqu'un événement n'est pas présent. En effet, si l'instruction Control exécute son corps lorsqu'un événement est présent, il est clair que l'utilisation de l'absence d'un événement est une flagrante contradiction. Par contre le contrôle d'un comportement par des configurations And et Or ne pose aucun problème.

L'utilisation des configurations dans l'instruction Control (sans le test d'absence d'un événement) peut être implémentée avec un événement auxiliaire de contrôle et un comportement réactif en parallèle qui détecte si la configuration est satisfaite. Voici un exemple d'une telle implémentation :

```
Jr.Control(Jr.And(Jre.Presence("A"),
                  Jre.Presence("B")),
           behavior()
          )
```

programme équivalent :

```
Jre.Local("control",
    Jr.Par(Jr.Loop(Jr.Seq(Jr.Seq(
                     Jr.Await(Jr.And(Jre.Presence("A"),
                                     Jre.Presence("B"))),
                     Jre.Generate("control")),
                     Jr.Stop() )
                   ),
           Jre.Control("control",
                   behavior())
          )
    )
```

L'instruction Freezable Cette instruction présente aussi la particularité de ne pas utiliser de configurations événementielles sans une réelle justification. Le problème se pose lorsque l'on veut récupérer les instructions gelées de, par exemple, la configuration `freezable( "A" || "B" )`. Que retourne la méthode `getFrozen`? Le parallèle de `getFrozen("A")` et `getFrozen("B")`? et dans le cas de `freezable( "A" && "B" )`? il est clair qu'il n'y a pas de sémantique intuitive pour cette opération. La solution serait d'utiliser un événement auxiliaire (comme pour l'instruction Control) adapté à l'application.

L'autre particularité de l'instruction Freezable est l'absence de *handler* comme celui de l'instruction Until. Cette absence n'est pas justifiable bien que l'on puisse implémenter un comportement similaire avec un comportement en parallèle comme pour l'instruction Control.

L'instruction Until Le problème de l'instruction Until est que sa sémantique est trop expressive; elle décrit plusieurs comportements en fonction du drapeau du corps (TERM, STOP et SUSP), de la configuration de préemption (satisfaite, pas satisfaite ou on ne sait pas encore) et de la détection de la fin de l'instant (vrai, faux). Par exemple si on ne considère que les cas lorsque le corps n'est pas suspendu, on a les 4 cas suivants :

Le corps	La configuration	Fin de l'instant	Action
TERM	X	X	Terminaison instantanée
STOP	satisfaite	vrai	exécution du handler à l'instant $i+1$
STOP	satisfaite	faux	exécution du handler à l'instant i (préemption instantanée)
STOP	non satisfaite	X	corps à l'instant $i+1$

le caractère X signifie que la valeur n'est pas importante pour prendre la décision.

Ce qui est particulièrement anormal dans ce tableau est le cas connu comme préemption instantanée (avec un handler vide):

```
Jr.Until( configuration ,
          Jr.Seq(Jre.Print("atom 1"), Jr.Seq(
                 Jr.Stop(),
                 Jre.Print("atom 2")))
          Jr.Nothing())
```

on exécute une instruction Stop qui se trouve dans le corps de l'Until et, au lieu d'arrêter l'exécution de l'instruction Until (et de la branche) pour l'instant courant, l'instruction Until finit son exécution.
L'autre cas qui cause des problèmes est le fait de ne pas savoir si le handler sera exécuté dans l'instant courant ou au prochain.

L'instruction Loop Le problème de l'instruction Loop est la terminaison instantanée de son corps qui est potentiellement dangereuse. Le plus souvent on tombe sur ce problème lorsqu'on exécute une instruction qui peut finir instantanément, par exemple l'instruction Until ou l'instruction When. Une solution à ce problème consiste à mettre en parallèle avec le corps une instruction Stop; cette solution n'élimine pas la désynchronisation possible due à la complexité de l'instruction Until ou à l'utilisation de l'instruction When.

3.2.1 Implémentation du parallélisme

La sémantique de l'opérateur de parallélisme (l'instruction Par) ne spécifie pas un ordre particulier pour exécuter ses deux branches ; cette sémantique permet d'implémenter l'instruction de façon déterministe (en choisissant toujours le même ordre) ou pas. En Rewrite et Replace l'instruction Par a été implémentée de façon déterministe (on exécute d'abord la branche gauche puis la droite) et en Simple l'instruction Par est non déterministe à cause de l'algorithme utilisé.

Il est important de dire que dans la première implémentation du modèle réactif en Java (les SugarCubes) la sémantique de l'instruction Par (appelée merge) était une sémantique déterministe et que celle-ci a été adaptée en Junior pour modéliser le fonctionnement de Simple. On peut voir la complexité de cette adaptation dans la sémantique du Par qui est définie par 6 règles de réécriture qui définissent toutes les permutations possibles de ses branches. Par exemple, si on a trois branches en parallèle `Jr.Par(Jr.Par(P1,P2), P3)` et si on ne considère pas les deux premières règles de réécriture du Par (de la figure 2.7) on obtient les 4 premiers cas décrits dans la figure 3.2 (au-dessus de la ligne foncée). Par contre si on considère toutes les règles on obtient les six cas possibles de la figure 3.2 (y compris les deux cas qui sont audessous de la ligne foncée) qui considèrent les permutations entre les instructions du Par qui contient les programmes `P1` et `P2` et le Par qui contient le premier Par et le programme `P3`.

Le choix entre parallélisme déterministe et non-déterministe n'est pas évident. Un parallélisme déterministe est très utile pour débugger et vérifier des propriétés des programmes réactifs. Le parallélisme non-déterministe est en général plus rapide et plus simple à implémenter car on n'a pas besoin de respecter l'ordre d'exécution.

3.2.2 Les instructions primitives

A la différence d'autres langages (comme Esterel), la plupart des instructions en Junior sont des instructions primitives, c'est-à-dire, que le comportement n'est pas exprimable par d'autres instructions. En fait, il y a seulement une instruction réactive qui n'est pas primitive, l'instruction Await. Le comportement de l'instruction Await peut être reproduit par le programme suivant :

P1	P2	P3
P2	P1	P3
P3	P1	P2
P3	P2	P1
P1	P3	P2
P2	P3	P1

FIG. 3.2: Combinaisons du Par

```
Jr.Await( configuration ) = Jr.Until( configuration ,
                                Jr.Loop(Jr.Stop())
                            )
```

Par ailleurs il existe un autre élément de Junior qui est en général primitif et qui n'est pas une instruction réactive : les configurations événementielles.

3.2.3 Scanner, Callback ou Interrupt

Lorsqu'on génère des événements valués en Junior, on est obligé d'attendre la fin de l'instant pour être sûr que l'on a toutes les valeurs. Cette attente oblige le programmeur à concevoir le programme réactif en deux phases (ou instants): une pour générer les événements et une autre pour les traiter. Si l'événement valué qui nous intéresse ne génère qu'une valeur, alors on n'est pas obligé d'attendre la fin de l'instant et on peut récupérer la valeur avec la méthode `currentValues()`. Dans certains cas, on sent bien que cette attente est inutile et que l'on pourrait traiter les événements au fur et à mesure qu'il sont générés. Plusieurs propositions ont été faites pour remédier à ce problème et jusqu'à maintenant celle qui a été le mieux acceptée est l'instruction Scanner. Voici une description des instructions qui ont été proposées:

L'instructions Listener L'idée de cette instruction est d'exécuter du code Java entre deux instants. L'exécution inter-instant est très utile car on sait que le système se trouve dans un état stable et que l'on a toute l'information événementielle pour faire des calculs cohérents. En général, le programmeur peut exécuter du code Java inter-instant comme on l'a fait dans l'exemple 2.3 [1], cependant l'inconvénient de le faire de cette façon est que le code réactif ne contient pas la description inter-instant et il est donc plus difficile à le comprendre et à le porter sur un système où l'exécution de code Java inter-instant n'a pas été prévue. Une autre façon de s'assurer que l'on a toute l'information événementielle serait de dédier un instant aux réactions événementielles et un autre instant aux calculs. Cette solution est toujours possible mais elle est plus lente et pose le problème de la synchronisation.

L'instruction Scanner Cette instruction existe en SugarCubes et elle s'appelle CallBack. En Junior on l'appelle Scanner et son comportement consiste à exécuter une action atomique, à chaque fois qu'est généré un événement valué en cours d'instant. Pour être sûr que l'on réagit à toutes les générations valuées de l'événement, l'instruction exécute l'action atomique (une micro-réaction) et puis elle rend SUSP tant que la fin de l'instant n'a pas été déclarée ; à la fin de l'instant elle rend TERM. En SugarCubes, il existe une variante de cette instruction qui au lieu de rendre TERM à la fin de l'instant rend STOP. Autrement dit, l'instruction se comporte comme `Jr.Loop(Jr.Scanner(action atomique))`. Pour éviter de possibles dérapages, on s'interdit la génération de l'événement de contrôle dans l'action atomique.

L'instruction Interrupt Cette instruction a été proposée par M. Serrano dans l'implémentation qu'il a faite du réactif en Scheme. L'idée est d'implémenter l'instruction Scanner avec une réaction à la génération

[1] on code les actions inter-instants dans la méthode qui fait réagir la machine réactive

valuée d'un événement le plus vite possible Pour cela il exécute les actions atomiques immédiatement lorsque l'événement est généré, et non à la prochaine micro-réaction.

La figure 3.3 illustre graphiquement le comportement de ces instructions.

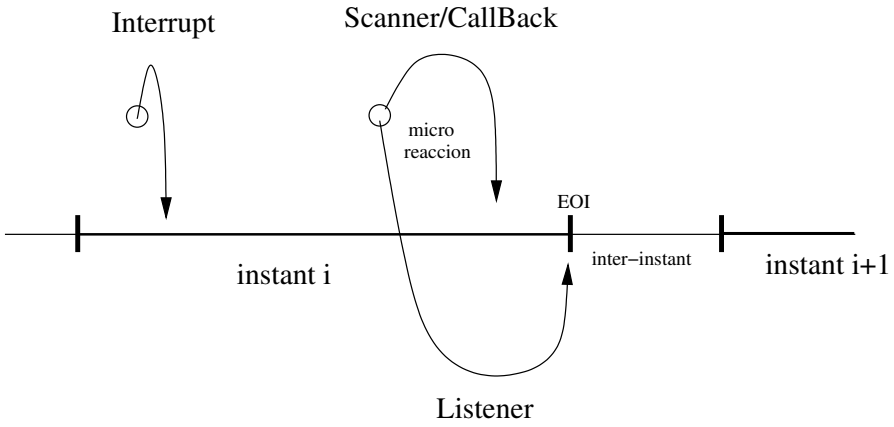

FIG. 3.3: Différents types de réactions

3.2.4 Run et Dynapar

En Junior, à la différence de SugarCubes, il n'existe pas d'instruction réactive qui permette de créer un programme réactif et de l'ajouter dans la machine à un point particulier qui ne soit pas en parallèle avec toutes les autres composantes. Ce type d'instruction s'avère très utile lorsqu'on a construit une entité réactive et que l'on veut étendre ses capacités; on veut que le nouveau comportement soit aussi soumis aux événements de contrôle définis dans l'entité, par exemple, le tuer avec un événement.

En SugarCubes, il existe les instructions Shell et AddShell qui permettent de faire ce type d'opérations. L'instruction `Shell(identif1, prog1)` associe l'identifieur `identif1` de type String au comportement réactif `prog1`; l'instruction `AddShell(identif2, prog2)` ajoute une copie du comportement réactif `prog2` dans toutes les instructions Shell qui ont l'identifieur `identif2`. L'ajout des copies se fait à la fin de l'instant. Si par erreur l'instruction `AddShell(identif, prog1)` est exécutée avant l'exécution de l'instruction `Shell(identif, prog2)` le programme `prog1` se perd.

Cette façon d'implémenter l'ajout dynamique présente deux problèmes: 1) l'ajout nécessite un espace de nom différent de celui des événements, et 2) cet espace de nom (défini en Java et pas dans le réactif) introduit une notion supplémentaire à formaliser [2]. D'ailleurs, il faut rappeler que jusqu'à maintenant on n'a pas formalisé ce qui se passe dans le monde Java; on se contente de dire qu'il y a un "effet de bord" mais on dit pas lequel. C'est pour cette raison que l'on a proposé une nouvelle instruction en Junior qui implémente une sorte de Shell basé sur la notion d'événement. Voici un exemple élémentaire d'utilisation de l'instruction Dynapar :

```
Jr.Until("kill",
         Jr.Dynapar("add",
                    Jr.Loop(Jr.Seq(Jr.Print("*"),
                                   Jr.Stop())))
                   )
        )
```

Ce programme imprime `*` à chaque instant. Supposons que l'on génère l'événement valué `"add"` à l'instant k, avec la valeur suivante :

[2] Ce n'est que récemment que J-F Susini a donné une sémantique formelle aux SugarCubes.

```
Jr.Loop(Jr.Seq(Jr.Print("-"),
                Jr.Stop()))
       )
```

A partir de l'instant $k+1$, le programme se mettra alors à imprimer `*-`, une fois par instant. Ce comportement se poursuit jusqu'à une génération de `"kill"` ou une nouvelle génération de `"add"`. L'événement `"kill"` tue ainsi le comportement original qui imprime `*` mais aussi celui qui imprime `-`.

L'ajout d'un comportement réactif à un point particulier de l'arbre du programme, que permet Dynapar, ne pouvait pas être implémenté dans la version originale de Junior. Dynapar ajoute donc de la puissance expressive à Junior. La sémantique de l'instruction DynaPar est la suivante:

$$\frac{t, E \xrightarrow{SUSP} t', E'}{DynaPar(e,t), E \xrightarrow{SUSP} DynaPar(e,t'), E'}$$

$$\frac{t, E \xrightarrow{TERM} t', E'}{DynaPar(e,t), E \xrightarrow{TERM} Nothing(), E'}$$

$$\frac{t, E \xrightarrow{STOP} t', E'}{DynaPar(e,t), E \xrightarrow{SUSP} DynaPar*(e,t'), E'}$$

$$\frac{eoi(E) = false}{DynaPar*(e,t), E \xrightarrow{SUSP} DynaPar*(e,t)), E}$$

$$\frac{eoi(E) = true \quad e \in E}{DynaPar*(e,t), E \xrightarrow{STOP} DynaPar(e, Par(t, currentValues(e))), E}$$

$$\frac{eoi(E) = false \quad e \notin E}{DynaPar*(e,t), E \xrightarrow{STOP} DynaPar(e,t), E}$$

Une des particularités de cette instruction, définie clairement par la sémantique, est que l'instruction termine lorsque le corps finit même si dans le même instant on reçoit l'événement d'ajout.

Evidemment, on aurait pu choisir d'autres sémantiques: 1) si le corps finit et l'événement valué est généré, on ajoute le comportement en parallèle, 2) même si le corps finit, l'instruction ne finit pas et on peut à tout moment ajouter des nouveaux comportements. La deuxième sémantique décrit le comportement de `dynaPar` comme une sorte de puit; le système de règles est encore plus simple car on remplace la deuxième et troisième règle par:

$$\frac{t, E \xrightarrow{\alpha} t', E' \quad \alpha! = SUSP \quad DynaPar*(e,t'), E'' \xrightarrow{\beta} t'', E''}{DynaPar(e,t), E \xrightarrow{\beta} t'', E''}$$

L'instruction DynaPar est contraignante à cause de l'ajout du nouveau comportement à l'instant d'après. On a donc fait une autre proposition pour ajouter immédiatement des nouveaux comportements réactifs à un point précis dans l'arbre d'exécution : l'instruction Run. L'instruction Run évalue un `ProgramWrapper` (que l'on décrira plus loin) qui lui rend un programme réactif puis elle se comporte comme celui-ci. Voici un exemple d'utilisation de Run et le `ProgramWrapper` associé :

```
Jr.Loop(Jr.Until("newBehavior",
                 Jr.Run( new ImplemProgramWrapper() ) ) )

public class ImplemProgramWrapper
{
  int i=0;

  Program evaluate(Environment env)
```

```
  {
      i++;
      if(i%2==0) return  Jr.Loop(Jr.Seq(Jr.Println("-"),
                                        Jr.Stop() )
                                 );
      else   return Jr.Loop(Jr.Seq(Jr.Println("*"),
                                   Jr.Stop() )
                            );
  }
}
```

Lorsqu'on exécute ce programme la première fois, l'instruction Run évalue le ProgramWrapper qui lui rend le programme réactif P (qui imprime `*` à chaque instant). Ensuit le Run se comporte comme P jusqu'à ce que l'événement `"newBehavior"` soit généré. Lorsque l'événement `"newBehavior"` est généré l'instruction Run est préemptée et la boucle (instruction Loop) est relancée : l'instruction Run réévalue le ProgramWrapper et se met alors à imprimer `-`. La sortie de ce programme est une chaîne de caractères formée par `"*"` et `"-"`. Le changement de caractère est le résultat de la génération de l'événement `"newBehavior"`.

Voici la règle de réécriture de l'instruction Run :

$$\frac{pw, E \rightsquigarrow p, E \quad p, E \xrightarrow{\alpha} p', E'}{Run(pw), E \xrightarrow{\alpha} p', E'}$$

L'opérateur $\rightsquigarrow$ évalue le wrapper pw (de type `Program`) dans l'environnement d'exécution[3] E pour obtenir un programme p. L'instruction Run évalue son wrapper, puis elle se comporte comme le programme obtenu.

L'instruction Run est très expressive, à tel point que l'on peut l'utiliser pour implémenter l'instruction Dynapar (on verra une implémentation possible dans la section 6.1.5). La question qui se pose maintenant est de savoir si on garde l'instruction Dynapar, on a décidé de la garder car 1) la sémantique est plus claire à partir des règles de réécriture que du programme Junior équivalent, 2) elle ne pose pas de problèmes d'appels récursifs à l'infini comme le Run, et 3) elle s'implémente plus efficacement sans l'instruction Run.

La table 3.2 résume tous les types d'ajouts de comportements conçus jusqu'à maintenant.

Ajout \ Quand	instantané	pas instantané
local	Run	Shell, DynaPar
machine		add

Tab. 3.2: Ajouts dynamiques de comportements réactifs

3.2.5 Par et Seq n-aires

Les deux instructions réactives de composition en Junior sont la séquence et le parallélisme. Ces instructions sont les plus utilisées lorsqu'on construit un programme réactif et elles sont aussi les plus exécutées. L'implémentation de ces instructions doit donc être la plus efficace pour réagir le plus rapidement possible. Dans le chapitre 4 on a mesuré le temps d'exécution de plusieurs types de programmes et on a trouvé que l'instruction Par avait un impact direct sur le temps d'exécution mais aussi sur la taille maximale des programmes que l'on pouvait

[3]Une description plus précise consisterait à utiliser l'environnement Java dans les règles; on ne l'a pas fait ici car l'évaluation n'affecte pas l'environnement réactif et parce que, pour le moment, on ne s'intéresse pas aux modifications faites à l'environnement Java.

exécuter [4]. Pour réduire le temps d'exécution et augmenter la taille maximale des programmes réactifs que l'on peut exécuter, on a proposé l'utilisation des instructions de composition n-aires.

Cette sous-section présente la formalisation de ces instructions. Voici la description de leurs algorithmes et de leur sémantique :

- L'instruction Seq.

  Algorithme

  On utilise une liste (ou tableau) et au moins un pointeur `current` (ou index) qui pointe sur l'instruction de la liste à exécuter (ce pointeur pointe sur la première instruction). L'exécution consiste à activer l'instruction pointée par `current` et analyser le code de retour: s'il est TERM, on pointe le pointeur à la prochaine instruction, sinon (STOP ou SUSP) on rend la même valeur et on ne change pas le pointeur. L'instruction termine lorsqu'on active la dernière instruction et elle rend TERM.

  Sémantique

  La sémantique est celle de Reactive-C [BOU 96e].

- L'instruction Par.

  Algorithme

  On utilise trois listes : 1) une qui contient les instructions initiales (*InitList*) à exécuter. Lorsque l'instruction Par est sous le contrôle d'une instruction Loop, cette liste sert à réinitialiser les instructions, 2) une qui contient les instructions suspendues (*SuspList*) à exécuter dans l'instant courant (état original de toutes les instructions), et 3) une qui contient les instructions stoppées (*StopList*), à exécuter à l'instant d'après (liste originalement vide). L'exécution consiste à activer toutes les instructions suspendues et à analyser le code de retour: s'il est TERM, on l'élimine de la liste, s'il est STOP on la déplace dans la liste des instructions stoppées, et s'il est SUSP on passe à l'instruction suivante. L'instruction rend SUSP si au moins une instruction qui la compose a rendu SUSP. L'instruction rend STOP lorsque toutes les instructions suspendues ont été déplacées dans la liste des instructions stoppées (et on les transfère dans la liste des instructions suspendues pour les activer à l'instant d'après). L'instruction rend TERM lorsqu'il n'y a plus aucune instruction dans les deux listes *SuspList* et *StopList*.

  Sémantique

  La sémantique du Par n-aire n'a jamais été définie auparavant. Elle est définie par des règles de réécriture de le forme suivante :

$$InitList\ SuspList\ StopList, E \xrightarrow{\alpha} InitList'\ SuspList'\ StopList', E'$$

où *InitList* est la liste initiale des instructions réactives,

SuspList est la liste des instructions suspendues, et

StopList est la liste des instructions stoppées.

1. Pour chaque instruction Par n-aire, on crée trois listes et on les réécrit :

$$\frac{[b_1, ..., b_n][][], E \xrightarrow{\alpha} InitList\ SuspList\ StopList, E'}{[b_1, ..., b_n], E \xrightarrow{\alpha} InitList\ SuspList\ StopList, E'}$$

2. On fait réagir une fois toutes les instructions qui sont dans la liste initiale.
   Si l'instructions termine, on l'élimine de la liste initiale :

[4] La taille dépend de la taille de la pile, laquelle est très utilisée dans les algorithmes récursif que l'on utilise pour parcourir l'arbre du programme réactif. Certaines implémentations comme Simple ou Glouton n'utilisent pas un arbre pour exécuter les programmes réactifs mais elles commencent l'exécution à partir d'un arbre; le parcours de cet arbre présente donc le même problème de pile.

$$\frac{b_1, E \xrightarrow{TERM} b_1', E' \quad [b_2, ..., b_n]\ SuspList\ StopList, E' \xrightarrow{\alpha} InitList'\ SuspList'\ StopList', E''}{[b_1, b_2, ..., b_n]\ SuspList\ StopList, E \xrightarrow{\alpha} InitList'\ SuspList'\ StopList', E''}$$

Si l'instructions est suspendue, on la met dans la liste d'instructions suspendues :

$$\frac{b_1, E \xrightarrow{SUSP} b_1', E' \quad [b_2, ..., b_n][SuspList, b_1']\ StopList, E' \xrightarrow{\alpha} InitList'\ SuspList'\ StopList', E''}{[b_1, b_2, ..., b_n]\ SuspList\ StopList, E \xrightarrow{\alpha} InitList'\ SuspList'\ StopList', E''}$$

Si l'instructions rend STOP, on la met dans la liste d'instructions stoppées :

$$\frac{b_1, E \xrightarrow{STOP} b_1', E' \quad [b_2, ..., b_n]\ SuspList\ [StopList, b_1'], E' \xrightarrow{\alpha} InitList'\ SuspList'\ StopList', E''}{[b_1, b_2, ..., b_n]\ SuspList\ StopList, E \xrightarrow{\alpha} InitList'\ SuspList'\ StopList', E''}$$

Lorsqu'on finit d'activer toutes les instructions (la liste initiale est vide) et qu'il reste des instructions dans la liste d'instructions suspendues, on rend SUSP :

$$[][b_1, ..., b_n][r_1, ..., r_n], E \xrightarrow{SUSP} [b_1, ..., b_n][][r_1, ..., r_n], E$$

3. Lorsqu'on a activé au moins une fois toutes les instructions réactives initiales (la liste est vide) et qu'il n'y a plus d'instructions suspendues, on rend STOP et on déplace les instructions stoppées vers la liste d'instructions suspendues pour les exécuter à l'instant d'après :

$$[][][b_1, ..., b_n], E \xrightarrow{STOP} [b_1, ..., b_n][][], E$$

4. Lorsque les trois listes sont vides, l'instruction Par termine et on rend TERM.

$$[][][], E \xrightarrow{TERM} [][][], E$$

3.3 Travaux similaires

Reactive-C

Reactive-C [BOU 91], créé par F. Boussinot, est une implémentation (la première) de ce que l'on appelle le modèle réactif synchrone; elle a été faite en C et dispose d'une sémantique formelle donnée avec des règles de réécriture. L'objectif est d'obtenir un assembleur réactif pour implémenter divers modèles de calcul fondés sur des notions de réaction ou d'instant. Parmi les implémentations déjà réalisées, on peut citer :

- Les réseaux de processus réactifs (réseaux "de Kahn" avec une notion d'instant) dans lesquels le test de la file vide devient possible tout en préservant le déterminisme (environ 200 lignes de RC).
- Le langage SL qui est une version d'Esterel sans préemption forte, dans lequel la réaction à l'absence d'un signal est reportée à l'instant suivant, ce qui permet d'éliminer les cycles de causalité (environ 800 lignes de RC).
- Le modèle POR ("Programmation par Objets Réactifs") implémenté par Soft Mountain sous le nom RLib, sur lequel Modcomp-AEG a construit l'outil PROTOP/IX. Cet outil est utilisé pour contrôler la cimenterie d'Angoulème des Ciments Lafarge (l'application comprend plusieurs milliers d'agents POR exécutés en parallèle et a été décrite à RTS'94).

Indépendamment du langage de programmation utilisé par Junior et Reactive-C (Java et C respectivement), on peut dire que le modèle d'exécution de Reactive-C est un modèle plus expressif que celui de Junior. En effet, Reactive-C permet le contrôle fin de l'exécution d'un programme réactif que Junior ne permet pas. Par exemple, en Reactive-C le programmeur a à sa disposition une primitive pour suspendre l'exécution d'une branche et donner le contrôle à une autre. Ceci a deux conséquences :

- On peut construire un ensemble plus large de programmes et donc l'expressivité du langage est plus grande. Avec Reactive-C, on pourrait implémenter le moteur d'exécution de Junior et ses programmes mais pas l'inverse. Les programmes Junior sont un sous-ensemble des programmes générés avec Reactive-C.
- Le coût à payer est de pouvoir construire des programmes qui sont syntaxiquement corrects mais sémantiquement incorrects en ce qui concerne la gestion des micro-réactions. Par exemple, si on implémente Junior avec Reactive-C, il faudrait s'assurer que toutes les branches suspendues de l'instruction Par soient réactivées au cours du même instant pour qu'elles voient toutes les modifications qui ont été faites lorsque l'instruction était suspendue (voir le même problème en SugarCubes ci-dessous).

Par rapport à Junior, Reactive-C est vraiment un langage de bas niveau[5] (assembleur réactif) car il n'existe pas d'événements ; lorsque le programmeur les implémente, il doit faire attention à l'efficacité en évitant des attentes actives. Reactive-C définit quelques instructions dont l'opérateur de parallélisme déterministe (Merge) est l'un des plus importants.

SugarCubes

Les SugarCubes [BOU 98c] sont la première implémentation du modèle réactif en Java. La première implémentation (V1) fut très prometteuse car elle réunissait les meilleures caractéristiques de Reactive-C, de Java et surtout de l'expérience accumulée dans la construction des systèmes réactifs. L'équipe de développement (L. Hazard, F. Boussinot et J-F. Susini) a très vite proposé des améliorations (V2) et pendant ce processus se sont présentés deux problèmes: 1) la nature du modèle est telle que l'on peut l'implémenter de plusieurs façons, et 2) certaines de ces implémentations peuvent conduire à la construction de programmes incohérents que l'on voudrait éliminer. Pour éviter ces problèmes, il a été proposé Junior; Junior définit un ensemble d'instructions réduit (sans les instructions potentiellement dangereuse) et donne une sémantique formelle aux instructions pour bien préciser les comportements réactifs à implémenter.

Pour mieux comprendre les complications de la programmation de bas niveau, voyons comme est utilisée la notion de micro-instant en SC. Les micro-instants sont créés en utilisant l'instruction Suspend et l'instruction Close. Une instruction suspendue se comporte comme l'instruction Stop si elle n'est pas enveloppée d'une instruction Close. L'instruction Close active son corps tant qu'il n'est pas stoppé ou fini. L'instruction Suspend divise l'instant en micro-instants et l'instruction Close combine les micro-instants pour créer un seul instant.

Voici un exemple:

```
SC.Close( SC.Merge( SC.Seq(SC.Print("Suspendig"),
                           SC.Suspend(),
                           SC.Print("1"),
                           SC.Stop(),
                           SC.Print("2")),
                    SC.Seq(SC.Print("A"),
                           SC.Stop (),
                           SC.Print "B"))
                  )
        )
```

Ce programme imprime `Suspending A1` au premier instant et `2B` au deuxième instant, puis termine.

Voici un exemple d'un programme en SugarCubes qui pose des problèmes:

```
SC.Merge( SC.Seq(SC.Await("E"),
                 SC.Print("Action_Atomique"),
          SC.Seq(SC.Suspend(),
                 SC.Generate("E"))
        )
```

[5]En fait, Reactive-C n'est pas un langage avec un compilateur, il s'implémente avec des macros. L'utilisation des macros ne permet pas de faire des optimisations.

Ce que l'on attend de ce programme est l'exécution de l'action atomique à l'instant de la génération de l'événement `"E"`. Cependant l'action atomique est exécutée avec un décalage d'un instant car l'environnement ne détecte aucun changement (aucune génération d'un événement pendant la première activation) et donc la fin de l'instant est déclarée.

L'instruction `Suspend` est une des instructions potentiellement dangereuses qui a été éliminée en Junior.

Les différences principales entre SugarCubes (V3) et Junior sont:

- Nombre d'instructions
  Il y a moins d'instructions en Junior (16) qu'en SugarCubes (32).
- Style de programmation
  Le style de programmation de SugarCubes est basé sur l'utilisation de l'instruction Cube qui n'existe pas en Junior. Le Cube est défini comme

  $$Cube = Link + Shell + Freezable$$

  et il n'est pas seulement une instruction; il joue aussi le rôle de machine réactive ce qui permet une programmation par hiérarchies.
- Différences sémantiques
  - La sémantique de l'instruction Freezable de Junior est différente de celle de SugarCubes : l'imbrication d'instructions Freezable en Junior est exécutée de l'intérieur ver l'extérieur; en SugarCubes c'est l'inverse.
  - L'instruction de parallélisme en Junior est non-déterministe tandis que celle de SugarCubes est déterministe (l'ordre est donné par la position des instructions).
  - L'instruction Loop de SugarCubes stoppe les boucles instantanées. En particulier les boucles sont stoppées après avoir exécuté deux fois le corps dans le même instant. En Junior les boucles instantanées sont permises.

FairThreads

Les FairThreads [BOU 02], créées par F. Boussinot, définissent un modèle de programmation concurrente et parallèle. Ce modèle a été défini par une sémantique formelle et il a été implémenté en plusieurs langages (Java, C, Scheme).

Le modèle des FairThreads ressemble fortement au modèle réactif synchrone en plusieurs aspects:

1. Les entités concurrentes sont exécutées de façon coopérative avec une notion d'instant à la Junior, c'est-à-dire dans laquelle tous les composants sont exécutés à chaque instant, d'où le nom de Fair (équitable),
2. Les entités peuvent interagir à l'aide d'un mécanisme de communication par diffusion d'événements,
3. Il est possible de créer de nouveaux composants dynamiquement.

Les FairThreads se distinguent de Junior par :

1. Le langage de programmation est un sous-ensemble de celui de Junior; en particulier la plupart des instructions événementielles n'existe pas et l'instruction Until, d'une part, n'existe pas en tant qu'instruction et d'autre part a la sémantique du Kill de SL.
2. Le parallélisme n'existe pas en tant qu'instruction; on le trouve comme dans les threads, au plus haut niveau (on obtient un parallélisme plat).
3. Les FairThreads proposent quelques caractéristiques qui ne sont pas présentes en Junior:

- Le détachement de comportements réactifs pour les exécuter de façon préemptive. Ceci est seulement possible lorsqu'on exécute les FairThreads dans une plate-forme qui supporte l'interface POSIX. Autrement dit, les FairThreads sont un habillage au-dessus de l'API POSIX.
- L'attente d'événements avec une instruction `await(E, N)` qui attend la génération de l'événement `E` pendant au plus N instants. Pour implémenter efficacement l'attente d'événement les FairThreads utilisent des files d'attente.
- Il est possible de générer du code efficace sous la forme d'un automate.

Implémentations pour l'embarqué

Spirit machine. Cette machine réactive, créée par J-F Susini, est une implémentation du modèle réactif qui vise la construction de systèmes qui disposent de très peu de ressources, comme les systèmes embarqués. Pour atteindre son objectif, la Spirit machine utilise un algoritme à base de bytecode, avec des *optcode* réactifs. La Spirit machine est implémentée en C et définit un ensemble d'instructions réactives à la Junior : des instructions événementielles comme Generate et Await mais aussi des instructions réactives de bas niveau (une sorte de goto réactif et des tests) pour implémenter les instructions non événementielles comme les boucles. La Spirit machine est en développement. Le problème principal de la Spirit machine est sa programmation de bas niveau qui se fait avec de macros et des offsets. Dans la section 6.4, on présente un langage de programmation de haut niveau (RAUL) qui génère du code pour ce moteur.

Jrc. Jrc est un langage de programmation réactive qui vise les systèmes embarqués à faibles ressources. Cette implémentation, faite par O. Parra [PAR 03] a été conçue pour mélanger des primitives réactives avec un langage efficace tel que le C. Jrc s'inspire de Reactive-C et de Junior et utilise les différentes techniques d'exécution de Junior (arbre d'exécution et files d'attente) ainsi que ses primitives événementielles ; de Réactive-C, Jrc utilise l'opérateur de parallélisme déterministe, le Merge, et la structuration des programmes sous forme de fonctions.

La Spirit machine et Jrc restent des prototypes et il faut attendre leur maturité pour pouvoir bien apprécier l'étendue de leurs intérêt. Cependant on peut déjà observer les avantages du bytecode (en termes d'espace mémoire) et ceux de la génération de code sans interprétation (vitesse d'exécution).

L'approche réactive fonctionnelle

Plusieurs implémentations de l'approche synchrone ont été effectuées, Lustre, Signal, Lucid Synchrone. Dans cette section on va présenter les deux principaux portages de l'approche réactive synchrone.

Librairie réactive en Standard ML Cette librairie réactive a été implémenté par Pucella [PUC 98] avec l'objectif de porter Reactive-C en Standard ML; elle est donc très proche du modèle de programmation de Reactive-C.

Les similitudes sont:

- La possibilité de programmer un contrôle fin à l'intérieur des instants, c'est-à-dire de programmer les micro-instants avec des instructions `suspend` et `close`.
- Une sémantique formelle (SOS) et une implémentation directe de la sémantique, par exemple le cœur de l'implémentation est la fonctions `step` qui joue le rôle de la réécriture dans la sémantique.

Parmi les différence on trouve:

- La différence entre instruction réactive et expression réactive. Les expressions réactives peuvent être définies comme des types de données (*datatype*).
- La préemption est implémentée avec le mécanisme d'exception.
- Il n'existe pas de mécanisme de communication autre que les variables partagées.
- La possibilité de programmer des fonctions d'ordre supérieur. Un travail similaire a été fait par Caspi et Pouzet [CAS 96] en Lustre.

Senior [DEM 01], créé par J. Demaria, est une librairie réactive pour le compilateur Scheme Bigloo. Senior a été créé avec l'objectif d'expérimenter le mélange de la programmation fonctionnelle et de Junior. En particulier, d'avoir accés à l'expressivité de Scheme (adoption d'une syntaxe moins lourde par rapport à Junior) et à l'ordre supérieur conforme à la sémantique de Junior.

Le résultat est une programmation très puissante dans laquelle :

- Les événements sont représentés par des symboles Scheme.
- Il y a un meilleur lien entre les données de Scheme et les programmes réactifs.
- On retrouve l'utilisation de l'ordre supérieur à tous les niveaux (implantation et utilisateur) ce qui perpétue la puissance et la souplesse du fonctionnel.
- Il est possible de créer des programmes réactifs récursifs (des fonctions récursives).
- L'existence d'une lambda réactive implémentée avec une fonction Scheme qui renvoie un programme réactif (arité variable, typage des paramètres).

Quelques problèmes restent sans solution, par exemple, les programmes réactifs ne sont pas réentrants et chaque invocation entraîne l'allocation d'un nouveau programme; le programme réactif ne peut donc être lié qu'à un seul objet.

3.4 Pistes

Dans cette section on va décrire trois expérimentations qui, bien qu'elles n'aient pas abouti à des solutions satisfaisantes, montrent les différents choix d'implémentation que peut faire le programmeur (qu'il soit développeur ou utilisateur du modèle): l'environnement et la QoS en Junior.

En Junior, l'adoption d'une nouvelle instruction se fait lorsqu'elle concilie le mieux possible la sémantique, la performance et l'expressivité. Ces trois facteurs sont considérés dans les instructions réactives que l'on présente dans les sous-sections suivantes.

3.4.1 D'autres façons de voir l'environnement

A l'heure actuelle, l'environnement d'exécution d'un programme réactif est constitué, principalement, par un ensemble d'identificateurs d'événements. Pour la plupart des applications, cet ensemble plat de noms est suffisant. Cependant pour d'autres applications (pour des raisons sémantiques ou de vitesse) l'ensemble plat n'est pas un bon modèle d'environnement.

Parmi les propositions destinées à améliorer la vitesse d'accès aux événements, on peut citer *l'environnement configurable* proposé par M. Serrano dans la construction de quelques applications graphiques avec les FairThreads en Scheme. L'idée est de voir l'environnement comme une structure qui prend le format des données utilisées; par exemple si on a une application qui utilise la souris pour générer des événements de la forme $(x, y)name$ (où x et y sont les cordonnées définies par le clic de la souris sur une fenêtre et $name$ est le nom du bouton de la souris qui a été enfoncé), il est préférable d'implémenter l'environnement comme un tableau de deux dimensions. La recherche de l'événement "$(x, y)name$" dans la table de hash est plus lente que si l'on cherche dans un tableau bidimensionnel en indexant le registre [x][y].
Cette proposition montre que l'implémentation actuelle de l'environnement en Junior est trop rigide car la notion d'identifier ne permet pas de faire de recherches adaptées au format de données. A l'heure actuelle, le programmeur doit créer sa propre version de Junior.

Maintenant on va décrire deux extensions sémantiques de la notion d'environnement.

1. Le pattern matching.

   L'idée du pattern matching est de pouvoir écouter plusieurs événements (identifiés par un patron) et de réagir à chaque événement qui satisfait le patron (dans le cas de l'instructions Scanner) ou au premier événement (dans le cas d'autres instructions événementielles comme Await). La description du patron dépend, en général, de l'application; néanmoins vu que l'on utilise très souvent des chaînes de caractères, le patron peut être donné sous la forme d'un langage formel. Les deux langages formels les plus utilisés pour décrire des chaînes de caractères sont les expressions régulières et les grammaires (voir [AHO 86]); les grammaires sont plus expressives que les expressions régulières mais sont plus lourdes à traiter. On a réalisé une expérimentation dans laquelle, pour simplifier l'implémentation, on utilise des expressions régulières pour décrire le patron des événements de type `String`.

   Pour simplifier la construction des identifiers, on a défini une méthode (`patternStr(String)`) qui les construit à partir d'une expression régulière. Grâce à cette méthode et aux modifications que l'on a faites en Junior (il faut modifier l'environnement pour que les instructions événementielles puissent chercher dans la table de hash un ou plusieurs événements qui satisfont le patron), on peut écrire les programmes suivants :

   ```
   Jr.Await( patternStr("^prefix!") );
   // Traitement du premier événement qui satisfait le patron

   Jr.Await(Jr.Or(Jr.Presence( patternStr("^prefix!"),
                               Jr.Presence( patternStr("suffix$");
   ```

   La description des expressions régulières est maintenant bien connue et même standardisée. Les expériences réalisées ont démarré avec la version 1.2 de Java qui ne dispose pas d'une librairie pour les utiliser (la version 1.4 offre cette fonctionnalité); on a donc utilisé la librairie regexp de GNU.

   Les expériences ont montré que l'efficacité de cette implémentation repose sur l'efficacité de la recherche dans la table de hash pour traiter un grand nombre d'événements. De même, l'extraction et la conversion des données transmises par les événements doivent être implémentées très efficacement.

2. Instructions événementielles paramétrables.

   L'objectif de ce type d'instructions est d'avoir un environnement constitué par des sous-ensembles d'événements. Le programmeur pourrait ensuite programmer des instructions événementielles qui n'auraient que la capacité de détecter les générations d'événements dans un des sous-ensembles. La création de sous-ensembles d'événements permet de rechercher les événements plus efficacement et de définir une portée des événements qui n'est pas hiérarchique comme celle de l'instruction Local. La portée hiérarchique pose des problèmes lorsqu'on utilise des événements auxiliaires pour contrôler un comportement réactif. Voici un exemple qui exécute le comportement `behavior()` un instant sur trois :

   ```
   Jre.Local("step",
             Jr.Par(Jr.Loop(Jr.Seq(Jre.Generate("step"),
                                   Jre.Repeat(3, Jr.Stop()) )),
                    Jre.Control("step",
                                behavior() )
                   )
            )
   ```

   Dans ce programme, on contrôle le comportement `behavior()` par l'événement `"step"`: en effet, `behavior()` n'est exécuté que lorsque l'événement `"step"` est généré. L'utilisation de l'instruction Local a comme effet de bord que `behavior()` ne peut plus recevoir l'événement `"step"` provenant de l'extérieur.

   En fait, ce que on cherche à faire en Junior, la plupart du temps, est de générer un événement qui n'est pas vu par d'autres comportements réactifs. Cet objectif peut être atteint d'autres façons, la question étant de savoir quelle est la plus efficace et la plus claire sémantiquement. Par exemple, en Java on peut utiliser un objet comme un événement qui ne peut être vu que par les comportements qui ont une référence sur lui. Cette affirmation est valide car Java garantit que tous les objets sont différents, autrement dit on utilise Java comme générateur d'événements uniques. Voici le programme précèdent dans lequel on utilise un objet comme événement (l'`ObjectIdentifier` est défini comme dans la section 2.3.1).

```
ObjectIdentifier eveObj = new ObjectIdentifier();

Jr.Par(Jr.Loop(Jr.Seq(Jre.Generate( eveObj ),
                      Jre.Repeat(3, Jr.Stop()) )),
       Jre.Control( eveObj,
                    behavior() )
      )
)
```

Les désavantages de cette solution sont doubles :

(a) Elle nécessite un générateur d'événements uniques qui n'existe pas dans tous les langages.

(b) La portée d'un événement n'apparait pas syntaxiquement et est difficile à déterminer.

Les instructions événementielles paramétrables permettraient de résoudre en partie ce problème de la façon suivante :

```
Environment E = new SubEnvironment();

Jr.Par(Jr.Loop(Jr.Seq(Jr.Generate(E, "step"),
                      Jr.Repeat(3, Jr.Stop())  )),
       Jr.Control(E, Jre.Presence(E, "step"),
                  behavior())
)
```

L'événement `"step"` n'est vu que par l'instruction Control ce qui implique que : 1) son corps `behavior()` peut recevoir ou émettre l'événement `"step"`, et 2) l'instruction Local n'est plus nécessaire. La portée de l'événement `"step"` est identifiée par la variable `E`.

Pour utiliser les sous-ensembles, il faudrait créer un nouveau type de configuration Presence (`Jr.Presence( environnement, identifier)`) pour chercher les événements dans un sous-ensemble donné. L'implémentation de la configuration Presence ne serait cependant pas suffisante et il faudrait aussi modifier le moteur réactif pour détecter la génération d'un événement dans un sous-ensemble et réagir à cet événement.
Les instructions événementielles paramétrables sont à l'étude et on n'a pas encore une implémentation et une formalisation de leur description. Cet étude fait partie des futures recherches à réaliser.

Ces deux propositions permettent de décrire des comportements réactifs que l'on ne peut pas programmer dans la version originale de Junior. Ces deux mécanismes sont complémentaires car le pattern matching ne peut pas être implémenté avec les instructions événementielles paramétrables et vice-versa. Le pattern matching permet d'attendre plusieurs événements en même temps tandis que les instructions événementielles paramétrables ne testent qu'un événement à la fois. Inversement, avec les instructions événementielles paramétrables on peut s'assurer que les événements générés dans un sous-ensemble ne seront pas écoutés pas d'autres comportements alors qu'avec le pattern matching tout le monde peut se mettre à écouter n'importe quel événement.

3.4.2 La QoS en Junior

Dans cette section on va décrire quelques expérimentations faites en Rewrite et Replace pour essayer de faire de la QoS (*Quality of Service*, Qualité de Service) en Junior. L'idée est de pouvoir garantir un certain nombre de propriétés, par exemple, que le système est capable de réagir à un événement en un certain temps. La technique générale (qui est largement utilisée dans les protocoles de communication des ordinateurs) consiste à limiter les ressources utilisées. Par exemple, on pourrait limiter le temps d'exécution de toute action atomique ou limiter le nombre d'instructions à exécuter.

Une première expérimentation faite par F. Boussinot [BOU 00b] a consisté à construire un automate qui implémente le programme réactif. Le problème de cette technique est que les automates ont tendance à exploser en taille dès que l'on programme un comportement "complexe". Pour limiter l'impact de ce problème, il est

proposé la construction d'automates partiels. C'est-à-dire que le programme n'est pas entièrement construit à la compilation et les parties manquantes sont compilées et exécutées à l'exécution lorsqu'elles sont sollicitées.

Mon idée consiste à modifier Junior pour limiter le temps d'exécution d'un atome et limiter le nombre d'instructions à exécuter; l'idée est de ne pas construire l'automate équivalent du programme réactif et d'utiliser la structure d'arbre de tout programme Junior. Voici la description de ces expérimentations.

Limitation du nombre d'instructions à exécuter

Limiter le nombre d'instructions à exécuter a comme objectif de limiter le temps de réaction. Le temps de réaction d'un programme réactif dépend : 1) du temps d'exécution de chaque type d'instruction, 2) du nombre d'instructions exécutées de chaque type et, 3) du temps d'exécution du code Java exécuté dans les atomes et dans les wrappers. Pour simplifier les choses, on va considérer que le temps d'exécution de chaque type d'instruction est le même et que le temps d'exécution des atomes et wrappers est nul. Avec ces deux considérations, on va estimer le temps d'exécution d'un programme comme la simple multiplication du nombre d'instructions par une constante.

Avec l'estimation du temps de réaction d'un programme et en connaissant le nombre de programmes exécutés dans la machine, on va pouvoir "garantir" que tout le système réagit en moins d'un temps donné. Pour éviter que le temps de réaction ne dépasse une certaine valeur, on fait deux choses : 1) on ne charge pas de nouveaux programmes réactifs, et 2) on exécute les programmes réactifs normalement jusqu'au moment où ils dépassent un certain nombre d'instructions, à ce moment là on les préempte.

Pour limiter le nombre d'instructions à exécuter, on a utilisé la structure d'arbre qui existe dans certaines des implémentations de Junior. L'idée consiste à modifier Junior pour avoir deux compteurs, un qui compte la hauteur de l'arbre et un autre qui compte le poids de l'arbre. Le coût du comptage est minime car de toute façon on visite les instructions lorsqu'on les active. Le seul problème avec cette implémentation est la façon dont on compte les instructions; en Simple, par exemple, il n'y a pas d'arbre d'exécution, et entre Rewrite et Replace on exécute un nombre différent d'instructions pour le même programme (Replace ne crée pas des instructions Seq dans les boucles). Cette différence entre Rewrite et Replace finit par être utile car, par exemple, elle montre comment Rewrite consomme plus de ressources que Replace. On n'a pas fait l'expérimentation en Simple ou Glouton mais l'implémentation des compteurs (malgré l'absence d'un arbre d'exécution) ne pose aucun problème.

L'avantage du mécanisme de comptage d'instructions à la volée, est que l'on ne perd pas du temps en construisant et en analysant l'automate du programme. D'autre part, il présente l'inconvénient de ne pas pouvoir prendre la décision d'exécuter un programme avant même de l'exécuter. Il y a deux façons de procéder:

1. Ne pas exécuter un programme réactif en fonction d'une estimation du nombre d'instructions à exécuter. Pour cela on peut utiliser le nombre d'instructions exécutées précédemment, le nombre maximal d'instructions que l'on pourrait exécuter et tout cela modulo un facteur de marge.
2. Arrêter l'exécution d'un programme réactif au cours de l'activation lorsqu'il dépasse l'utilisation des ressources allouées, par exemple, la hauteur, le poids, etc.

La deuxième option présente le problème général de la préemption forte: on peut laisser le système dans un état instable. Les expérimentations faites ont consisté à mettre au point le comptage des instructions exécutées dans Rewrite et Replace. Ensuite, il suffit d'arrêter l'exécution d'un programme en utilisant une instruction limitant les ressources à utiliser; pour limiter les ressources on pourrait imaginer l'instruction suivante :

```
Jr.Limite(Characteristic, Value, Body, Handler)
```

où `Characteristic` définit le type de ressource à limiter (par exemple la hauteur maximale de l'arbre d'exécution), `Value` définit le nombre de ressource que l'on peut utiliser, `Body` est le programme réactif à limiter et `Handler` est le programme réactif exécuté lorsque le corps est interrompu car il a dépassé les limites.

Le programme suivant utilise cette instruction avec la constante `Jr.Height` qui limite la hauteur de l'arbre :

```
Jr.Limite(Jr.Height, 2,
          Jr.Seq(Jr.Print("QoS"),
                 Jr.Loop(Jr.Stop()),
          Jr.Print("QoS Préemption")
         )
```

Le programme exécute son corps jusqu'au point où il dépasse la hauteur permise (2), c'est-à-dire il exécute les instructions Seq et Loop, et lorsqu'il va commencer l'exécution de l'instruction Stop, le programme est préempté et le *handler* est exécuté (on imprime `QoS Préemption`).

Les atomes bornés en temps

Une autre expérimentation qui a été faite est l'implémentation des *actions atomiques bornées en temps*. L'idée est de pouvoir stopper les programmes qui, pour une raison quelconque, ne finissent pas ou prennent beaucoup de temps. Limiter le temps d'exécution d'une action atomique a deux objectifs:

1. Garantir l'évolution globale du système. Voici un programme réactif qui arrête l'exécution de tout le système :

   ```
   Jr.Atom{  for(;;); }
   ```

   Dans cet exemple on a simplifié la syntaxe réelle des actions atomiques (celle utilisée dans la section 2.3.5) pour faciliter la lecture. Dans la suite on utilisera la syntaxe simplifiée.

   Limiter le temps d'exécution de ces programmes, empêcherait la propagation de ce genre d'erreurs aux autres comportements réactifs exécutés en parallèle.

2. Garantir le fonctionnement du système dans des conditions acceptables, autrement dit, faire d'une autre façon de la QoS. Par exemple, si on limite le temps d'exécution d'un programme, on peut assurer que le système ne s'écroulera pas lorsqu'on ajoutera un nouveau comportement. Si c'est le cas, on peut refuser l'ajout.

Les expérimentations faites ont montré qu'un mécanisme efficace de threads préemptifs est nécessaire car ce que l'on veut faire est de stopper le thread qui exécute le programme réactif et puis reprendre l'exécution dans une instruction réactive "sûre". Tout le problème consiste à reprendre l'exécution à partir d'une instruction réactive désignée. Par exemple, pour le programme précédent, il faudrait créer deux threads, un thread principal qui exécute les instructions réactives Loop et Seq, et un thread auxiliaire qui exécute l'action atomique. Le thread principal se suspend lorsqu'il exécute l'atome et, à ce moment, il passe la main au thread auxiliaire qui exécute le code Java. Si le thread auxiliaire dépasse le temps alloué, on le tue et on réveille le thread principal avec un code d'erreur. Si le thread auxiliaire finit normalement, on redonne la main au thread principal avec un code de succès.

Le problème de cette implémentation en Java est la création d'un nouveau thread à chaque fois que l'on exécute un atome. La création étant assez lourde, il vaut mieux le suspendre lorsque l'atome finit normalement et le relancer à la prochaine exécution d'un atome. Malheureusement l'opération pour suspendre un thread Java a été éliminée (dans le jargon de Java, on dit *deprecated*) et il faut donc implémenter un mécanisme à soi pour le suspendre de façon propre. Mais ce n'est pas tout le problème :

1. Il ne faut pas implémenter la suspension avec une attente active car sinon on peut consommer des ressources inutilement. On doit utiliser `sleep` ou les primitives `wait` et `notify`.

2. Il faut gérer correctement les différents threads créés: le thread principal, le thread auxiliaire et le thread qui implémente le timer. En particulier il faut faire attention aux priorités.

En conclusion j'ai eu du mal à mettre au point une version qui marche bien et qui passe à l'échelle.

Pour réduire la création de threads et avoir un contrôle plus fin sur le temps d'exécution d'un programme, on pourrait créer une instruction qui préempte un comportement réactif et pas seulement une action atomique.

Appelons cette instruction `preemption`, que l'on utilise de la façon suivante :

```
Jr.Preemption(Nombre d'instructions, Body, Handler)
```

Voici un exemple de cette instruction :

```
Jr.Preemption(30,
              Jr.Loop(J.Seq(Jr.Atom{ for(;;); },
                            Jr.Stop() ) ),
              Jr.Atom{ System.out.println("Timer elapsed"); }
             )
```

Ce programme préempte l'action atomique (constituée par l'instruction `for`) lorsqu'elle prend plus de 30 secondes en s'exécutant. Lorsque la préemption est effective, le handler est exécuté. Le problème de cette instruction est la sémantique que l'on donne à un programme qui exécute une instruction événementielle, par exemple l'instruction Await du programme suivant:

```
Jr.Par(Jr.Preemption(30,
                     Jr.Seq(Jre.Await("E"),
                            Jr.Atom{ System.out.println("E est présent"); },
       Jr.Seq(Jre.Repeat(25, Jr.Stop()),
              Jre.Generate("E"))
      )
```

L'instruction Wait peut prendre beaucoup de temps à finir, même lorsque l'événement est généré durant la première activation; par exemple, lorsqu'on utilise Replace et que l'on tombe sur un programme de type cascade inverse. Ensuite, il faut voir comment interpréter le temps que prend l'instruction lorsque l'événement n'est pas présent durant instant; il faudrait également ne pas considérer le temps d'exécution inter-instant.

Finalement, il faut dire que l'implémentation des actions atomiques bornées en temps doit être accompagnée par une instruction qui permet de détecter lorsque l'atome dépasse le temps. Les tests que l'on a faits ont montré que, par exemple, la création d'objets peut prendre beaucoup de temps et que, lorsque cela arrive, il faut souvent arrêter tout le comportement et gérer le problème au cas par cas. Autrement dit, il faut un mécanisme de contrôle d'erreurs; la section suivante présente une proposition pour résoudre ce problème.

3.4.3 Les exceptions

Lorsqu'on exécute un programme réactif il est possible de tomber sur deux types d'erreurs: 1) les erreurs qui résultent de l'exécution du code Java qui se trouve dans les actions atomiques et les wrappers, et 2)les erreurs qui concernent l'exécution des instructions réactives, par exemple l'exécution d'une loop instantanée. A l'heure actuelle, il n'existe aucune instruction qui permet de gérer ces erreurs et lorsqu'on exécute, par exemple, un atome qui génère une exception, c'est tout le système qui s'arrête. Pour résoudre ce problème avec un mécanisme d'exceptions à la Java, il faut voir :

1. de quelle façon on va implémenter les exceptions. Le plus facile est d'utiliser le mécanisme d'exceptions de Java mais ce choix implique que des futures implémentations de Rejo dans d'autres langages poseraient des problèmes.

2. quelle sémantique on va donner. En Reactive-C il existe déjà un mécanisme d'exception, la question est de voir si ce mécanisme s'adapte bien en Junior et sinon comment l'adapter. Si on implémente les exceptions en utilisant le mécanisme d'exception de Java, il faudrait voir quelles sont les répercussions sémantiques.

3. étudier les répercussions des propositions. Le problème de la gestion des exceptions est le fait que l'on peut implémenter une préemption forte qui peut laisser le système dans un état incohérent (certaines branches ne sont pas exécutées). Il faudrait voir dans quelles conditions la péemption forte, provoquée par une exception, est sans danger.

La proposition que l'on a faite en Junior pour traiter les exceptions consiste en une instructions réactive qui a la syntaxe suivante:

```
Jr.Try(Body, Handler)
```

L'instruction se comporte comme son corps `Body` tant qu'il n'existe pas d'erreur (aucune exception est générée). Si lors de l'exécution du corps une exception est levée, on arrête l'exécution et on exécute le `Handler`.

La formalisation de l'instruction Try, comme celle de l'instruction If et celle de l'instruction Scanner, pose quelques problèmes; la sémantique des ces instructions comporte la description du code Java que, jusqu'à maintenant, on n'a pas formalisé. En particulier, la sémantique de Junior ne décrit pas ce qui se passe lorsqu'on exécute un wrapper ou une action atomique. Les modifications de l'environnement Java (par opposition à l'environnement réactif que l'on dénoté par `E` dans les règles) sont considérées comme des effets de bord qui n'affectent pas le modèle réactif.

La sémantique formelle de l'instruction Try devrait modéliser le mécanisme d'exception de Java; ce mécanisme, qui affecte l'environnement Java, ne peut pas être représenté par les règles de réécriture de Junior; néanmoins si on introduit dans ces règles la notion de post-condition, la sémantique de l'instruction Try aurait la forme suivante :

$$\frac{t, E \xrightarrow{\alpha} t', E' \quad \alpha \neq TERM \quad exception() = false}{try(t, u), E \xrightarrow{\alpha} try(t', u), E'}$$

$$\frac{t, E \xrightarrow{TERM} t', E' \quad exception() = false}{try(t, u), E \xrightarrow{TERM} Nothing(), E'}$$

$$\frac{t, E \xrightarrow{\alpha} t', E' \quad exception() = true \quad u, E' \xrightarrow{\alpha} u', E''}{try(t, u), E \xrightarrow{\alpha} u', E''}$$

ou $exception()$ est un prédicat que l'on utilise pour savoir si une exception a été levée; autrement dit, lorsque $exception()$ est vrai, on est en train d'exécuter la section `catch` d'une instruction Try en Java.

Cette proposition ne modifie pas le mécanisme général d'exécution d'un programme réactif comme c'est le cas de Reactive-C. Reactive-C définit un drapeau supplémentaire appelé EXC qui est remonté dans l'arbre. Ce drapeau est généré par une instruction réactive spéciale, appelée `raise`, qui joue le rôle de l'instruction throw de Java. On peut voir le drapeau EXC comme la modélisation des exceptions Java à l'intérieur du réactif. Cette modélisation influence les autres instructions réactives, en particulier l'instruction Merge qui doit prendre en compte le nouveau drapeau. L'instruction Merge est définie de telle sorte que lorsqu'elle reçoit le drapeau EXC d'une de ses branches, elle arrête l'exécution des branches restantes.

L'implémentation de l'instruction Try se comporte comme la description que l'on vient de faire sans l'introduction du drapeau EXC et la définition de l'instruction raise. L'idée est de faire le moins de modifications possibles à Junior et d'utiliser tout ce qui est disponible en Java et dans le réactif pour ne pas dupliquer des fonctionnalités.

Voyons maintenant les problèmes de cette implémentation :

1. Il n'est pas possible de spécifier l'exception que l'on veut attraper. L'implémentation de l'instruction réactive Try a été codée en Java avec une instruction Try pour 1) ne pas reimplémenter un mécanisme d'exception et, 2) réutiliser les exceptions définies en Java. L'instruction Try en Java ne permet pas de déterminer à l'exécution le type d'exception que l'on veut intercepter, autrement dit, on ne peut pas déclarer (dans la section catch) le type d'exception à intercepter comme une variable. Or, c'est justement ce dont on a besoin; on voudrait écrire des programmes réactifs qui utilisent l'instruction réactive Try avec une exception particulière (par exemple l'exception `RuntimeException`) et avoir une seule implémentation de l'instruction réactive Try qui, lorsqu'on crée une instance, est capable d'utiliser une exception quelconque (celle que l'on a codée).

   L'implémentation actuelle de l'instruction réactive Try, intercepte toutes les exceptions générées par l'exécution du corps. Ensuite, c'est le programmeur qui doit gérer les exceptions; par exemple, il peut analyser la chaîne de l'exception pour savoir si l'on l'a traitée ou on l'a remontée. Voici un exemple:

```
Jr.Try(Jr.Atom{ throw new Exception("préemption forte"); },
       Jr.Atom{
                 Exception e = (Exception)env.getData("exception");
                 if( e.getmessage().equals("préemption forte")==0 )
                    System.out.println("Exception de préemption");
                 else
                    throw e;
               } )
```

2. Il est possible de coder un comportement qui arrête l'exécution d'un programme et qui le laisse dans un état instable, c'est-à-dire une sorte de préemption forte. Avec cette instruction on pourrait implémenter la configuration Or que l'on ne peut pas implémenter avec des instructions réactives. Voici un exemple :

```
Jr.Try(Jr.Par(Jr.Seq(Jre.Await("A"),
                     Jr.Atom{ throw new Exception("préemption: A"); } ),
              Jr.Seq(Jre.Await("B"),
                     Jr.Atom{ throw new Exception("préemption: B"); } ) ),
       Jr.Atom{ System.out.println("Exception: A || B"); } )
```

   En fait, le problème est lié au fait que l'on laisse une branche sans finir et que l'on donne la main à une autre. Une solution serait donc d'éviter de donner la main est de traiter l'exception avant, autrement dit, il faudrait limiter l'usage de l'instruction Try dans des comportements réactifs qui ne contiennent pas d'instructions Par. Cette limitation peut être réalisée soit par le compilateur soit par le programmeur. La version actuelle de Rejo ne fait aucune analyse pour limiter l'usage de l'instruction Try et c'est donc la responsabilité du programmeur.

3.5 Conclusions

On a présenté dans ce chapitre comment la sémantique formelle de Junior (vue dans le chapitre précédent) a été le point de départ de nombreuses implémentations de Junior mais aussi de ses variantes. La première implémentation de Junior, Rewrite, est une implémentation directe et naïve de la sémantique qui a tout de suite montré son défaut : une exécution extrêmement lente. Les travaux menés ces dernières années en Junior ont consisté à améliorer la vitesse d'exécution tout en conservant une sémantique formelle claire. L'évolution de Junior s'est déroulée par raffinements: détection des problèmes (création inutile d'objets, mauvaise gestion d'événements inter et intra instants, instructions binaires, etc), formalisation et implémentation.

Par ailleurs, on a analysé les instructions réactives de Junior pour mieux les comprendre et, dans certains cas, les améliorer. On a proposé quelques nouvelles instructions pour : 1) introduire plusieurs instructions de SugarCubes avec une sémantique formelle (l'instruction Dynapar), 2) avoir un meilleur mécanisme de modularité (l'instruction Run), 3) avoir un mécanisme pour traiter les erreurs (l'instruction Try). Quelques propositions sont encore à l'étude (celles pour faire de la QoS et la portée des événements).

En conclusion Junior présente les désavantages et les avantages suivants :

Désavantages

- Au niveau du langage :
  - Syntaxe difficile. La syntaxe de Junior est une syntaxe semblable à celle des langages fonctionnels : tout est fonction et il faut donc toujours écrire deux parenthèses, plus les virgules. Cette syntaxe fonctionnelle rend difficile l'utilisation, par exemple, des instructions de composition comme la Séquence et le Parallélisme.
  - Programmation de bas niveau. Malgré la programmation de Junior dans un langage de haut niveau, le langage Junior reste un langage de bas niveau dû, principalement, à la création et l'utilisation de wrappers et des actions atomiques.

  - Une sémantique compliquée pour quelques instructions. En particulier l'instruction Until et les cas où les instructions réagissent instantanément : préemption instantanée, loop instantanée, terminaison instantanée. La dé-synchronisation qui découle de la sémantique de quelques instructions est un autre problème.
  - Pas de programmation par objets. Junior est un langage impératif dont le seul mécanisme existant pour la programmation par objets est l'instruction Link. L'instruction Link modélise bien le mécanisme utilisé dans les langages à objets dans lesquels l'accès aux données est fait par un pointeur. Malheureusement, le programmeur est responsable de récupérer ce pointeur, de l'utiliser comme préfixe et de faire les casts nécessaires.
  - Deux types d'exécution. Lorsqu'on exécute un programme en Junior il faut savoir qu'il y a toujours une première exécution impérative classique en Java et puis l'exécution réactive. Cet ordre est très important car le comportement d'un objet réactif peut dépendre de quand il a été créé; en particulier certaines données ne sont disponibles qu'à l'exécution réactive.
  - Il existe certaines instructions qui n'ont pas encore de sémantique formelle. En particulier les instructions Scanner, Try-catch qui sont largement utilisées dans les Ricobjs (voir section 7.4.2) et les méthodes `currentValues` et `previousValues` de l'environnement.

- Violation des caractéristiques de l'approche réactive.
  Si pour une raison quelconque l'environnement évolue plus vite que le machine réactive (par exemple, en générant des événements à un débit très élevé) celle-ci n'est pas capable de déterminer la fin de l'instant. Junior ne dispose d'aucun moyen pour gérer ou tout simplement détecter cette situation.

- La vitesse d'exécution.
  Une des faiblesses du modèle réactif synchrone lorsqu'on veut l'appliquer dans la construction de systèmes temps réel est la vitesse d'exécution. Par rapport aux implémentations de l'approche synchrone, celles de l'approche réactive synchrone sont moins rapides. Un effort important a été réalisé ces dernières années pour réduire le temps d'exécution. Cet effort se reflète dans le nombre des implémentations de Junior qui ont été faites (au moins 7 algorithmes différents). Le problème de vitesse est lié à la façon dont on exécute les programmes réactifs (on les interprète) mais aussi à la mauvaise gestion des événements. Les dernières implémentations commencent à éliminer ces problèmes avec de meilleurs algorithmes de gestion d'événements à base de files d'attente et en réduisant le temps d'interprétation en utilisant des machines à *byte-code*.

- Les systèmes construits avec Junior sont de systèmes non-déterministes.
  Au contraire de SugarCubes, Junior offre un opérateur Par non déterministe. Le non déterminisme est un problème pour le débugage de programmes et pour la vérification de propriétés.

- Les systèmes réactifs synchrones doivent converger.
  La durée d'un instant doit converger pour garantir l'évolution du système. La convergence de ce type de systèmes peut être perdue par deux raisons : soit le programmeur ne construit pas un système qui converge, soit il réutilise du code qui, sans le savoir, ne rend pas la main. En général, Junior limite les cas où le programmeur peut bâtir des programmes réactifs qui ne convergent pas; cependant il reste le cas des boucles instantanées. L'utilisation de programmes qui ne rendent pas la main est aussi un problème présent dans les systèmes coopératifs décrits dans le point suivant.

- Les problèmes des systèmes coopératifs.
  Junior construit des systèmes coopératifs et ceux-ci sont une lame à double tranchant. Parmi les problèmes des systèmes coopératifs on trouve:

  - ils sont contraints de coopérer pour que le système évolue; dans le cas de Junior la violation de cette condition peut survenir dans les actions atomiques ou lors de l'évaluation d'un wrapper.
  - La portabilité et la réutilisation de programmes est fortement restreinte par la coopération. La coopération n'est pas toujours implémentée et elle n'est pas toujours implémentée de la même façon.
  - La coopération empêche l'utilisation de systèmes multiprocesseurs.

Les avantages des systèmes coopératifs sont décrits dans le point suivant.

Avantages

- Les avantages des systèmes coopératifs.
  Ils sont plus rapides que les systèmes préemptifs car ils ne font pas de changements de contexte inutiles. Le changement de contexte est une opération coûteuse faite par le noyau du système d'exploitation.

- Junior se présente comme une alternative à la programmation par threads.
  Les deux mécanismes les plus largement utilisés pour programmer la concurrence nécessaire dans les systèmes informatiques sont les processus et les threads; ces derniers présents en Java. Ces deux mécanismes présentent le problème du passage à l'échelle; par exemple en Linux le nombre de threads qu'un utilisateur peut créer est limité soit par le système (codé en dur) soit par le mécanisme de création de threads. Junior offre un opérateur de parallélisme sans threads qui permet une programmation concurrente fine et efficace, et surtout qui passe mieux à l'échelle que les threads.

- Sémantique formelle du langage.
  La sémantique formelle de Junior permet une compréhension facile et sans ambiguïté du langage. Cette description formelle du langage s'avère très utile lors de la programmation de comportements concurrents. La description formelle du langage est un autre atout du langage sur la programmation par threads de Java qui manque de sémantique.

  L'article [BOU 00a] fait une présentation de ces trois derniers avantages du modèle réactif.

- Dynamisme.
  Par rapport à d'autres langages, en particulier ceux de l'approche synchrone, Junior offre la possibilité d'ajouter dynamiquement de nouveaux programmes. Cette propriété permet de construire des systèmes que l'on ne peut pas construire avec l'approche synchrone où les problèmes de causalité surgissent lorsque l'on met en parallèle deux comportements réactifs.

Chapitre 4

Workbench

Lorsqu'existe un grand nombre d'implémentations d'un algorithme (ici il s'agit du modèle réactif) on est souvent amené à se demander quelle est la meilleure ou dans quelles conditions une implémentation est meilleure que les autres. Simple est une implémentation de Junior qui a longtemps été considérée comme la meilleure en termes de vitesse. Son problème principal est la complexité de l'algorithme utilisé (il est difficile de garantir la sémantique formelle). Plusieurs implémentations de Junior ont été créées avec l'objectif d'avoir des performances similaires à Simple tout en ayant une sémantique et une implémentation claire : SugarCubes, Glouton, Rvm et Jrc. Ces implémentations empruntent la notion de file d'attente de Simple et utilisent des nouveaux mécanismes (pas d'interprétation pour certaines instructions et exécution de bytecode) pour s'exécuter encore plus vite.

L'objectif de ce chapitre est de présenter les résultats obtenus à partir d'une plate-forme de tests, appelée *workbench*, que l'on a réalisée autour du modèle réactif pour vérifier le comportement des différentes implémentations et si possible les améliorer. Pour cela, il est impératif de : 1) bien construire le même programme; Vu que l'instruction Par est non-déterministe, on a été confronté au problème de ne pas tester le même programme dans la Rvm; la Rvm ajoute les comportements réactifs en sens inverse, 2) ne pas considérer les différences entre les langages utilisés (Java et C), et 3) considérer ou non le temps de construction ou de modification du programme réactif, etc.

Avant de commencer notre analyse, on voudrait rappeler qu'il existe plusieurs types d'analyses de programmes (temporelle, spatiale, complexité de l'implémentation) et que cette étude ne comporte qu'une analyse temporelle. L'analyse n'est pas complète car les différentes implémentations testées sont en évolution. De plus, au fur et à mesure que l'on faisait les tests et que l'on trouvait des "erreurs" d'implémentation, les différents programmeurs concernés ont créé des nouvelles versions de leur implémentation.

De mon côte, j'ai aussi fait quelques propositions :

- Une implémentation de Junior en C. Cette implémentation, appelée Cr, est née du besoin de :
  1. Appliquer le réactif dans la construction des systèmes où Java est mal adapté, par exemple, les systèmes embarqués.
  2. Avoir une version en C du langage Rejo (présenté dans les chapitres 5 et 6); ce langage s'appelle RAUL (présente dans la section 6.4) et il a besoin d'un moteur réactif en C (à la Junior).
- Des modifications à quelques instructions réactives pour avoir des meilleurs performances : 1) on a implémenté des instructions N-aires de composition (Par et Seq) en Cr, 2) on a éliminé la méthode `reset` en Cr-replace.

Tous les tests ont été réalisés sur une plate-forme qui a les caractéristiques suivantes :

PowerPC G4, 500Mhz, 512MB, Mac OS X ver 10.2.6.
Java version 1.4.1_01, Java HotSpot(TM) Client VM.

gcc (GCC) 3.1 20020420 (prerelease)
Copyright (C) 2002 Free Software Foundation, Inc.

Parmi les tests qui restent à faire, il faudrait utiliser différents :

1. types de plate-formes pour voir l'impact de l'architecture et du système d'exploitation (SE), par exemple des machines qui utilisent des microprocesseurs basés sur la technologie Intel avec des SEs comme Windows et Linux. Le SE de Mac OS X utilise un noyau de type UNIX ; en particulier, son architecture est celle qui a été définie dans les SEs BSD, par exemple OpenBSD, NetBSD et FreeBSD.

2. versions de Java (comme la 1.2, 1.3, et la 1.4) pour voir si les nouvelles versions n'introduisent pas d'*overhead*. La plupart de tests ont été réalisés avec la version 1.4 de Java pour Mac qui est sortie récemment (cette version est sortie plus tard en Mac qu'en Linux ou Windows à cause de problèmes d'efficacité).

3. mécanismes d'exécution de bytecode. Tous les tests que l'on va présenter ont été effectués avec une JVM qui utilise le JIT.

Voici la description des différents tests réalisés et les résultats obtenus. Les graphes que l'on va présenter représentent sur l'axe Y le temps, donné en secondes, et sur l'axe X la variable considérée. Les temps des programmes Java n'incluent pas le temps de démarrage de la JVM qui est beaucoup plus important que celui d'un programme en C.

Dans les tests que l'on va présenter dans la suite, on ne teste pas l'efficacité de toutes les instructions réactives car certaines instructions n'ont pas été définies dans tous les formalismes : 1) la Rvm ne définit pas les instructions Link et Freezable, 2) Glouton n'implémente pas l'instruction Freezable, l'instruction Control est une traduction et l'instruction Until est en réalité un Kill, 3) Loft n'implémente pas la plupart des instructions événementielles définies en Junior mais il en définit d'autres comme link et unlink, et 4) SugarCubes implémente un parallélisme déterministe, le Merge (on analyse ce cas dans la section 4.2). La section 4.7 présente un test qui illustre la différence en efficacité liée à l'expressivité du langage.

4.1 Le problème de la pile

La première idée que l'on a eue pour tester les implémentations de Junior a été de construire un programme en Rejo qui contient N instructions et pas un programme qui exécute en boucle M instructions ($M < N$) jusqu'à ce que l'on attend l'exécution des N instructions. Par exemple, pour tester le temps d'exécution de N instructions en séquence, on crée un programme Rejo qui définit une méthode réactive mettant en séquence les N instructions. Comme on va tester le même programme pour plusieurs valeurs de N, on crée un autre programme qui construit le premier et auquel on passe la valeur N comme paramètre.

Malheureusement, cette façon de faire a montré tout de suite ses défauts. En réalité, la première chose que l'on teste est la capacité du compilateur à compiler des gros programmes, et pas le moteur réactif lui-même. La figure 4.1 illustre les limites trouvées pour chaque compilateur lorsqu'on a compilé le programme de la cascade inverse (présenté dans la section 4.5). Ce graphe montre que la taille des programmes que l'on peut exécuter ne dépend pas de l'implémentation utilisée car, par exemple, avec Rejo toutes les implémentations s'arrêtent à 600 composants.

Pour résoudre ce problème il existe trois solutions :

1. Une solution consiste à lancer l'application, c'est-à-dire le compilateur, avec une pile plus grande. Ceci se fait différemment en Java (avec un paramètre) ou en C (une option dans le shell utilisé).

2. Si le compilateur n'utilise pas la pile du système pour parcourir l'arbre syntaxique, on peut créer un compilateur qui utilise une pile interne plus grande (parfois il suffit de passer la taille de la pile comme paramètre).

3. Une dernière solution consiste à créer un programme Rejo qui lui-même construit le programme entier à partir de bouts de code qui sont ensuite ajoutés dans la machine. De cette façon, on est sûr de tester le moteur réactif et pas le compilateur.

On a opté pour la troisième solution pour être le plus souple possible.

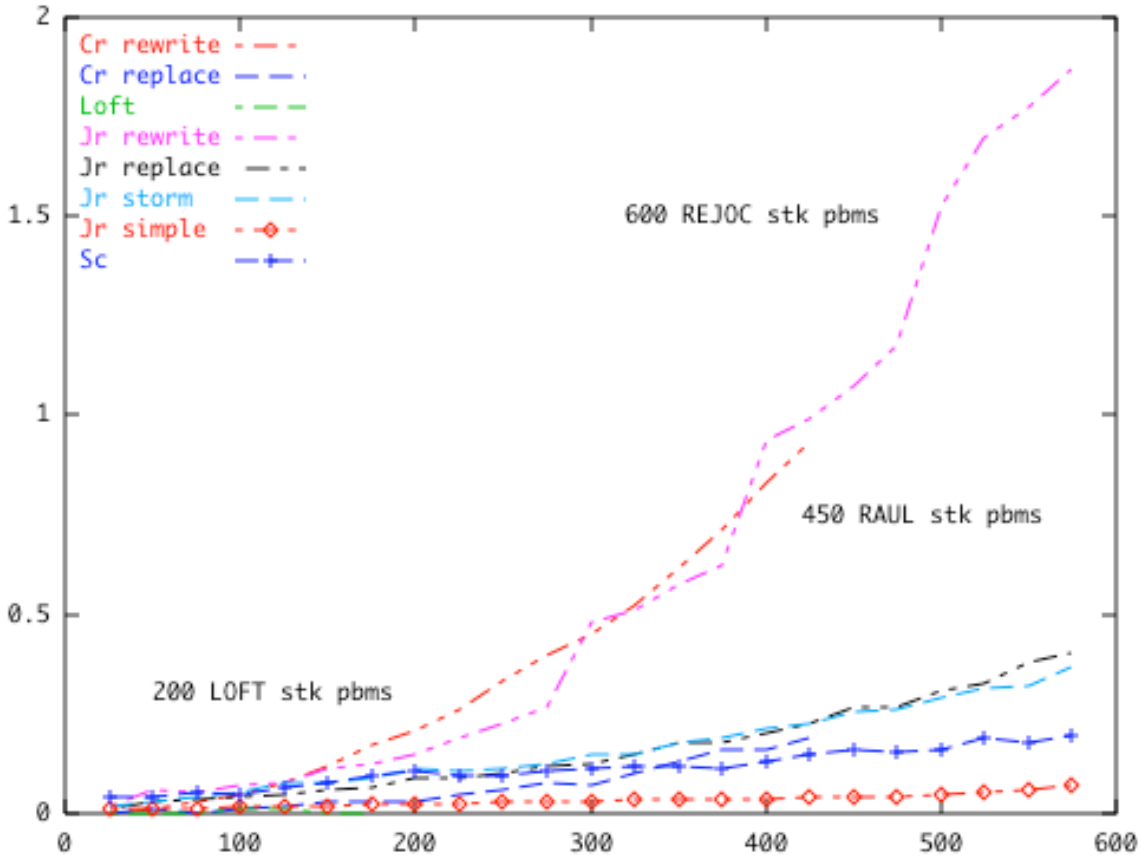

FIG. 4.1: Problème de pile des compilateurs.

4.2 L'instruction Par n-aire

Une fois réglé le problème de pile du compilateur, on a commencé les tests des implémentations avec des valeurs de l'ordre d'un million. Malheureusement, le problème de pile s'est présenté encore une fois dans presque toutes les premières implémentations de Junior : les appels récursifs qui implémentent les règles de réécriture font exploser la pile, surtout en Rewrite qui les implémente fidèlement.

On a contourné le problème de deux façons :

1. Balancer l'arbre. Etant donné que la méthode `add` de la machine réactive crée des arbres en peigne à partir des instructions Par, on a construit les programmes réactifs de telle sorte que l'on obtienne des arbres balancés. La question que pose cette stratégie est de savoir si on doit prendre en compte le temps de balancement, En effet, Storm et SugarCubes balancent l'arbre d'exécution tandis que Simple et Glouton modifient le programme pour l'exécuter. Dans les tests que l'on va présenter, on ne considère pas le temps utilisé pour construire un arbre balancé.
2. Utiliser des tableaux. La spécification des tableaux (Par et Seq) en Junior (V2.1) les définit comme une traduction en un arbre en peigne. C'est pour cette raison que Simple propose une instruction particulière (l'instruction `MultPar` qui n'appartient pas à l'interface Jr) pour les exécuter. La version originale de Glouton proposait aussi la même instruction mais la dernière version (celle que l'on a utilisée pour les tests) utilise des listes de façon transparente (le balancement n'a apporté aucune amélioration).

Pour essayer de casser le problème de la pile, on a implémenté une instruction Par n-aire en Cr-replace (voir 3.2.5). L'implémentation du Par n-aire utilise un algorithme à base de listes qui est extrêmement efficace en temps d'exécution et en mémoire utilisée.

Programme de test

Pour mesurer le temps d'exécution de l'instruction Par lorsqu'on a des branches qui finissent à des instants différents (cas où l'arbre en peigne est inefficace), on a construit un programme composé de N branches qui exécutent le programme : `repeat(M) stop;`. Les branches contiennent différentes valeurs de M données par la série : $N, N-1, N-2, 1, 2, 3, N-3, N-4, N-5, 4, 5, 6, ...$; la série s'arrête quand $N-i$ atteint la valeur $i+1$. Voici un exemple pour $N = 10$:

```
Jr.Par(Jr.Repeat(10,Jr.Stop()),Jr.Par(
       Jr.Repeat(9,Jr.Stop()),Jr.Par(
       Jr.Repeat(8,Jr.Stop()),Jr.Par(
       Jr.Repeat(1,Jr.Stop()),Jr.Par(
       Jr.Repeat(2,Jr.Stop()),Jr.Par(
       Jr.Repeat(3,Jr.Stop()),Jr.Par(
       Jr.Repeat(7,Jr.Stop()),Jr.Par(
       Jr.Repeat(6,Jr.Stop()),Jr.Par(
       Jr.Repeat(5,Jr.Stop()),Jr.Par(
       Jr.Repeat(4,Jr.Stop()),
       Jr.Repeat(5,Jr.Stop())
      ))))))))))
```

Résultats

La figure 4.2 (l'axe X représentant la valeur de N) montre quatre implémentations qui ont des problèmes de pile[1]. On voit que l'utilisation de tableaux (noté avec un T ; les listes sont notées avec un L et les arbres balancés avec un B) : 1) augmente la taille des programmes que l'on peut exécuter; les limites des implémentations lorsque les programmes ne sont pas balancés sont : 2025 en Jr-Simple, 4053 en Jr-Replace et 4233 en Cr-Replace, et 2) améliore la vitesse d'exécution.

La figure 4.3 montre le temps d'exécution de toutes les implémentations confondues ; on n'a dessiné que les implémentations les plus rapides de la figure 4.2. Ce test montre que, pour un programme sans événement,

[1] D'autres implémentations, comme Rewrite (en Java et C), ont le même problème mais on ne les a pas représentées ici.

l'implémentation la plus rapide en Java est Replace ; étant donné que toutes les branches sont toujours parcourues, l'algorithme de Simple qui gère des files est plus cher. Ce qui nous a surpris dans ce test est que Jr-replace soit plus rapide que Cr-replace ; pour l'instant on n'a aucune piste pour s'expliquer ce comportement. Un autre résultat observé est le comportement bizarre de Storm qui s'explique soit : 1) par le fait que Storm n'utilise en fait pas le même algorithme d'exécution que SugarCubes (contrairement à ce que l'on pensait), soit 2) par le fait que l'algorithme de balancement ne fonctionne pas correctement. Il faut dire que la version de SugarCubes avec laquelle on a commencé les tests présentait aussi des problèmes de mauvais balancement, alors que celle que l'on utilise maintenant résout ce problème. La version de Glouton avait aussi des problèmes de pile qui ont été résolus avec des listes ; la version actuelle est insensible au balancement.

Dans la suite, on ne va présenter que les résultats des implémentations les plus rapides, autrement dit pour Simple on utilisera des tableaux (on notera avec T), pour Replace en C des listes (on notera L) et pour la version en Java un arbre balancé (on notera B).

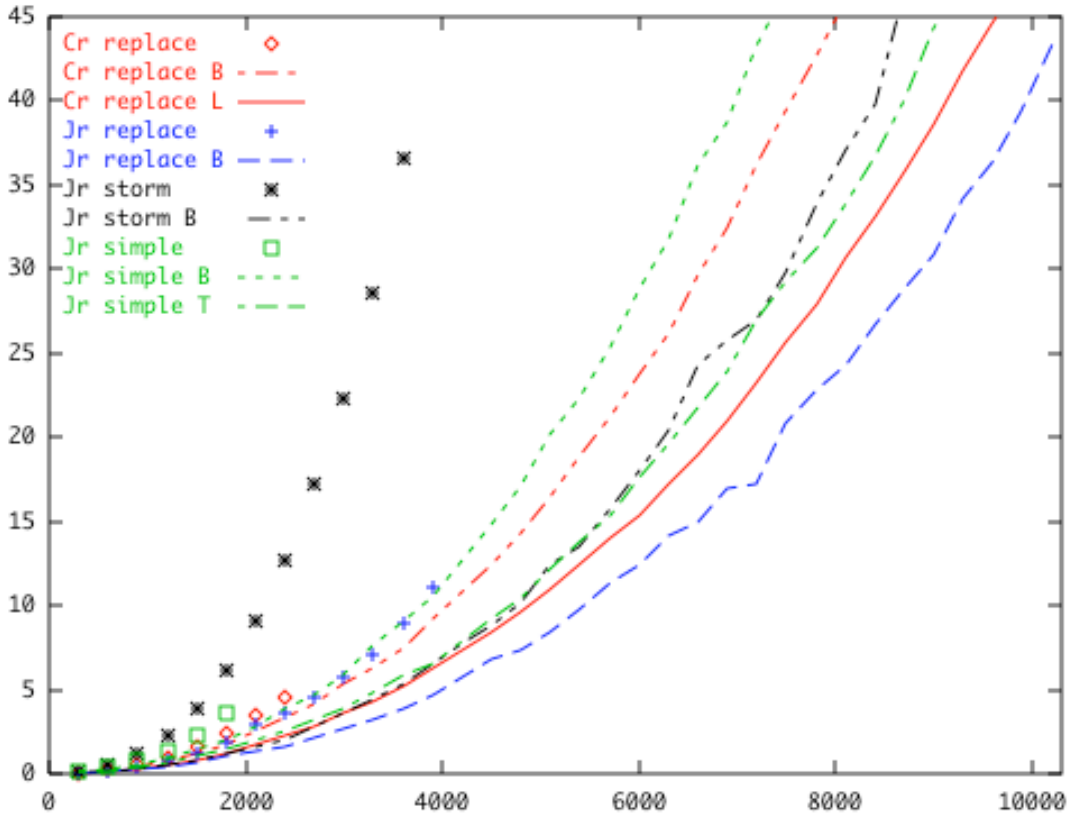

FIG. 4.2: Par terminant à des instants différents.

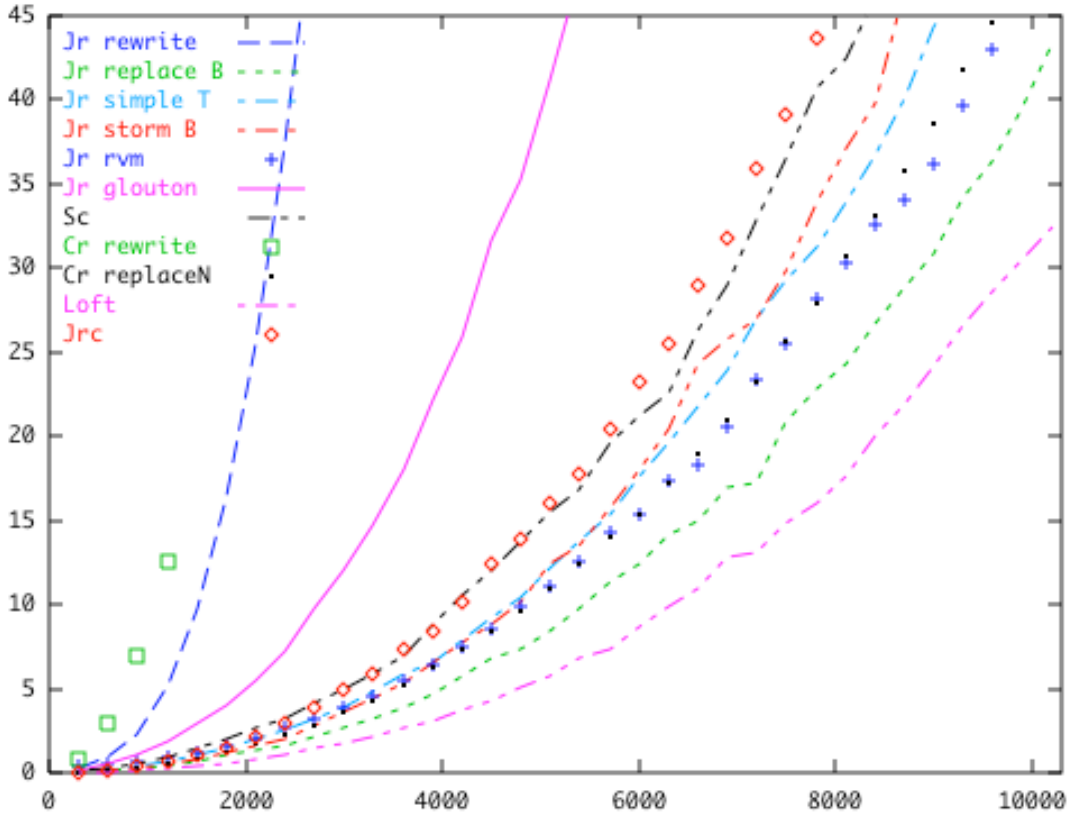

FIG. 4.3: La figure précédente avec toutes les implémentations.

4.3 L'instruction Seq n-aire

Junior propose deux instructions principales pour composer des programmes : l'instruction Par et l'instruction Seq. On a vu dans la section précédente que la performance du Par peut être substantiellement améliorée en utilisant des listes. Une autre façon d'implémenter le parallélisme, utilisée en Simple et Glouton, est l'utilisation de files d'attente. En réalité, dans ces implémentations l'instruction de parallélisme disparaît ; elle est transformée et exécutée par le moteur réactif qui déplace les bouts de code d'une file à une autre.
Autrement dit, en Simple et Glouton on a un gain considérable dans le parallélisme car non seulement on gagne en vitesse (on n'a plus d'arbre à parcourir) mais aussi en espace mémoire (il y a juste la gestion des files d'attente). Par contre, ce qui reste à améliorer est la séquence; en Simple la séquence est une instruction réactive binaire (la version qui utilise un tableau est dans un état expérimental) et en Glouton elle est transformée dans une notion plus générale qui est la notion de groupe. Il existe d'autres implémentations comme les FairThreads et les implémentations à base de bytecode qui n'ont pas de séquence ou qui l'implémentent très efficacement (à base de saut vers une adresse, `goto`).

Programme de test

Pour analyser le coût de la séquence dans les différentes implémentations, on a créé un programme qui met 1500 instructions Stop en séquence et qui les exécute N fois avec une boucle réactive. Voici l'exemple :

```
Jre.Repeat(N,
           Jr.Seq(Jr.Stop(),Jr.Seq(    // 1
                  Jr.Stop(),Jr.Seq(    // 2
                     ...
                  Jr.Stop()            // 1500
                 )) ... ))
```

Ce programme a été exécuté avec une version particulière de Cr dans laquelle on a fait deux modifications : 1) on a créé une implémentation n-aire de la séquence pour éviter le problème des arbres en peigne et leur balancement, et 2) on utilise un reset sans descente (comme Glouton). L'idée du reset en Replace et Storm (voir section 3.1) est d'éviter la copie du corps des boucles (Loop et Repeat) en utilisant une méthode `reset()` qui parcourt le programme (descente de l'arbre) en réinitialisant les instructions. L'idée du reset sans descente est que les instructions ne sont plus réinitialisées par la boucle qui les contrôle mais par elles-mêmes, lorsqu'elles finissent. Ce reset à la terminaison évite le parcours du programme mais nécessite des tests supplémentaires dans certaines instructions (Control, When, Run, etc.) pour détecter leur terminaison. On pense que le coût des tests est minime par rapport au gain que l'on obtient avec le reset sans descente ; il faudrait cependant mener une expérimentation (que l'on n'a pas eu le temps de faire) pour trouver à partir de quel moment les tests supplémentaires deviennent significatifs.

On a choisi la valeur 1500 un peu arbitrairement ; elle est en dessous de la valeur maximale de la plus grande séquence que l'on a pu construire avec les différentes implémentations : 1900 en Jr-simple, 2200 en Jr-rewrite, 2600 en Jr-glouton et 3400 en Jr-replace. SugarCubes n'a pas présenté de limite car il balance automatiquement l'arbre des séquences. Les implémentations qui utilisent des listes (Cr-replace, Loft et Glouton) n'ont pas présenté de limite à ce niveau.

Résultats

La figure 4.4 (l'axe X représentant N) montre les temps d'exécution des implémentations les plus lentes en Java (Rewrite, Replace et Simple) ainsi que les différentes versions de Cr-replace construites : avec des listes (noté avec un L) et avec la modification du reset (noté avec un R ; LR combine les deux améliorations) ; dans les tests que l'on va présenter dans les sections suivantes on utilisera un seul L pour dire que l'on utilise des instructions n-aires de séquence et de parallélisme.
La figure 4.5 montre toutes les implémentations sauf les quatre plus lentes. Les résultats montrés ne dépendent pas du balancement de l'arbre car on n'a qu'une seule branche.

Dans ce test, on voit que les implémentations qui utilisent une séquence binaire explosent, alors que les implémentations qui utilisent des listes ont des temps très proches et qu'elles ne sont battues que pour les implémentations qui n'interprètent pas de code comme Loft et la Rvm; l'écart n'est cependant pas très grand si on considère que l'on a exécuté 15 millions d'instructions (1500 fois 10000).

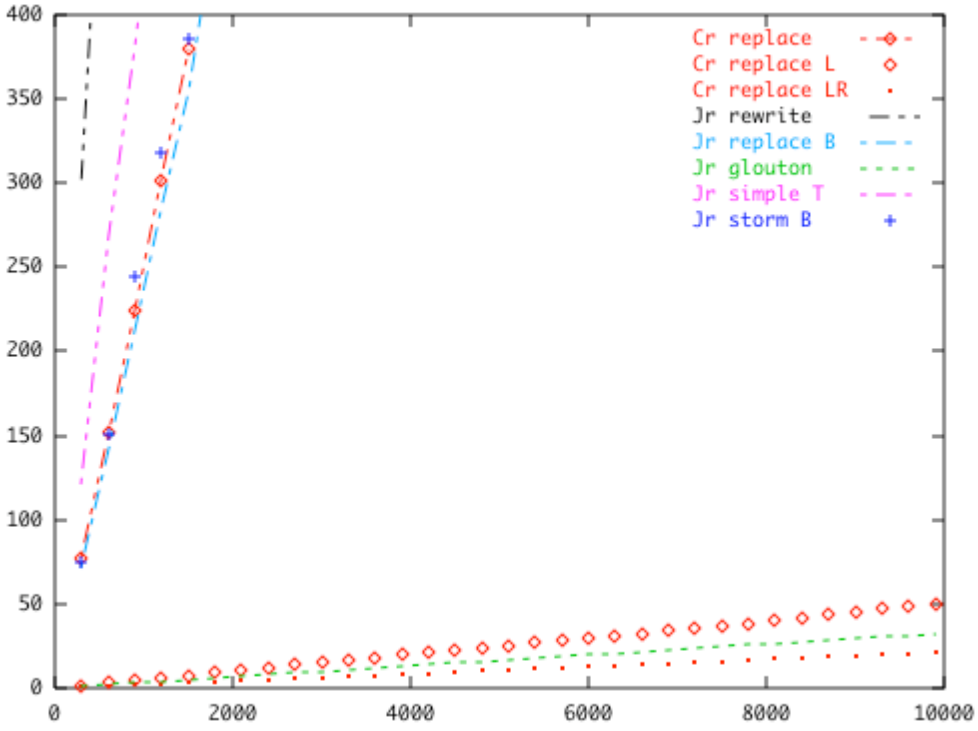

FIG. 4.4: Sequence d'instructions Stop.

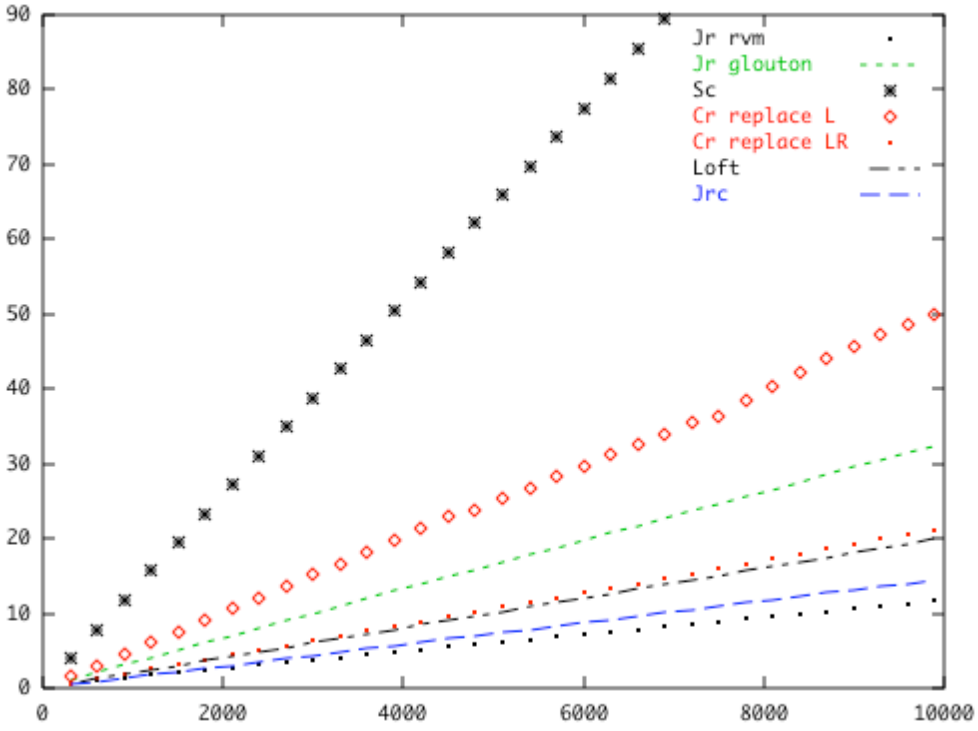

FIG. 4.5: Agrandissement de la figure précédente.

4.4 L'attente d'événements

Comme il a été dit dans la section 3.1, Simple a été créée avec l'objectif de traiter un grand nombre de composants en parallèle et un grand nombre d'événements. Pour exécuter efficacement un grand nombre d'événements, Simple utilise le même mécanisme que l'on trouve dans les systèmes d'exploitation, c'est-à-dire, l'attente passive[2]. Dans cette section on va présenter trois tests que l'on a conçus pour analyser les différents types d'attente que l'on a identifiés dans l'approche réactive.

Dans l'approche réactive, on peut parler de deux types d'attentes : l'attente inter-instant et l'attente intra-instant. L'attente intra-instant est générée par l'utilisation du drapeau SUSP ; ce drapeau sert à suspendre l'exécution d'une instruction, lorsqu'on n'a pas toute l'information pour l'exécuter, pour pouvoir activer d'autres branches qui éventuellement permettront de continuer. Le problème du drapeau SUSP est qu'il ne distingue pas les instructions suspendues qui doivent forcement être exécutées dans l'instant (comme l'instruction When) de celles que l'on pourrait laisser endormies (comme l'instruction Wait). L'instruction Wait devrait en effet attendre passivement entre les instants jusqu'à la satisfaction de sa configuration ; Wait pourrait d'ailleurs également attendre passivement durant le premier instant (comme le fait Storm, grâce au drapeau WAIT) ce qui serait utile dans certains cas comme, par exemple, la cascade inverse. L'attente intra-instant est analysée dans la section suivante.

L'idéal serait d'avoir une implémentation qui exécute une première fois les instructions événementielles pour les identifier, et puis de ne plus les ré-exécuter que lorsque toutes les conditions nécessaires à leur reprise sont réunies (durant l'instant de la première activation ou après). L'évolution des implémentations de Junior qui formalisent ces deux types d'attente a commencé par la formalisation de l'attente intra-instant (Storm) puis par celle de l'attente inter-instant (Glouton).

1er Programme de test

Pour déterminer qui implémente le plus efficacement l'attente inter-instant et voir l'écart existant avec les implémentations qui font de l'attente active, on a construit un programme qui met en parallèle 1000 branches qui attendent l'événement i (i variant de un à mille). Voici le programme utilisé :

```
Jr.Par(Jre.Await(1), Jr.Par(     // branch 1
       Jre.Await(2), Jr.Par(     // branch 2
       ...
       Jre.Await(1000)   ))      // branch 1000
      )
```

L'événement i n'est jamais généré et le programme est exécuté durant N instants.

Résultats

Les figures 4.6 et 4.10 montrent le temps d'exécution du programme (l'axe X représente N). Ces graphes corroborent que : 1) L'attente active inter-instant est implémentée par Rewrite, Replace et Storm ; Loft semble implémenter aussi une attente active mais de façon très efficace, 2) L'attente passive inter-instant est implémentée par Simple et Glouton.

2ème programme de test

Une autre façon de programmer l'attente en Junior est d'attendre l'absence d'un événement. Pour analyser le comportement d'un programme réactif qui réagit à l'absence d'un événement qui est généré à chaque instant, on a construit un programme similaire au précédent, mettant en parallèle 1000 branches qui attendent l'absence d'un événement i (i variant de un à mille). Voici le programme utilisé que l'on a exécuté pendant N instants :

```
int i=0;

Jr.Par(Jre.Repeat(4,
                  Jr.Seq(Jr.Atom{ i++; },Jr.Seq(
                         Jre.Generate(i),
                         Jr.Stop()))
```

[2]L'attente passive est souvent implémentée à l'aide de sémaphores qui gèrent l'utilisation d'une ressource à l'aide de files d'attente.

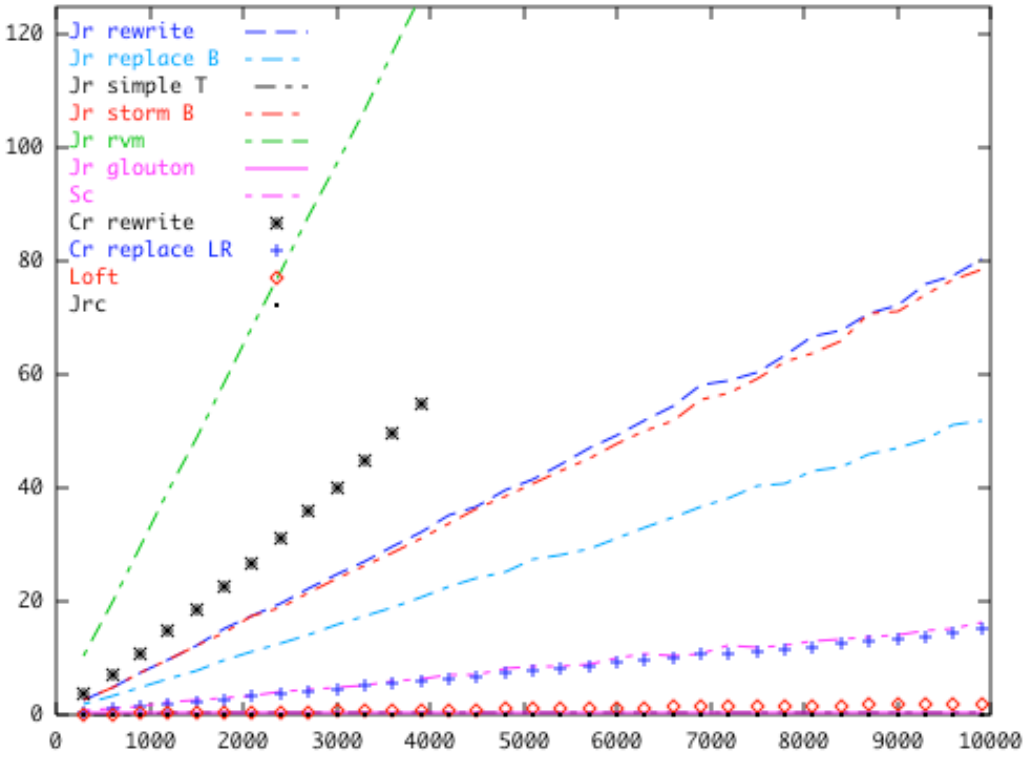

FIG. 4.6: L'attente inter-instant de la présence.

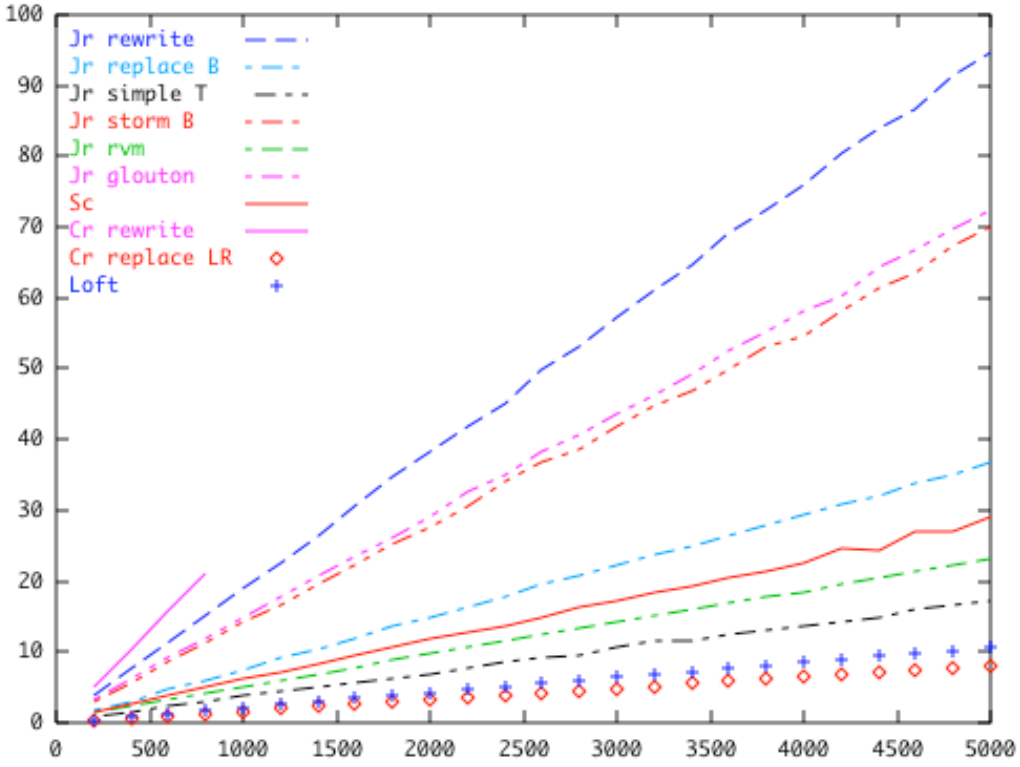

FIG. 4.7: L'attente inter-instant de l'absence.

```
            ),Jr.Par(
    Jr.Await(Jr.Not(Jre.Presence(1))), Jr.Par(     // branch 1
    Jr.Await(Jr.Not(Jre.Presence(2))), Jr.Par(     // branch 2
    ...
    Jr.Await(Jr.Not(Jre.Presence(1000)))  )))      // branch 1000
)
```

Résultats

La figure 4.7 montre le temps d'exécution du programme. Ce graphe montre que la réaction à l'absence est plus coûteuse que la réaction à la présence car : 1) Les implémentations comme Simple ou Glouton qui faisaient de l'attente passive font maintenant de l'attente active, 2) les implémentations qui faisaient de l'attente active ont vu leur temps augmenter : Jr-replace a augmenté d'approximativement 10 secondes lorsqu'on l'a exécuté 5000 instants (noter qu'il y a eu un changement d'échelle, de 1000 à 5000). Les implémentations qui parcourent toujours toutes les branches à la fin de l'instant (comme Loft et Cr-replace) ont les meilleurs temps d'exécution. Comme d'habitude, les implémentations de Rewrite (en C ou Java) sont plus lentes que celles de Replace. Le seul comportement bizarre que l'on a observé est le temps d'exécution de Glouton, beaucoup plus grand que Simple ; il apparaît que la gestion de files en absence d'événement est très chère. Storm a aussi un temps d'exécution très grand mais différent de celui de SugarCubes, ce qui nous fait penser à un mauvais balancement de l'arbre du programme.

3ème programme de test

Une des particularités de Junior est qu'il existe une instruction pour réagir à la présence d'un événement à chaque instant (instruction Control) mais qu'il n'existe pas d'instruction pour réagir à l'absence d'un événement à chaque instant (la réaction retardée d'un instant). Le programme suivant a été construit avec le but de voir le coût de la réaction à l'absence à chaque instant. Le programme attend l'absence d'un événement i qui est généré à chaque instant. Voici la partie du programme qui réagit à l'absence de i en exécutant Stop :

```
Jr.Par(Jr.Loop(Jre.When(i,
                        Jr.Stop(),
                        Jre.Generate(i+"abs"))),
       Jre.Control(i+"abs",
                   Jr.Loop(Jr.Stop())))
```

On a aussi construit un programme réactif qui réagit à la présence d'un événement en boucle ; le programme que l'on a utilisé est semblable au précédent mais sans la première branche du Par.

Résultats

Les figures 4.8, 4.9 et 4.11 montrent les résultats de l'exécution des programmes décrits précédemment pendant N instants (l'axe X). Ces graphes montrent que : 1) Il n'y a pas de différence entre l'attente d'un événement une seule fois (avec Await) ou N fois (avec Until), et 2) l'attente de l'absence d'un événement est plus chère lorsqu'on l'attend à chaque instant. La réaction à l'absence en boucle est plus chère pour deux raisons : 1) les moteurs réactifs ont été conçus (et optimisés) pour attendre la présence et non l'absence (résultat corroboré dans les tests précédents), et 2) la réaction à l'absence à chaque instant nécessite une structure particulière (et non une seule instruction) ce qui rend l'exécution plus lente. Dans les sections 4.7 et 4.8 on montrera d'autres cas qui renforcent cette affirmation.
En conclusion, on peut dire qu'il vaut mieux créer des systèmes réactifs réagissant à la présence d'événements plutôt qu'à leur absence.

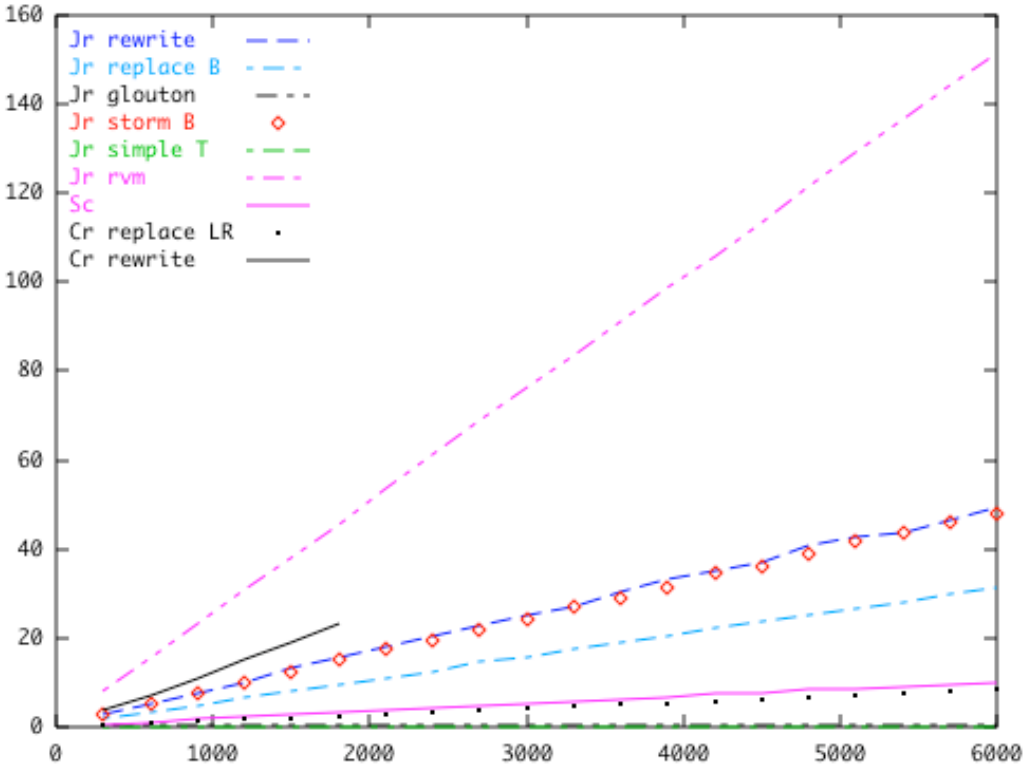

FIG. 4.8: L'attente inter-instant en boucle de la présence.

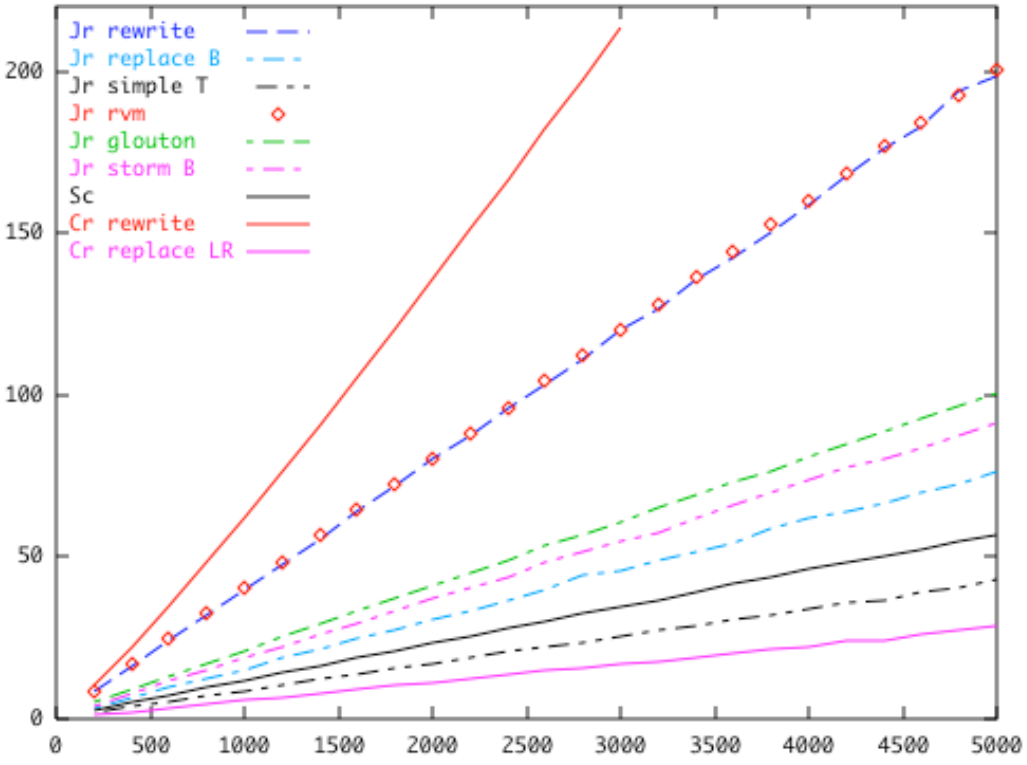

FIG. 4.9: L'attente inter-instant en boucle de l'absence.

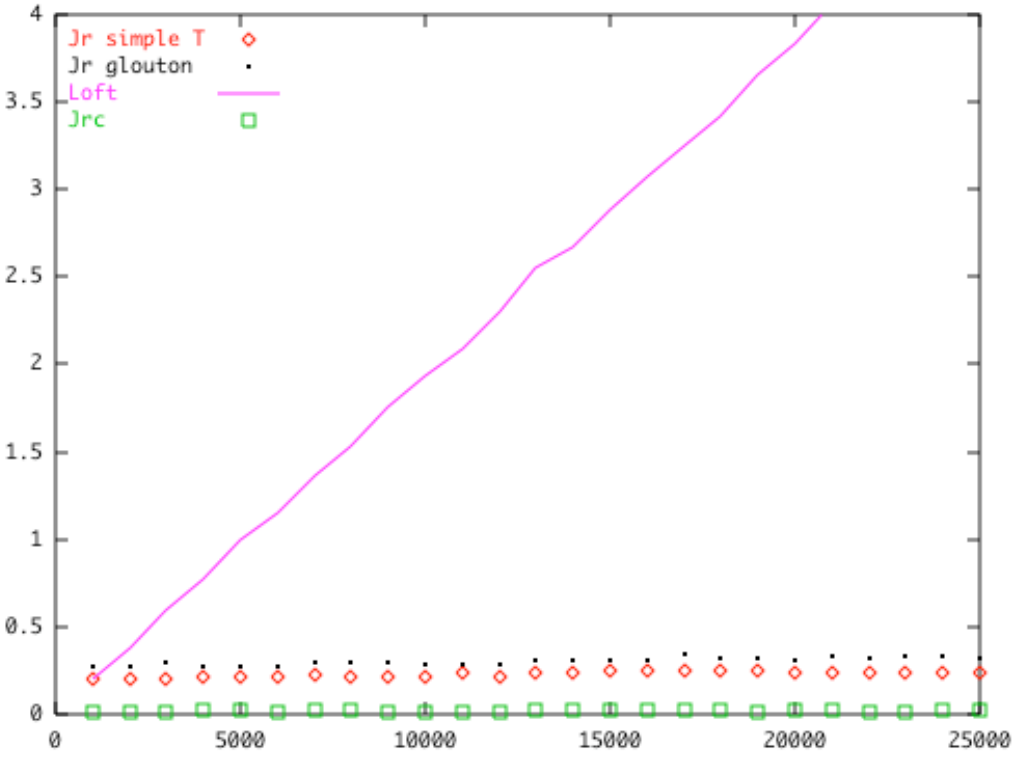

FIG. 4.10: Agrandissement de la figure 4.6.

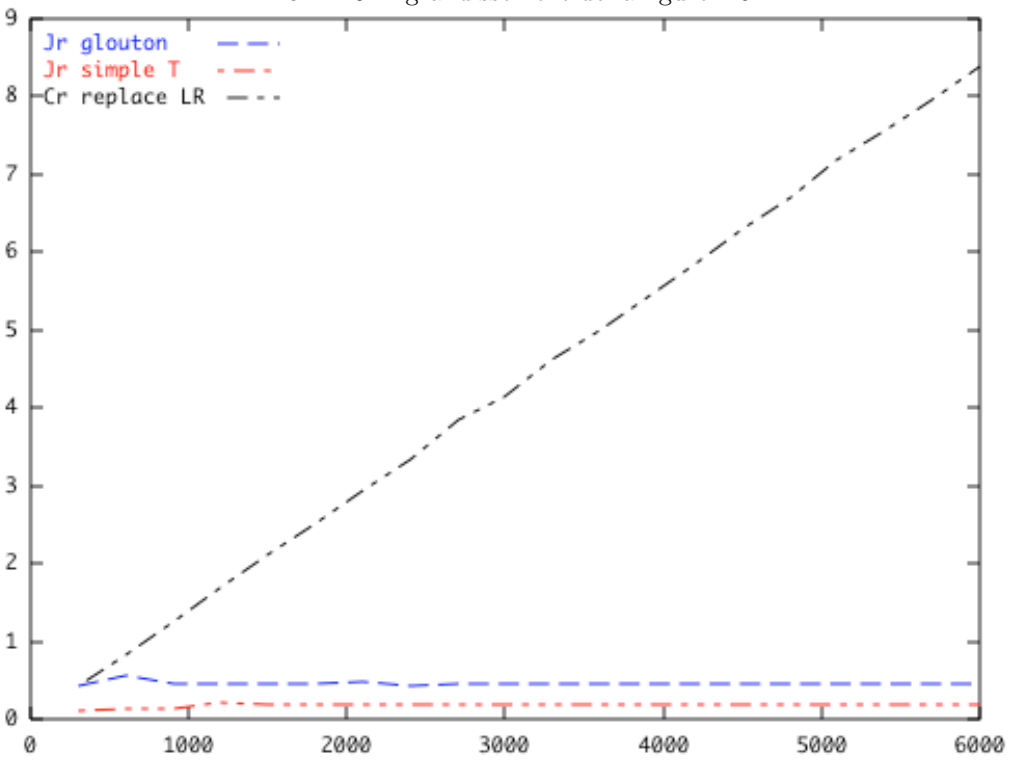

FIG. 4.11: Agrandissement de la figure 4.8.

4.5 Les cascades inverse et directe

La cascade inverse (problème que l'on a vu dans la section 3.1) est un programme conçu pour montrer l'impact du parcours inutile (durant un instant) de l'arbre d'exécution en Rewrite et Replace. Au contraire des exemples présentés dans la section précédente (attente inter-instant), dans la cascade inverse (attente intra-instant) tous les événements sont générés au cours du même instant et ce que l'on teste est la vitesse de propagation d'une réaction (la vitesse de convergence d'un instant).

Le test de la cascade inverse a toujours été utilisé pour mesurer l'efficacité des nouvelles implémentations ; cependant, je pense que la cascade inverse n'est qu'un type de programme parmi d'autres et que les systèmes réactifs que l'on construit "normalement" ont des structures diverses où la cascade inverse n'est qu'un cas extrême isolé. Baser l'efficacité d'une implémentation sur la cascade inverse serait une erreur. C'est pour cela que dans ce chapitre on a mené plusieurs tests et que dans cette section on va présenter également une cascade directe.

Programme de test

Voici les programmes que l'on a utilisé pour mesurer le temps d'exécution des cascades inverse et directe :

a) Cascade inverse

```
Jr.Seq(Jre.Generate(N),
       Jr.Par(Jre.Await(1),Jr.Par(
              Jr.Seq(Jre.Await(2), Jre.Generate(1)),Jr.Par(
              Jr.Seq(Jre.Await(3), Jre.Generate(2)),
              ...
              Jr.Seq(Jre.Await(N),Jre.Generate(N-1))
             ))...)
)
```

b) Cascade directe

```
Jr.Seq(Jre.Generate(1),
       Jr.Par(Jr.Seq(Jre.Await(1), Jre.Generate(2)),Jr.Par(
              Jr.Seq(Jre.Await(2), Jre.Generate(3)),Jr.Par(
              ...
              Jr.Seq(Jre.Await(N-1), Jre.Generate(N)),
              Jre.Await(N)
             ))...))
)
```

Résultats

Les figures 4.12 et 4.13 montrent le temps d'exécution de la cascade inverse et les figures 4.14 et 4.15 montrent les temps d'exécution de la cascade directe (l'axe des X représente N). Ces graphes corroborent que : 1) L'attente active intra-instant est implémentée par Rewrite (la pile explose tout suite, aussi bien en Jr qu'en Cr), Replace (Jr et Cr), Storm et Loft ; 2) L'attente passive intra-instant est implémentée par Simple, Glouton, Rvm[3], SC et Jrc [4].

En ce qui concerne la cascade directe on observe deux choses : 1) A l'exception de Simple, les implémentations les plus rapides dans la cascade inverse ne le sont plus dans la cascade directe, et 2) L'écart entre les implémentations n'est pas très "important" (écart linéaire) surtout si on le compare avec l'écart qui se présentait dans la cascade inverse (écart exponentiel).
Etant donné que dans la cascade directe on parcourt toutes les branches une seule fois, les implémentations les plus rapides (Loft et Cr-replace) sont celles qui perdent moins de temps en choisissant les instructions à exécuter ; Simple et Glouton dans ce cas passent en fait leur temps à déplacer les instructions événementielles entre les files.

[3]La version actuelle a des problèmes d'allocation mémoire qui empêchent l'exécution des grands programmes.
[4]Pendant le stage d'Olivier Parra, on a fait ce test pour voir comment se comportait son implémentation. Le résultat est très encourageant.

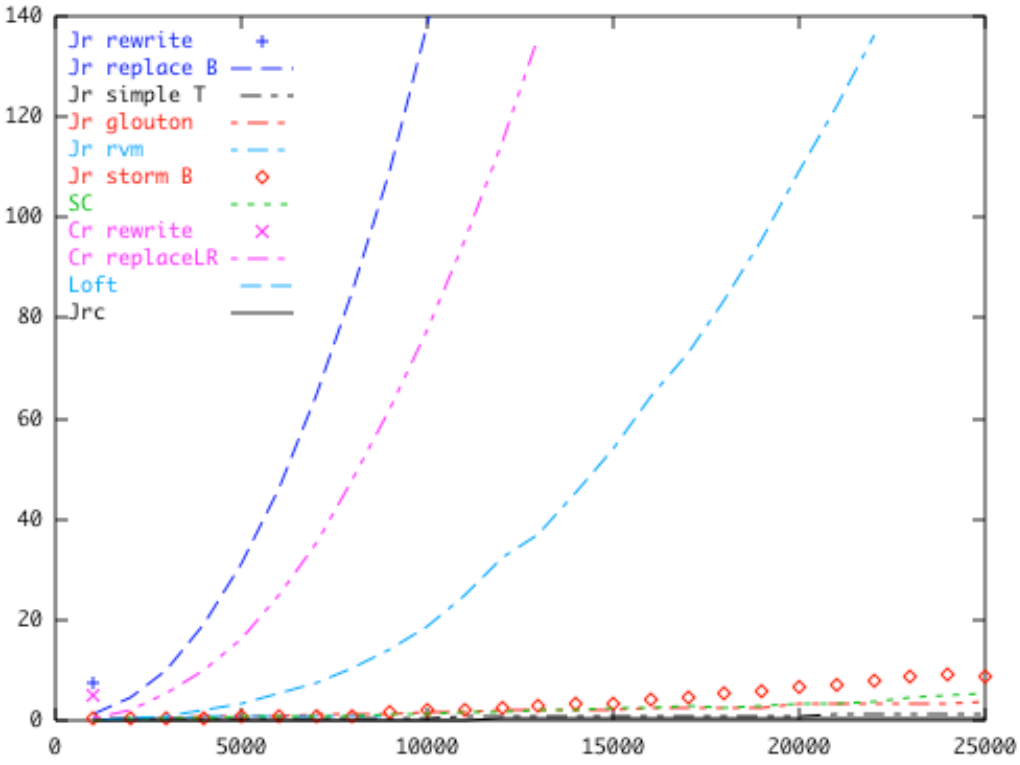

FIG. 4.12: Cascade inverse.

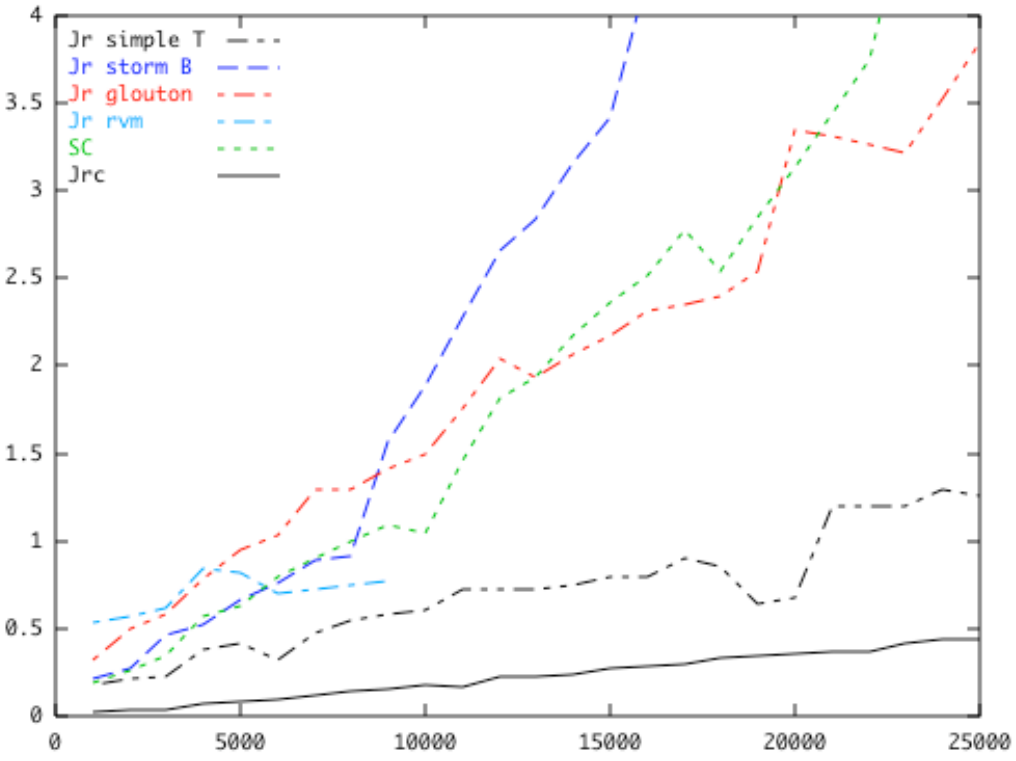

FIG. 4.13: Agrandissement de la figure précédente.

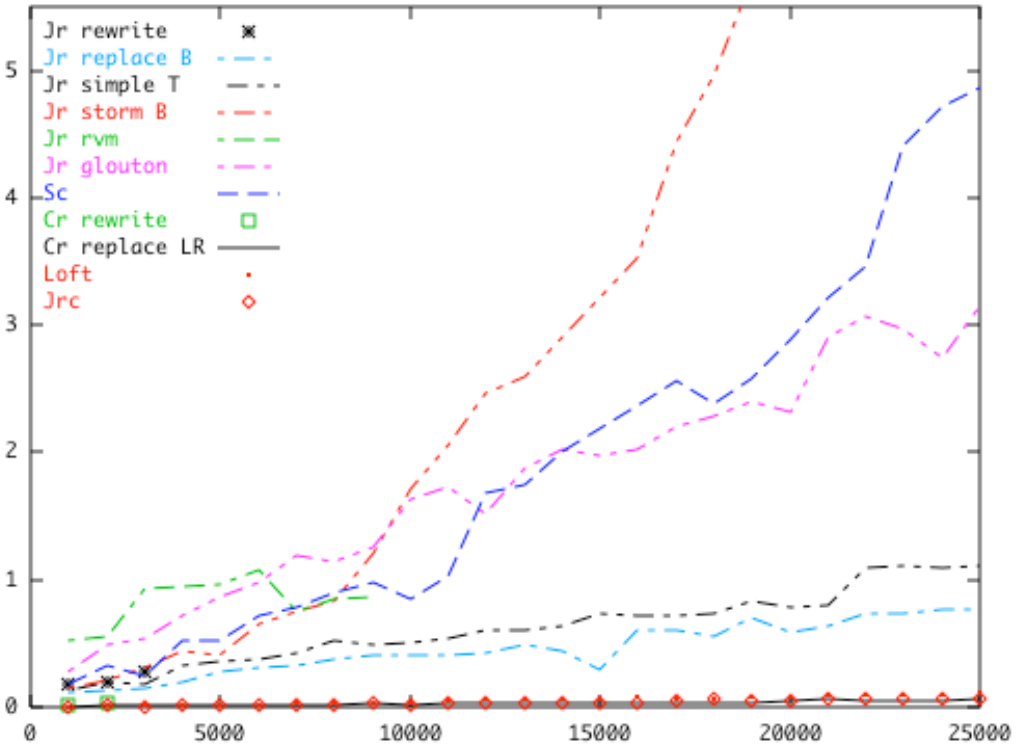

FIG. 4.14: Cascade directe.

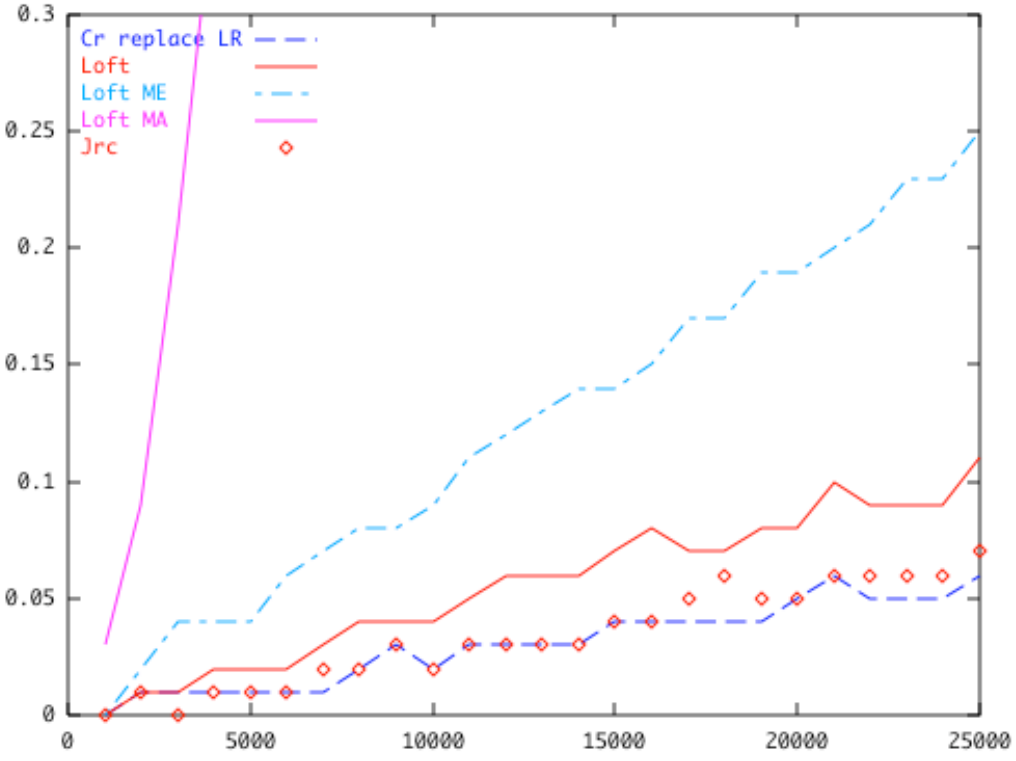

FIG. 4.15: Agrandissement de la figure précédente.

Un autre résultat important que l'on a pu observer dans la cascade directe (void figure 4.15), est l'impact de l'allocation mémoire en Loft ; le coût de la création d'un thread est très élevé (croissance exponentielle notée MA) et le coût de la création des événements aussi (linéaire mais comparable au coût d'exécution, noté ME). Ce comportement souligne que la gestion mémoire est un aspect important de la comparaison des implémentations en Java et C. Le GC de Java se déclenche périodiquement et son temps d'exécution est pris en compte par nos tests, surtout lorsque les tests durent longtemps. En loft et en Cr, la gestion de la mémoire est laissée (pour l'instant) au programmeur ; dans les tests que l'on a faits, on n'a jamais libéré la mémoire utilisée pour les événements. En Java au contraire, le temps mesuré prend en compte une partie de la gestion mémoire ; pour éliminer ce problème il faudrait désactiver le GC (test que l'on n'a pas fait) ou gérer correctement les événements.

4.6 Les cascades inverses avec Stops

On présente maintenant un test qui mélange les deux problèmes précédents, c'est-à-dire un programme qui teste à la fois l'attente inter-instant et l'attente intra-instant. Pour mélanger les deux aspects on a modifié le programme de la cascade inverse (on a ajouté une instruction Stop entre les instructions Await et Generate) pour la réaliser en N instants (N est le nombre de composants) au lieu d'un seul. On s'attend à voir trois comportements : 1) des temps d'exécution très grands pour les implémentations qui utilisent l'attente active, 2) des temps moyennement grands pour les implémentations qui utilisent l'attente passive intra-instant, et 3) des temps petits pour les implémentations qui utilisent l'attente passive dans tous les cas.

Programme de test

Voici le programme que l'on a utilisé pour mesurer le temps d'exécution :

```
Jr.Seq(Jre.Generate(N),
       Jr.Par(Jre.Await(1),Jr.Par(
              Jr.Seq(Jr.Seq(Jre.Await(2), Jr.Stop()), Jre.Generate(1)),Jr.Par(
              Jr.Seq(Jr.Seq(Jre.Await(3), Jr.Stop()), Jre.Generate(2)),
              ...
              Jr.Seq(Jr.Seq(Jre.Await(N), Jr.Stop()), Jre.Generate(N-1))
             ))...)
      )
```

Résultats

Les figures 4.16 et 4.17 montrent le temps d'exécution (l'axe des X représente la valeur de N). Ces graphes corroborent que Simple et Glouton sont les implémentations les plus rapides car elles font toujours de l'attente passive. Par contre, selon la classification que l'on a présentée dans le paragraphe précédent, on ne distingue pas les deux premiers groupes ; il apparaît que les implémentations qui utilisent l'attente passive intra-instant (Storm et SugarCubes) sont caractérisées par leur mauvais comportement (attente active) inter-instant. Seul Loft qui implémente très efficacement l'attente active inter-instant se situe entre les deux groupes précédents.

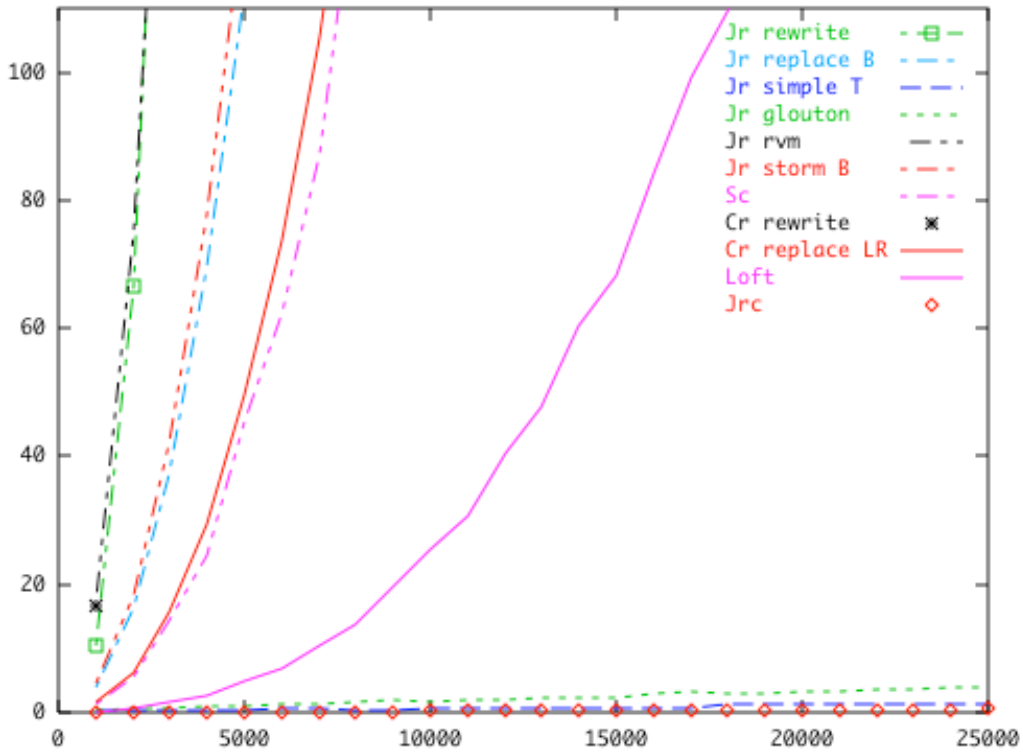

FIG. 4.16: Cascade inverse avec Stops.

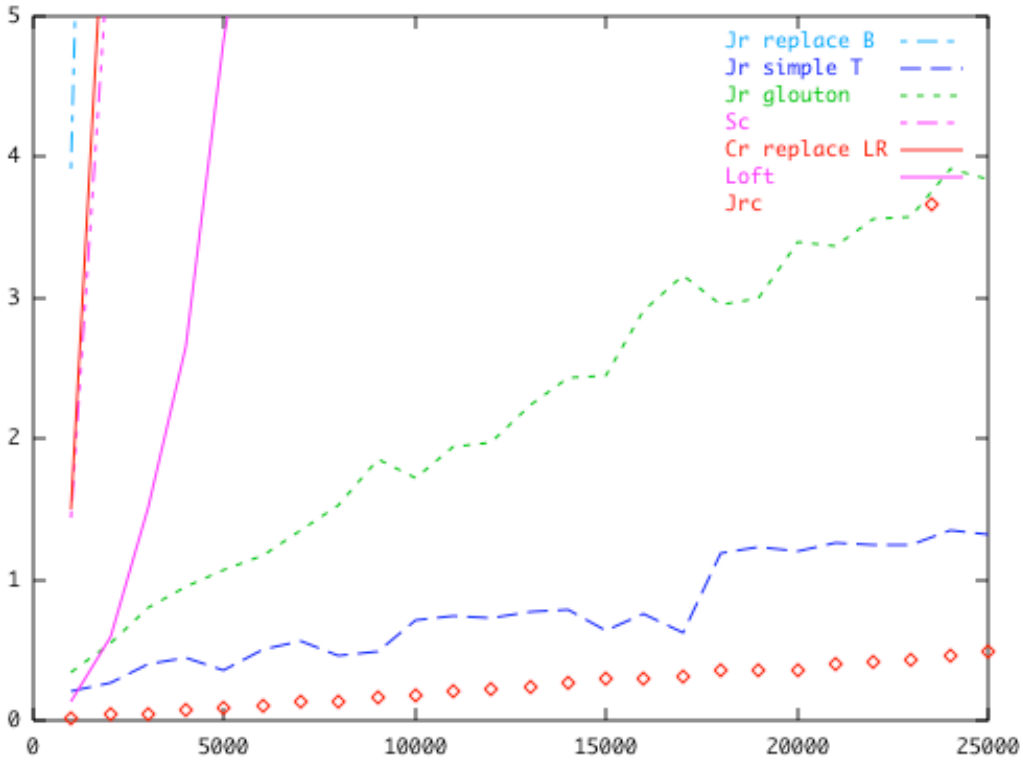

FIG. 4.17: Agrandissement de la figure précédente.

4.7 L'impact de l'expressivité du langage

Jusqu'à maintenant, Loft (implémentée avec les FairThreads) est une des implémentations qui est restée parmi les plus rapides. Si on se place du point de vue des architectures de microprocesseurs, les FairThreads correspondent à une architecture RISC, dans le sens où ils introduisent un ensemble d'instructions réduit. Cela signifie deux choses : 1) l'implémentation est plus rapide pour des programmes "simples" [5] car elle a moins de choses à faire (l'interprétation[6] des instructions est plus rapide), et 2) la programmation est en général de bas niveau. Par rapport à Junior, Loft n'a pas les instructions Control, Local et Freezable, ni les configurations ; d'autres instructions sont implémentées différemment : pas de Par imbriqué (seul le Par de plus haut niveau existe) et pas d'instruction Until.

Programme de test

Pour analyser le coût de la programmation en Loft d'un comportement réactif à la Junior, on a créé un programme qui implémente une instruction qui n'existe pas en Loft, l'instruction Until. Plus précisément, on a créé un programme qui exécute en parallèle N comportements `loop { action_atomique; stop;}` qui, lorsqu'un des comportements est préempté (avec un événement `i`), génère un événement (`i+1`) qui préempte en cascade directe tous les autres. On utilise la cascade directe pour réduire le temps lié à l'enchaînement d'événements.
Voici le programme en Junior :

```
Jr.Until(i ,
         Jr.Loop(Jr.Seq(Jre.Generate(0),
                        Jr.Stop())),
         Jre.Generate(i+1))
```

L'instruction Until n'existe pas en Loft mais elle peut être implémentée avec le mécanisme d'arrêt d'un thread ; le seul problème pour implémenter le programme en Loft est le `handler` de l'Until qui doit être codé avec un programme (mis en parallèle avec le Loop) qui attend la préemption pour générer l'événement `i+1`.

Résultats

La figure 4.18 (l'axe X représente N) montre le temps d'exécution du programme. On peut voir que le comportement de Loft se dégrade de façon non-linéaire par rapport aux graphes 4.14 et 4.15 (cascade directe) qui présentent un comportement similaire. L'autre particularité de ce test est le temps d'exécution de Glouton qui passe d'une croissance linéaire, dans le test de la cascade directe, à une croissance exponentielle. Ceci s'explique par la sémantique de l'instruction de préemption de Glouton (un Kill) ; le Kill attend la fin de l'instant pour réaliser la préemption, ce qui veut dire qu'au lieu de terminer en un seul instant, il faut N instants.

Par ailleurs, pour implémenter un nouveau comportement réactif qui n'est pas primitif (comme le `handler` en Loft), que ce soit en Loft ou en Junior, il faut : 1) créer un ou plusieurs nouveaux comportements parallèles qui surveillent la satisfaction d'une situation (la préemption dans notre exemple), et 2) utiliser un ou plusieurs événements auxiliaires (l'événement `i+1` dans notre exemple) pour influencer le comportement réactif principal.

[5] Programmes sans comportements Par imbriqués ou sans comportements événementiels "complexes" comme ceux définis en Junior.
[6] Dans un microprocesseur, on parle de cycle *fetch*.

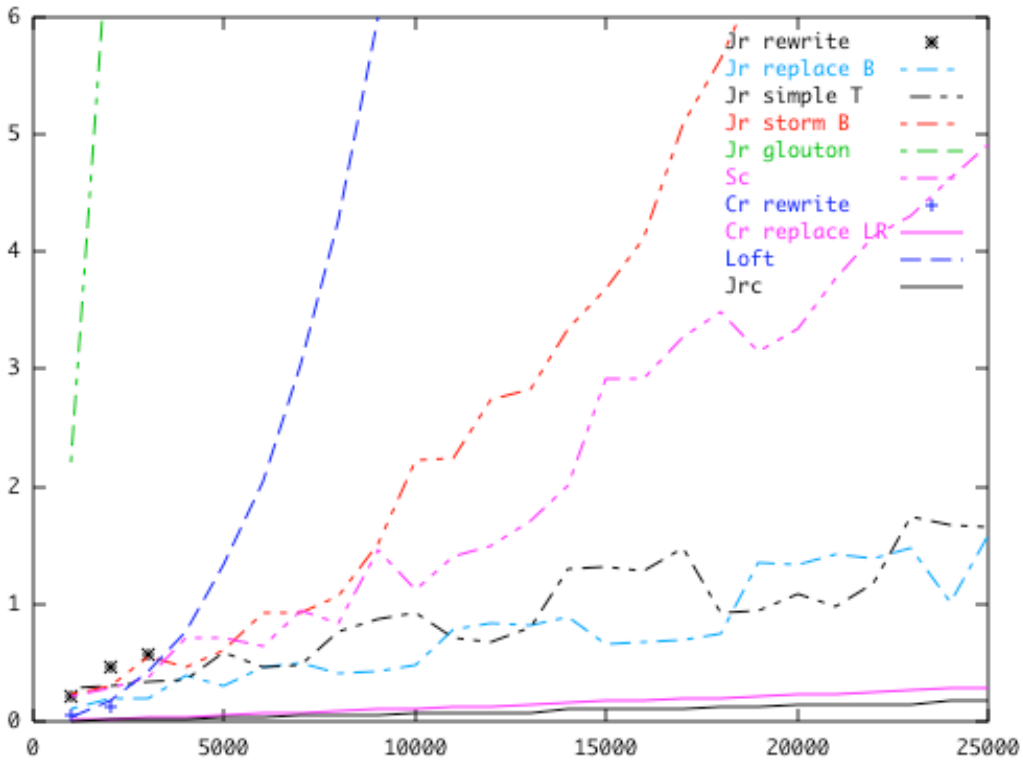

FIG. 4.18: Impact de l'expressivité.

4.8 Structure de contrôle

On finit cette série de tests par la mesure du temps d'exécution d'un programme réactif qui contrôle l'exécution d'un autre comportement réactif. Le contrôle utilise trois événements pour suspendre, résumer, et tuer le comportement réactif contrôlé. On a de plus un événement spécial pour ajouter des nouveaux comportements réactifs en parallèle avec le programme réactif contrôlé. On expliquera en détail le fonctionnement de ce programme dans la section 7.3.3.

L'objectif de ce test est de mesurer le temps d'un programme "réel" où il n'y a pas, en principe, de cas extrêmes comme ceux vus jusqu'à maintenant.

Programme de test

Voici le programme de contrôle qui utilise `Jre.Loop(Jr.Stop())` comme programme à contrôler. On a remplacé l'instruction Freezable par une instruction Until parce que l'instruction Freezable n'a pas été implémentée dans certaines versions et que l'instruction Until a un comportement similaire.

```
Jre.Until( name + "!kill",
  Jr.Par(
     Jr.Loop(Jr.Seq(
          Jre.Until( name + "!susp",
               Jr.Loop(Jr.Seq(
                    Jre.Generate(name + "step"),
                    Jr.Stop()))),
          Jre.Await( name + "!resume"))
     )
  ,
     Jre.Control( name +"step",
       Jr.Par(
          Jr.Loop(Jr.Seq(                        // Until au lieu
             Jr.Until( name + "!ctl",            // de Freezable
                Jre.Loop(Jr.Stop()))             // Corps contrôlé
             Jre.Generate( name + "!kill" ))
          )
        ,Jr.Par(
          Jr.Loop(Jr.Seq(
              Jre.Await(Jr.Or(name + "!add",
                        Jr.Or(name + "!ser",
                              name + "!migra"))),Jr.Seq(
              Jre.Generate( name + "!ctl"),
              Jr.Stop()))
          )
        ,Jr.Loop(Jr.Seq(
              Jre.Await( name + "!add"),
              Jr.Stop())
          )
        ))
     )
  )
)
```

FIG. 4.19: Programme qui implémente la structure de contrôle.

Le programme de test a été exécuté N instants pendant lesquels on a généré plusieurs fois les événements de contrôle pour suspendre et résumer le corps. Le corps a été tué à la fin.

Résultats

On a fait deux tests, le premier (voir figure 4.20 où l'axe X représente la valeur de N) qui teste la structure de contrôle avec toutes les implémentations pour voir laquelle l'exécute le plus efficacement, et le deuxième (voir figure 4.21) qui teste une instruction réactive (décrite dans la section 7.3.3) qui implémente le même comportement. Le but du deuxième test est de montrer que parfois il vaut mieux implémenter un comportement réactif en une seule instruction au lieu de plusieurs. L'implémentation en une seule instruction rend la sémantique plus claire et l'implémentation plus efficace (dans la figure 4.21 l'écart de temps augmente linéairement jusqu'à atteindre 40% dans l'exécution de 30000 instants).

La figure 4.20 montre que Simple, Glouton et SugarCubes (qui gèrent bien le parallélisme interne et les événements) sont parmi les implémentations les plus rapides. L'implémentation la plus rapide est Cr-replace-LR qui cependant a une croissance supérieure aux implémentations en Java. L'avantage de Cr-replace-LR est qu'il utilise toujours des listes (Par et Seq n-aire, même pour les instructions imbriquées) et que la structure de contrôle n'est pas un programme qui utilise les événements de contrôle de façon intensive.
Le seul cas bizarre est le temps d'exécution de Cr-rewrite qui explose, même par rapport à sa version en Java ; on n'a pas étudié ce phénomène.

On voudrait finir cette section en rappelant que les tests présentés ici, ne sont nullement complets ; il faudrait vérifier que les résultats sont les mêmes sur d'autres plate-formes, c'est-à-dire qu'ils sont indépendants de la machine et du système d'exploitation.

4.9 Conclusion

On a fait un workbench pour tester plusieurs implémentations de l'approche réactive (Junior, Loft, Cr) dans le but d'étudier leur vitesse d'exécution. Ceci a permis de mettre en lumière quelques défauts des implémentations mais aussi de faire quelques propositions.

De ce workbench on a obtenu les résultats suivants :

- Dans les implémentations en Java, on n'a pas identifié des sauts significatifs dans les courbes correspondant à de grosses périodes d'activité du garbage collecteur (GC). On a confirmé cette hypothèse avec un logiciel de profiling dans lequel on a vu que le GC se déclenchait régulièrement de telle sorte que la gestion de la mémoire ne générait pas des pics de temps. La gestion de la mémoire des implémentations du réactif en C reste un sujet à étudier; aucun des langages utilisés (Cr, Jrc et FairThreads) ne dispose d'un algorithme de gestion de mémoire et c'est de la responsabilité du programmeur de gérer l'allocation des événements.
- Lorsqu'on utilise souvent un comportement réactif, il faut se poser la question de la définition et de l'implémentation d'une nouvelle instruction particulière. La création d'une nouvelle instruction réactive améliore la performance mais aussi la compréhension, surtout si on lui donne une sémantique formelle.
- Il vaut mieux construire des systèmes réactifs en utilisant des présences d'événements que des absences. Tous les moteurs réactifs ont été optimisés pour mieux gérer l'attente de la présence d'un événement que l'attente de son absence.
- Le parallélisme est une instruction cruciale dans les moteurs réactifs ; elle doit être implémentée le plus efficacement possible car la performance générale du moteur en dépend. Mon implémentation en C de Junior (Cr-replace) m'a permis d'observer deux choses :

  1. Une instruction de parallélisme déterministe (Merge en SugarCubes) est plus coûteuse qu'une instruction non-déterministe (Par en Simple et Glouton). Le déterminisme nécessite une structure particulière pour mémoriser l'ordre d'exécution entre les branches, mais aussi un algorithme plus coûteux pour choisir la prochaine instruction à exécuter. Dans le cas de Cr-replace, le déterminisme s'est traduit dans un algorithme qui utilise des listes triées et non des listes traitées comme des ensembles (comme le fait Glouton).
  2. Le Par de plus haut niveau (celui qui exécute les programmes réactifs lorsqu'on fait `add` dans la machine) joue un rôle très important. Ce Par de plus haut niveau peut-être optimisé de façon particulière par rapport aux Par imbriqués.

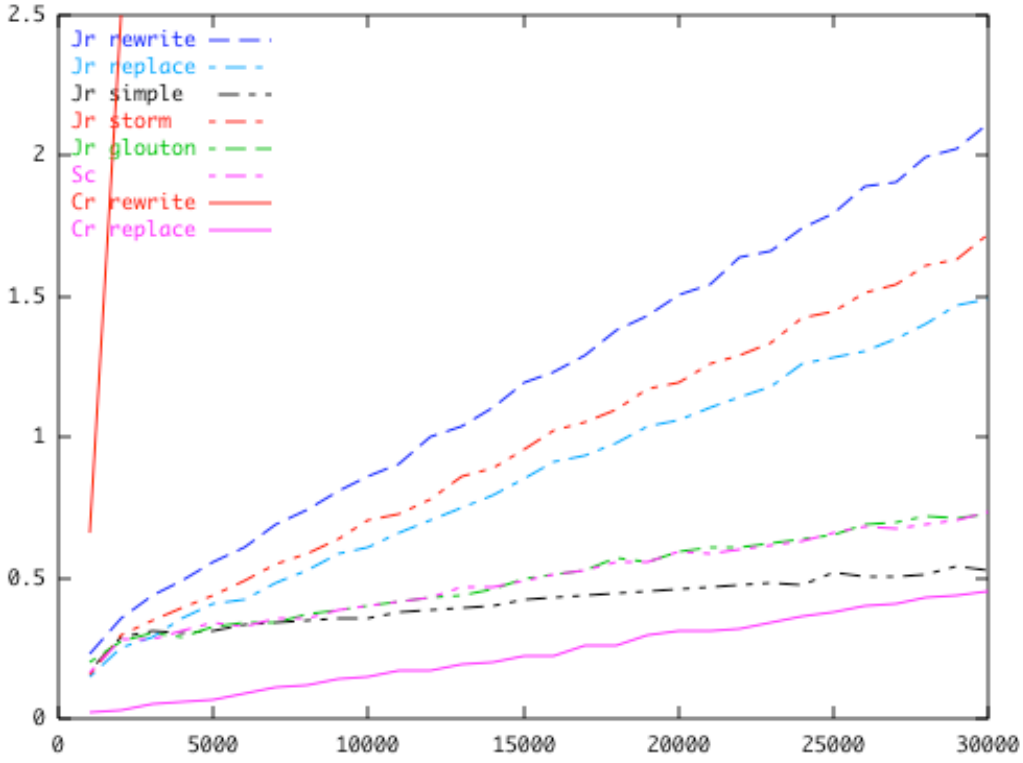

FIG. 4.20: Structure de contrôle.

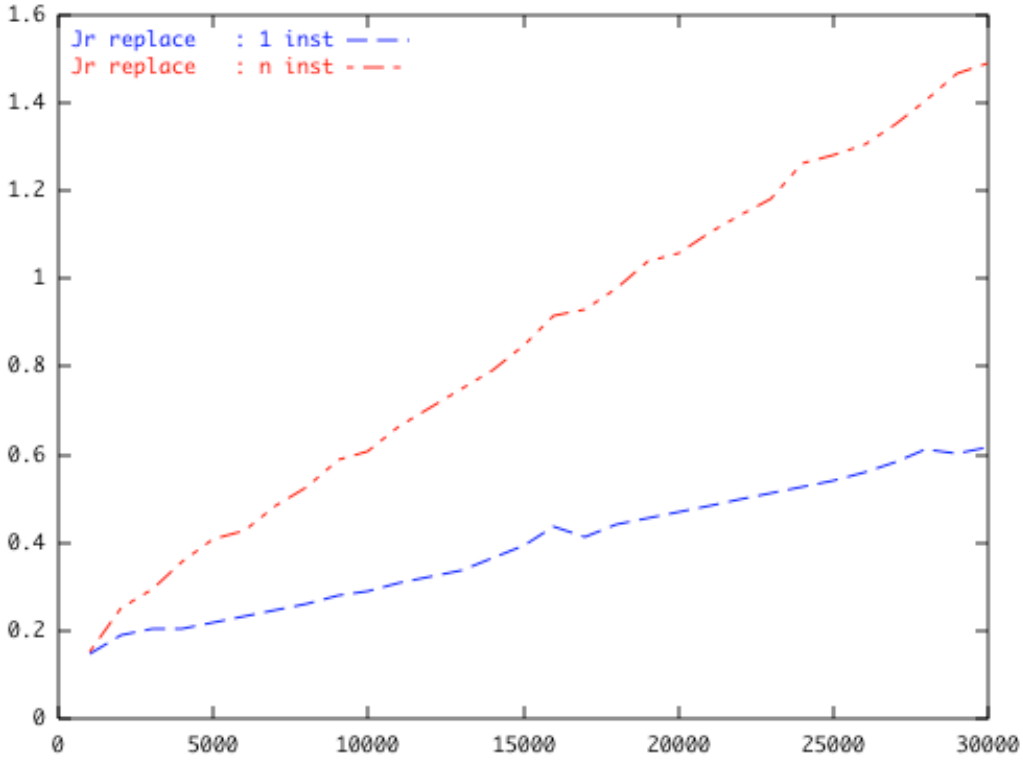

FIG. 4.21: Structure de contrôle : Une instruction versus plusieurs.

- Les résultats par implémentation sont les suivants :

  1. *Rewrite* est la pire implémentation. Dans les seuls cas où on aurait pu avoir des bons résultats à cause de la réécriture (Par et Seq en forme de peigne), la création d'objets était trop pénalisante. L'utilisation de listes pour les instructions n-aires s'est révélée bien plus efficace.
  2. *Replace* est une implémentation avec des performances antagoniques. Lorsqu'il n'y a pas d'instructions événementielles ou les événements sont présents avant l'exécution des instructions événementielles elle est parmi les plus rapides; par contre lorsqu'il y a des comportements en cascade (sur un ou plusieurs instants) elle est très lente. Pour avoir la meilleure performance il faut balancer l'arbre (dans les instructions Par et Seq); son atout principal est sa simplicité.
  3. *Simple* est une très bonne implémentation en termes de vitesse pourvu que l'on utilise un tableau dans l'implémentation du Par. Malheureusement la primitive qui l'utilise n'est pas définie dans l'interface Jr et c'est de la responsabilité du programmeur de l'utiliser.
  4. *SugarCubes* est une implémentation rapide qui est en constante évolution. Contrairement à d'autres implémentations, son auteur, J-F Susini, a continué son développement et ainsi on a pu tester de nouvelles versions améliorées (par exemple, pour le balancement de l'arbre). Son problème principal est l'attente active inter-instant et l'utilisation d'instructions binaires de composition.
  5. *Storm* est une implémentation qui n'implémente pas le même algorithme que SugarCubes (principalement le balancement) ; ceci s'est traduit dans la possibilité d'exécuter des programmes grands sans avoir le problème de la pile mais avec des temps trop pénalisants.
  6. *Glouton* est une implémentation très efficace qui est proche de Simple mais jamais plus rapide. Son atout principal par rapport à Simple est sa sémantique formelle. Glouton est une implémentation récente qui commence à faire son chemin et je pense qu'elle va évoluer et s'améliorer pour peut-être un jour rattraper Simple.
  7. *Rvm* est une implémentation rapide qui promettait beaucoup dans le domaine des machines à byte-code. Malheureusement elle est très lente dans certains cas et la dernière version disponible contient encore quelques bugs (j'ai pu en corriger certains mais d'autres, comme la taille de mémoire utilisée, restent un mystère).
  8. *Loft* est une implémentation très rapide tant que l'on n'utilise pas de programmes avec des comportements réactifs complexes. Son seul problème est l'attente active.

- On ne peut pas comparer de façon fiable les implémentations en Java avec celles qui ont été développées en C car les fonctionnalités et les algorithmes utilisés, en particulier la gestion mémoire, sont souvent différents.

  Néanmoins, on peut établir quelques points de comparaison :

  1. Cr-rewrite est plus lent et explose en pile plus vite que sa version en Java. L'explosion de la pile en Cr est due à la taille par défaut (512K) qui est celle de bash (le shell utilisé pour lancer les applications) ; cette taille de pile est plus petite que celle de Java.
  2. Cr replace est presque toujours plus rapide que sa version en Java balancée. Le seul cas plus lent est le test numéro un ; pour l'instant on n'a pas encore d'explication à ceci. La vitesse de Cr replace est due à plusieurs optimisations que l'on a réalisées : 1) pas de paramètre ni de valeur de retour dans la méthode `rewrite` (qui en Jr-replace n'a pas de paramètre mais uniquement une valeur de retour), 2) le code a été compilé avec l'option -O3 de GCC qui optimise la vitesse du code généré, 3) on a utilisé des instructions n-aires de composition et une méthode `reset()` à la terminaison. On n'a pas eu le temps de créer une version similaire en Java pour bien la comparer avec Cr-replace ; ceci reste dans les perspectives de recherche.

La table 4.1 présente les positions des différentes implémentations dans les tests que l'on a fait. Il y a onze colonnes qui représentent, de gauche à droite, les onze tests que l'on a réalisés ; ainsi la première colonne représente le test de la section 4.2 et les colonnes 3-6 les tests de la section 4.4. La classification a été réalisée de la façon suivante :

1. On a noté un P lorsqu'il a eu quelques problèmes pour exécuter l'implémentation. En 1) Loft, les Ps sont dus au fait qu'il n'existe pas d'instruction Control, 2) Rvm, les Ps sont des bugs qui soit génèrent des erreurs à la compilation soit font que les programmes ne finissent jamais.

2. La position ne considère pas les différences en croissance mais on a noté avec la même valeur lorsqu'il était impossible de distinguer une claire différence entre les implémentations.

3. La moyenne a été calculée avec les valeurs disponibles ; ainsi la moyenne de Loft a été calculée sur huit valeurs et celle de Cr-replace sur les onze.

4. Certaines implémentations ont des courbes qui se croisent. Dans ces cas, on a noté avec une fraction A/B pour indiquer que l'implémentation commence avec la position A puis passe à la position B ; dans la moyenne générale, on a considéré la moyenne de A et B.

Implémentations	Tests : 1-11											Moyenne
Jr simple T	5	9	2	3	1	2	2	5	2	4	2	3.36
Sc	7	6	6	5	4	3	4/5	6/7	5	5	3	5.00
Jr replace B	2	7	7	6	5	4	9	4	7	3	4	5.27
Jr glouton	9	5	3	8	2	6	5/4	7/6	3	10	3	5.63
Jr rvm	3	1	10	4	8	7	P3	8	9	P	P	5.88
Jr storm B	6	8	8	7	6	5	6	9	8	6	5	6.72
Jr rewrite	10	10	8	9	6	6	P11	P11	9	9	6	8.63
Jrc	8	2	1	P	P	P	1	2	1	1	P	2.28
Cr replace L	4	4	5	1	3	1	8	1	6	2	1	3.27
Loft	1	3	4	2	P	P	7	3	4	7	P	3.87
Cr rewrite	11	P11	9	10	7	8	P10	P10	P11	8	7	7.90

Tab. 4.1: Résultat du Workbench.

En conclusion, Simple est l'implémentation en Java qui semble la meilleure en moyenne. Elle est en particulier la plus efficace lorsqu'on a des comportements réactifs avec des longues attentes car elle évite l'attente active. Malheureusement son implémentation est compliquée et la vérification de sa sémantique et la création de nouvelles instructions sont difficiles. Ce sont ces difficultés qui ont motivé la création d'autres algorithmes plus "simples" (Storm, Glouton, Jrc) inspirés de Simple. Le problème principal de Simple, mais aussi des autres algorithmes, est de garantir que la sémantique est respectée par l'implémentation. La technique que l'on a utilisée est une batterie de tests qui bien sûr n'assure pas l'absence d'erreurs. Pour cette raison, on préfère les implémentations qui tendent à rester le plus près possible de la sémantique formelle. Au niveau de l'efficacité, les nouveaux algorithmes ne sont pas encore aussi performants que Simple mais ils s'en rapprochent. Il faudra attendre leur maturité pour évaluer les bénéfices des techniques qu'ils utilisent.

Certaines implementations mélangent plusieurs techniques (par exemple files d'attentes, byte code réactif dans le cas de la Rvm) dont la combinaison est difficile à évaluer. Dans ce workbench on a montré que dans les implémentations sans files d'attente les instructions n-aires et un reset en remontée améliorent nettement la performance. Ce workbench a aussi montré que pour être réellement efficace des files d'attente sont indispensables pour éviter l'attente active. Eviter l'interprétation de code améliorerait également certainement l'exécution (la Rvm le propose par l'introduction de byte code réactif).

Un point est à noter : de façon stricte, seules Rewrite, Replace, Simple et Storm en Java implémentent la même sémantique. Les autres implémentations considèrent soit des variantes de certaines instructions (par exemple

Glouton utilise une instruction Kill au lieu d'une instruction Until), soit ne les implémentent pas toutes (par exemple la Rvm n'implémente pas l'instruction Freezable).

Chapitre 5

Rejo: modèle et programmation 1ère partie

Rejo est un langage de programmation qui a démarré avec des objectifs très modestes et qui a évolué pour s'adapter aux besoins du programmeur. La première version cherchait à rendre la programmation de SugarCubes plus simple : on voulait éliminer l'utilisation répétitive des parenthèses et l'utilisation des instructions binaires de composition qui rend la programmation pénible. La version actuelle de Rejo cache d'autres aspects de bas niveau qui existent en SugarCubes et Junior (comme les wrapper et les actions atomiques) et cherche à définir un langage indépendant du moteur réactif utilisé. La version actuelle de Rejo génère, principalement, du code Junior mais il existe aussi une version expérimentale qui génère du code SugarCubes. Rejo est un langage de haut niveau qui définit son propre modèle d'objet réactif se distinguant de celui de SugarCubes. L'utilisation de Rejo pour programmer les agents mobiles de la plate-forme Ros et pour programmer les comportements réactifs de la plate-forme PING a influencé fortement son développement. Les structures des agents mobiles et des entités de la plate-forme Ping sont différentes ; Rejo génère donc du code indépendant de toute plate-forme d'exécution. Pour l'utilisateur qui veut programmer des agents mobiles, le compilateur offre une option spéciale permettant de générer du code pour la plate-forme Ros (voir le chapitre 7).

On présente le langage Rejo dans ce chapitre et dans le suivant. La présentation sera la plus indépendante possible du moteur réactif utilisé, sauf pour les aspects communs comme le modèle d'exécution (section 5.1) ou l'exécution des applications (section 5.3). La présentation du langage (section 5.2) se fait instruction par instruction, avec de petits exemples accompagnés de leur sortie et d'une explication. Finalement, la section 5.4 présente deux exemples complets de programmes Rejo qui résolvent le problème des philosophes de deux façons différentes.

Les contenus de ce chapitre et du suivant ont été publiés dans les articles [ACO 00b] et [ACO 02].

5.1 Modèle d'exécution

Comme dans tous les langages orientés objets, le modèle d'objet de Rejo définit la relation qui existe entre les données et le code. La particularité de Rejo est qu'il existe deux types de codes : le code Java (appelé méthode Java) et le code réactif (appelé méthode réactive). La relation qui existe entre les données et les méthodes Java, et entre deux méthodes Java, est définie par le modèle d'objets de Java. Par contre, ce qui est nouveau ce sont les différentes relations qui résultent de l'introduction du code réactif. La table 5.1 considère les diverses combinaisons possibles.

Comme on peut voir dans cette table, le modèle d'objet de Rejo est le résultat de la combinaison des deux modèles existants, plus la définition de nouvelles règles d'interaction (nouvelles instructions) entre les méthodes réactives.

Relation entre		***Définie par***
Donnée	Code Java	Modèle Java
Code Java	Code Java	Modèle Java
Donnée	Code réactif	Modèle Junior: Atom et Wrapper
Code Java	Code réactif	Modèle Junior: Atom et Wrapper
Code réactif	Code réactif	Modèle Junior(inline) + call

TAB. 5.1: Modèle de Rejo

Voici une description de chaque relation :

Donnés - Code Java et ***Code Java - Code Java***. Il existe plusieurs documents et pages web qui décrivent les règles d'interaction entre ces deux éléments, en particulier le site de Sun [JavaDocs] qui donne pas mal d'informations. La documentation qui nous intéresse est celle du langage Java [JavaLang]. Les définitions les plus importantes pour implémenter le modèle de Rejo sont la notion de portée des variables et des classes, l'invocation d'une méthode et l'héritage.

Donnés - Code réactif et ***Code Java - Code réactif***. Le code réactif peut accéder à toutes les variables de la classe sans aucune restriction. L'accès se fait, du point de vue du modèle réactif, de façon atomique, c'est-à-dire que l'accès commence et finit dans le même instant. Une autre façon d'accéder aux variables (avec un comportement similaire) est l'utilisation des Wrappers de Junior. Dans les instructions atomiques et dans les Wrappers, on peut exécuter n'importe quel type de code Java comme des invocations de méthodes, des expressions arithmétiques ou des instructions de contrôle de flot.

Code réactif - Code réactif. Il existe deux façons de construire un programme réactif:

1. La première manière consiste à construire les comportements réactifs séparément et à les exécuter en parallèle. Les comportements réactifs n'interagissent alors que par le mécanisme de communication de base qu'est la diffusion d'événements.
2. L'autre manière consiste à insérer des comportements réactifs préalablement définis dans un nouveau comportement. Il existe deux types d'insertions, l'***insertion statique*** sous la forme de `inline`, et l'***insertion dynamique*** avec l'instruction `call`.

Un intérêt de la première technique est qu'elle génère du code portable car les comportements réactifs ne gardent aucune référence vers les autres comportements. Le problème de cette technique est qu'il faut payer cette portabilité par l'implémentation d'un protocole qui détecte la présence ou l'absence des autres comportements ; dans certains systèmes où la commutativité entre les comportements est une propriété essentielle, ce protocole n'est pas nécessaire. Par contre, la deuxième technique est plus simple car elle n'a pas besoin de ce protocole, puisque l'on connaît déjà tous les autres comportements réactifs utilisés. Un autre avantage est que l'on obtient une construction modulaire plus souple, dans laquelle on peut, par exemple, construire des comportements réactifs imbriqués et dépendants des événements.

En fait, les différences entre l'insertion statique et l'insertion dynamique sont plus profondes qu'il peut sembler au premier abord :

- L'insertion statique est équivalente à la construction d'une macro ; ceci veut dire que l'exécution est rapide mais pas très souple. Les paramètres sont des valeurs lues à la compilation et non à l'exécution ; autrement dit, ce sont des constantes.
- L'insertion dynamique est équivalente à l'invocation de méthode ; ceci veut dire que l'exécution est légèrement plus lente que l'insertion statique mais aussi beaucoup plus souple. L'exécution est plus lente car il faut créer le programme à l'exécution ce qui entraîne que les paramètres sont de vraies variables.

- L'insertion statique (comme dans les macros) ne permet pas les définitions récursives, et est donc plus sûre, les erreurs étant détectées à la compilation. L'insertion dynamique (comme les méthodes) permet la récursion et risque donc de boucler et de générer des erreurs d'épuisement de pile (*stack overflow*).

La figure 5.1 résume toutes les relations mentionnées. Dans cette figure on ne voit que les quatre relations entre un comportement réactif et les autres éléments présents dans l'objet. La figure 5.2 montre toutes les relations possibles d'un comportement réactif lorsqu'il interagit avec d'autres objets réactifs de Rejo (que dans la suite on appellera simplement Rejo), ou quand il est construit par héritage.

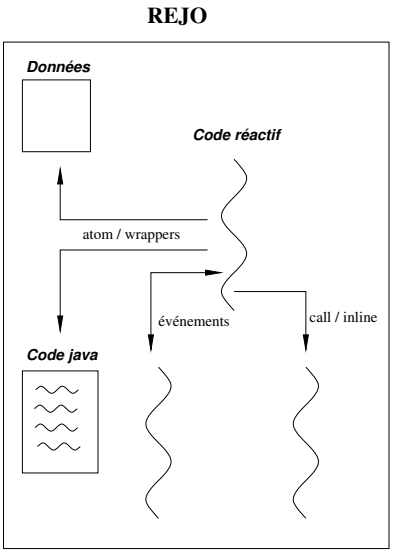

FIG. 5.1: Interactions d'un comportement réactif

Une des propriétés de l'insertion dynamique, qui est absente en Java, est la possibilité d'insérer du code qui n'était pas défini à la construction de l'objet. En effet, en Java on peut pas ajouter de nouvelles méthodes à un objet ; le dynamisme ne porte que sur la création de nouveaux objets (données). L'implémentation de l'ajout dynamique de méthodes réactives peut poser des problèmes de typage . Si la méthode ajoutée a été définie dans une autre classe, il faut vérifier (à l'aide de casts) qu'elle n'utilise pas de variable non définie dans la classe (voir les détails d'implémentation dans la section 6.2.1).

La figure 5.2 montre d'une part l'expressivité du modèle de Rejo ainsi que la complexité qui peut surgir lors de la construction d'un système. Heureusement, le réactif est là pour simplifier les choses, ce qui n'est pas le cas dans la programmation standard par threads, par exemple.

Finalement, il ne faut pas oublier que le moteur d'exécution d'un programme Rejo est toujours celui de Junior. Autrement dit, une application construite avec Rejo n'est rien d'autre qu'un ensemble de comportements réactifs (définis par des méthodes réactives) mis en parallèle. On obtient donc un arbre dans lequel les branches sont composées d'instructions réactives qui ont accès à deux ensembles de variables : celles de la classe (variables globales) et celles de la méthode (variables locales). Ceci est décrit sur la figure 5.3. L'exécution d'une méthode réactive (qu'elle soit insérée statiquement ou dynamiquement) se comporte comme l'appel d'une méthode Java, c'est-à-dire que l'appelant est bloqué tant que la méthode appelée n'a pas fini son exécution ; autrement dit, l'appel de méthode réactive est synchrone.

Une autre façon de voir les méthodes réactives est de les considérer comme des coroutines auxquelles on rajoute une notion d'instant. Une coroutine exécute un ensemble d'instructions jusqu'à atteindre un point d'arrêt où elle rend la main ; la prochaine fois que l'on exécute la coroutine, l'exécution démarre là où on s'était arrêté. Ce qui est nouveau est l'utilisation d'un code de retour (STOP, SUSP, TERM) relié à la notion d'instant. On obtient ainsi une suite d'invocations qui définissent un instant. Comme dans les coroutines, les paramètres d'une méthode réactive sont des variables locales qui sont initialisées une seule fois, au moment de la première invocation.

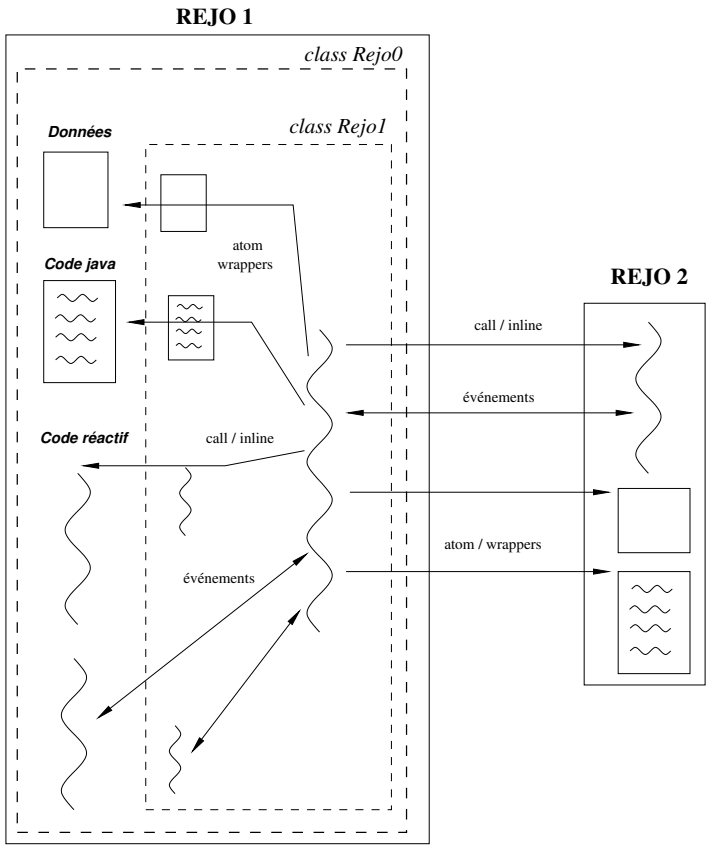

FIG. 5.2: Interactions d'un comportement avec héritage.

5.2 Programmation 1ère partie

Le langage Rejo est constitué des 14 instructions réactives présentes dans la version actuelle de Junior (la 2.1), plus 4 instructions réactives qui sont à l'étude[1] ainsi que quelques macros. Pour simplifier l'explication de Rejo, on présente dans un premier temps les instructions[2] suivantes :

atom	par	loop	repeat	if	call
wait	when	until	control	local	generate

Les autres instructions (freezable, link, scanner, try, dynapar) ainsi que les macros et d'autres aspects de la programmation en Rejo sont présentés dans le chapitre suivant.

5.2.1 Structure des programmes Rejo

Les programmes Rejo sont, comme les programmes Java, formés de classes. La structure d'une classe Rejo est semblable à celle d'une classe Java à laquelle on aurait ajouté de nouvelles méthodes appelées *méthodes réactives*. Les méthodes réactives ont leur propre syntaxe décrite dans la section suivante et leur sémantique est donnée par leur traduction en Junior. Une classe Rejo peut contenir plusieurs méthodes réactives apparaissant n'importe où dans la classe. En résumé, la seule différence avec une classe Java standard est la présence de méthodes réactives :

[1] Les instructions Scanner, Run, Dynapar et Try-catch.
[2] Les instructions Seq et Nothing de Junior n'apparaissent pas en Rejo car elles sont générées automatiquement par le compilateur.

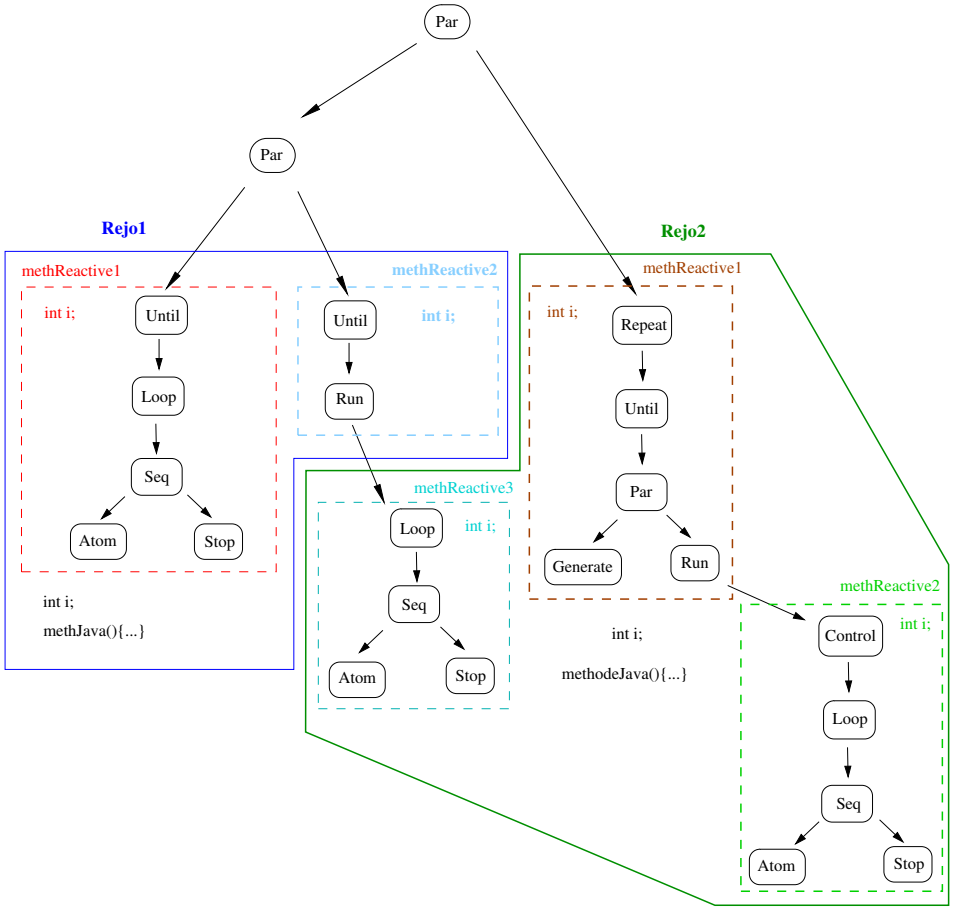

FIG. 5.3: Arbre d'exécution d'un programme Rejo.

```
package nom_du_package
import nom_du_package

public class nom_de_la_classe
{
  Variables;
  Méthodes Java
  ...
  Classes internes (inner-classes)

  Méthodes réactives
}
```

$$Rejo = Donnée + Code\ Réactif + Code\ Java$$

Après avoir créé un programme Rejo, le programmeur a deux options pour l'exécuter :

1. il peut, comme il le ferait en Junior, créer une méthode principale (`main`) qui ajoute les instructions réactives dans la machine et qui, ensuite, active cycliquement la machine pour faire réagir le programme ;
2. il peut charger la classe Rejo dans une plate-forme ROS, soit en chargeant manuellement une instance de la classe, soit en utilisant le shell réactif (Rsh) fourni par le système ROS. Dans ce dernier cas, Rsh crée une instance de la classe et l'ajoute automatiquement dans le système.

Plus d'informations concernant l'exécution des programmes Rejo est donné dans la section 5.3.

5.2.2 Méthodes réactives et instructions réactives de base

Les méthodes réactives ont une sémantique et une syntaxe très semblables à celles des méthodes Java. A la base, ce sont des séquences d'instructions ayant un nom et pouvant possèder des variables locales et des paramètres :

```
modificateurs reactive nom_de_la_methode( parametres )
{
  variables locales;
  instructions réactives
}
```

Dans le corps de la méthode, `instructions réactives` est une séquence d'instructions réactives. L'instruction qui met en séquence des instructions réactives est ainsi cachée au programmeur. Il existe également une instruction Par, décrite dans la sous-section 5.2.6, permettant de mettre des instructions réactives en parallèle.

Dans les méthodes réactives, le type de retour est remplacé par le mot-clé `reactive`. Les instructions réactives qui apparaissent dans les méthodes réactives sont de trois types :

1. Les instructions réactives spécifiques de Rejo, comme `par`, `loop`, `repeat`, etc., décrites dans les sous-sections suivantes ;
2. Le sous-ensemble des instructions Java composé des expressions. Ce sous-ensemble contient par exemple les invocations de méthodes et les expressions utilisant l'opérateur d'affectation ;
3. Les déclarations de variables Java. Ces variables sont locales aux méthodes réactives, comme dans les méthodes Java standard.

La sémantique des expressions et des variables est la même qu'en Java. Celle des instructions réactives est donnée dans les paragraphes suivants.

Pour faire coexister instructions Java et instructions réactives, les instructions Java sont exécutées en tant qu'*instructions atomiques*. Une instruction atomique est une instruction qui s'exécute d'une seule traite et qui termine instantanément, c'est-à-dire qui commence et termine son exécution au cours du même instant. Les instructions qui se terminent instantanément sont à opposer à celles dont l'exécution peut prendre plusieurs instants, parmi lesquelles on trouve, par exemple, l'instruction d'attente d'événement.

Les instructions composant le corps d'une méthode réactive sont exécutées en séquence et le passage à l'instruction suivante a lieu dès que la précédente est complètement terminée. L'instruction Stop stoppe le passage en séquence: l'exécution ne repartira en séquence qu'à l'instant suivant.

La fin de l'instant courant est définie de la même manière qu'en Junior, lorsque toutes les méthodes réactives exécutées dans l'instant sont soit complètement terminées, soit stoppées, soit bloquées en attente d'événements absents. Il faut noter qu'en général, le programmeur n'a pas à se soucier des retours TERM, STOP ou SUSP qui sont utilisés par la machine réactive pour propager le contrôle. Pour une compréhension fine du fonctionnement des instructions réactives, la prise en compte de ces retours est cependant parfois nécessaire.

Ce paragraphe se termine par l'exemple très simple du programme qui imprime `Hello World`. Ce programme utilise l'instruction atomique `System.out.print` qui affiche une chaîne de caractères à l'écran :

```
import ros.kernel.*;

public class HelloWorld implements Agent
{
  public reactive rmain(String[] args) {
    System.out.println(" Hello World ");
  }
}
```

5.2.3 Instructions atomiques

On ne peut utiliser qu'un sous-ensemble de Java pour écrire les instructions atomiques apparaissant dans le corps des méthodes réactives. Cette restriction est introduite pour éviter que le programmeur écrive des instructions qui n'auraient pas de sens (pas de sémantique) dans le modèle réactif, ou qui ne pourraient pas être exécutées par la machine réactive. Supposons que l'on écrive, par exemple:

```
for(int i=0; i<10; i++)
  stop;
```

La machine réactive rencontrerait un problème lors de l'exécution de la boucle for. En effet, cette instruction n'étant pas une instruction réactive, se poserait le problème de l'interprétation du code STOP retourné par l'instruction Stop.

Pour contourner cette restriction, le programmeur peut définir une nouvelle méthode Java chaque fois qu'il veut utiliser une structure de contrôle Java, puis invoquer cette méthode (les invocations de méthodes Java sont des expressions valides). L'instruction Atom de Rejo est là pour éviter ces définitions de méthodes, en permettant de considérer n'importe quelle instruction Java comme une instruction atomique.

A partir de maintenant, on va présenter une version abrégée de la syntaxe de chaque instruction réactive avec l'objectif de se faire rapidement une idée de l'instruction. La syntaxe complète (en notation BNF) est donnée dans l'annexe A et ici, à coté de la syntaxe, on donne le numéro de ligne dans cette annexe.

Syntaxe

lignes: 300-307

```
atom{ instructions_en_Java }
```

Exemple

Cet exemple imprime atomiquement la chaîne `0 1 2 3 4`.

```
int j=10;
atom{
  int i=5;
  if(i < j)
    for(int k=0; k<i; k++)
       System.out.print(k + " ");
}
```

5.2.4 L'instruction Call et les macros avec inline

Le paragraphe précédent montrait comment considérer les instructions Java comme des instructions réactives exécutées de manière atomique. En l'absence du mot-clé `atom`, seul un sous-ensemble des instructions Java peut être utilisé. Ce sous-ensemble inclus les invocations de méthodes ; ainsi, pour invoquer une méthode, il suffit de taper son nom et ses paramètres, comme par exemple dans :

```
public reactive rmain(String[] args)
{
  methode_Java();
}
```

Bien entendu, on peut récupérer la valeur retournée (si elle est définie) par un appel de méthode Java.

Pour appeler une méthode réactive, Rejo offre deux options :

Les macros avec le mot-clé inline. Lorsqu'on utilise le mot-clé `inline` on construit un comportement réactif à la compilation et on l'insère à la place d'`inline`. Ainsi, l'appel d'une méthode réactive consiste à insérer une copie du corps de la méthode réactive à l'endroit de l'appel. Etant donné que la construction est faite au moment de la compilation, c'est-à-dire que la construction est statique, les définitions récursives de méthodes réactives sont interdites avec `inline`. Une autre conséquence de la construction statique est que les noms des variables que l'on utilise avec inline sont traités comme des constantes (null pour les objets et zéro pour les autres types) car leur valeur ne peut exister qu'à l'exécution.

L'utilisation d'`inline` permet d'insérer non seulement des méthodes réactives, mais aussi des instructions Java retournant des instructions Junior, comme par exemple dans :

```
1. inline methode_Java();
   ...
   Program methode_Java() { return Jr.Loop(Jr.Stop()); }

2. Program p = Jr.Stop();
   inline p;

3. inline Jr.Repeat(5, Jr.Stop());
```

L'instruction Call. L'instruction Call permet de construire et d'exécuter du code réactif dynamiquement. La construction du code réactif est faite à l'exécution et elle permet donc les appels récursifs ainsi que l'utilisation de variables qui ne sont pas traitées comme des constantes, par opposition à `inline`.

Par exemple, considérons le code suivant :

```
public reactive rmain(String[] args)
{
  int var=3;

    call reactive_method(var);
}

reactive reactive_method(int i) {
  System.out.println("i="+i);
}
```

son exécution construit une instruction atomique à partir de la méthode `reactive_method(int i)` (avec la valeur de 3) et l'exécute ; la sortie du programme est `i=3`.

Il n'y a aucune différence entre `call` et `inline` lorsqu'ils apparaissent dans des méthodes réactives qui n'utilisent ni la récursion ni des paramètres avec des valeurs calculées à l'exécution.

Les exemples de la section 5.4 montrent des utilisations de `call` et `inline`.

5.2.5 Le système à Runtime

Etant donné que tout programme réactif s'exécute dans une machine réactive, il est indispensable d'avoir accès à celle-ci. Cet accès se fait en Rejo à travers la variable `env` qui est omniprésente dans toute méthode réactive. Cette variable donne accès aux primitives définies dans l'interface `machine` de Jr, c'est-à-dire:

```
add()          currentValues()     linkedObject()
generate()     previousValues()
```

5.2.6 Instruction Par

L'instruction Par implémente un parallélisme dans lequel les diverses instructions mises en parallèle réagissent toutes à chaque instant, et non l'une après l'autre comme c'est le cas lorsqu'elles sont mises en séquence. L'instruction Par se termine lorsque toutes ses branches ont terminé. Il ne s'agit toutefois que d'un parallélisme primitif dans la mesure où les instructions ne sont pas simultanément exécutées, comme ce pourrait être le cas avec une machine multiprocesseur. En fait, le parallélisme est sous contrôle de l'utilisateur, en particulier par l'utilisation de l'instruction Stop. Ce parallélisme correspond à une variante coopérative de la programmation par threads, dans laquelle l'instruction réactive Stop remplacerait `yield`.

La sémantique du `par` ne spécifie pas l'ordre dans lequel les branches sont exécutées ; la programmation en Rejo ne doit donc pas reposer sur un ordre particulier d'exécution de celles-ci.

Syntaxe

lignes:346-347

```
par{ instructions_réactives ||...|| instructions_réactives }
```

Exemple

Le programme suivant montre l'exécution au cours des instants de plusieurs instructions mises en parallèle à l'aide de l'instruction Par et contrôlées avec des instructions Stop.

```
                                Instants
                               #1   #2   #3
par
{
   System.out.print("1");       |
   stop;                        |
   System.out.print("4");            |
   stop;                             |
||
   System.out.print("2");       |
   stop;                        |
   System.out.print("5");            |
   stop;                             |
   System.out.print("6");                 |
   stop;                                  |
||
   System.out.print("3");       |
   stop;                        |
}
```

La sortie de ce programme est : `123456`.

5.2.7 Instruction Loop

Une boucle exécute plusieurs fois une suite d'instructions, que l'on appelle le corps de boucle. Dans le cas d'une boucle finie, le nombre de cycles est fini ; sinon, la boucle est dite infinie. En Rejo comme en Junior, deux instructions distinctes permettent de créer des boucles finies (Repeat) ou infinies (Loop). On présente dans cette sous-section l'instruction Loop ; l'instruction Repeat est présentée dans la sous-section suivante.

L'instruction Loop réexécute son corps dès qu'il se termine. On dit que l'on a une *boucle instantanée* lorsque une boucle infinie commence et termine l'exécution de son corps au cours d'un même instant. Les boucles instantanées sont dangereuses dans la mesure où elle peuvent bloquer l'exécution de la totalité du programme ; rappelons-nous que l'on est dans un cadre coopératif dans lequel il est impératif de donner le contrôle aux autres composants pour que le système global puisse évoluer.

Syntaxe

ligne: 344

```
loop corps
```

La syntaxe de l'instructions Loop suit les mêmes règles que celles de Java. Par exemple, des espaces et des caractères de contrôle peuvent apparaître entre les divers composants du corps. Comme en Java, les parenthèses "{", "}" sont nécessaires lorsque le corps comprend plus d'une instruction. Cette régle s'applique d'ailleurs également aux autres instructions Rejo décrites dans les sections suivantes. Bien entendu, il est recommandé en Rejo, comme dans tout autre langage de programmation, d'indenter les programmes pour les rendre plus lisibles.

Exemple

Cet exemple montre la forme typique permettant d'exécuter une instruction atomique à chaque instant :

```
loop{
  System.out.print("Hello World");
  stop;
}
```

Ce programme imprime `Hello World` à chaque instant.

Le second exemple est celui d'une boucle instantanée. En fait, ce programme bloque la machine car il ne passe jamais le contrôle aux autres composants (on ne doit donc jamais écrire un tel programme ; lorsque `loop` est utilisé, le corps doit contenir au moins un `stop`):

```
loop{
  System.out.print("Hello ");
  System.out.print("World");
}
```

Ce code imprime `Hello World` sans arrêt durant le premier instant.

Le troisième et dernier exemple est un programme que l'on peut utiliser pour éviter les boucles instantanées. La technique consiste à placer une instruction Stop en parallèle avec le corps de l'instruction Loop ; voici comment est transformé l'exemple précèdent :

```
loop{
  par{
     System.out.print("Hello ");
     System.out.print("World");
  ||
     stop;
  }
}
```

Ce programme imprime maintenant une fois `Hello World` à chaque instant.

5.2.8 Instruction Repeat

L'instruction Repeat implémente les boucles finies ; en d'autres termes, elle exécute son corps un nombre fini de fois, ce nombre étant déterminé la première fois que l'instruction est exécutée. A la différence des boucles infinies, le fait que le corps termine instantanément n'est pas un problème car, dans tous les cas, la boucle terminera. Ainsi, dans une boucle `repeat`, il n'est pas dangereux d'omettre les instructions `stop` (tout en gardant à l'esprit que la sémantique ne sera évidemment plus la même !).

Syntaxe

ligne: 366

```
repeat( expression_arithmétique ) corps
```

Exemple

```
repeat(5){
  System.out.print(" Hello ");
  stop;
}
```

Ce programme imprime cinq fois `Hello`, une fois par instant.

5.2.9 Variables réactives

En Rejo comme en Junior, on peut utiliser des variables dans une instruction réactive. Le problème de ces variables est qu'elles doivent être lues lorsque l'instruction est exécutée et non lorsqu'elle est créée. Pour résoudre ce problème, Junior définit des *wrapper* de variables, comme les IntegerWrappers et les BooleanWrappers. Un des avantages de Rejo est que les *wrappers* sont introduits automatiquement par le compilateur lorsque cela est nécessaire. La seule chose dont doit s'occuper le programmeur est, comme d'habitude, la correction du typage.

Par exemple, dans une instruction `repeat`, on peut utiliser une variable pour définir le nombre de cycles de la boucle. La sémantique de l'instruction `repeat` spécifie que ce nombre est calculé à la première exécution de la boucle. Ainsi, dans l'exemple suivant, l'instruction `repeat` sera exécutée avec la valeur 3, lue lorsque l'instruction est exécutée pour la première fois. Cette valeur reste la même, même si celle de la variable change par la suite :

```
int i=2;
i++;
repeat(i){
  System.out.print(i+", ");
  i++;
  stop;
}
```

Ce programme imprime: `3, 4, 5`.

5.2.10 Instruction If

L'instruction If permet de déterminer puis de poursuivre l'exécution d'une des deux branches qui la composent, en fonction du résultat de l'évaluation d'une condition booléenne. La première branche (celle après `if`) est exécutée si la condition est vraie ; dans le cas contraire, c'est la seconde branche (après `else`) qui est exécutée. Les conditions booléennes sont des conditions booléennes de Java, avec une restriction : il leur est interdit de tester la présence ou l'absence des événements ; si cela est nécessaire, il faut utiliser l'instruction When décrite en 5.2.14.

Syntaxe

ligne:338-340

```
if( condition_booléen )
    corps
```

```
if( condition_booléen )
    corps_1
else
    corps_2
```

Exemple

Dans le programme suivant, une instruction If teste si la variable `i` est égale à 3 et, lorsque c'est le cas, il imprime la valeur de `i` 3 fois, une par instant. L'exécution de l'instruction Repeat est effectuée 3 fois même si la valeur de `i` change aux instants suivants. La sortie du programme est : `i=3 i=4 i=5` .

```
int i=3;

if( i == 3 ){
  repeat( i ){
```

```
        System.out.print("i="+ i + " ");
        i++;
        stop;
    }
}
```

5.2.11 Evénements et conditions événementielles

Les événements sont un concept-clé du modèle réactif synchrone. En fait, la plupart des instructions réactives définissent des comportements dépendant d'événements. Les événements peuvent par exemple être utilisés pour contrôler ou préempter des instructions réactives. Les événements sont définis de la façon la plus générale comme étant des identificateurs, du type `Identifier`; cependant, en Rejo, le programmeur peut également utiliser les événements simplement sous forme de chaînes de caractères ou des entiers. La syntaxe que l'on peut utiliser pour construire les événements sera dans la suite appelée `expression_événementielle`; elle est définie dans le chapitre A ligne 386. La sémantique des événements est celle de Junior, c'est-à-dire que l'on retrouve en Rejo le paradigme des événements réactifs, avec le rejet de la réaction instantanée à l'absence qui élimine les problèmes de causalité.

On peut en Rejo définir des conditions événementielles (chapitre A lignes 373-380) qui permettent de tester si plusieurs événements sont présents au cours d'un même instant (opérateur AND) ou si l'un d'entre eux l'est (opérateur OR). On peut également tester si un événement n'est pas présent à un instant (opérateur NOT). Ce sont les trois seuls opérateurs utilisables pour construire des conditions événementielles.

En reprenant les notations de Java, la syntaxe Rejo de ces trois opérateurs est &&, || et !. Comme en Java, les opérateurs AND et OR ont la même précédence (et s'évaluent donc de droite à gauche) et l'opérateur NOT a une précédence plus grande que AND et OR. Les parenthèses "(" et ")" servent, comme en Java, à changer l'ordre d'évaluation.

On termine ce paragraphe par deux exemples de conditions événementielles. La condition 2 montre l'utilisation de chaînes de caractères pour désigner des événements (l'opérateur + désigne la concaténation) :

```
condition 1 = !( "a" && "b") || "c"
condition 2 = ! "a" && var1  || var2 + "b"
```

Les paragraphes suivants montrent comment générer des événements et programmer des réactions en fonction de conditions événementielles.

5.2.12 Instruction Generate

L'instruction Generate génère un événement dans l'environnement d'exécution, puis termine instantanément (c'est une instruction atomique). Les événements sont définis de la façon la plus générale comme étant du type `Identifier`. Ainsi, les événements peuvent être implémentés par n'importe quels objets, à la seule condition que soit définie une fonction d'égalité. A l'aide de cette abstraction, un programmeur peut implémenter les événements par n'importe quel type de base, par exemple les entiers. De plus, pour éviter au programmeur d'avoir à implémenter quoi que ce soit, Junior, et donc Rejo, définissent le type `StringIdentifier` qui permet de traiter les événements comme des chaînes de caractères. Les événements sous forme de chaînes de caractères sont, bien sûr, traités de façon transparente par Rejo. Pour plus de détails sur le type `Identifier`, voir la spécification de Junior.

Un objet peut être attaché à un événement lorsqu'il est généré ; en d'autres termes, on a des événements valués. Ainsi, les événements sont utiles non seulement pour synchroniser des comportements, mais aussi pour la communication de données.

Syntaxe

lignes: 327-328, 332

```
nom_de_l'événement !;
(gen || generate) expression_événementielle [, objs];
```

Exemple

```
"evenement" !;                              // Exemple 1
gen "evenement";                            // Exemple 2
gen "evenement" + variable + method();      // Exemple 3
gen variable;                               // Exemple 4
```

Dans la version actuelle de Rejo, la syntaxe utilisée dans l'exemple 1 n'accepte que des chaînes de caractères sans objet attaché. La syntaxe `gen` accepte, par contre, des chaînes (exemple 3), des objets attachés et des identificateurs généraux. Dans l'exemple 4, la variable peut être du type `String` ou `Identifier` (l'instruction `gen` teste le type).

5.2.13 Instruction Wait

L'instruction Wait implémente un comportement réactif qui attend qu'une condition événementielle soit vraie et termine lorsque c'est le cas. Elle s'implémente en testant si la condition est fausse ou bien si la fin d'instant est décidée. Lorsque la fin d'instant est décidée, l'instruction Wait retourne STOP, c'est-à-dire qu'elle se comporte à cet instant comme une instruction Stop.

Syntaxe

ligne: 330-332

```
nom_de_l'événement ?;
wait condition_événementielle [, objs];
```

Exemple

```
"evenement" ?;                              // Exemple 1
wait "evenement";                           // Exemple 2
wait "evenement" + variable + method();     // Exemple 3
wait ("a" && variable) || ! method();       // Exemple 4
```

La syntaxe utilisant le caractère "?" ne permet pas de traiter les conditions événementielles dans toute leur généralité, mais uniquement les événements désignés par des chaînes de caractères. Dans l'exemple 4, l'instruction Wait teste le type de l'événement pour déterminer le type d'`Identifier` à utiliser avec `variable` et `method()`.

On utilise les atomes suivants pour récupérer les valeurs associées aux générations des événements :

```
Object[] obj;

obj = env.currentValues( evenement);
obj = env.previousValues( evenement);
```

Ces deux instructions retournent un tableau d'`Objects` dans la variable `obj`. Il est de la responsabilité du programmeur de récupérer le bon type des valeurs générées (par des *cast* Java) et d'utiliser le bon type d'`Identifier`. Par exemple, pour récupérer les valeurs de l'événement `"e"` on exécute :

```
obj = env.previousValues(J.makeIdentifier("e"));
```

où `J.makeIdentifier()` est une méthode définie en Rejo pour créer des événements de type `String`.

Reprenons l'exemple de la cascade inverse décrit dans le chapitre 3. On peut l'écrire en Rejo de deux manière distinctes : 1) de façon exhaustive, en écrivant toutes les branches à la main (voir figure 5.4a) et 2) de façon récursive, en utilisant l'instruction Call (voir figure 5.4b).

```
public reactive waterfall()
{
   par
   {
        gen 1;
   ||
        wait N;   // N=1500;
   ||
        wait (N-1);
        gen   N;
     ...
   ||
        wait i;
        gen  (i+1);
     ...
   ||
        wait 2;
        gen  3;
   ||
        wait 1;
        gen  2;
   }
}
```

a) cascade classique.

```
public reactive waterfall()
{
   par
   {
        gen 1;
   ||
        int N=1500;
        call cascade(N);
   }
}

public reactive cascade(int N)
{
   par
   {
      wait N;
      gen (N+1);
   ||
      if(N!=1)
        call cascade(N-1);
   }
}
```

b) cascade produite récursivement.

FIG. 5.4: Cascade inverse

5.2.14 Instruction When

L'instruction When permet de choisir puis de poursuivre l'exécution d'une des deux branches qui la composent, en fonction d'une condition événementielle. Si la condition est vraie, alors la première branche (après when) est exécutée ; sinon, c'est la seconde branche (après else).

La particularité de cette instruction est qu'elle cache une instruction Stop lorsque l'évaluation de la condition événementielle nécessite d'attendre la fin de l'instant courant. C'est par exemple le cas lorsqu'elle teste l'absence d'un événement ; dans ce cas, l'exécution de la branche choisie ne peut débuter qu'à l'instant suivant ; tout se passe donc comme si l'on avait ajouté une instruction Stop au début de la branche choisie.

Syntaxe

ligne: 334-336

```
when( condition_événementielle )
    corps
```

```
when( condition_événementielle )
    corps_1
else
    corps_2
```

Exemple

Le programme suivant montre l'effet de l'instruction Stop cachée dans l'instruction When. Ce programme est composé de deux branches : la première génère l'événement `E` à l'instant N et la seconde branche teste sa présence.

Il peut se faire que la seconde branche ne puisse pas détecter l'événement parce qu'elle est désynchronisée à cause du `stop` caché dans le `when`. La trace 1 montre l'exécution lorsque N vaut 3 : l'événement est toujours vu comme absent. Dans la trace 2, N vaut 2 et l'événement est bien vu présent au troisième instant. La sortie du programme est `a a a` avec N = 3 et `a p a` avec N = 2.

```
                                    Trace 1                 Trace 2
                      instants ->   1  2  3  4  5  6    1  2  3  4  5
par
{
   repeat(N)                        |  |  |             *  *
      stop;                         |  |  |             *  *
   "E" !;                           |                   *
||
   repeat(3){                       |  |  |  |  |       *  *  *  *  *
     when("E")                      |  |  |             *  *  *
        System.out.print("p ");                         *
     else
        System.out.print("a ");     |  |  |             *  *
     stop;                          |  |  |             *  *  *
   }
}
```

5.2.15 Instruction Until

L'instruction `until` implémente la préemption, en forçant la terminaison d'une suite d'instructions réactives (le corps de l'instruction `until`) lorsqu'une condition événementielle devient vraie. La préemption est faible dans la mesure où le corps garde la possibilité de réaction à l'instant de préemption ; la préemption l'empêche en fait uniquement de réagir aux instants suivants. Junior, et par conséquent Rejo, utilisent cette forme de préemption pour éviter les problèmes de causalité.

L'instruction `until` a une partie `handler` optionnelle qui n'est exécutée que lorsque la préemption est effective.

La préemption réalisée par `until` peut ou non être instantanée. La préemption est instantanée lorsque la condition est vraie alors que l'instant courant n'est pas encore terminé. Dans ce cas, l'exécution du `handler` démarre instantanément (au cours du même instant). Dans le cas contraire, c'est-à-dire lorsque le fait que la condition de préemption soit vraie n'est connu qu'aprés que la fin d'instant ait été décidée, l'exécution du `handler` ne démarre qu'à l'instant suivant.

Syntaxe

ligne:368-369

```
until( condition_événementielle )
   corps
```

```
until( condition_événementielle )
   corps_1
handler
   corps_2
```

Exemple

Le programme suivant est composé de deux comportements. Le premier (la première branche de l'instruction Par) imprime `b` à chaque instant, tant que l'événement `préemption` n'est pas présent. Le second comportement (la deuxième branche du par) génère `préemption` au sixième instant :

```
par
{
   until("préemption")
     loop{
        System.out.print("b");
        stop;
     }
   handler
     System.out.println("h");
||
   repeat(5)
     stop;
   "préemption" !;
}
```

La sortie du programme est : `bbbbbbh`.

Le deuxième programme que l'on va présenter utilise l'instruction Until pour préempter une boucle potentiellement instantanée, en réalisant ainsi une sorte de `while` réactif. Ce type d'utilisation de boucle instantanée est potentiellement dangereux et il recommandé soit d'éviter un tel usage, soit d'être très prudent si on l'utilise.

```
int i=1;

until("kill")
  loop{
    if( i <= 5 ){
      System.out.print("*");
      i++;
    }else{
      gen "kill";
      stop;
    }
  }
```

La sortie du programme est : `*****`.

5.2.16 Instruction Control

L'instruction Control ne fait réagir son corps qu'aux instants où un événement est présent ; aux autres instants, l'instruction se comporte comme `stop` (elle retourne STOP). L'instruction Control termine lorsque son corps termine. L'instruction Control sert, principalement, à définir des sous-horloges.

Syntaxe

ligne:361-362

```
control( expression_événementielle )   corps
```

Exemple

Le programme suivant est composé de deux comportements. Le premier (la première branche de l'instruction Par) imprime `c` si l'événement `E` est présent. Le second comportement (la deuxième branche du Par) génère `E` aux instants 3 et 7:

```
par
{
   repeat(2){
     control("E")
       System.out.print("c");
     stop;
   }
||
   int i=1;
   repeat(10){
     if( i==3 || i==7 )
       "E" !;
     i++;
     stop;
   }
}
```

La sortie du programme est `cc` et celui-ci termine au onzième instant (la première branche termine au huitième instant, la seconde au onzième). Par contre, si l'on remplace 2 par 3 dans le repeat de la première branche, alors la sortie est la même mais l'instruction Par ne termine plus (la première branche n'est pas terminée et elle retourne STOP indéfiniment, en attente de l'événement `E` qui n'est plus généré).

5.2.17 Instruction Local

L'instruction Local déclare un événement local à son corps. L'événement déclaré doit être une constante.

Syntaxe ligne:357-359

```
local( expression_événementielle ) corps
```

Exemple

Dans l'exemple suivant on va générer l'événement `"E"` (ligne 18) qui ne doit pas être vu par le corps de l'instruction Local (l'instruction de la ligne 10) puis on va générer `"E"` dans le corps de l'instruction Local (ligne 7) pour vérifier que les instructions qui se trouvent en dehors de la portée de l'instruction Local ne le voient pas (l'instruction de la ligne 21).

Ce programme imprime `"Ext"` uniquement au premier instant et `"Int"` uniquement à l'instant suivant.

```
1  par
2  {
3      local("E")
4          par{
5              stop;
6              "E" !;
7          ||
8              loop{
9                wait "E";
10               System.out.print("Int");
11               stop;
12             }
13         }
14 ||
15     "E" !;
16     loop{
17         wait "E";
18         System.out.print("Ext");
19         stop;
20     }
21 }
```

5.3 Compilation et exécution

Il existe deux options pour exécuter un programme réactif créé avec le langage Rejo :

1. Construire le programme en suivant les étapes décrites pour Junior.
2. Construire le programme en le considérant comme un agent qui sera exécuté dans la plate-forme ROS.

L'exécution d'un programme Rejo en tant qu'agent est expliquée dans le chapitre 7.2. Tous les détails du processus de compilation sont expliqués dans la section 6.2.

Comme en Java, un programme Rejo doit être compilé avant de pouvoir être exécuté par une machine virtuelle.

Un exemple concret

Cet exemple est celui de la section 2.3.5 écrit en Rejo. Le programme exécute une action atomique à chaque instant. L'action atomique consiste à imprimer la chaîne `atom`. Après la création du programme et de la machine, le programme est chargé dans la machine puis activé dix instants.

```
1  import junior;
2  import ros.kernel.*;
3
4  public class Exemple2
5  {
6    public reactive rmain(String[] args)
7    {
8       loop{
9          System.out.println(" atom ");
10         stop;
11       }
12   }
13
14   public static void main(String[] args)
15   {
16     Machine M = Jr.Machine();
17
18     M.add( new Example2().rmain(args));
19     for(int i=1; i<=10; i++)
20        M.react();
21   }
22 }
```

Listing 5.1: Exemple 2 d'un programme Rejo

5.4 Exemples

5.4.1 Le problème des philosophes

Le problème des philosophes [DIJ 65, ARI 82] est un problème d'accès concurrents à des ressources partagées. Le but de cet exemple n'est pas de donner une solution complète au problème des philosophes, mais de montrer une utilisation conjointe de méthodes Java, de méthodes réactives et d'instructions réactives. Bien entendu, on veut également montrer la simplicité de l'accès à des ressources partagées lorsque l'on est dans un cadre coopératif. En effet, l'utilisation de sémaphores ou d'autres mécanismes de synchronisation n'est alors plus nécessaire, car il n'y a pas de risque d'être préempté durant un accès.

Le programme est composé d'une classe `Phil` qui déclare quatre philosophes (methode `rmain`, lignes 7-15). Le comportement des philosophes est représenté par une méthode réactive (`phi(int id)`, lignes 17-24) qui exécute une boucle infinie mettant en séquence les trois comportements élémentaires: `thinking`, `hungry` et `eating`. Les fourchettes sont modélisées par un tableau de booléens (`f[]`) qui indiquent si la fourchette est prise ou non. Le comportement de famine (la méthode `starving(int id)`, lignes 31-40) vérifie si les deux fourchettes voisines du philosophe sont libres avant de les prendre (méthode `takeForks` ligne 36). Le code est le suivant :

```
1  import ros.kernel.*;
2
3  public class Phil1 implements Agent {
4  int NF = 4;
5  boolean f[]={true, true, true, true};
6
7    public reactive rmain(String[] args)
8    {
9      par {
10        call phi(0);
11     || call phi(1);
12     || call phi(2);
```

```
13      || call phi(3);
14      }
15    }
16
17    public reactive phi(int id)
18    {
19      loop {
20         call thinking(id);
21         call starving(id);
22         call eating(id);
23      }
24    }
25
26    public reactive thinking(int id)
27    {
28       call waiting(id, "T");
29    }
30
31    public reactive starving(int id)
32    {
33      until("forksTaken"+id)
34        loop{
35           prn(id*2, "H("+id+")");    // Pas de lock pour
36           if( takeForks(id) )        // prendre les fourchettes
37             gen "forksTaken"+id;
38           stop;
39         }
40    }
41
42    public reactive eating(int id)
43    {
44      call waiting(id, "E");
45      freeFork(id);
46    }
47
48    public reactive waiting(int id, String str)
49    {
50      int i=5;
51      repeat(i){
52        prn(id*2, str+"("+id+") ");
53        stop;
54      }
55    }
56
57    boolean takeForks(int left)
58    {
59      int righ = (left==0) ? NF-1 : left-1;
60      if( f[left] && f[righ] ){
61         f[righ] = f[left] = false;
62         return true;
63      }else return false;
64    }
65
66    void freeFork(int left)
67    {
68      int righ = (left==0) ? NF-1 : left-1;
69      f[righ] = f[left] = true;
70    }
71
```

```
72    void prn(int nTabs, String str)
73    {
74      for(int i=1; i<=nTabs; i++)
75        System.out.print("t");
76      System.out.println(str);
77    }
78  }
```

Listing 5.2: Le problème des philosophes

La sortie obtenue lorsqu'on exécute `Phil` dans le système ROS est la suivante:

```
<localhost | > Phil1
T(0)    T(1)    T(2)    T(3)
T(0)    T(1)    T(2)    T(3)
    ...
H(0)
E(0)    H(1)    H(2)
                E(2)    H(3)
E(0)    H(1)    E(2)    H(3)
   ...
<localhost |  > gen "Phil!kill" # This command kills the Rejo
```

On peut voir qu'il peut y avoir plus de deux philosophes qui mangent simultanément, s'ils ne sont pas l'un à côté de l'autre.

5.4.2 Producteurs/Consommateurs

L'exemple précèdent montre comment profiter de la programmation coopérative à la base de Rejo pour résoudre un problème de concurrence sans verrous. Néanmoins, il reste un problème dans la solution proposée, qui consiste en l'attente active des philosophes lorsqu'ils ont faim. Ce problème se présente aussi dans un autre problème classique d'accès aux ressources partagées appelé, les Producteurs/Consommateurs [ARI 82]. Dans cette sous-section, on va présenter une solution au problème Producteurs/Consommateurs ; on a choisi ce problème car il est plus général que celui des philosophes dans le sens où les ressources ne sont pas préalablement assignés : une fourchette n'est accessible qu'à deux philosophes tandis que les cellules d'un buffer peuvent être lues ou écrites par plusieurs producteurs ou consommateurs.

On va s'intéresser tout particulièrement à la façon d'éviter l'attente active. En particulier, il faut endormir 1) le producteur lorsque le buffer est plein et 2) le consommateur lorsqu'il n'y a rien à consommer ; de plus, il faut réveiller 1) le producteur lorsque le buffer est libéré et 2) le consommateur lorsqu'un producteur produit quelque chose. La gestion des entités endormies est faite à l'aide d'une file d'attente pour les producteurs et d'une autre pour les consommateurs.

Voici le code :

```
1  import ros.kernel.*;
2  import java.util.*;
3
4  public class ProdConso extends Phil1
5  {
6   Vector buffer = new Vector();
7   Vector prod   = new Vector();
8   Vector conso  = new Vector();
9   int MAX=10;
10  boolean reduce=false;
11
```

```
12    public reactive rmain(String[] args)
13    {
14      par{
15         call producer("1");
16      || call producer("2");
17      || call producer("3");
18
19      || call consumer("4");
20      || call consumer("5");
21      }
22    }
23
24    reactive producer(String name)
25    {
26     int id = identif(name);
27
28       loop {
29          if( buffer.size() >= MAX ){        // Si le buffer est plein
30             prod.addElement(name);
31             prn(id*2, "Pd("+name+")");
32             reduce=true;
33             wait name+":wakeup";            // on s'endort
34          }else{                             // sinon on produit la donnée ""
35             buffer.addElement("");          // (pas de lock grâce au réactif)
36             prn(id*2, "Pp("+name+")"+buffer.size());
37             if( buffer.size() == 1 ){                // et on réveille les
38                Enumeration names = conso.elements(); // consommateurs
39                String c = "";                        // en attente
40
41                repeat(conso.size()){
42                   c = (String) names.nextElement();
43                   generate c+":wakeup";
44                }
45                conso.clear();
46             }
47          }
48          stop;
49       }
50    }
51
52    reactive consumer(String name)
53    {
54     int id = identif(name);
55
56       until(name)
57         par
58         {
59            loop {
60               if( buffer.isEmpty() ){        // Si le buffer est vide
61                  conso.addElement(name);
62                  prn(id*2, "Cd("+name+")");
63                  wait name+":wakeup";        // on s'endort
64               }else{                         // sinon on consomme la donnée
65                  buffer.remove(0);           // (pas de lock grâce au réactif)
66                  prn(id*2, "Cc("+name+")"+buffer.size());
67                  if( buffer.size() == MAX/2 ){          // et on réveille
68                     Enumeration names = prod.elements(); // les producteurs
69                     String c = "";                       // en attente
70
```

```
71                      repeat(prod.size()){
72                         c = (String) names.nextElement();
73                         generate c+":wakeup";
74                      }
75                      prod.clear();
76                   }
77                }
78                stop;
79             }
80          ||          // les consommateurs se tuent pour
81             loop{    // introduire du non-déterminisme
82               if(reduce){
83                 reduce=false;
84                 repeat(4) stop;
85                 generate name;
86               }
87               stop;
88             }
89          }
90       }
91
92       int identif(String str)
93       {
94         try{
95            return Integer.parseInt(str);
96         }catch(NumberFormatException e){
97            return 0;
98         }
99       }
100
101   }
```

Listing 5.3: Le problème des philosophes

La sortie obtenue lorsqu'on exécute le programme est la suivante :

```
<localhost | > ProdConso
Pp(1)1          Pp(2)2          Pp(3)3          Cc(4)2          Cc(5)1
Pp(1)2          Pp(2)3          Pp(3)4          Cc(4)3          Cc(5)2
        ...
Pp(1)8          Pp(2)9          Pp(3)10         Cc(4)9          Cc(5)8
Pp(1)9          Pp(2)10         Pd(3)           Cc(4)9          Cc(5)8
Pp(1)9          Pp(2)10                         Cc(4)9          Cc(5)8
Pp(1)9          Pp(2)10                         Cc(4)9          Cc(5)8
Pp(1)9          Pp(2)10                         Cc(4)9          Cc(5)8
Pp(1)9          Pp(2)10                         Cc(4)9          Cc(5)8
Pp(1)9          Pp(2)10                                         Cc(5)9
Pp(1)10         Pd(2)                                           Cc(5)9
Pp(1)10                                                         Cc(5)9
Pp(1)10                                                         Cc(5)9
Pp(1)10                                                         Cc(5)9
Pp(1)10                                                         Cc(5)9
Pp(1)10
Pd(1)
<localhost |  >
```

Cette solution a la particularité de réveiller tous les consommateurs lorsqu'un producteur génère une nouvelle donnée. Ce comportement est similaire à celui que l'on obtient en Java lorsqu'on appelle la méthode `notifyAll()`.

5.5 Conclusion

On a présenté Rejo, un nouveau langage de haut niveau pour la programmation réactive. Il s'agit en fait d'une extension de Java pour créer des objets réactifs en Java (Rejo signifie*REactive Java Objects*).

Les programmes Rejo sont traduits dans du code 100% Java. Plus précisemment, les instructions réactives sont traduites par le compilateur Rejo (appelé Rejoc) en instructions Junior (un ensemble de classes Java de bas niveau).

Rejo définit un ensemble d'instructions et de méthodes réactives pour construire des programmes réactifs. Rejo utilise un modèle particulier d'objets réactifs qui repose sur le modèle et la sémantique de Junior ainsi que sur le modèle d'objets de Java.

On obtient un langage ayant les caractéristiques suivantes:

- *Concurrence* : les objets réactifs de Rejo s'exécutent en concurrence (dans un mode coopératif). Comme Junior, Rejo constitue donc une alternative aux threads Java [BOU 00a] ;
- *Interactivité* : les objets Rejo communiquent en utilisant des événements diffusés instantanément ;
- *Réactivité* : les comportements des agents sont construits à partir de primitives permettant de réagir à la présence et à l'absence des événements diffusés ;
- Dynamicité : les agents peuvent être modifiés, ajoutés et enlevés du système dynamiquement, en cours d'exécution ;
- Multiplates-formes : la portabilité est la conséquence de l'utilisation du langage Java (byte code Java).

Chapitre 6

Rejo: programmation 2ème partie et implémentation

DANS ce chapitre on continue la présentation du langage Rejo (section 6.1) pour ensuite passer à la description et à l'analyse du code généré par le compilateur (sections 6.2 et 6.3). La section 6.4 présente une implémentation expérimentale de Rejo en C dans laquelle on a testé quelques nouvelles idées pour améliorer l'intégration du code réactif avec le code impératif. La section 6.5 fait le point sur les différents problèmes qui se présentent lors de la construction d'un système réactif avec Rejo. Finalement, la section 6.6 présente les travaux similaires et la section 6.7 les conclusions.

6.1 Programmation - 2ème partie

Dans cette section on présente les cinq instructions restantes, servant à implémenter la migration (l'instruction Freezable), à traiter les erreur d'exécution (l'instruction Try), à modifier un comportement (l'instruction Dynapar), à implémenter la notion d'objet réactif (l'instruction Link) et enfin à traiter plusieurs occurrences valuées d'un événement (l'instruction Scanner).

On finit cette section avec une brève description de l'héritage en Rejo et un exemple de son utilisation.

6.1.1 Instruction Freezable

L'instruction Freezable gèle un programme réactif en fonction de la présence d'un événement. L'événement est identifié à *runtime* lors de la première activation de l'instruction. A l'instant où l'événement est présent, le `corps` est exécuté une dernière fois, à la manière de la préemption. Le programme gelé est stocké dans l'environnement en parallèle avec les autres instructions gelées par le même événement, s'il y en a.

Un point important est à noter lorsqu'on utilise cette instruction : la machine nettoie les programmes gelés à chaque instant. Donc pour ne pas perdre un programme gelé, il faut impérativement le retirer à l'instant qui suit la génération de l'événement de gel.

Syntaxe

ligne: 361-362

```
freezable( expression_événementielle ) corps
```

Exemple

Le programme suivant gèle un comportement réactif (une boucle finie qui imprime `*`) lorsque l'événement `gele` est généré (au troisième instant). Une fois le comportement gelé, on le récupère à l'instant d'après, puis on laisse passer 3 instants avant de le recharger et de le réexécuter. La table 6.1 détaille ce que fait le programme instant par instant.

```
par{
    repeat(2)
       stop;
    generate "gele";
||
    freezable( "gele" ){
       repeat(5){
          System.out.println("*");
          stop;
       }
    }
    stop;
    Program p = env.getFrozen(J.makeIdentifier("gele"));
    repeat(3)
       stop;
    env.add(p);
}
```

Instant	Action
1	imprime "*"
2	imprime "*"
3	imprime "*" et gel
4	on récupère le programme
5	rien
6	rien
7	rien
8	ajout du programme
9	imprime "*"
10	imprime "*"

TAB. 6.1: Description du programme par instant

6.1.2 Instruction Link

L'instruction Link associe un objet Java à un programme réactif. L'objet associé est connu à *runtime* après avoir évalué une expression Java retournant cet objet.

Cette instruction joue un rôle important dans l'implémentation de la notion d'objet réactif. On discutera de son utilisation et de son impact en Rejo dans la section 6.2.

Syntaxe

ligne: 364

```
link( expression_objet ) corps
```

Exemple

Ce programme est composé de deux branches, une qui génère l'événement valué `"data"` avec la valeur `5` (l'objet `Integer`) et une autre qui attend le même événement et qui, lorsqu'il est généré, récupère la valeur et l'imprime à chaque instant.

```
par{
    generate "data", new Integer(5);
||
    Integer obj;

    wait "data", obj;
    link( obj ){
       Integer obj = (Integer)env.linkedObject();
       loop{
          System.out.println(" " + obj.longValue());
          stop;
       }
    }
}
```

La sortie de ce programme est `5` à chaque instant.

6.1.3 Instruction Scanner

Cette instruction attend toutes les occurrence de l'événement `expression_événementielle` et pour chaque occurrence valuée de l'événement, elle exécute du code Java.

Pour récupérer la valeur en question il faut exécuter la méthode `getData("scanner")` de l'environnement.

Syntaxe

ligne: 349-355

```
scanner( expression_événementielle ) code_Java
```

Exemple

Cet exemple illustre l'utilisation de l'instruction Scanner dans la construction d'un comportement réactif qui réagit lorsque la touche `A` est appuyée. Ce programme attend l'événement `KEY_PRESSED` qui est généré dans la méthode Java `keyPressed(KeyEvent key)` (voir les Ricobj dans la sous-section 7.4.2). Le code `env.getData("scanner");` récupère la valeur de l'événement `KEY_PRESSED` pour ensuite le comparer avec le code de la touche que l'on veut traiter (la touche `A` qui a le code `81`).

```
loop{
   scanner( KEY_PRESSED ){
      KeyEvent key = (KeyEvent) env.getData("scanner");
      int keyCode = key.getKeyCode();

      switch(keyCode){
         case 81:
                System.out.println("key A pushed");
                break;
      }
   }
}
```

6.1.4 Instruction Try-catch

Cette instruction est composée de deux ensembles d'instructions réactives `corps1` et `corps2`. Le comportement général de l'instruction Try-catch est de se comporter comme `corps1` tant qu'il n'y a aucune anomalie, c'est-à-dire tant qu'aucune exception n'est levée. En cas d'erreur durant l'exécution de `corps1`, l'exécution s'arrête et on exécute `corps2`.

Syntaxe

ligne: 342

```
try{ corps1 } catch { corps2 }
```

Exemple

Cet exemple illustre l'utilisation des exceptions dans une boucle finie qui est arrêtée avant de finir. Le programme utilise une condition Java pour savoir quand générer l'exception (au quatrième instant).

```
int i=1;

try{
   repeat(5){
      if( i <=3 ){
        i++;
        System.out.print("*");
        stop;
      }else{
        throw new Exception("Exception");
      }
   }
}catch{
   System.out.println("try-catch error");
}
```

La sortie de ce programme est `***Exception`.

6.1.5 Instruction Dynapar

Cette instruction se comporte comme son corps et lorsque un événement de contrôle est généré, elle analyse les valeurs de l'événement : si les valeurs sont de type `Program`, elle en crée des copies et les met en parallèle avec son corps ; si les valeurs ne sont pas de type `Program`, elles sont ignorées. La mise en parallèle se fait à la fin de l'instant pour ne pas perturber le corps. L'instruction Dynapar termine lorsque le corps finit, même si l'événement d'ajout est reçu à cet instant.

Syntaxe

ligne: 371

```
dynapar( expression_événementielle ) corps }
```

Exemple

Le premier exemple que l'on va présenter est un programme qui imprime la valeur de la variable `i` à chaque instant et qui met en parallèle un nouveau comportement qui imprime la valeur de `i` incrémentée d'une unité lorsque l'événement `"add"` est généré (tout les 5 instants).

```
int i=1;

par
{
   dynapar( "add" )
      call makeProg(i++);
||
   loop{
      repeat(5) stop;
      generate "add", makeProg(i++);
   }
}

reactive makeProg()
{
   loop{
```

```
        System.out.print(i+"");
        stop;
    }
}
```

la sortie de ce programme est la suivante `11111121212121212123123123 ...`

Le deuxième exemple que l'on va présenter est une implémentation de l'instruction Dynapar avec l'instruction Call. En effet, l'instruction DynaPar n'est pas primitive et on peut l'implémenter à l'aide de l'instruction Call et Freezable.

```
reactive dynaPar(String E, Program body)
{
 local("ctl", "kill", "reload")
   par
   {
      freezable("ctl")
        inline body;
      handler{
        wait "reload";
        call reLoad(E);
      }
      generate "kill";
   ||
      until("kill"){
        loop{
          wait E;
          generate "ctl";
          wait ! "kill";
          generate "reload";
        }
      }
   }
}

reactive reLoad(String E)
{
   freezable("ctl")
     par{
          call  env.getFrozen("ctl");
     ||   call  Jr.Par((Program)env.previousValues(E));
     }
   handler{
     wait "reload";
     call reLoad(E);
   }
}
```

6.1.6 Macros

Rejo définit quelques macros qui rendent la programmation plus simple. Ces macros sont :

- `wait eve, var;`
  Cette macro attend un événement valué `eve` et stoke la valeur dans la variable `var`. Pour s'assurer d'avoir toutes les valeurs associée à l'événement, il faut attendre la fin de l'instant ; la macro est traduite de la façon suivante à l'aide d'une instruction Stop :

```
wait eve ;
stop ;
var = env.previousValues(eve);
```

- `stop N; halt;`
  Ces macros sont traduites en :

```
repeat(N)          loop
  stop;              stop;
```

6.1.7 Héritage et Polymorphisme

Dès le début de la conception de Rejo, on a cherché à intégrer le plus possible les instructions réactives avec le langage Java. L'objectif est double :

- Eviter la création d'un nouveau langage. Créer un nouveau langage à partir de zéro (from scratch) pose principalement deux problèmes : les programmeurs doivent apprendre un nouveau langage (syntaxe et sémantique) et le processus de création est long et difficile à débugger.
- Profiter des tous les atouts de Java. Les principaux avantages de Java sont bien connus : portabilité, modèle d'objet simple, langage simple (sans pointeurs), etc.

Etant donné que Rejo génère du code Java, il hérite de la syntaxe et de la sémantique de la plupart des éléments de Java. En particulier, il hérite du modèle d'objet de Java, autrement dit Rejo est un langage orienté objets avec un héritage simple. La section 6.2.1 explique comment la traduction d'un programme Rejo profite de toutes ces caractéristiques de Java.

On va maintenant présenter une nouvelle implémentation du problème des philosophes que l'on a vu dans la sous-section 5.4.1 ; cette version profite de l'héritage pour ne pas reimplémenter tous les comportements.

6.1.8 Les philosophes avec événements

Dans la sous-section 5.4.1 on a présenté une solution au problème des philosophes en Rejo. Cette solution profite pleinement de la caractéristique coopérative de l'instruction de parallélisme pour éviter d'utiliser des verrous. Dans la solution présentée, un philosophe trouve (en fonction de son nom) qui sont ses voisins. Autrement dit, l'algorithme de détection des voisins est statique. Ceci apparaît clairement car les philosophes ne génèrent qu'un seul événement qui n'est pas utilisé pour communiquer avec les autres philosophes.

On présente maintenant une autre solution au problème des philosophes dans laquelle on utilise un protocole de communication entre les philosophes pour se détecter et pour décider qui entre dans la section critique (prises de fourchettes). A la différence de la première solution, cette fois-ci les fourchettes sont des entités actives qui ont un comportement réactif ; le comportement d'une fourchette décide (parfois aléatoirement, pour éviter la famine) qui la prend lorsque les deux philosophes concernés la sollicitent au même instant.

La figure 6.1 illustre graphiquement le protocole de communication ; le cas représenté est l'échange de messages entre une fourchette et deux philosophes qui la demandent au même instant.

Cette solution profite de l'héritage pour réutiliser les comportements: `thinking` et `eating`, ainsi que la méthode `prn`.

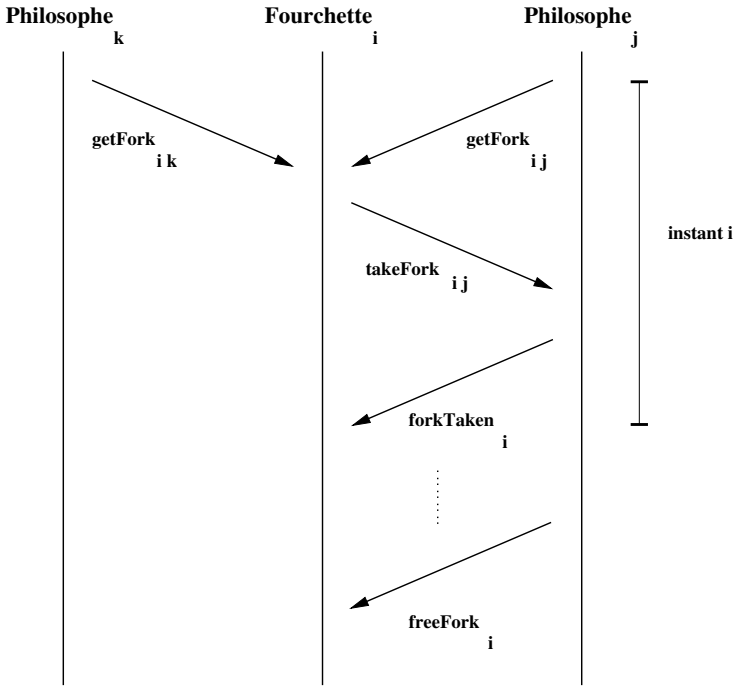

FIG. 6.1: Protocole de communication entre la fourchette et les philosophes

```
1  import ros.kernel.*;
2  import java.util.Random;
3
4  public class Phil2 extends Phil1
5  {
6   Random rand = new Random();
7   int NF=4;
8
9    public reactive rmain(String[] args)
10   {
11     par{
12        call fork("1");
13     || call fork("2");
14     || call fork("3");
15     || call fork("4");
16
17     || call phil("1");
18     || call phil("2");
19     || call phil("3");
20     || call phil("4");
21     }
22   }
23
24   reactive fork(String nameFork)
25   {
26    String taken;
27    boolean p1=false, p2=false;
28    int id = identif(nameFork), next;
29
30      next = (id==NF)? 1: id+1;
```

```
31        par
32        {
33           loop {
34              wait "getFork" + nameFork + nameFork;
35              p1=true;
36              generate nameFork;
37              stop;
38           }
39        ||
40           loop {
41              wait "getFork" + nameFork + next;
42              p2=true;
43              generate nameFork;
44              stop;
45           }
46        ||
47           loop {
48              wait nameFork;
49
50              if( p1 && p2 ) taken = (rand.nextInt(2)==1)?
51                                    nameFork+nameFork : nameFork+next;
52              else if( p1 )  taken= nameFork + nameFork;
53              else           taken= nameFork + next;
54
55              generate "takeFork"+taken;
56              when( "forkTaken"+nameFork ) {
57                wait "freeFork"+nameFork ;
58                stop;
59              }
60              p1=false;
61              p2=false;
62           }
63        }
64     }
65
66     reactive phil(String namePhil)
67     {
68      int id = identif(namePhil), prev;
69
70       prev = (id==1)? NF: id-1;
71       loop{
72          call thinking(identif(namePhil));
73          par{
74             call starving(namePhil);
75          ||
76             wait namePhil+"!eat";
77             call eating(identif(namePhil));
78             generate "freeFork"+ namePhil;
79             generate "freeFork"+ prev;
80          }
81       }
82     }
83
84     reactive starving(String namePhil)
85     {
86      int id = identif(namePhil), prev;
87
88       prev = (id==1)? NF: id-1;
89       until(namePhil+"!end")
```

```
90          loop{
91            prn(id*2, "H("+namePhil+")");
92            generate "getFork"+ namePhil + namePhil;
93            generate "getFork"+ prev + namePhil;
94
95            when( "takeFork"+ prev + namePhil &&
96                  "takeFork"+ namePhil + namePhil ){
97              generate "forkTaken"+prev;
98              generate "forkTaken"+namePhil;
99              generate namePhil+"!eat";
100             generate namePhil+"!end";
101             stop;
102           }
103         }
104     }
105
106     int identif(String str)
107     {
108       try{
109          return Integer.parseInt(str);
110       }catch(NumberFormatException e){
111          return 0;
112       }
113     }
114
115 }
```

Listing 6.1: Le problème des philosophes

La sortie obtenue lorsqu'on exécute `Phil2` dans le système ROS est la suivante :

```
<localhost | > Phil2
               T(1)          T(2)          T(3)          T(4)
               T(1)          T(2)          T(3)          T(4)
               T(1)          T(2)          T(3)          T(4)
               T(1)          T(2)          T(3)          T(4)
               T(1)          T(2)          T(3)          T(4)
               H(1)          H(2)          H(3)          H(4)
                                           E(3)
               H(1)          H(2)          E(3)          H(4)
               H(1)          H(2)          E(3)          H(4)
               E(1)
               E(1)          H(2)          E(3)          H(4)
               E(1)          H(2)          E(3)          H(4)
...
```

6.2 Implémentation

Rejo a été implémenté en utilisant JavaCC [WEB JavaCC]. JavaCC est un générateur de compilateurs, comme Yacc en UNIX. Le parseur JavaCC inclus aussi le générateur d'analyse lexicale (comme le fait Lex en UNIX). Les deux principales différences entre JavaCC et Yacc/Lex sont que les actions en JavaCC sont codées en Java et que la notation pour spécifier les règles de production et l'automate qui reconnaît les tokens est plus facile à utiliser.

L'idée principale dans la construction de Rejo est d'étendre la grammaire de Java pour pouvoir écrire des programmes Java qui contiennent des instructions réactives. Pour pouvoir mélanger les instructions Java et les

instructions réactives, Rejo propose la création de méthodes réactives qui contiennent des instructions réactives. Les instructions réactives se traduisent en des appels aux méthodes Java de la bibliothèque de Junior[1] qui implémente les instructions réactives. De cette façon, un programme écrit en Rejo est totalement traduit dans un programme Java, ce qui garantit la portabilité des objets Rejo.

La documentation de JavaCC inclus la grammaire de plusieurs versions de Java ; on a choisi celle de la version 1.1 [2] à laquelle on a ajouté les nouvelles règles de production et leurs actions. Pour faciliter l'usage du compilateur, il a été créé un script shell, appelé Rejoc (*Rejo Compiler*), qui facilite l'utilisation du compilateur. La figure 6.2 montre les phases de la création du compilateur de Rejo ainsi que la phase de compilation.

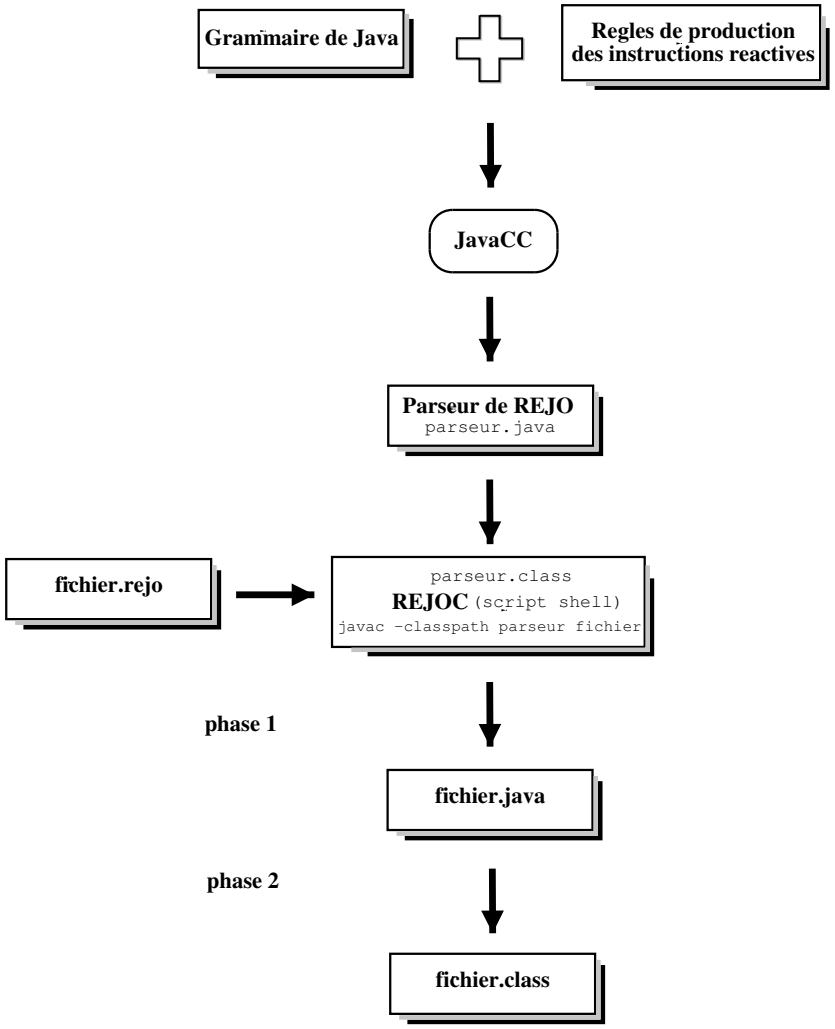

FIG. 6.2: Phases de compilation de Rejoc et d'un Rejo

Le programmeur peut contrôler le processus de compilation à l'aide des options suivantes :

pars : effectue la traduction en Junior, sans compiler le fichier Java produit (la seconde phase n'est pas exécutée) ;

[1] Il existe une version expérimentale qui génère du code en SugarCubes; cette version a été utilisée dans la constructions des tests présentés dans le chapitre 4.

[2] La version 1.1 ne présente pas de grosses différences par rapport aux nouvelles versions de Java, voir [JavaLang] pour plus d'information.

ros : ajoute au code généré des variables et des instructions Java permettant de produire un agent pouvant être exécuté par la plate-forme ROS. Cette option est décrite plus en détails dans le chapitre 7.

Voici par exemple comment compiler le programme `HelloWorld`:

```
$ rejoc HelloWorld.rejo
Compiling HelloWorld.rejo
Compiling HelloWorld.java
```

La grammaire de Java est composée de deux parties :

- L'*analyse lexicale* : Cette section définit les tokens du langage, par exemple les mots-clés : `if`, `for`, `class`, `int`, `break`, etc.
- L'*analyse grammaticale* : Cette section définit la structure des différentes composantes du langage. Elle contient trois parties principales :
  - La *structure d'un programme*, par exemple, la structure des classes, des méthodes, des interfaces, de la déclaration de variables, etc.
  - Les *expressions*, par exemple les affectations, les conditions, etc.
  - Les *structures de contrôle*, par exemple les instructions `if-else`, `while`, `for`, `do-while`, `switch`, etc.

Les principales modifications faites à la grammaire de Java pour implémenter le langage Rejo sont :

- On a ajouté la notion de *méthode réactive*. Les méthodes réactives sont définies par une règle de production, appelée `RMethodDeclaration`, ajoutée à la définition de classe.
- On a ajouté un ensemble de règles de production qui définissent la *structure* d'une méthode réactive. Ces règles définissent les instructions réactives, les variables réactives et les expressions réactives.
- Pour éviter la redéfinition des règles de production (surtout en ce qui concerne la définition des expressions), on a défini une méthode *action* qui choisit soit de copier le code Java tel quel (lorsqu'on parse du code non réactif) soit de le modifier ou de le stocker pour le traiter ultérieurement (lorsqu'on parse du code réactif).

L'appendice A contient les règles de productions de Rejo (en notation BNF). Cet appendice commence par la grammaire de Java suivie des règles qui ont été ajoutées. Ces dernières ainsi que la règle qui introduit les méthodes réactives ont été mises en gras.

Le tableau 6.2 montre quelques statistiques concernant les modifications faites. Dans cette table, on peut remarquer que les modifications les plus importantes ont porté sur les structures de contrôle.

Section	***Java***	***Rejo***
Structure d'un programme	39	14
Expressions	31	9
Structure de contrôle	15	20
Total	85	43

TAB. 6.2: Règles de production de Java et Rejo.

6.2.1 Code Généré

Dans cette section, on va donner quelques exemples du code generé par Rejo pour mieux comprendre ce qui se passe lors du parsing et comment le langage profite des caractéristiques de Java pour, par exemple, "implémenter" l'héritage.

Traduction d'une méthode réactive

La traduction d'une méthode réactive consiste en la génération de deux choses : 1) Une méthode Java qui contient la traduction des instructions réactives, et 2) Une classe qui contient les variables locales (les paramètres sont aussi des variables locales), les actions atomiques et les wrappers utilisés dans les instructions atomiques.

Le tableau 6.3 montre un petit exemple de la traduction d'une méthode réactive appelée `m`. Voici quelques explications sur le code :

- L'utilisation du mot-clé `reactive` a deux objectifs, il sert à : 1) généraliser le type de retour des méthodes réactives (`Program` pour Junior et `Instruction` pour SugarCubes), et 2) distinguer les méthodes réactives des méthodes Java qui rendent aussi des programmes réactifs; les méthodes réactives sont traduites par le compilateur Rejoc tandis que les méthodes Java n'ont aucun traitement.
- La méthode `m` en Java contient une variable `lv` de type `_m_1_`. Ce nom contient: 1) un numero pour distinguer les méthodes qui s'appellent de la même façon mais ont une signature différente, et 2) des caractères "_" pour générer des variables et des classes differentes de celles qui sont utilisées par le programmeur.
- La classe `_m_i_` contient une méthode `actions` qui exécute les actions atomiques et qui évalue les wrappers. Chaque action atomique et chaque wrapper est identifié par un index `_i_`. Ces actions atomiques et wrappers sont utilisés par des instances des classes auxiliaires `Action`(qui implémente l'interface action) et `NewTypeWrapper` (qui implemente l'interface TypeWrapper) qui ont été définies en Rejo.

Cette façon d'implémenter les méthodes réactives, en créant des nouveaux objets, est due au fait que la majorité des implémentations de Junior sont des implémentations à état. L'état d'exécution d'un comportement réactif est stocké dans chaque instruction et elles ne peuvent donc pas être partagées. Ceci a deux conséquences : 1) l'implémentation n'est pas très efficace en ce qui concerne l'espace mémoire utilisé par le code, et 2) l'implémentation est très rapide car on ne perd pas de temps dans les changements de contexte (sauver et restaurer des données dans la pile).

Un des avantages de traduire les méthode réactives comme on l'a vu dans la figure 6.3 est que l'on n'a pas besoin d'une instruction Link pour lier les données avec le code réactif ni des cast nécessaires. La clé de cette traduction est la méthode `actions` qui a été définie à l'intérieur de la classe `_m_1_` ; grâce à cette localisation, le code des wrappers et des actions atomiques a accès aux variables locales de la méthode réactive (comme la variable `arg`) et aux variables globales définies dans l'objet. En fait, ce que l'on exploite ici ce sont les droits d'accès définis en Java pour les classes internes (les *inner-class*) ; les classes internes peuvent accéder au code et aux données sans aucune restriction. En plus de ne pas utiliser l'instruction Link, on évite les casts et les exceptions résultant de l'utilisation de l'instruction Link.
Malheureusement, cette implémentation sans Link limite le dynamisme; lorsqu'on définit une méthode réactive, celle-ci reste attachée pour toujours au Rejo qui l'a défini [3]. Lorsqu'on construit un Rejo $R1$ et que l'on lui ajoute dynamiquement un nouveau comportement défini dans une méthode réactive d'un Rejo $R2$, le nouveau comportement est exécuté sous le contrôle de R1 mais il continue à affecter les données de $R2$.
Ce problème est fortement lié à la définition d'objet réactif. On est en train de créer une nouvelle version[4] du compilateur dans laquelle on trouvera un compromis entre vitesse et sécurité (lorsqu'il n'y a pas l'instruction Link, les casts et leurs exceptions).

[3] L'invocation d'une méthode réactive crée une instance des instructions réactives et de la classe auxiliaire qui contient les variables locales ; l'instance de la classe auxiliaire ne peut exister que s'il existe une instance de la classe du Rejo qui la contient. Cette contrainte s'explique car l'instance de la classe interne n'aurait pas des données à utiliser.

[4] Pour la petite histoire, la première version de Rejoc utilisait l'instruction Link et mettait un cast et un préfixe devant chaque variable locale pour accéder aux données et au code de l'objet lié (l'objet réactif).

```
reactive m(String arg)
{
   loop{
     System.out.println("atom");
     stop;
   }
}
```

```
Program m(String arg){
  _m_1_ lv = new _m_1_(arg);

  return
  J.Loop(J.Seq(
         J.Atom(new Actions(lv, 1, this)),
         J.Stop()));
}

class _m_1_ implements WrappersAndActions{
  String arg;

  _m_1_(String arg){
     this.arg=arg;
  }

  ...

  Object actions(..., int _i_, ...){
     ...
     switch(_i_){
        case 1:
                System.out.println("atom");
                break;
     }
     ...
  }
}
```

FIG. 6.3: Traduction d'une méthode réactive

Une autre façon d'implémenter des variables locales aux instructions réactives (atomes at wrappers) est la définition d'une instruction réactive spécifique. Cette stratégie, définie dans les SugarCubes (on utilise l'instruction `SC.localVariable(name, type, program)`), a l'avantage de ne pas utiliser des casts (code plus sûr) mais l'inconvénient de ralentir un peu l'exécution (l'arbre d'exécution est plus profond).

En résumé, l'implémentation des variables d'une méthode réactive repose soit sur l'instruction Link, soit sur la portée des variables définie en Java. C'est cette dernière technique qui est utilisée dans la version actuelle de Rejo pour définir quatre types de variables :

V1 Les variables de la classe. On a accès à ces variables depuis le code des méthodes Java et réactives, et depuis les classes internes. Par rapport à la notion d'instant, ces variables peuvent vivre un ou plusieurs instants.

V2 Les variables des méthodes réactives. Ces variables sont créées dynamiquement dans le tas et elles sont définies dans la classe auxiliaire créée pour chaque méthode réactive. Ces variables existent tant que la méthode réactive n'a pas terminé (un ou plusieurs instants); on peut voir ces variables comme des variables statiques d'une fonction C.

V3 Les variables des méthodes Java. Ces variables sont complètement gérées par Java dans la pile. Les variables ne sont vues que par les instructions de la méthode et, en général, elles sont détruites à la terminaison de la méthode. La durée de vie de ces variables est celle de l'action atomique dans laquelle l'appel de méthode a lieu.

V4 Les variables locales aux atomes. Ces variables sont définies dans la méthode `actions` et leur portée est celle du bloc dans lequel elles apparaissent. La durée de vie est celle d'une action atomique.

Toutes ces variables obéissent aux règles de Java. Par exemple, la définition de la classe suivante, qui illustre les quatre types de variables mentionnées, est valide.

```
public class Vars
{
 Objet V1;

   void m(){Objet V2; }

   reactive n()
   {
    Objet V3;

      atom{
       Objet V4= new Objet(); V1=V4;
      }
   }
}
```

Traduction des instructions de composition

Les deux instructions de composition offertes par Rejo sont la séquence et le parallélisme. La séquence est une instruction implicite et le parallélisme est une instruction explicite. L'algorithme de traduction de ces deux instructions est le même, il consiste à mémoriser la traduction des deux branches réactives avant de les écrire. Les instructions sont écrites lorsqu'on commence à analyser une troisième (on écrit la première instruction mémorisée et une instruction de composition) où lorsqu'on détecte la fin (on écrit la dernière instruction de composition avec les deux instructions réactives mémorisées et toutes les parenthèses accumulées). La table 6.4 montre la traduction de deux exemples, un pour la séquence et l'autre pour le Par.

```
reactive s()
{
  stop;
  stop;
  stop;
  stop;
}
```

```
Program _s_1()
{
  ...

  return
  J.Seq(J.Stop(), J.Seq(
        J.Stop(), J.Seq(
        J.Stop(),
        J.Stop() ))
       );
}
```

```
reactive p()
{
  par{
     stop;
  || stop;
  || stop;
  || stop;
  }
}
```

```
Program _p_1()
{
  ...

  return
  J.Par(J.Stop(), J.Par(
        J.Stop(), J.Par(
        J.Stop(),
        J.Stop() ))
       );
}
```

FIG. 6.4: Traduction des instructions de composition

Traduction des instructions atomiques

Les instructions atomiques sont traitées d'une façon particulière : au lieu de générer une instruction atomique par instruction Java, on génére une instruction atomique par séquence d'instructions Java. Ceci permet de générer du code efficace même lorsqu'on utilise le mot-cle `atom`. La traduction des atomes en séquence utilise aussi un algorithme qui mémorise les actions atomiques et ne les écrit que jusqu'à l'apparition d'une nouvelle instruction réactive qui n'est pas elle-même un atome. La tableau 6.5 montre de telles traductions.

La traduction d'une séquence d'actions atomiques n'a pas été complètement optimisée et il reste des cas que l'on pourrait réduire encore plus. Par exemple, si on a une instructions réactive If qui ne contient que des atomes dans ses deux branches, on pourrait ne générer qu'un seul atome, puis s'il est entouré d'autres atomes, les fusionner en un seul atome. Un autre cas que l'on pourrait aussi traiter comme un atome est l'instruction Generate ; l'instruction Generate est une instruction atomique que l'on peut exécuter dans une action atomique au lieu de la considérer comme une instruction réactive. Par exemple, pour générer l'événement `"eve"` on écrit :

```
EnvironmentImpl e = (EnvironmentImpl)env;
e.generate(J.makeIdentifier("eve"));
```

```
reactive m()
{
   i++;
   System.out.println("i="+i);
   loop{
     System.out.println("atom2");
     atom{
        for(int j=0; j<=5; j++)
           System.out.println("atom2");
     }
     stop;
   }
}
```

```
Program m()
{
  _m_1_ lv = new _m_1_();

  return
  J.Seq(J.Atom(new Actions(lv, 1, this)),
        J.Loop(J.Seq(
               J.Atom(new Actions(lv, 2, this)),
               J.Stop())))
  );
}

class _m_1_ implements WrappersAndActions{
     ...
     switch(_i_){
        case 1:
                 i++;
                 System.out.println("i="+i);
                 break;
        case 2:
                 System.out.println("atom2");
                 for(int j=0; j<=5; j++)
                    System.out.println("atom2");
                 break;
     }
     ...
}
```

FIG. 6.5: Traduction des actions atomiques

Ce programme n'est valide qu'en Rewrite, Replace et Storm car dans la version actuelle de Junior, on ne peut pas accéder à la machine d'exécution dans les actions atomiques ; on ne peut exécuter ni `add` ni `generate` à partir de la variable `env` qui est de type `Environment`. La machine est divisée en deux vues, celle définie par l'interface `Machine` et celle définie par l'interface `Environment` [5].
A l'heure actuelle, Rejoc n'optimise pas les instructions réactives If et Generate car, comme on verra dans la suite, Rejoc est un compilateur à une passe unique.

En fait, il existe trois grand types d'optimisations : 1) les optimisations pour créer un seul atome, 2) les optimisations associées à la simplification des instructions Stop, et 3) les optimisations qui résultent des interactions événementielles. La figure 6.6 montre quelques exemples d'optimisations du type 2 et 3. Les optimisations des instructions Stop et des interactions événementielles sont, en général, réalisées par le programmeur et la version actuelle de Rejoc ne les effectue pas. En fait, ces cas d'optimisation se présentent plus à l'exécution qu'à la compilation ; ces types d'optimisations font partie des futurs sujets de recherche, comme ceux liés à la traduction des instructions réactives en automates.

```
stop;                          repeat(N) stop;
stop;          ⇒
...
stop;
```

```
par{                           stop;
   stop;
|| stop;
  ...          ⇒
||
   stop;
}
```

```
gen "E";                       gen "E";
wait "E";      ⇒
```

```
until("E")                     wait "E";
  loop         ⇒
    stop;
```

FIG. 6.6: Optimisations possibles

[5]La classe qui implémente l'environnement en Rewrite, Replace et Storm s'appelle `EnvironmentImpl`; cette classe permet de générer des événements mais pas à travers l'interface `Environment`.

Traduction de conditions événementielles

Les conditions événementielles sont traduites en instructions Junior qui utilisent des `Config`. La particularité de ces instructions est qu'elles peuvent utiliser des expressions Java d'un type quelconque. Il faut donc définir un identifier et un wrapper pour chaque événement. Rejoc utilise soit la méthode `makeWrapper`, soit l'objet `VarIdentifierWrappers` pour construire l'événement et son wrapper, selon le type de donnée utilisée (String, Identifier). La méthode `makeWrapper` est utilisée pour construire des événements constants et l'objet `VarIdentifierWrappers` est utilisé pour construire des événements définis par des expressions Java.

En plus des expressions Java, la plupart des instructions événementielles peuvent être constituées de conditions événementielles, appelées Configurations en Junior; ces conditions événementielles sont traduites de gauche à droite par des configurations `Jr.And`, `Jr.Or` et `Jr.Presence` (une par événement). La tableau 6.7 montre la génération d'une condition événementielle.

```
reactive m()
{
  wait "eve1" && var || ! "eve3";
}
```

```
Program m()
{
  _m_1_ lv = new _m_1_();

  return
  J.Await(
   J.And(J.Presence(J.makeWrapper("eve1")),
         J.Or(J.Presence(
                new VarIdentifierWrapper(lv,1,this)),
              J.Not(J.Presence(J.makeWrapper("eve1")))
             ))
  );
}

class _m_1_ implements WrappersAndActions{
     ...
     switch(_i_){
        case 1:
                o=makeIdentifier(var);
                break;
     }
     ...
}
```

Fig. 6.7: Traduction des conditions événementielles

Traduction des invocations

Rejo utilise deux types d'invocations : les invocation statiques (macros) et les invocations dynamiques. Les invocations statiques sont traduits, comme toutes les macros, par simple insertion du code Java correspondant. Par contre, les invocations dynamiques sont traduites par une instruction réactive Run qui invoque la méthode `actions` ; c'est grâce à l'invocation de la méthode `actions` à l'exécution que l'on peut passer des paramètres et construire dynamiquement un nouveau comportement réactif. Le tableau 6.8 montre la traduction d'une méthode réactive qui effectue une invocation statique (utilisation d'inline) et une invocation dynamique (utilisation de call) recevant un paramètre, la variable `var`.

Compilateur en une passe

Le compilateur de Rejo est un compilateur très rapide car il traduit les instructions réactives à la volée. Autrement dit, il construit un arbre sémantique du programme, pour ensuite l'analyser et générer du code efficace. En fait, lors de la définition de la grammaire, on a remarqué que la syntaxe des instructions réactives était très simple et que l'on pouvait les traduire directement ; lorsque la syntaxe se complique (comme c'est le cas pour l'optimisation des actions atomiques), on a redéfini la grammaire de telle sorte que la traduction soit directe.

La compilation en une passe a ses limites, par exemple, il existe quelques programmes que l'on ne peut pas optimiser sans une vision globale. La figure 6.9 montre un programme réactif composé des quatre instructions réactives qui pourraient être simplifiés en un seul atome, comme le montre la figure 6.9b.

```
reactive m()
{
 int var;

  methode_Java();
  var=5;
  call methode_reactive_1(var);
  inline methode_reactive_2();
}
```

```
Program  m()
{
  _m_1_ lv = new _m_1_();

  return
  J.Seq(J.Atom(new Actions(lv, 1, this)),J.Seq(
        J.Run(new VarProgramWrapper(lv, 2, this)),
        reactive_methode_2())
       );
}

class _m_1_ implements WrappersAndActions{
     ...
     switch(_i_){
        case 1:
                 methode_Java();
                 var=5;
                 break;
        case 2:
                 o=methode_reactive_1(var);
                 break;
     }
     ...
}
```

FIG. 6.8: Traduction des invocations

```
if( cond )
   atome1;
else{
   repeat( 10 ){
      atome2;
   }
}
```

a) quatre instructions.

```
if( cond )
   atome1;
else{
   for(int i=1; i<=10; i++){
      atome2;
   }
}
```

b) un atome.

FIG. 6.9: Simplification des programmes réactifs

6.3 Analyse du compilateur

La version actuelle du compilateur Rejo, la 2.0b1, est une version assez stable et il ne reste que quelques petit problèmes de syntaxe et d'optimisation en suspend. H.P. Charles [HPC 03] a défini les caractéristiques que doit avoir un bon compilateur. Voici une analyse du compilateur de Rejo en fonction des points mentionnés par Charles :

Générer du code fiable . Vu que Rejoc effectue une traduction presque directe des instructions réactives vers Junior, la fiabilité du code généré est haute et en général il est très facile de vérifier que le code produit est celui que l'on a codé. Seules les instructions de séquence et de parallélisme sont traduites en plusieurs instructions ; la vérification de ces instructions reste aussi simple car on utilise toujours le même format (la même indentation) et lorsqu'il y a une erreur, elle est en général détectée à la compilation (première phase). Pour générer du code fiable, on a créé une batterie de tests qui teste la syntaxe et la sémantique des programmes Rejo. Cette batterie a permis d'éliminer pas mal d'erreurs syntactique et sémantique en Rejo (surtout lorsqu'on créait une nouvelle version) mais également des erreurs en Junior (par exemple le problème de la récursion infinie dans les boucles réactives). Cette batterie a été aussi utilisée pour le portage de Rejoc en C.

Générer du code rapide et peu volumineux . La vitesse d'exécution du code généré par Rejoc dépend presque entièrement du modèle d'exécution sous-jacent. Le seul overhead introduit par Rejoc est l'invocation de la méthode `actions` et la création des classes auxiliaires qui implémentent les variables locales. La

méthode `actions` est utilisée pour généraliser les invocations des méthodes d'évaluation des wrappers et des atomes, effectuées par le moteur d'exécution de Junior ; l'overhead à payer est une invocation de la méthode `actions` pour chaque wrapper et atome exécutés. Pour éliminer cet overhead, il faudrait mélanger le code réactif avec le code Java des wrappers et des atomes, autrement dit il faudrait éliminer l'interprétation du code réactif et générer du code ad-hoc. Le code ad-hoc est plus rapide mais présente quelques difficultés pour le migrer.

Le seul vrai problème de Rejoc est que le code généré est parfois un petit peu volumineux ; il génère de gros switch qui pourraient être simplifiés lorsque les actions atomiques ou les wrappers se répètent (ce qui améliorerait le temps de compilation).

Rapide à l'exécution . Vu que le compilateur de Rejoc traduit les programmes réactifs en une seule passe sans construire d'arbre sémantique, le compilateur est très rapide et utilise très peu de mémoire. L'algorithme de Rejoc pour parser les programmes réactifs est connu comme un parseur LL(1) ; les parseurs LL(1) sont plus rapides que les parseurs LL(K) mais aussi moins expressifs. En fait, le parseur de Rejoc (généré par JavaCC) est en général un parseur LL(1) qui devient LL(K) (grâce à la fonctionalité `LOOKAHEAD` de JavaCC) lorsqu'il parse certaines instructions.
Les parseurs LL(1) sont connus pour être plus lents que les parseurs LR(1) (comme Yacc) mais aussi pour être plus simple à programmer et mieux adaptés pour gérer les erreurs.

Générer des messages d'erreur compréhensibles . Lorsqu'on compile et on exécute un programme réactif il peut y avoir quatre types d'erreur :

1. Les erreurs trouvées dans la première phase de compilation. Ces erreurs sont détectées par Rejoc et sont correctement traitées ; le parseur généré par JavaCC donne de très bons messages d'erreurs et il n'est pas nécessaire de l'améliorer.
2. Les erreurs trouvées dans la deuxième phase de compilation. Ces erreurs sont détectées par Javac et ne sont pas gérées correctement car elles font toujours référence au code Java alors que parfois (pour les wrapper et les atomes) il faut faire référence au code réactif à partir duquel elles ont été produites. Jusqu'à la version 1.3 de Java, il n'y avait rien pour résoudre ce problème, comme ce qui existe dans le langage C (la macro inline). Les futures versions de Rejoc utiliseront le mécanisme JSR (*Debugging Support for Other Languages*) de la version 1.4 de Java.
3. Les erreurs trouvées à l'exécution pendant la création du programme réactif. Ces erreurs sont détectées par la JVM et ne sont pas gérées correctement. Ces erreurs sont plus difficile à traiter car il faudrait informer le moteur réactif (Junior) de l'origine de l'erreur qu'il a trouvé. Heureusement, le compilateur Rejoc est très robuste et il génère rarement ce type d'erreurs (provenant par exemple d'une instruction Loop sans body ou d'une instructions événementielle sans wrapper).
4. Les erreurs trouvées pendant l'exécution du programme réactif. Ces erreurs sont détectées par le moteur réactif de Junior et elles ne sont pas non gérées correctement. A l'heure actuelle le moteur réactif n'est pas au courant de l'existence de Rejoc, c'est-à-dire qu'il n'y a rien en Junior qui fasse référence aux définitions de Rejo. Par exemple, Junior n'a aucun moyen de savoir quelles sont les instructions réactives qui appartiennent à une méthode réactive et donc à un objet réactif.

   En Rejo, seule l'instruction Call (ou Run en Junior) possède une String qui contient le nom de la méthode qui l'invoque.

Intégré à un ensemble d'outils Les outils classiques que l'on utilise pour programmer sont un éditeur de texte, un débugger (un dévermineur) et des outils pour analyser le code. Rejoc n'est pas un produit commercial et il n'offre pas donc un environnement classique de programmation. Cependant, rien n'empêche son utilisation dans un environnement particulier. J'ai ainsi défini dans mon éditeur préféré (vim) les mot-clé de Rejo. Par ailleurs, lorsque j'ai du débugger des programmes Rejo, j'ai remarqué que l'on passe beaucoup de temps à débugger le moteur réactif (on parcourant l'arbre) ; un environnement de programmation réactive devrait cacher les traits caractéristiques de chaque moteur réactif et devrait cacher l'utilisation de la séquence et l'utilisation des instructions Par binaires.

Cependant, la version actuelle de Rejoc souffre des défauts suivants :

1. Limitations du langage. L'idée étant d'intégrer le réactif avec Java, on voudrait pouvoir écrire les mêmes programmes avec la même sémantique. Malheureusement l'intégration a un prix :

   (a) Pas de méthodes statiques. Etant donné que les méthodes réactives de Rejo sont impléméntées avec la création dynamique de plusieurs classes (les classes des instructions réactives et des variables locales) il n'est pas possible de créer des méthodes réactives statiques. Néanmoins, il est possible d'avoir un comportement similaire en construisant deux classes, une qui contient la définitions des comportements réactifs et une qui contient une variable statique de la classe qui définit les comportements ainsi qu'une méthode statique pour chaque méthode réactive. Voici un exemple que l'on trouve dans l'implémentation de Ricobjs (voir section 7.4.2) :

```
public class Behaviors
{
  static Behav b= new Behav();

  static public Program inertia(Object self)
  {
      return b.inertia(self);
  }

   ...

}
```

```
public class Behav ...
{
 ...

  public reactive inertia(Object self)
  {
    Icobj ico = (Icobj)self;
       loop{
          ...
         stop;
       }
  }
}
```

   (b) On ne peut pas utiliser les mots-clés de Rejo en dehors des méthodes réactives. La plupart des langages prohibe l'utilisation des mots-clés dans, par exemple, la déclaration de variables ou de méthodes. En Rejo cette contrainte ne s'applique de la même façon car il y a deux types de code (code Java et code réactif); on devrait pouvoir utiliser les 18 mot-clés de Rejo (sauf le mot-clé `reactive`), en dehors des méthods réactives (là où ils posent aucun problème sémantique). La version actuelle de Rejoc n'accepte que l'utilisation de quelques mot-clé.

   (c) Quelques expressions Java ne sont pas encore acceptées. Les limitations actuelles sont liées au fait que la compilation se fait en une passe et qu'il est difficile d'intégrer les instructions de contrôle comme le `for`. Par ailleurs, dans la version actuelle, il existe quelques expressions qui ne sont pas acceptées par la compilateur, comme les atomes qui commencent par un cast (par exemple `(Integer)variable.parseInt("45");`) et qui seront intégrées dans la prochaine version.

2. Les problèmes de Junior se retrouvent en Rejo, où ils sont même propagés au modèle d'objet. Par exemple, il existe quelques données qui ne sont connues qu'à l'exécution réactive de l'objet ; il est donc nécessaire de configurer l'objet à l'exécution et non à la création dans le constructeur.

6.4 Version en C de Rejo: RAUL

Comme on l'a vu jusqu'à maintenant, il existe plusieurs implémentation du modèle réactif en Java et Rejo offre une vision commune à toutes ces implémentations (principalement grâce à l'interface Jr). Cependant les différentes implémentations que l'on a commencé à réaliser en C ne disposent pas d'un langage de haut niveau et dans le cas de FairThreads on sent que Loft ressemble à une version restreinte de Rejo. On peut se demander pourquoi ne pas avoir une version de Rejo en C ou, mieux encore, si on ne pourrait pas concevoir un langage réactif qui ne dépende pas de Java, de C ou d'un autre langage impératif (possiblement orienté objet comme C++ ou Objective-C).

Pour explorer l'idée d'un langage réactif qui unifie les langages réactifs déjà existants, on a créé le langage RAUL (Reactive Algorithms Unified Language). Ce langage n'est pour l'instant qu'une version de Rejo en C avec les différences qui en découlent naturellement :

1. Il n'y a pas d'objets. Etant donné que l'on commence à utiliser le réactif dans la construction de systèmes embarqués, on n'a pas voulu ajouter la notion d'objet car elle peut-être lourde à gérer dans ce contexte. De plus, il faudrait choisir la meilleure notion d'objet, principalement entre celle de C++, celle d'Objective-C et celle de C#.
2. La syntaxe des actions atomiques et wrappers est celle de C, c'est-à-dire que l'on a l'arithmétique de pointeurs. Pour l'instant cette syntaxe ne pose aucun problème pour générer du code en C++ ou Objective-C, mais par contre elle en pose pour Java.

Le compilateur de RAUL, créé avec Yacc et Lex, est un compilateur en 2 passes, c'est-à-dire qu'à la différence de Rejoc, on construit un arbre de syntaxe abstraite qui est parcouru au moins deux fois : une fois pour optimiser le code et une autre pour le générer. RAUL définit deux instructions réactives qui n'existent pas en Rejo : `while` et `for`; ces instructions sont implémentées à l'aide d'une seule nouvelle instruction réactive (qui n'existe pas en Junior), le While réactif.

Les optimisations faites dans le compilateur de RAUL sont celles que l'on a en Rejoc (en particulier, création d'une seule action atomique par séquence d'atomes) plus d'autres comme, par exemple, la création d'une action atomique à partir des structures de contrôle, comme le If ou le While réactifs, ne contenant que des atomes.

RAUL génère du code dans deux moteurs réactifs : Cr et la Spirit Machine. On a ainsi un langage réactif de haut niveau (comme Rejo) qui cache les détails de l'implémentation des différents moteurs réactifs. D'autres moteurs réactifs pourraient aussi être utilisés : FairThreads en C, Rvm en C ou Jrc. Etant donné que RAUL génère du code en C, on ne peut pas utiliser le mécanisme de Rejo (la portée des *inner class*) pour définir les variables locales. RAUL utilise donc l'instruction réactive Link (définie en Cr et dans la Spirit machine) pour avoir accès aux variables locales de la fonction réactive. La technique d'exécution des actions atomiques et des wrappers est la même que celle de Rejo (méthode `actions` avec un switch ; voir section 6.2.1).

Voici un exemple en RAUL qui imprime * 5 fois, une à chaque instant.

```
#include <stdio.h>

reactive f()
{
 int j;

   printf("begin\n");
   for(j=1; j<=5; j++){
     printf("*");
     stop;
   }
   printf("end\n");
}

int main()
{
 Instruction *p;
 Environment env;
 int i;

   initEnv(&env);
   p = f();
   add(&env, p);

   i=0;
   do{
     i++;
     printf("\ni(%d):\n", i);
   }while( !reac(&env) );
```

```
    return 0;
}
```

Listing 6.2: Exemple d'un programme en RAUL

Pour vérifier la syntaxe et la sémantique de RAUL, on a utilisé la même batterie de tests que l'on a créée en Rejo. Néanmoins, RAUL est encore dans un état expérimental et certains problèmes restent encore à résoudre. Un des problèmes est l'utilisation du mot clé `typedef`; la définition de nouveaux types de données avec `typedef` pose des problèmes dans le parsing des programmes et nécessite de parser les librairies. Le parsing des librairies comme `stdio.h` dépend de la plate-forme et du compilateur utilisé. Par exemple, GCC et ses librairies utilisent du code C qui n'est pas conforme au standard AINSI ; le compilateur de Cygwin utilise des types prédéfinis qui n'apparaissent dans aucun fichier.

Pour terminer la description de Rejo, on va maintenant considérer quelques difficultés que l'on trouve dans la programmation dans ce langage, qui sont en fait plus généralement des difficultés de la programmation réactive.

6.5 Difficultés de la programmation réactive

Lorsqu'on construit un système réactif avec Rejo, il faut veiller à un certains nombre d'aspects pour garantir le bon fonctionnement du système avec la meilleure performance possible. Les problèmes en Rejo sont, pour la plupart, les problèmes généraux du modèle réactif que l'on retrouve bien sûr en Junior et dont Rejo hérite. Voici une liste des problèmes les plus souvent rencontrés :

1. Granularité des instants. La granularité des instants a un impact direct sur la performance et sur la réactivité du système. En effet, si la granularité est trop petite, elle provoque un grand nombre de "changement de contextes" (dont certains inutiles) qui se traduit en une baisse du temps laissé pour les calculs. Par contre, une granularité trop grande implique plus de temps pour les calculs et donc moins de temps pour réagir.

   Par exemple, si on construit une boucle `for` qui imprime 5 étoiles, on peut le faire dans un seul instant (dans une action atomique comme dans la figure 6.10a), ou en 5 instants (comme dans la figure 6.10b).

```
for(i=1; i<=5; i++){
    System.out.println("*");
}
```

a) un instant.

```
repeat(5){
    System.out.println("*");
    stop;
}
```

b) cinq instants.

Fig. 6.10: Granularité

2. Coopération. Pour garantir l'évolution des systèmes construits avec Rejo (propriété de vivacité), ceux-ci doivent coopérer ; toute situation qui retarde ou empêche la coopération est donc signe de mauvais fonctionnement. En général, les instructions réactives ont été conçues pour garantir la coopération ; la plupart des instructions événementielles coopèrent implicitement et le programmeur programme des points de coopération explicite avec l'instruction Stop. Néanmoins, il reste quelques cas où on peut construire des systèmes non coopératifs. En particulier, on peut construire : a) des boucles instantanées avec l'instruction Loop, b) des méthodes réactives récursives qui n'ont pas de cas terminal pour arrêter la récursion au cours de l'instant, c) des actions atomiques qui ne finissent jamais, et d) des wrappers (pour les configurations événementielles et les conditions booléennes) dont l'evaluation ne finit pas.

```
loop ;
```
a) boucle instantanée

```
reactive f(){ call f(); }
```
b) récursion instantanée

```
for(;;);
```
c) atome infini.

FIG. 6.11: Violation de la coopération.

3. Désynchronisation. Comme le fonctionnement d'un programme Rejo est basé sur la notion d'instant, il est très important que les comportements réactifs soient bien synchronisés, autrement dit, que les instructions réactives événementielles soient exécutées au même instant que la génération des événements. Si une instructions événementielle n'est pas exécutée au même instant de la génération de l'événement utilisé, le système perd sa capacité à réagir. La principale cause de désynchronisation est due au fait que l'on retarde d'un instant : 1) la réaction à l'absence ou 2) l'ajout des nouveaux comportements.

   Pour comprendre comme on peut construire un programme qui se désynchronise, analysons le programme de la figure 6.12a. Si l'événement `"E"` n'est pas généré, l'instruction When se comporte comme une instruction Stop et à l'instant d'après elle exécute la partie `else`; Lorsque la partie `else` n'est pas écrite on exécute une instruction Nothing.
   Le problème de ce programme est la terminaison instantanée du corps de la branche when. En effet, si le corps finit instantanément, la boucle Loop peut également finir instantanément. Pour éviter ce problème, il faut analyser le code de la branche when (la méthode `m`) pour voir si la terminaison instantanée est normale et, si elle l'est, modifier le code pour éliminer la boucle instantanée. La solution la plus simple est d'ajouter une instruction Stop après l'instruction when (voir figure 6.12a); c'est justement cette instruction Stop qui, lorsque l'événement `"E"` n'est pas généré, introduit un instant supplémentaire au cours duquel le programme ne réagit pas. Si on ajoute l'instruction Stop dans le corps de l'instruction When, on a aussi un problème de désynchronisation qui se présente à la fin de l'exécution de `m`.

```
loop{
   when( "E" )
      call m();
}
```
a) boucle instantanée probable.

```
loop{
   when( "E" )
      call m();
   stop;
}
```
b) sans boucle instantanée.

FIG. 6.12: Désynchronisation

4. Déterminisme vs Non-déterminisme. L'instruction de parallélisme de Rejo a la même sémantique que celle de Junior, autrement dit, Rejo est un langage non-déterministe. Cependant, comme on a vu dans la sous-section 3.2.1, il existe quelques implémentations de Junior qui disposent d'une instruction Par déterministe (comme l'instruction Merge de SugarCubes). Lorsqu'on utilise ces implémentations et que l'on programme des comportements réactifs en se basant sur l'ordre du Par, il ne faut pas oublier que ce code n'est pas portable dans les autres implémentations. La seule façon d'implémenter un ordre d'exécution réellement portable est l'utilisation des événements.

5. Gel des instructions. Le gel des instructions peut être utilisé pour implémenter la migration de code réactif et sa sauvegarde (persistance). Malheureusement, la sérialisation ou la migration de ce code est dépendante de l'implémentation de Junior utilisée ; par exemple l'opération de sérialisation en Java sauve l'état des objets (les objets qui implémentent les instructions réactives) et cet état n'est pas le même pour toutes les implémentations. Ceci veut dire que la sérialisation et la migration ne sont portables que si on utilise toujours la même implémentation et que l'on ne la modifie pas. Notons que ce problème pourrait être résolu en definissant un format standardisé de sauvegarde des programmes Junior.

6.6 Travaux similaires

L'introduction de la concurrence dans les langages à objets pose trois types de problèmes, (1) l'articulation entre objets et processus, (2) la communication entre objets et (3) la synchronisation. L'expression de la synchronisation peut, selon le niveau auquel s'effectue le contrôle, concerner l'objet, ses méthodes ou les instructions de ses méthodes. Ainsi les problèmes posés par des accès concurrents de méthodes qui modifient un attribut peuvent être résolus en restreignant les possibilités d'exécutions concurrentes (en interdisant l'exécution concurrente de méthodes, la concurrence entre certaines méthodes).
Le lecteur peut se référer à plusieurs articles de synthèse qui présentent les différentes façons dont les concepts énumérés précédemment ont été mis en oeuvre [WEL 96, BRI 96, FUR 95].

Le modèle de Rejo se fonde sur un modèle à objets actifs pour lequel les méthodes sont exécutées dans un processus (celui qui fait réagir le moteur réactif). Dans la suite, on va présenter les différentes modèles d'objet actif qui ont été proposés dans le domaine des langages synchrones. Les mécanismes de synchronisation auxquels on s'intéressera, considèrent les méthodes comme unités de synchronisation.

Reactive Scripts

Les Reactive Scripts [BOU 96a, BOU 96b] forment un langage interprété et dynamique, qui possède des instructions événementielles inspirées de celles de SL. Il existe deux versions de Reactive Scripts, une implémentée au-dessus de Reactive-C et une autre au-dessus de SugarCubes.

Comme les Reactive Scripts génèrent du code dans un langage réactif synchrone, on peut les voir comme un langage de haut niveau comme Rejo. Cependant, il y a deux différences principales :

- Les Reactive Scripts sont un langage interprété tandis que Rejo est un langage compilé. La première version de Reactive Scripts fut implémentée au-dessus de Reactive-C et dans ce contexte il était très utile d'avoir un langage interprété pour dépurer les programmes écrits en C.
- Les Reactive Scripts sont un langage qui rend plus facile l'écriture des programmes mais qui, par rapport au langage sous-jacent, n'ajoute rien de nouveau. Dans la version SugarCubes, il existe une correspondance directe (à une exception près) entre les instructions des deux formalismes. Rejo, par contre, ne rend pas seulement plus facile la programmation mais aussi ajoute un nouveau concept : la programmation orienté objet.

Les différences principales entre Rejo et les Reactive Scripts (implémenté avec SugarCubes) sont :

- *Contrôle des instants.* Etant donné que les Reactive Scripts sont un langage interprété, le programmeur a la possibilité de contrôler la cadence des instants. De plus, le programmeur dispose de l'instruction `next` qui relance automatiquement l'exécution du prochain instant. A l'inverse, en Rejo, le programmeur n'a aucun contrôle sur les instants dans la partie réactive, il doit placer le code inter-instant dans la méthode qui fait réagir la machine. Les deux techniques ont le même pouvoir expressif.
- *Programmation modulaire.* Pour obtenir une programmation modulaire, les Reactive Scripts offrent au programmeur la notion de module (`behavior`). Le module a une sémantique différente de celle de methode réactive en Rejo :
  - Les modules sont des comportements réactifs qui peuvent changer d'un instant à l'autre, soit par leur redéfinition soit par l'ajout de nouvelles instructions mises en parallèle. Les méthodes réactives sont des comportements immuables. L'avantage d'avoir des méthodes immuables permet au compilateur de génère du code plus sûr, par contre les méthodes dynamiques . En Rejo on peut reproduire le comportement d'un module dynamique grâce à l'instrucion Link; une version dynamique des méthodes réactives est à l'étude.

  - La définition d'un module est une instruction réactive qui ne peut pas être redéfinie au cours d'un même instant. L'exécution d'un module (par l'instruction `run`) n'est possible qu'après sa définition. La définition d'une méthode réactive n'est pas une instruction réactive (la machine réactive ne connaît pas son existence); il n'y a pas donc de problèmes pour exécuter une méthode qui est définie après son invocation.
  - Les paramètres en Rejo sont des données qui n'ont aucune sémantique associée au comportement réactif tandis que les paramètres utilisés dans les modules peuvent être de deux types : événements ou objets. Lorsqu'on passe un événement comme paramètre, il faut spécifier si c'est un événement d'entrée (avec la sémantique de `InputLocalEvent` de SugarCubes) ou de sortie (avec la sémantique de `OutputLocalEvent` de SugarCubes). En Rejo on ne peut reproduire la sémantique des signaux d'entrée/sortie que celle qui est définie par l'instruction Local.

- *Instructions-Macros.* Reactive Scripts et Rejo définissent quelques instructions qui sont en réalité des macros. La macro principale en Reactive Scripts est l'instruction **object** qui définit un comportement réactif utilisant l'instruction Until et les modules. Rejo définit aussi quelques instructions-macro comme l'instruction Wait valuée.

- *Code réactif convergent.* L'instruction Loop en Reactive Scripts est blindée contre la construction de boucles instantanées : elle ajoute systématiquement une instruction Stop en parallèle pour les éviter. En Rejo, le programmeur doit l'ajouter à la main; par contre il peut coder des programmes qui pourraient boucler un nombre indéfinie de fois.

- *Interfaçage avec Java.* L'intégration des instructions réactives avec le code Java est réalisée dans deux situations bien identifiées : l'exécution d'atomes et l'évaluation de wrappers. Cette intégration présente deux problèmes: 1) c'est le point d'entrée de l'exécution de code non coopératif, et 2) quelle syntaxe faut-il utiliser. La syntaxe de Reactive Scripts cherche à simplifier l'intégration ; ils proposent une syntaxe, a base d'accolades, qui n'est pas transparente au programmeur. Au contraire, Rejo réalise une intégration complètement transparente en utilisant la syntaxe de Java.

- *Différences entre Junior et SugarCubes.* Les différences ont été déjà vues dans la section 3.3.

Rhum

Rhum [LAU] est une plate-forme interlogiciel (*middleware*), implémentée en Java et dédié à la programmation d'applications réactives et distribuées. Rhum mélange les principes de l'approche réactive et les concepts généraux d'environnement de programmation distribuée. Le modèle de programmation utilisé est appelé DROM (Distributed Reactive Object Model)[BOU 98a]; DROM définit un objet réactif comme étant une entité :

1. Accessible grâce à une référence, valide dans le cas d'une application distribuée.
2. Offrant une interface réactive. Les méthodes de cette interface peuvent être appelées à travers sa référence.
3. Réagissant à des appels suivant les spécifications de son comportement.

Une application Rhum est un système distribué constitué de domaines qui sont implémentés par des machines réactives Junior. Les domaines sont dans la plupart des cas situés dans des sites physiques distincts. Chaque domaine exécute sa propre séquence d'instants ; ils sont asynchrones car il ne partagent pas la même horloge. Les objets réactifs doivent être attachés à un domaine pour exécuter les différents instants de leur comportements. L'appel des méthodes réactives d'un objet réactif doit obéir à la sémantique suivante :

1. Quand l'entité appelante s'exécute dans le même contexte synchrone (même domaine) que l'objet réactif appelé, l'événement correspondant à l'appel est généré dans le même instant. Ainsi, l'objet appelé peut réagir instantanément à l'invocation.
2. Quand l'entité appelante s'exécute dans un contexte différent (environnement asynchrone, domaine distinct), l'événement est généré dans l'instant suivant du domaine dans lequel l'objet invoqué s'exécute.

Si on compare Rejo et Rhum, on voit que Rhum est aussi un langage de haut niveau bâti au-dessus de Junior. A la différence des Reactive Scripts, Rhum a les mêmes objectifs que Rejo : rendre la programmation plus facile et définir un modèle d'objet particulier. La principale différence entre Rejo et Rhum est qu'en Rejo on a voulu construire un langage d'usage général tandis que Rhum est un langage dédié aux plates-formes réparties.

Caractéristiques	**Rejo**	**Rhum**
Modularité	méthode réactive récursivité, appel instantané	méthode réactive une exécution par instant(DROM)
Héritage	possible	pas possible
Code Java	mélangé avec le réactif	séparé du code réactif (fichier à part) qui ne fait que des invocations

Tab. 6.3: Rejo vs Rhum

Icobjs

Les Icobjs ne sont pas à proprement parler un langage de programmation réactive orienté-objets mais ils proposent un modèle d'objet réactif (que l'on va expliquer en détail dans la section 7.4.2) très puissant au-dessus de Junior. Bien entendu, on a utilisé Rejo pour programmer les Icobjs et on a rencontré quelques difficultés qui nous ont fait réfléchir à l'évolution du langage. Le problème principal dans l'implementation des Icobjs est lié à la propriété de dynamisme : on voudrait pouvoir ajouter des champs dans les Icobjs à run-time. En Java, ces champs seraient soit des variables soit des méthodes. Malheureusement Java, n'est pas un langage de programmation qui permet changer la définition d'un objet, c'est-à-dire de redéfinir sa classe comme on peut le faire en Smalltalk. Cette propriété d'ajouter des champs dans un objet est l'essence du modèle d'exécution des Icobjs. Pour surmonter la limitation de Java, les Icobjs utilisent une structure dynamique (une table de hash) pour simuler l'ajout et l'élimination de champs. Les méthodes réactives ajoutées à un objet doivent réaliser un binding dynamique avant d'exécuter la moindre ligne de code et ensuite réaliser une série de tests (des casts) pour être sûr que l'objet possède les champs nécessaires. Cette série de tests pose des problèmes de sécurité mais aussi d'efficacité.

Le modèle des Icobjs a été implementé avec Rejo (voir 7.4.2) et le système résultant relève deux sujets à analyser : 1) comment limiter les problèmes de sécurité associés à l'ajout dynamique et 2) comment rendre transparent l'accès aux champs ajoutés. Dans la version actuelle de Rejo l'accès aux variables et les méthodes d'un Rejo s'implémente avec le mécanisme de porte des variables (expliqué dans la section **??**) qui ne nécessite ni du binding ni des casts. Cette implémentation respecte le modèle d'exécution de Rejo, presenté dans la section 5.1, qui attache de façon permanente une méthode réactive à un Rejo. Ceci veut dire que lorsqu'on ajoute un comportement réactif défini dans un Rejo A à un Rejo B le nouveau comportement : 1) continue de modifier l'objet A mais sous le controle de l'objet B, et 2) comme l'ajout se fait a run-time, le nouveau comportement réactif n'existe pas du point de vu du Rejo B. Le Rejo B ne peut utiliser les syntaxes *objet.variable* et *objet.method*(). Le Rejo B n'a aucun moyen de savoir quels ont été les champs ajoutés.
Le résultat de l'implémentation des Icobjs avec Rejo, que l'on appelle Ricobjs, est un mélange de deux modèles : 1) le modèle de Rejo qui facilite la programmation de la partie statique d'un Ricobj par la définition des méthodes réactives dans l'objet ou par héritage et 2) le modèle des Icobjs qui facilite la programmation dynamique par l'ajout des champs gérés avec une table de hash. La programmation des comportements réactifs ajoutés dans d'autres Ricobjs exige quelques règles : 1) ils ne doivent contenir que des references aux variables locales à la méthode réactive, et 2) on doit faire un binding dynamique explicite pour accéder aux champs du Ricobjs.

Les caractéristiques du modèle de Ricobjs, en particulier celles de l'ajout dynamique, font penser à la définition d'un nouveau modèle d'exécution de Rejo qui rend transparent l'ajout des nouveau champs tout en évitant d'introduire des problèmes de sécurité et en ayant une implémentation efficace. L'implémentation de l'ajout de nouveaux champs dans un Rejo demande une analyse sur qui doit gérer les ajouts, soit la machine réactive (comme le font les Reactive Scripts implémentés avec les SugarCubes) soit le objet lui-même (comme les Icobjs).

Dans la suite, on va présenter quatre langages qui mélangent la notion d'objet avec les concepts de langage synchrone à l'Esterel. Par rapport aux travaux précédents, ces langages utilisent une approche synchrone statique qui empêche l'ajout de nouveaux champs dans l'objet à run-time [6], par exemple des nouveaux comportements réactifs. Par contre, ces langages offrent la possibilité de vérifier des propriétés à l'aide d'outils automatisés de vérification.

Triveni

Triveni [COL 98] est un formalisme Java pour programmer des systèmes concurrents en utilisant des threads et des événements. Le modèle d'exécution de Triveni mélange les concepts d'Algèbre de processus, l'Approche synchrone et la Programmation orienté objet. La notion principale de Triveni est celle de *comportement abstrait* qui, dans un système concurrent, modélise l'interaction du système avec son environnement. La communication est réalisée par des événements qui représentent les canaux de communication. Un programme Triveni est constitué d'instructions dans le langage hôte (qui dans l'implémentation actuelle est Java) et de comportements abstraits programmés avec des *combinators* (des instructions réactives inspirées d'Esterel).

Triveni est compatible avec les threads Java et les threads POSIX (comme les pthreads); il est aussi compatible avec les modèle d'événements basés sur l'*Observer pattern* [GAM 95].

Le modèle de Triveni est très proche de celui d'Esterel. Parmi les fonctionnalités qu'offre Triveni, on trouve :

- Triveni étend la programmation par threads avec un opérateur de parallélisme. Les composants qui sont mis en parallèle peuvent être implémentés séparément et Triveni garantit que toutes les branches reçoivent les mêmes événements. Par rapport à Esterel, Triveni offre une communication asynchrone et une description de comportements autonomes.
- Il existe un mécanisme d'exceptions basé sur l'opérateur de préemption (`do-watching`).
- Triveni dispose d'une sémantique formelle [COL 99] constituée par une sémantique opérationnelle (qui garantit l'équité entre les branches parallèles) et une sémantique dénotationnelle basée sur l'équité des traces.
- Il est possible de spécifier des propriétés de sûreté en utilisant une logique temporelle (il existe un ensemble d'outils qui automatise les tests). Le programmeur peut insérer des conditions (à l'aide de l'instruction *assert*) pour débugger ou vérifier le bon fonctionnement du programme. Les conditions peuvent, en particulier, tester des séquences d'événements.

Objets à contrôle réactif

Le modèle des objets à contrôle réactif, développé par M. Augeraud, F. Bertrand [AUG 99a], permet la conception de programmes à objets concurrents. Ce modèle sépare le contrôle de la concurrence de l'expression des transformations de l'état des objets. Pour cette raison, le contrôle est exprimé dans un composant de l'objet, ***le contrôleur réactif***, qui a en charge de contrôler l'exécution des méthodes, entités dont le rôle est d'exprimer les transformations. Le contrôleur exprime les conditions liées à l'état d'exécution des méthodes et il coordonne les exécutions des méthodes affectant un même objet. Son objectif est de réaliser l'ordonnancement des méthodes pour maintenir la cohérence structurelle de l'objet en assurant des propriétés comme l'exclusion mutuelle.

Le modèle des objets à contrôle réactif permet une conception plus claire en séparant les problèmes de synchronisation de ceux de la programmation des transformations. Il permet aussi une expression plus aisée de la synchronisation en permettant l'utilisation d'un langage de synchronisation, BDL [AUG 99b] (*Behavorial Description Language*) de haut niveau. La séparation répond aux exigences d'indépendance entre synchronisation et transformation nécessaires à la réduction des anomalies d'héritage.

Dans BDL, il existe trois types d'entités : les identificateurs de méthodes, les qualificateurs d'exécution de méthodes et les opérateurs d'ordonnancement. Ces opérateurs ont été adaptés du langage réactif asynchrone Electre [CAS 95]. Le contrôleur d'exécution assure trois fonctions au sein d'un objet concurrent :

[6] En fait le langage Marvin permet l'ajout de nouveaux comportements sous certaines conditions (dans les modules comportementaux d'un même niveau) et le moteur d'exécution vérifie à run-time la cohérence du système grâce à un algorithme d'ordonnancement [GON 79] qui recalcule l'ordre d'execution des modules.

1. La gestion des exécutions : le contrôleur crée, pour chaque méthode dont l'exécution est demandée, un fil d'exécution destiné à exécuter le code de la méthode.

2. La synchronisation des exécutions à partir du programme BDL associé au contrôleur. Pour réaliser cela, le contrôleur doit connaître en permanence l'état d'exécution des méthodes qu'il contrôle.

3. La mémorisation de demandes normales qui ne peuvent être satisfaites immédiatement. Dans un tel cas, la demande est mémorisée puis réitérée lorsqu'une méthode se termine car c'est à cet instant que les demandes en attente ont la possibilité d'être satisfaites.

Les synchronisations étant exprimées par l'intermédiaire d'un langage réactif, l'automate représentant le comportement peut être éventuellement visualisé, le fonctionnement de l'objet peut être simulé et il est possible de vérifier que des propriétés comportementales souhaitées sont satisfaites. Ces propriétés peuvent être des propriétés d'ordonnancement entre les méthodes de l'objet mais également des contraintes, entre méthodes, résultant de constructions comme l'héritage ou l'agrégation. L'utilisation d'un langage réactif apporte aussi une plus grande confiance dans le processus de conception dans la mesure où le code produit par les langages réactifs et sur lequel porte la vérification est le même que celui utilisé lors de l'exécution du programme.

Les programmes BDL sont traduits en programmes Esterel qui est utilisé comme langage cible car il offre des structures de contrôle de haut niveau et également pour la richesse de son environnement de développement (notamment son interfaçage avec différents outils de vérification). Cependant il a été conservé l'hypothèse d'asynchronisme du langage réactif Electre en ne traitant qu'un seul événement en entrée à la fois. La traduction des opérateurs BDL en Esterel et la réalisation du compilateur permettant l'obtention de ce module est décrite dans [BER 96].

Marvin

Marvin [RIC 01] est un langage de programmation de mondes virtuels habités par des agents. Le modèle d'exécution de Marvin est le modèle INVIWO (INtuitive VIrtual WOrdls) qui se compose de trois sous-modèles : 1) un modèle d'agent autonome (un ensemble minimal de composants), 2) un modèle d'avatar (un agent à part entière, plus ou moins contrôlé), et 3) un modèle de monde virtuel habité constitué uniquement d'agents communicants (agents autonomes et avatars).

Marvin est fortement inspiré de la syntaxe du langage ESTEREL, auquel ont été ajoutées des caractéristiques orientées-objets et orientées-agents permettant de décrire à la fois la structure et le comportement d'agents INVIWO et les données manipulées par ces agents.

Un agent est composé d'un ensemble d'attributs, d'un ensemble de composants comportementaux (capteurs, effecteurs et modules comportementaux) et d'un gestionnaire de modules comportementaux. L'agent centralise d'une part la réalisation des primitives d'actions concernant la modification de la structure et l'envoi de stimuli, d'autre part la réception des stimuli internes, externes et utilisateurs.

Le langage Marvin est programmé dans une architecture de sélection de l'action, basée sur des modules synchrones concurrents et un mécanisme d'arbitrage. Le processus d'un agent est composé d'un ensemble de modules comportementaux concurrents, chargés de proposer des actions à effectuer en fonction des stimuli perçus et de l'état courant de l'agent ; dans un deuxième temps, les arbitres choisissent les actions à réaliser à partir des propositions qui leur ont été faites.

Les agents INVIWO sont parfaitement autonomes, dans le sens où aucune entité ne peut modifier l'état interne d'un agent ou influencer son comportement sans son accord. Cela permet de décrire des mondes complètement répartis, dans lesquels les décisions sont prises localement au niveau de chaque agent. En contrepartie, la cohérence globale d'un monde INVIWO n'est pas garantie.

synERJY

synERJY est un environnement de programmation reactive développé au GMD en Allemagne. L'environnement a été créé dans le projet SYNC (Synchronous embedded software) formé par R. Budde, A. Poigne et K.H. Sylla. L'environnement de programmation propose deux langages de programmation synchrone orienté-objets, un éditeur graphique de machines d'états, un simulateur et un compilateur qui génère du code en AINSI-C pour

pouvoir l'exécuter sur plusieurs architectures. L'environnement peut s'intégrer à d'autres outils comme VIS et SMV.

Les deux langage de programmation synchrone orienté-objets offerts sont SynchronousEifel [Web SynEifel] et synERJY [Web synERJY] (SYNchronous Embedded Reactive Java plus un Y pour la phonétique). Ces langages combinent deux paradigmes : la programmation orientée objets pour une conception robuste et flexible et le paradigme synchrone pour une modélisation précise de comportements. Les deux langages soulignent l'intégration des comportements réactifs avec le modèle de données orientées objets.

synERJY utilise la majeur partie de la syntaxe de Java TM et il a été conçu pour générer du code efficace, en particulier pour des microprocesseurs. synERJY intègre plusieurs caractéristiques des formalismes synchrones comme la programmation impérative, la génération de machines d'état et les équations de flot de données.

6.7 Conclusion

Rejo est un langage de haut niveau qui permet aux programmeurs de construire facilement des systèmes réactifs. Les instructions de Rejo sont traduites en un langage de bas niveau (Junior), cachant ainsi la complexité de celui-ci.

Les avantages principaux de Rejo par rapport à Junior sont les suivants :

- La complexité de l'utilisation des conditions réactives (appelées configurations en Junior) est complètement cachée en Rejo ;
- Les instructions atomiques deviennent transparentes ; le programmeur peut directement utiliser les instructions Java comme des instructions réactives ;
- Les notions de méthode et de variable réactive sont introduites en Rejo. Les variables réactives (par exemple les IntegerWrappers de Junior) peuvent être utilisées de manière transparente en Rejo. Ainsi Rejo offre un mécanisme d'encapsulation.
- L'intégration de Junior et Java est faite de telle façon que l'héritage et le polymorphisme sont préservés. En particulier on peut parler d'héritage de comportements réactifs.
- Rejo, en tant que langage de haut niveau, permet de faire quelques optimisations sur le code généré ainsi que de détecter des bugs dans le moteur d'exécution.

En résumé, Rejo définit un modèle d'objets réactifs au-dessus des modèles d'exécution de Java et de Junior. Ce modèle fait partie de la famille de modèles d'objets réactifs qui utilisent une approche synchrone dynamique (comme Rhum et les Icobjs) ou le nombre d'événements, et de comportements réactifs n'est pas fixe. Ce modèle contraste avec les modelés d'objets réactifs qui utilisent une approche synchrone statique (comme Triveni et synERJY) ou le trait principal est l'élimination des problemes de concurrence par vérification statique.

Chapitre 7

ROS: architecture et agents

Ce chapitre présente l'utilisation du modèle réactif dans la construction d'un Système d'Agents Mobiles (SAM). Vu l'énorme quantité des systèmes d'agents mobiles qui ont été créés et les problèmes qui subsistent dans ce domaine, on a préféré, au lieu de créer un SAM intégrant les derniers gadgets, créer un SAM "minimal" qui servirait à tester l'apport du modèle réactif dans la construction d'agents mobiles. L'idée est de construire un système modulaire avec un minimum de services, pour ensuite l'étendre en utilisant le dynamisme du modèle réactif et la technologie orientée objet.
Le construction du SAM que l'on propose utilise Junior et Rejo. Rejo sert à programmer les agents mobiles et Junior à construire la plate-forme qui les accueille. Les agents que l'on va programmer avec Rejo sont ceux que l'on a présenté dans la section 1.2, c'est-à-dire des entités autonomes qui migrent entre deux SAM. Rejo et le SAM facilitent la programmation des agents autonomes : l'instruction Try de Rejo permet de récupérer des erreurs d'exécution et le SAM relance l'exécution des agents lorsque la migration échoue. La programmation de la migration des agents se fait de façon transparente car elle est déclenchée par la génération d'un événement particulière de migration. Cette façon de provoquer la migration par un événement permet de faire de la migration reactive (l'événement est généré en dehors du corps de l'agent) ou proactive (génération interne). Un des avantages de Junior pour implémenter la migration est que l'on obtient une migration forte parce que l'on migre également l'etat des instructions réactives.

La section 7.1 explique l'intérêt d'utiliser le modèle réactif dans la construction d'un SAM. La section 7.2 détaille l'architecture du système construit, appelé Ros (de l'anglais *Reactive Operating System*), et son API de programmation. La section 7.3 présente la structure des agents : mécanisme de contrôle et migration des agents ; on montrera les deux façons d'implémenter les agents 1) par un programme Rejo (méthodes réactives) ou 2) grâce à une nouvelle instruction qui simplifie la sémantique et améliore la vitesse d'exécution. La section 7.4 décrit les deux applications principales qui ont été créées : un Shell réactif (Rsh) et un environnement graphique de programmation réactive (Ricobjs). Finalement, on décrit dans les sections 7.5 et 7.6 les travaux similaires et les conclusions.

7.1 Introduction

De nombreux langages permettant l'interaction entre agents existent déjà depuis une dizaine d'années. L'association internationale FIPA (Foundation for Intelligent Physical Agents) tente actuellement de définir une normalisation, appelée ACL (Agent Communication Language). Cependant, de nos jours, des propositions de nouveaux langages utilisant différentes approches continuent à apparaître. Cet ensemble de langages a augmenté beaucoup avec l'arrivée du langage Java parce que Java offre la facilité de créer des objets et les faire migrer en utilisant RMI. Parmi les SAMs basés sur Java, on peut citer, entre autres : Concordia [WEB Concordia], Aglets [WEB Aglets] et Voyager [WEB Voyager].

Une des approches qui a montré sa puissance dans plusieurs domaines est l'approche réactive. Cette approche fournit un ensemble de primitives qui permettent la construction de systèmes, comme l'interaction entre des

agents qui réagissent à des événements. L'approche réactive offre un bon moyen pour créer des agents ; cependant, les langages réactifs existant (Esterel, Signal, Lustre, etc.) ne permettent pas la migration de code. De plus, on a besoin d'un langage qui propose des fonctions spécifiques pour la manipulation des agents.

Le premier essai pour construire un SAM dans le cadre réactif a été RAMA [NIK 99]. Cependant la programmation en RAMA est de bas niveau par rapport aux besoins liés à la présence d'agents. Le résultat est une programmation difficile qui ne cache pas les détails de l'implémentation de la plate-forme et qui laisse la responsabilité de l'affectation des noms d'agents au programmeur, ce qui peut générer divers problèmes, par exemple de sécurité.

Rejo est un nouveau langage réactif, plus précisément une extension à Java pour créer des Objets Java Réactifs (REactive Java Objets). On propose d'utiliser Rejo pour construire des agents mobiles réactifs. Les objets réactifs peuvent ainsi migrer de façon naturelle dans le cadre réactif et les fonctions nécessaires typiques d'Agent Mobile (comme l'affectation d'un nom) sont fournis par une plate-forme appelée ROS, décrite dans ce chapitre. Cette plate-forme exécute les objets et fournit les opérations typiques d'un SAM. La figure 7.1 montre la position de Rejo parmi les diverses approches utilisées.

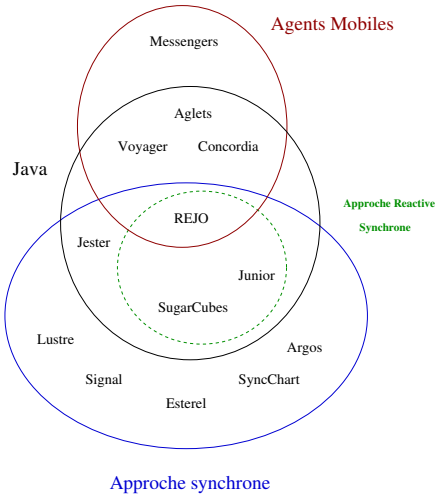

FIG. 7.1: Situation de Rejo par rapport aux objectifs

De même que les langages d'agents, les SAM [FUG 98] ont évolué vers des systèmes de plus en plus complexes. Parmi les caractéristiques que possèdent les SAM, on peut identifier un ensemble basique de propriétés communes : mobilité, nommage, communication (pour la collaboration), sécurité. Il existe une analogie forte avec les Systèmes d'Exploitation Répartis (SER)[TAN 95]. Un SER contient un administrateur de processus et un administrateur de processeurs pour diriger la migration des processus. La migration des processus peut être utilisée pour équilibrer la charge du système. De même, dans un SAM, un agent est un processus ayant la propriété de migration et la migration peut être utilisée pour équilibrer la charge du système (et éventuellement celle du réseau).

Le système RAMA a montré que l'approche réactive est bien adaptée à la construction des SAM. Cependant, l'architecture de RAMA ne permet pas d'ajouter facilement de nouvelles fonctionnalités. La principale raison est que son dessin est monolithique : il mélange l'interface graphique de l'utilisateur et la notion de groupe avec les composants réactifs.

Puisqu'un SAM est très proche d'un système d'exploitation, on a construit un système, ROS, basé sur le modèle actuel des SERs, c'est-à-dire, utilisant un ***micro-kernel***. Le micro-kernel implémente les fonctions basiques dont les agents ont besoin et permet d'ajouter des nouveaux modules pour augmenter sa fonctionnalité. Si on a besoin d'un petit kernel (rapide et qui utilise peu de mémoire), on peut enlever certains modules.

ROS est un système qui permet l'exécution des instructions réactives et en particulier l'exécution des objets Rejo. L'architecture de ROS est similaire à celle d'un SER et il a été implémenté avec Junior.

On va maintenant présenter un des problèmes principaux qui existent en Java pour implémenter la migration. Ce problème est qu'en Java on peut pas migrer les threads ; ceci veut dire que, lorsqu'on migre un agent, il migre sans son état d'exécution (migration faible). Plusieurs propositions décrites dans la suite ont été faites pour pouvoir également migrer l'état ; dans la sous-section 7.3.2, on présente la migration de nos agents.

La migration en Java

Il existe principalement trois approches pour aborder le problème de capture et de restauration de l'état des flots Java : une approche dite explicite, une approche dite implicite se basant sur un pre-processeur de code et une approche implicite qui étend la JVM.

La première approche, appelée gestion explicite, consiste à laisser entièrement à la charge du programmeur de l'application la gestion de la capture et de la reconstruction de l'état de son application. Le programmeur doit donc ajouter du code qui effectue le travail, en des points précis de l'application dits points de sauvegarde. En un point de sauvegarde, le code ajouté doit enregistrer l'état de l'exécution : il note la dernière instruction exécutée ainsi que l'état courant de l'application. Lors de la reprise de l'exécution, le premier traitement effectué est un branchement vers le dernier point de sauvegarde enregistré ; ainsi, l'exécution peut reprendre au point où elle a été interrompue. Cette solution manque de souplesse car la gestion des points de sauvegarde est entièrement à la charge du programmeur. De plus, l'ajout ou la modification de points de sauvegarde induit une modification de l'application elle-même. Cette gestion explicite est utilisée dans les applications faisant appel à des plates-formes à agents mobiles [CHE 95] fournissant une migration faible tels que les Aglets [WEB Aglets] ou Mole [WEB Mole].

Les deux autres approches, dites implicites, fournissent un service de capture/restauration de l'état des flots de contrôle. Le service fourni est indépendant du code de l'application et se présente sous la forme d'une primitive appelée par l'application elle-même ou par une application externe. Ces deux approches diffèrent par leur mise en oeuvre :

1. La première approche implicite consiste à fournir un pré-processeur qui injecte du code dans le code de l'application Java, soit dans le code source, soit dans le code intermédiaire Java. Lors de l'exécution de l'application, le code injecté crée un objet Java, que l'on nommera backup, et l'associe à l'application. Au fur et à mesure que l'application s'exécute, le code injecté sauvegarde dans l'objet backup l'état de l'exécution (les appels de méthodes et les valeurs des variables locales des méthodes). Lorsque l'application désire capturer son état, il lui suffit d'utiliser l'objet backup que le pre-processeur lui a associé. Pour restaurer l'état d'une application, les informations sauvegardées dans l'objet backup sont utilisées pour re-initialiser l'état de l'application : une re-exécution partielle de l'application est effectuée pour reconstruire la pile d'exécution et re-initialiser les valeurs des variables locales. Le principal intérêt de cette approche est qu'elle ne modifie pas la JVM. Mais son inconvénient est que, d'une part, elle induit un sur-coût non négligeable des performances de l'application (du au code injecté par le pré-processeur) [BOU 00c] et que, d'autre part, le coût de l'opération de restauration de l'état est important à cause de la re-exécution partielle de l'application. [FUN 98] propose une plate-forme à agents mobiles se basant sur un pré-processeur qui instrumente le code source des applications Java et [TRU 00] ou encore [SAK 00] proposent des implantations de mécanismes de capture d'état de flots de contrôle Java basés sur un pré-processeur qui instrumente le code intermédiaire de l'application.

2. La seconde approche implicite consiste à étendre la machine virtuelle Java pour en extraire l'état des flots et le rendre accessible par les programmes Java. Cette extension doit permettre de capturer l'état courant d'un flot et de le stocker dans un objet Java. L'extension doit également permettre de créer un nouveau flot et de l'initialiser avec un état préalablement capturé. Un service de capture/restauration construit en suivant cette approche n'est certes utilisable que sur des machines virtuelles étendues. Nous avons tout de même choisi cette approche pour les deux raisons suivantes :

   (a) Elle réduit le sur-coût induit par le service sur les performances de l'application (pas de code injecté) et elle réduit le coût du service de capture/restauration car sa mise en oeuvre dans la machine virtuelle Java est principalement native (en C).

   (b) Du fait que ce service possède plusieurs domaines d'applications, nous pensons que c'est un service de base que doit être intégré à la JVM.

Le mécanisme de migration que l'on va utiliser appartient, en quelque sorte, aux deux classifications ; il est du premier type car on insère du code (les instructions Stops) pour sauver l'état de l'agent. Le code est inséré par le programmeur et non par un pre-processeur mais le mécanisme est également du deuxième type car il nécessite une machine virtuelle spéciale (la machine réactive de Junior) pour exécuter le code et effectuer la capture de l'état (avec freezable qui place la copie dans la machine) et sa restauration (add dans la machine).

Pour plus d'information sur les mécanismes de migration de threads en Java, le lecteur peut lire [TRU 00].

7.2 Architecture

ROS a été construit en utilisant une philosophie *micro-noyau*. Cette philosophie a permis de construire un prototype minimal qui est constitué par trois modules : le module ***Engine Daemon*** (Démon de la machine) qui fait réagir la machine réactive, le module ***Migration Daemon*** (Démon de migration) qui permet la charge distante de nouveaux Rejos (des agents) dans le système, et le ***Kernel*** (Noyau) qui gère les Rejos (les charge localement, les élimine, etc.). La figure 7.2 montre cette architecture.

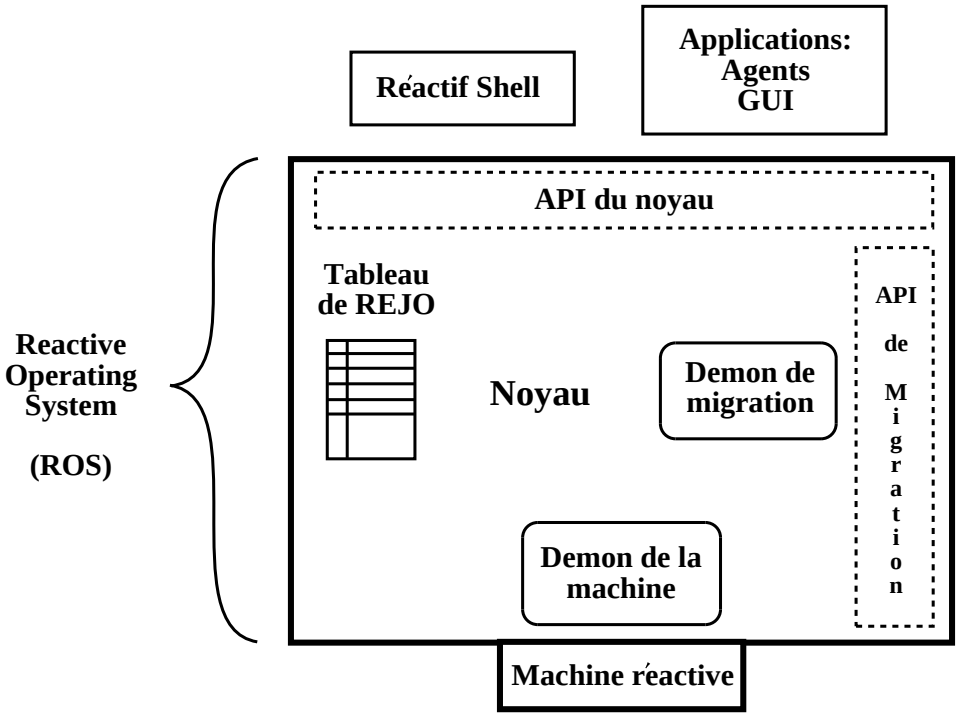

FIG. 7.2: Architecture de ROS

Les Rejos sont gérés par une table et un identificateur local unique qui fournit un nommage local. Cet identificateur est l'équivalent du PID (Process Id) d'UNIX. Pour faciliter l'usage du système, trois interfaces ont été construites : l'interface `API_kernel` qui décrit la façon d'utiliser le système (ajouter et enlever des Rejos), l'interface `MigrationServ` qui offre le service de migration et l'interface `Agent` qui définit un ensemble minimal de méthodes que doivent posséder les Rejos pour être exécutés en ROS.

Démarrage du système et utilisation

Le démarrage du système ROS se fait avec un script appelé `run` qui simplifie l'exécution en passant les bons paramètres, tels le CLASSPATH et la configuration du serveur RMI (voir figures 7.3).

Le système Ros démarre chaque module du système ; si un des modules basiques (le kernel et le démon de la machine) ne peut pas être exécuté, le système retourne un code d'erreur à l'utilisateur. Par contre, si le module

de migration ne peut pas être démarré, le système affiche un message d'erreur mais démarre tout de même, sans le service de migration.

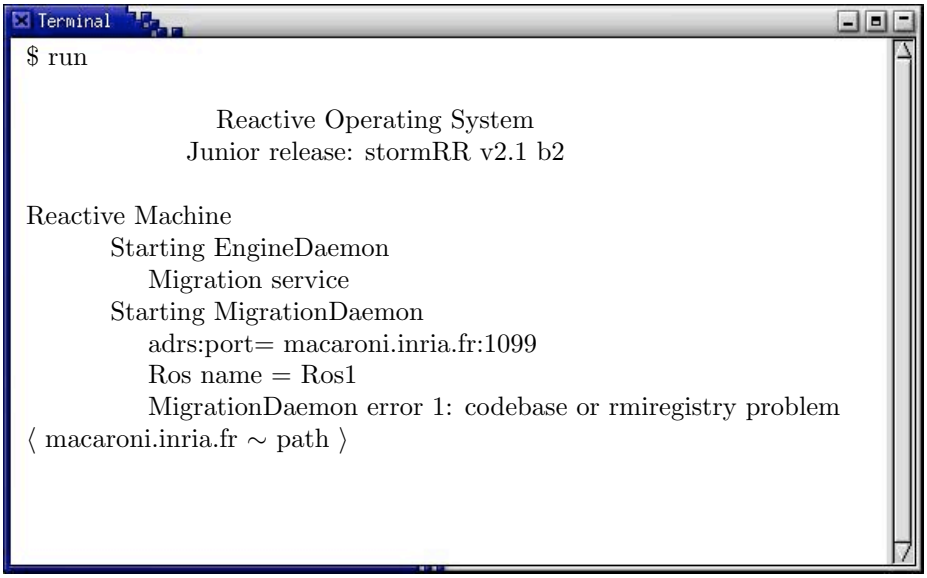

FIG. 7.3: Démarrage du système ROS

La commande `run` accepte quelques options qui permettent de lancer le noyau seul ou avec quelques applications. La table 7.1 décrit les options existantes. Si aucune option n'est donnée, le système démarre avec les options `-k` et `-t`, c'est-à-dire avec le noyau et ses modules ainsi qu'une application, le Rsh, décrite dans la section 7.4.1.

Option	Signification
-k	lance le Noyau
-g	lance l'Interface Graphique
-r	lance les Ricobjs
-t	lance le Shell réactif

TAB. 7.1: Les options de démarrage de Ros

Finalement, ROS peut être lancé à l'intérieur d'autres programmes (*embedded systems*) sans problème car il n'impose aucune interface graphique (comme le fait RAMA), ni la présence du service de migration. ROS peut utiliser une interface graphique en Tcl/Tk ou bien peut être inclus dans une page Web en utiliser le Browser comme interface. Dans la sous-section 7.4.2, on présente une application (les Ricobjs) qui tourne sur ROS et qui utilise des applets pour montrer graphiquement des comportements réactifs ; les applets peuvent être utilisés dans un browser comme Explorer ou par une application à part entière.

API

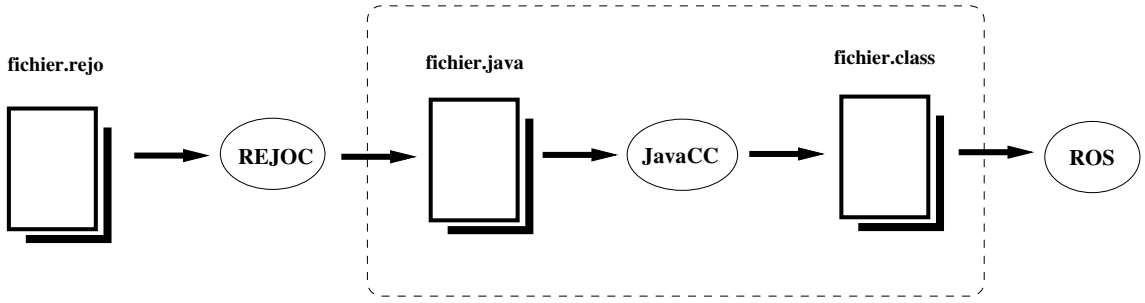

FIG. 7.4: Compilation et exécution d'un agent

Le système ROS offre 4 services (définis dans l'interface `API_kernel`). Ces services équivalent aux *system calls* d'un système d'exploitation et ils sont les suivants :

1. La gestion d'Agents (charge et décharge d'agents).

   ```
   int load(Agent ro);
   int load(Agent ro, Output o);
   int load(Agent ro, String[] args, Output o);
   ```

   Charge l'agent `ro` dans le système en lui affectant un nom. L'agent est chargé à l'instant d'après et, s'il a déjà été exécuté auparavant, on charge juste son résidu. Le paramètre `args` est une copie des variables que reçoit le système au démarrage.

   Lorsque le paramètre `Output` est présent, il est utilisé comme une deuxième sortie standard. L'agent peut l'utiliser dans le code réactif par les deux instructions atomiques d'impression `print` et `println` qui sont équivalentes aux instructions `System.out.print` et `System.out.println`. Pour pouvoir les utiliser sur une plate-forme ROS, il faut que l'agent implémente l'interface `Output`, ce qui est toujours le cas lorsque on compile avec l'option -ros.

   ```
   void loadInst(Program i);
   ```

   Charge un programme Junior dans la machine. Le programme n'a aucune structure de contrôle et donc n'a pas de nom non plus

   ```
   void unLoad(Agent ro);
   Program unLoad(String name);
   ```

   Élimine un agent en utilisant une chaîne `name` ou l'objet `ro`. L'élimination consiste en la génération de l'événement `kill` et l'élimination du tableau des agents.

   ```
   Hashtable getTable();
   ```

   Retourne une copie du tableau des agents du système. Le tableau est de type `Hashtable` de Java et il donne l'état du système à l'instant i-1 (ou i est l'instant d'exécution), plus les agents qui ont été chargés dans l'instant courant, avant l'appel de la méthode.

2. Interaction avec la machine réactive. Il offre la même API que la machine réactive de Junior, plus les méthodes suivantes :

   ```
   void generate(String name);
   void generate(String name, Object val);
   ```

   Génère un événement `name` dans la machine. L'événement peut être un événement valué qui utilise l'objet `val`. Cette génération implique l'utilisation du `StringIdentifier` défini dans le langage Rejo.

   ```
   void suspend();
   void resume();
   ```

   Ces méthodes suspendent et reprennent l'exécution du démon de la machine.

3. Quelques méthodes pour obtenir des informations sur l´état de la machine.

   `int getInstant();`

   Retourne le nombre d'instants au cours desquels la machine réactive a été activée.

   `int getRejos();`

   Retourne le nombre d'agents présents dans le système.

   `String getName();`

   Retourne le nom du système. Ce nom est utilisé par le service de migration et il sert à différencier plusieurs systèmes ROS tournant sur la même machine physique. Ce nom est donné par l'utilisateur au moment du démarrage avec l'option `-n`.

4. Un service de migration offert par le démon de migration qui exécute les appels de la méthode :

   `int Agent(Agent ag) throws java.rmi.RemoteException;`

   Cette méthode charge l'agent `ag` en utilisant la méthode `load(ag, ros);` où `ros` est un identifier du système ROS qui reçoit l'agent.

7.3 Les Agents

Un Agent en ROS n'est pas autre chose qu'un Rejo avec une structure particulière. Deux conditions cependant doivent être respectées pour pouvoir exécuter un programme Rejo sur la plate-forme ROS :

1. Une des méthodes réactives doit s'appeler `rmain`. Celle-ci joue le même rôle que la méthode `main` d'un programme Java, c'est-à-dire que c'est la méthode à laquelle le système donne le contrôle initialement ;
2. Le programme Rejo doit implémenter l'interface `Agent`. Cette interface définit l'API de l'agent, par exemple c'est elle qui spécifie que l'agent doit posséder la méthode `rmain` mais aussi d'autres méthodes pour récupérer des informations indispensables comme le nom de l'agent.

La plate-forme ROS offre deux services principaux aux agents Rejo : un service de nommage local et une structure de contrôle. La structure de contrôle permet de faire trois choses :

1. Contrôler l'exécution de l'agent. Le contrôle consiste à pouvoir le tuer, le suspendre et le résumer.
2. Effectuer deux opérations sur le corps de l'agent : 1) lui ajouter un nouveau comportement en parallèle, et 2) sauvegarder l'état d'exécution. En fait, il s'agit de la sauvegarde de tout l'agent, c'est-à-dire de son état d'exécution et de ses données.
3. Migrer l'agent. Cette opération consiste à envoyer une copie de l'agent à un site distant et à l'enlever localement. La migration est expliquée dans la section suivante.

Toutes ces opérations sont implémentées dans une méthode réactive appelée `shell` qui est exécutée par le système ROS lors de la charge d'un agent, par la méthode `load`. Les opérations de contrôle sont implémentées à l'aide de signaux (de type `String`) qui utilisent la convention suivante :

$$nom\ de\ l'agent\ +\ !\ +\ nom\ de\ l'opération$$

ainsi un agent qui s'appelle toto aura les signaux de contrôle décrits dans la figure 7.2.

Le signal de migration doit être un signal valué dont la valeur contient l'adresse du site distant où l'agent veut migrer. Les signaux `kill` et `migra` ont une sémantique particulière : ils exécutent du code Java pour permettre

Signal	Valeur	Action
"toto!susp"		Suspend l'agent.
"toto!resume"		Résume l'exécution de l'agent suspendu.
"toto!kill"		Tue l'agent.
"toto!migra"	Adresse	Migre l'agent.
"toto!ser"	Nom du fichier	Sauve l'état de l'agent.
"toto!add"	Programme	Ajoute un comportement réactif à l'agent.

TAB. 7.2: Signaux d'un agent en ROS

Signal	Java Méthode
"toto!kill"	`termin()`
"toto!migra"	`freeze()`: exécuté avant la migration. `warmup()`: exécuté avant d'exécuter le code réactif dans le site distant.

TAB. 7.3: Méthode Java et sa sémantique

que l'agent fasse une dernière action[1], par exemple libérer les ressources qu'il a alloué (les réinitialiser ou les détruire). La table 7.3 montre les méthodes Java exécutées lorsque les signaux `kill` et `migra` sont générés.

Etant donné qu'il existe 6 événements de contrôle, la table 7.2 n'est pas suffisante pour décrire le comportement d'un agent. En fait, il nous faut donner une description du comportement réactif pour chaque combinaison possible de la génération de deux ou plus événements de contrôle au cours d'un même instant. La table 7.4 décrit le comportement d'un agent comme un automate qui change d'état (au cours du même instant ou entre deux instant) lorsque les différents événements de contrôle sont générés. La description est exhaustive et très compacte car plusieurs cas sont traités de la même manière. Dans ce tableau, 3 caractères sont utilisés pour chaque événement : `G` qui veut dire que l'événement a été généré, `-` qui veut dire que l'événement n'a pas été généré, et `x` qui veut dire qu'il n'est pas important de savoir si l'événement a été généré ou pas pour prendre la décision. Par exemple, la première ligne indique que si l'événement de contrôle `kill` est généré à l'instant i, l'agent passera à l'état kill (état final qui veut dire tué) dans le même instant ou au prochain instant. Cette décision d'arrêter le comportement est prise indépendamment de la présence ou de l'absence des autres événements.

7.3.1 Implémentation

La structure actuelle d'un agent est trop complexe pour la présenter et l'expliquer dans une seule section. On va donc présenter la structure par étapes pour faciliter sa compréhension et s'abstraire de quelques détails peu importants.

La structure de contrôle d'un agent a été codée, principalement, avec une méthode réactive appelée `shell`. Dans cette méthode on peut identifier un programme réactif comme celui donné dans le listing 7.1.

```
1    local( "step" )
```

[1]Cette dernière action est similaire à celle qui a été définie en SugarCubes sous le nom de Notification. En SugarCubes, il existe des notifications pour la terminaison, le gel et le réveil d'une instruction, ainsi que pour le début et la fin d'instant. Les notifications de SugarCubes sont implémentées par des instructions réactives et non , comme en ROS, par des actions atomiques.

Etat i	susp	res	kill	add	ser	migra	Nouvel état	Action
run	x	x	G	x	x	x	kill (i ou i+1)	termin()
run	G	x	-	x	x	x	susp (i+1)	
run	x	x	-	G	G	G	freeze (i+1)	
susp	x	x	G	x	x	x	kill (i ou i+1)	termin()
susp	x	G	-	x	x	x	run (i)	
susp	G	-	-	x	x	x	susp (i+1)	
freeze	x	x	G	x	x	x	kill (i) ou run (k) sur machine distante	termin() et/ou freeze() et warmup()
freeze	x	x	-	x	x	x	run (i) ou run (k) sur machine distante	termin()

TAB. 7.4: Exécution des événements

```
2       until( "kill" )
3         par
4         {
5           loop{
6             until( "susp")
7               loop{
8                 gen "step";
9                 stop;}
10            handler{
11              stop;
12              wait "resume";
13            }
14          }
15        ||
16          control( "step" ){
17            call body();
18            gen "kill";
19          }
20        }
```

Listing 7.1: Contrôle de base d'un agent

Ce programme implémente les trois comportements de base : suspendre, résumer et tuer un comportement réactif. Les signaux de contrôle sont implémentés de la façon suivante :

- Pour tuer l'agent on utilise une instruction Until associée à l'événement `kill` (ligne 2).
- Pour suspendre et résumer l'agent, on utilise un événement auxiliaire appelé `step`. Lorsque l'événement est généré (ligne 8), l'agent s'exécute un instant (ligne 16). Pour suspendre l'agent, on arrête de générer le signal `step` (ligne 6). Pour résumer l'exécution, on attend la génération de l' événement `resume` (ligne 12) avant de recommencer à générer l'événement `step` à chaque instant (lignes 5 et 7).
- Lorsque le programme réactif termine normalement, on se débarrasse de la structure de contrôle qui reste (ligne 18 qui tue la première branche du Par).
- Pour ne pas avoir d'interférences avec l'événement auxiliaire `step`, on le déclare local (ligne 1).

Pour exécuter les méthodes Java qui ont une sémantique spéciale, on les invoque dans le code précèdent ; par exemple, la méthode `termin()` est invoquée comme un atome qui s'exécute lorsque l'instruction Until (ligne 2) termine et que l'on exécute son handler (qui n'est pas écrit).

Maintenant voyons comment modifier le programme précédent pour ajouter le comportement réactif qui permet la migration de l'agent. L'algorithme réactif peut être implémenté de deux façons : récursivement ou itérativement. La figure 7.5 présente l'implémentation récursive (à gauche) et l'implémentation itérative (à droite) ; ces codes remplacent l'instruction Control du listing 7.1. Dans les deux cas, la migration est implémentée par deux comportements : un qui attend l'événement de migration et qui après avoir récupéré le résidu essaie d'envoyer une copie, et un autre comportement chargé de ré-exécuter l'agent en cas d'échec. En fait, toute la différence entre ces deux mécanismes réside dans la façon de recharger le code en cas d'échec. Un élément clé dans la recharge est la méthode `body()`, qui n'est pas une méthode réactive codée en Rejo mais qui rend le programme réactif de l'agent (stocké dans une variable de l'agent).

Voici une description des ces deux algorithmes :

- L'implémentation récursive utilise une méthode réactive (la méthode `reLoad()` dans la troisième branche du Par) qui recharge l'agent en recréant la structure qui gèle l'agent (première branche de l'instruction Par de la méthode `reLoad`) et en se recréant elle-même (deuxième branche de la méthode `reLoad`) pour de futures tentatives de migration.
- L'implémentation itérative ré-exécute l'agent en réutilisant la même structure avec une instruction Loop. La particularité de cet algorithme est la façon dont on termine le programme lorsque l'agent finit normalement ; pour cela on utilise la variable `end` qui prend la valeur de `true` quand l'agent finit normalement et `false` quand l'agent est gelé. Cette particularité va nous être très utile dans la suite car elle va permettre d'éliminer un problème qui apparaît à cause de l'instruction `gen "kill"`' présente dans le corps de l'instruction Freezable de l'implémentation récursive.

Dans les deux algorithmes, on exécute la méthode `migrateTo()` qui se charge d'envoyer une copie de l'agent (par RMI) et la méthode `freeze()` qui est exécutée quand la migration a lieu sans problème.

Voyons maintenant comment modifier le code pour ajouter de nouveaux comportements réactifs et pour les sérialiser. L'ajout dynamique et la sérialisation vont générer des modifications importantes dues à deux problèmes : 1) plusieurs opérations doivent être effectuées sur le code de l'agent qui est en train d'être exécuté, et 2) ces opérations doivent être effectuées dans un ordre particulier. En fait, il nous faut construire un code réactif de façon non-structurelle, autrement dit on a besoin d'un mécanisme qui gère toutes les opérations ensemble et pas plusieurs mécanismes ou instructions. Par exemple, l'ajout dynamique (implémenté avec l'instruction Dynapar) et la migration (implémenté avec l'instruction Freezable) peuvent entrer en conflit. Voici les deux cas possibles :

1. `dynapar("eve1", freezable("eve2", body()))`

   lorsqu'on ajoute des nouveaux comportements réactifs à l'agent, on ne va pas les migrer car ils seront en dehors de l'instruction freezable. On pourrait ajouter les comportement réactifs avec une instruction freezable pour les migrer, mais cela impliquerait qu'un utilisateur lambda doit connaître les détails de l'implémentation (pour chaque version) pour migrer également les ajouts (bien sûr, si c'est ce qu'il veut faire).

2. `freezable("eve2", dynapar("eve1", body()))`

   lorsqu'on migre l'agent, on va migrer son corps et aussi l'instruction Dynapar qui fait partie de la structure de contrôle. Le problème est que le mécanisme de migration que l'on implémente, migre uniquement le code de l'agent et pas la structure de contrôle qui est recréée à chaque fois que l'agent est chargé. On a choisi de migrer juste le code de l'agent pour 1) réduire le temps de migration (on ne migre pas la structure constituée d'à-peu-près 100 instructions réactives), et 2) éviter de futurs conflits à cause de la structure choisie ; l'agent pourrait migrer (ou être exécuté à partir de sa sauvegarde en disque) sans problèmes entre deux versions différentes du système ROS.

```
control( "step" ){
  par
  {
    freezable( "migra" ){
       call body();
       gen "kill";
    }
  ||
    loop{
       wait "migra", obj;
       if( migrateTo(obj) ){
            freeze();
            gen "kill";
       }else
            gen "reload";
       stop;
    }
  ||
    wait "reload";
    run reLoad();
  }
}
```

```
public reactive reLoad()
{
  par
  {
     freezable("migra"){
        call body();
     }
  ||
     stop;
     wait "reload";
     call reLoad();
  }
}
```

a) Implémentation récursive

```
control( "step" ){
  par
  {
     loop{
        end=true;
        freezable("migra"){
           call body();
        }handler{
           wait "reload";
           end=false;
        }
        if(end){
          gen "kill";
          stop;
        }
     }
  ||
     loop{
       wait "migra", obj;
       if( migrateTo(obj) ){
          freeze();
          gen "kill";
       }else
          gen "reload";
       stop;
     }
  }
}
```

b) Implémentation itérative

Fig. 7.5: Migration

Une autre solution possible au problème de la migration de l'instruction Dynapar serait de l'éliminer avant de charger l'agent. Ceci pose le problème de pouvoir analyser un programme réactif indépendamment de l'implémentation, pour extraire le corps de l'instruction Dynapar. Jusqu'à maintenant, il n'existe pas d'interface de programmation pour faire ce genre de choses. Il existe aussi un autre problème à résoudre pour distinguer si la provenance de l'agent est locale ou distante. En effet, dans le cas d'une provenance locale, il ne faut pas éliminer l'instruction Dynapar si elle apparaît dans le corps de l'agent.

La solution adoptée consiste à utiliser l'instruction Freezable avec un algorithme itératif et un événement de contrôle. Le listing 7.2 montre le code de la nouvelle implémentation de l'instruction Control. La première branche est la même que celle que l'on a donné dans la figure 7.5 en changeant l'événement de contrôle `"migra"` par `"ctl"` (ligne 6). L'événement de contrôle est généré lorsqu'on veut faire une des opérations qui demande le gel du corps de l'agent (dernière branche du Par, ligne 37) ; si un de ces événements est généré, on génère également l'événement de contrôle (ligne 38) et à l'instant d'après (ligne 39) on traite tous les événements dans un ordre particulier. Le mécanisme pour traiter les événements dans un ordre particulier est simple mais volumineux ; le problème est l'impossibilité de savoir, à partir du moteur d'exécution, si un événement a été généré dans l'instant précédent. La 2ème, 3ème et 4ème branches du Par servent à détecter la génération des événements qui sont ensuite traités à l'instant d'après, par la suite d'instructions If dans la 5ème branche du Par. Cette suite définit l'ordre dans lequel on traite les événements, autrement dit, lorsque ces événements sont généré simultanément, on fait l'ajout d'abord, ensuite la sérialisation, et à la fin la migration. L'ordre que j'ai choisi est basé sur l'utilité des opérations mais a aussi un aspect arbitraire. Par exemple, il ne fait pas de sens de migrer l'agent, puis d'ajouter un comportement à quelque chose qui va être détruit. Par contre, l'opération de sérialisation peut parfaitement commuter avec l'ajout et la migration, en produisant des résultat différents, Dans ces cas, il importe que ces opérations soient effectuées de manière déterministe. Le traitement

des événements finit, en général, par la génération de l'événement `"reload"` qui sert à reprendre l'exécution normale ou en cas d'échec dans la migration.

Pour finir, on va présenter le code qui est utilisé pour récupérer les erreurs ; ce code, donné dans le listing 7.3, fait partie de la structure qui implémente les trois opérations de base (on a écrit les instructions Local et Until de la structure). Le code sert à :

1. Exécuter la méthode `warmup()` si l'agent est chargé après avoir migré.
2. Exécuter la méthode `termin()` lorsque l'agent termine. On exécute aussi la méthode `unLoad()` qui élimine alors l'agent du tableau d'agents du noyau de Ros.
3. Eviter la propagation des erreurs de l'agent. Pour cela on utilise l'instruction Try-catch qui attrape les exceptions et qui exécute également les méthodes `warmup()` et `termin()`.

Tous les événements utilisés jusqu'à maintenant sont précédés du code `locName()+` pour les distinguer de ceux des autres agents. On pourrait aussi les construire on utilisant une syntaxe du type :

```
locName() + "!event" || groupeName() + "!event"
```

pour implémenter une notion de groupe.

Les invocations des méthodes `freeze()`, `warmup()` et `termin()` sont aussi précédées de la méthode `This()` qui est utilisée pour distinguer le `this` du Rejo auxiliaire qui implémente la structure de contrôle, de celui qui implémente le code de l'agent.

7.3.2 Migration

Tous les mécanismes de migration en Junior sont basés sur les mêmes principes suivants :

1. Les instructions réactives sont implémentées comme des objets Java, ***objets-instruction***.
2. Un programme réactif est donc un ensemble d'objets-instructions.
3. L'exécution d'un programme réactif altère les données des objets-instructions. Les données des objets-instructions peuvent être vues comme le *program counter* (compteur-programme) qui décrit la prochaine instruction à exécuter.
4. La migration du code réactif (migration forte) signifie la simple copie des objets-instructions locaux dans la machine distante. Cette opération de copie distante est faite en Java par RMI.

Les mécanismes de migration qui ont été créés en Junior et SugarCubes reposent sur ces quatre principes et définissent : 1) celui qui fait la copie, l'agent lui-même ou une entité externe, 2) quand la fait-il, inter-instant ou intra-instant, 3) quels protocoles sont utilisés, RMI ou sockets, plus le protocole qui demande la migration au site d'accueil. Tous ces sujets sont traités dans la suite.

Pour implémenter la migration en Junior il existe deux options :

1. Construire l'agent avec le minimum nécessaire pour implémenter un mécanisme de migration quelconque. Autrement dit, l'agent contient juste l'instruction Freezable.
2. Construire l'agent avec l'instruction Freezable et un ensemble d'instructions qui implémentent un mécanisme particulier de migration : le protocole de haut niveau (qui repose sur un protocole de bas niveau comme TCP ou RMI) entre le site distant et le site local.

```
1  control( "step" ){
2    par
3    {
4       loop{
5          end=false;
6          freezable( "ctl" ){
7             call body();
8          }handler{
9             end=true;
10            wait "reload";
11         }
12         if(!end){
13            gen "kill";
14            stop;
15         }
16      }
17   ||
18      loop{
19         wait "add";
20         adding = true;
21         stop;
22      }
23   ||
24      loop{
25         wait "ser";
26         serializing = true;
27         stop;
28      }
29   ||
30      loop{
31         wait "migra";
32         migrating = true;
33         stop;
34      }
35   ||
36      loop{
37         wait "add" || "ser" || "migra" ;
38         gen  "ctl";
39         stop;
40         if( adding ){
41            adding(env);
42            gen "reload";
43         }
44
45         if( serializing ){
46            serial(env);
47            gen "reload";
48         }
49
50         if( migrating ){
51            if( migrateTo(env) ){
52               freeze();
53               gen "kill";
54            }else{
55               gen "reload";
56            }
57         }
58      }
59   }
60 }
```

Listing 7.2: Code de migration, sérialisation et ajout

```
1  try{
2    if( migra() )
3      warmup();
4    local("step")
5       until("kill"){
6           ...
7
8       }handler{
9           done=true;
10          termin();
11          unLoad(This());
12      }
13 }catch{
14   if( !done )
15     try{
16        termin();
17        unLoad(This());
18     }catch{ ; }
19   }
20   termin();
21   unLoad(This());
22 }
```

Listing 7.3: Traitement d'erreurs

Les premières implémentations de Rejo implémentaient la migration à partir de la deuxième option car celle-ci offre deux avantages. Elle permet de 1) Expérimenter facilement avec plusieurs mécanismes de migration qui devient alors un mécanisme orthogonal à l'exécution du Rejo, et 2) Migrer un Rejo qui n'a pas été conçu pour migrer. L'avantage est de pouvoir réutiliser du code qui n'a pas été créé pour la migration mais l'inconvénient est que pour la même raison le code réutilisé n'est pas toujours conforme au format requis, c'est-à-dire qu'il n'est pas toujours sérialisable.

Néanmoins, cette option s'avère difficile à utiliser : l'utilisateur doit en effet créer son propre mécanisme de migration ou en utiliser un déjà créé ; dans les deux cas il faut ajouter le programme dans la machine pour déclencher la migration. Pour cette raison, l'implémentation actuelle des agents crée les agents avec tout ce qu'il faut pour migrer.

Implémentation

Voici maintenant un récapitulatif de l'implémentation de la migration à base de RMI. Le code de la migration qui effectue l'appel distant se trouve dans la méthode `MigrateTo(...)` donné dans le listing 7.3.2.

La suite d'actions pour migrer un agent est la suivante :

1. Génération de l'événement qui déclenche la migration, par exemple :
   `generate "toto!migre", "polka.inria.fr/Ros1";`
   Comme chaque Rejo a un nom unique (ici toto), on obtient une migration sélective. L'adresse est composée du nom de la machine (polka.inria.fr) plus le nom du site Ros (Ros1 par défaut) séparés par le caractère "/".

2. Gel du code de l'agent. Le gel est fait grâce à l'instruction réactive `freezable(String)` qui retire de la machine les instructions réactives qui restent à exécuter.

3. Exécution du code réactif en charge de la migration (méthode `migrateTo`). On commence par la récupération de l'adresse de destination (lignes 5-11 de la méthode `migrateTo`).

4. Enlèvement des instructions réactives de la machine (ligne 13 de la méthode migrateTo). Le noyau de ROS appelle la méthode `getFrozenIntruction (String)` qui rend une copie des instructions réactives gelées qui se trouvent dans la machine.

5. Gestion de quelques erreurs et préparation de l'agent : on stocke le code réactif et on appelle la méthode `setMigra(true)` pour noter que l'agent est chargé après une migration.

6. Envoi d'une copie du Rejo à la machine distante (ligne 23 de la méthode `migrateTo`). On fait un appel distant (RMI) en envoyant l'objet comme paramètre.

7. Si la migration échoue, l'agent est rechargé dans le système, comme expliqué dans la section précédente.

8. Sinon, on exécute la méthode `freeze()` et on enlève les instructions de contrôle du Rejo (`gen "kill"`).

9. Chargement de l'agent dans la machine distante. Cette action est effectuée par la méthode `Load(Agent)` qui fait partie du démon de migration du noyau.

```
1  public boolean migrateTo(Object[] obj)
2  {
3   String dest="", nameRos="";
4
5     if( obj.length < 2 ){
6       System.out.println("Unknown address");
7       return false;
8     } else {
9       dest = (String)obj[0];
10      nameRos = (String)obj[1];
11    }
12
13    Program res = ros.unLoad( locName() + "!ctl" );
14    if( res == null){
15      System.out.println(" Empty program ");
16      return false;
17    }
18    setBody( res );
19    setMigra(true);
20    try{
21       MigrationServ SendingThe =(MigrationServ)lookup
22            ("//"+dest+"/MigrationServAt "+ nameRos );
23       if( SendingThe.Agent(rejo) )
24        return false;
25    }catch(Exception e) {
26      System.out.println("\n Migration Error of "+locName() +
27                         " To "+ dest +" "+ nameRos +"\n");
28      System.out.println(" "+ e.getMessage());
29      e.printStackTrace();
30      return false;
31    }
32    return true;
33 }
```

La figure 7.6 illustre les étapes de la migration par rapport aux instants de la machine :

Si la migration se déroule sans problème, elle s'effectue en deux instants et non en trois lorsque l'agent a juste l'instruction Freezable (il faut un instant supplémentaire pour charger le programme qui implémente la migration).

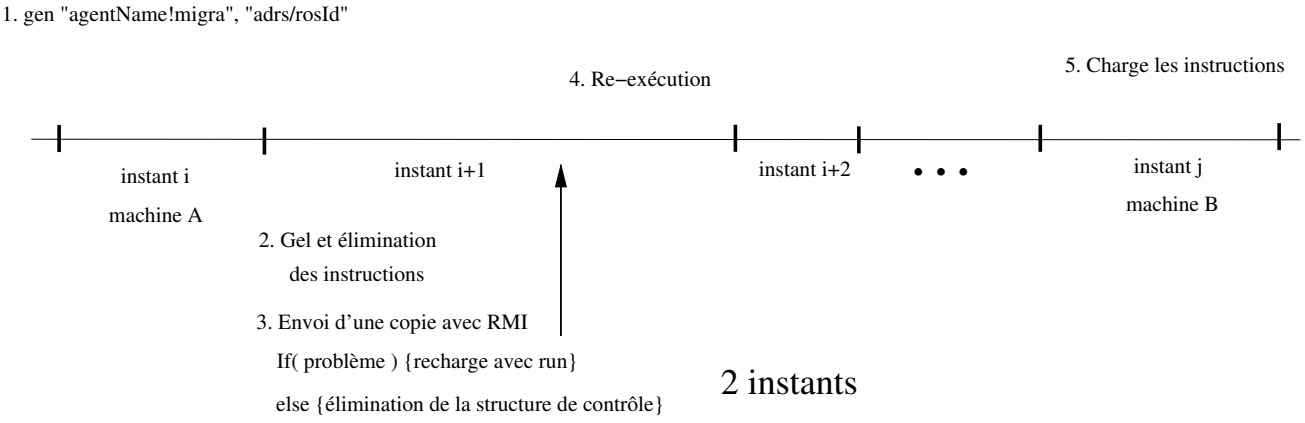

FIG. 7.6: Etapes de la migration

7.3.3 Deuxième implémentation des agents

Dans les sections précédentes on a montré comment on peut implémenter un agent mobile en utilisant Rejo ; cette implémentation présente cependant quelques problèmes :

1. La structure est assez complexe et il n'est pas facile de savoir, par exemple, ce qui ce passe lorsque ont été générés au même instant deux ou plusieurs événements de contrôle. La complexité provient en particulier du fait que la réaction à certains événements prend deux instants.
2. La structure est composée de plusieurs instructions dont certaines sont exécutées à chaque instant. Cette structure est donc très gourmande en ressources et elle passe mal à l'échelle. Par exemple, si on ne considère que la structure qui permet de suspendre, résumer et tuer un comportement réactif, on voit qu'elle est composée de 15 instructions dont 11 sont exécutées régulièrement.

Dans cette section on va présenter une nouvelle instruction réactive, appelée `ReactiveThread`, qui implémente un comportement très proche de celui défini par la méthode `shell` d'un agent. L'idée est de définir une instruction qui ait une sémantique plus claire et qui soit plus efficace. La nouvelle instruction est définie formellement en utilisant des règles de réécriture (voir la table 7.6); dans la sémantique on utilise ***rth*** au lieu de `ReactiveThread` et il y a deux instructions auxiliaires ***rtha*** et ***rths***. L'instruction `ReactiveThread` est décrite informellement dans la suite.

L'instruction `ReactiveThread` définit une sorte de thread réactif qui réagit à quatre événements : un événement pour le suspendre, un événement pour le résumer, un événement pour le tuer, et un événement pour lui ajouter un comportement en parallèle, qui sera aussi sous le contrôle des trois événements précédents. Si aucun événement de contrôle n'est généré, on exécute régulièrement le corps du thread ; si un ou plusieurs événements sont générés au cours d'un instant, on attend la fin de l'instant et, à ce moment, on prend une décision. Vu comme un automate, le thread réactif change d'état seulement entre les instants et non au cours d'un même instant (comme le fait un Agent en Ros). Le thread réactif change d'état en analysant tous les combinaisons possibles (qui nous intéressent) ; la table 7.5 montre la table des transitions de l'automate qui décrit le comportement du thread réactif.

Le thread réactif se distingue d'un agent en Ros par les points suivants :

1. On peut ajouter des nouveau comportement réactifs à un thread réactif même lorsqu'il est suspendu. Cette opération est implémentée par un simple test (représenté par une règle de ré-écriture dans la sémantique) mais en Ros son implémentation serait beaucoup plus complexe.

2. La terminaison d'un thread réactif a toujours lieu à la fin de l'instant tandis qu'un agent en ROS peut finir instantanément. La terminaison instantanée d'un agent n'est pas une caractéristique importante en ROS puisqu'il est impossible de mettre en séquence un programme réactif avec l'agent ; tous les agents sont exécutés en parallèle par des `add` dans la machine réactive.

Etat i	susp	res	kill	add	Etat i+1	Action
run	x	x	G	x	kill	
run	G	x	-	-	susp	
run	G	x	-	G	susp	Par(body, $\sum$ currentValue(a))
run	-	x	-	-	run	
susp	x	x	G	x	kill	
susp	x	G	-	-	run	
susp	x	G	-	G	run	Par(body, $\sum$ currentValue(a))
susp	x	-	-	-	susp	

TAB. 7.5: Exécution des événements

$$\frac{t,E \xrightarrow{\alpha} t',E' \quad \alpha != STOP}{rth(s,r,k,a,t),E \xrightarrow{\alpha} rth(s,r,k,a,t'),E'} \qquad \frac{t,E \xrightarrow{STOP} t',E' \quad rtha(s,r,k,a,t'),E' \xrightarrow{\alpha} t'',E''}{rth(s,r,k,a,t),E \xrightarrow{\alpha} t'',E''}$$

$$\frac{\dfrac{\dfrac{eoi=false}{rtha(s,r,k,a,t),E \xrightarrow{SUSP} rtha(s,r,k,a,t),E}\ sat(\ !k \wedge !s \wedge !a\ )}{rtha(s,r,k,a,t),E \xrightarrow{STOP} rth(s,r,k,a,t),E}\ sat(\ !k \wedge a \wedge s\ )}{rtha(s,r,k,a,t),E \xrightarrow{STOP} rths(s,r,k,a,Par(t,\sum currentValue(a))),E} \qquad \frac{\dfrac{\dfrac{sat(k)\ \ eoi=true}{rtha(s,r,k,a,t),E \xrightarrow{STOP} Nothing,E}\ sat(\ !k \wedge a \wedge !s\ )}{rtha(s,r,k,a,t),E \xrightarrow{STOP} rth(s,r,k,a,Par(t,\sum currentValue(a))),E}\ sat(\ !k \wedge !a \wedge s\ )}{rtha(s,r,k,a,t),E \xrightarrow{STOP} rths(s,r,k,a,t),E}$$

$$\frac{\dfrac{\dfrac{eoi=false}{rths(s,r,k,a,t),E \xrightarrow{SUSP} rths(s,r,k,a,t),E}\ sat(\ !k \wedge !r \wedge !a\ )}{rths(s,r,k,a,t),E \xrightarrow{STOP} rths(s,r,k,a,t),E}\ sat(\ !k \wedge r \wedge a\ )}{rths(s,r,k,a,t),E \xrightarrow{STOP} rth(s,r,k,a,Par(t,\sum currentValue(a))),E} \qquad \frac{\dfrac{\dfrac{sat(k)\ \ eoi=true}{rths(s,r,k,a,t),E \xrightarrow{STOP} Nothing,E}\ sat(\ !k \wedge !r \wedge a\ )}{rths(s,r,k,a,t),E \xrightarrow{STOP} rths(s,r,k,a,Par(t,\sum currentValue(a))),E}\ sat(\ !k \wedge r \wedge !a\ )}{rths(s,r,k,a,t),E \xrightarrow{STOP} rth(s,r,k,a,t),E}$$

TAB. 7.6: Sémantique du Thread Réactif

7.4 Exemples

Le but de cette section est de présenter quelques exemples complexes pour illustrer l'utilisation de Rejo/ROS dans la construction d'applications plus sérieuses.

7.4.1 Shell réactif

Le Shell Réactif (Rsh) fut créé avec deux objectifs : 1) faciliter l'exécution dynamique des programmes Rejo, et 2) pouvoir tester les programmes réactifs à l'aide d'un langage interprété, soit en tapant des commandes,

soit en exécutant un script. Rsh est similaire aux Reactive Scripts [BOU 96a] par plusieurs aspects comme la syntaxe et les instructions réactives de base ; cependant, Rsh est différent par le nombre d'instructions réactives acceptées (plus restreint) et par son modèle d'exécution basé sur la création d'un Rejo qui permet de contrôler les commandes.

Lorsque le Rsh démarre, il imprime un prompt de la forme `< hostname | path_actuel>` et se met à attendre des commandes. Rsh implémente trois types de commandes :

1. Des instructions réactives traduites en code Junior.
2. Des commandes atomiques pour effectuer quelques opérations de gestion de fichiers comme `ls`, `cd` et `cat`.
3. Quelques commandes pour interagir avec les agents et le système.

La table 7.7 énumère la liste de commandes implémentées et donne une petite description pour chacune.

Comme tout langage interprété, Rsh permet la création et l'exécution de scripts, c'est-à-dire de commandes stockées dans un fichier pour une exécution ultérieure. Les scripts doivent être sauvés dans fichier qui a l'extension `.rs` et ils sont également exécutés par un Rejo qui les contrôle.

L'implémentation de Rsh utilise un parseur et un comportement réactif en Rejo qui implémente l'attente et l'exécution des commandes. Lorsqu'une commande est entrée, on génère un événement valué (la commande) ; la partie réactive analyse la commande (le parseur) et, si la commande est valide, on la traduit en code Junior et on l'ajoute dans la machine sous la forme d'un Rejo.
Un exemple d'un script en Rsh est montre dans la figure 7.7.

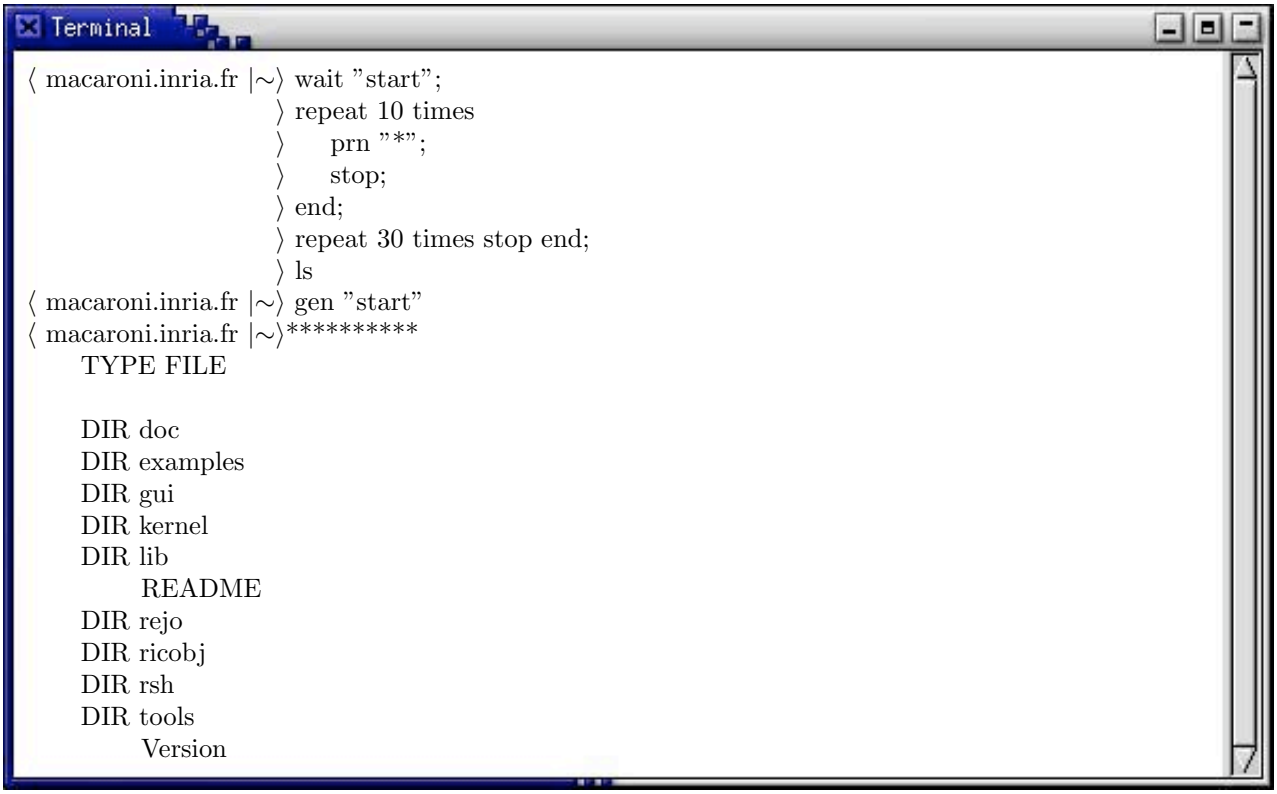

FIG. 7.7: Exécution du Rsh en Ros

Ce programme attend l'événement `"start"` est une fois généré il imprime dix fois le caractère `*` (un par instant), puis on laisse passer trente instants et on exécute la commande `ls` dans le répertoire actuel.

Command syntaxe	Description
file	crée et charge dans le système ROS le Rejo (Instruction Atomique=IA).
file.rs	exécute le script.
ls	liste les fichier du répertoire actuel (IA).
cd *string*	change de répertoire (IA).
cat *string*	montre le contenu d'un fichier (IA).
ps	liste les Rejo dans le système (IA).
kill *string*	tue un Agent de nom string. Cette commande est équivalente à gen "string!kill".
exit	termine l'exécution du shell mais pas de la machine réactive.
logout	termine l'exécution de tout le système, machine réactive comprise.
A ; B par A ‖ B end stop gen *expession* wait *condition* loop B end repeat *val* times B end if *condition_bool* then A else B end when *condition* then A else B end until *condition* then A [handler B] end control *expression* then A end local *expression* then A end	la sémantique est la même que celle de Junior, c'est-à-dire, si A et B sont deux programmes valides A ; B est la séquence de ces deux programmes. Les conditions peuvent contenir les opérateurs AND, OR, NOT.

Tab. 7.7: Les commandes du Rsh

7.4.2 Ricobj

Une des principales applications qui a été créées avec le modèle réactif est les Icobjs [BOU 96c]. Les Icobjs sont un formalisme qui a été proposé pour "programmer" des comportements réactifs facilement. L'objectif principal est de fournir une manière simple et évidente pour des non-spécialistes de composer des comportements au travers de manipulations liées à l'aspect graphique et à l'animation des objets. La programmation se fait donc graphiquement avec un mécanisme puissant de combinaison de comportements. Cette programmation repose sur une notion d'icobj qui regroupe dans une même entité un aspect comportemental (caractéristique objet), un aspect graphique (caractéristique icône) et un aspect animation. Cette programmation par Icobjs permet le parallélisme, la communication par événements diffusés ainsi que la migration à travers le réseau. Pour résumer, un icobj c'est du comportement, plus du graphique animé.

Il existe plusieurs versions des Icobjs. La première est implémentée au-dessus de Reactive Scripts (implémenté en C avec l'interpréteur de scripts réactifs rsi-tk et construit au dessus de Tcl/Tk), et les deux dernières sont implémentées au-dessus de SugarCubes et de Junior (qui utilisent l'interface graphique de Java). Noter qu'à priori, l'utilisateur des Icobjs n'a pas à utiliser les Reactive Scripts, SC ou Junior ni même à en connaître l'existence. Il existe une page web [WEB Icobj] qui explique en détails les Icobjs et qui contient de nombreuses démos.

Parmi les travaux de recherche autour des Icobj, on peut citer celui d'A. Samarin [SAM 02] qui les a utilisés pour implémenter des comportements qui obéissent aux lois physiques. En particulier, il propose une nouvelle technique de simulation de systèmes dynamiques continus qui introduit un temps discret explicite dans le processus de simulation. Il utilise 1) la notion d'instant qui est présente dans le réactif pour, par exemple, résoudre numériquement les équations différentielles qui décrivent un phénomène physique, 2) la diffusion des événements pour faire communiquer les comportements physiques, 3) l'ajout dynamique du réactif pour construire dynamiquement des phénomènes physiques sans devoir construire préalablement un modèle, et 4) les Icobjs pour montrer graphiquement les comportements physiques.

Un autre travail utilisant les Icobj est celui de C. Brunette. Celui-ci a étudié la cohérence dans les jeux en réseaux distribués [BRU 01] et à l'heure actuelle il est en train de créer une nouvelle version des Icobjs au-dessus de Junior [BRU 02]. La nouvelle version des Icobjs résout quelques problèmes des versions précédentes (comme l'utilisation de la souris et le rafraîchissement des images) et ajoute des nouvelles notions (par exemple, on peut construire un Icobj qui est aussi un Workspace).

Les Ricobjs

Pour implémenter les Icobjs avec les fonctionnalités de Rejo/ROS, j'ai créé une version des Icobjs que j'appelle Ricobjs. Les Ricobjs sont codés en Rejo et sont exécutés par Ros comme des agents migrants. Autrement dit, on cherche à :

1. Rendre la programmation des Icobjs plus simple. La création des comportements réactifs primitifs se fait forcement dans un langage de programmation. Les versions actuelles des Icobjs sont codées soit en SugarCubes soit en Junior. Rejo facilite cette programmation et, lorsque c'est utile, permet d'utiliser l'héritage de comportements réactifs.

2. Utiliser le mécanisme de migration de Ros. La simple création d'un Ricobj lui confère ainsi la possibilité de migrer. Pour garantir cette propriété, il a fallu analyser le code de tous les RIcobjs pour assurer qu'aucun ne contenait des objets non sérialisables.

3. Utiliser la structure de contrôle des agents. Par défaut on ne peut que tuer les icobjs ; grâce à la structure d'agent on peut les suspendre, les résumer, les tuer, ou bien les sauvegarder ou les migrer.

Les Ricobjs héritent des caractéristiques des Icobjs :

Un Ricobj peut être créé, activé ou bien détruit. Le comportement d'un Ricobj est la combinaison de son comportement propre et des influences qu'il subit de la part d'autres Ricobjs. L'exécution d'un Ricobj peut changer sa représentation, l'animer, changer son comportement ou encore agir sur les Ricobjs qui sont dans sa zone d'influence (Workspace). La création d'un Ricobj nécessite, dans certains cas, la saisie d'informations externes.

Un aspect essentiel des Ricobjs est la possibilité de composer leurs comportements. Il existe des Ricobj appelés créateurs dont la principale fonction est de créer de nouveaux Ricobjs : le Ricobj `Constructor`, crée un Ricobj par copie du comportement d'autres Ricobjs, et le Ricobj `Load` (qui n'existe pas dans les Icobjs), crée un Ricobj à partir d'une classe.

Les Ricobjs peuvent utiliser leurs zones d'influence pour communiquer entre eux. Ce type de communication est lié au graphique et à la proximité, et deux Ricobjs dont les zones d'influence sont disjointes ne peuvent communiquer par ce moyen. Les événements fournissent un autre moyen de communication entre Ricobjs, qui ne repose pas sur leur aspect graphique. Un événement peut être attendu ou généré par un Ricobj, lors de son exécution, et il est ainsi possible de contrôler les Ricobjs par des événements générés par d'autres Ricobjs.

La figure 7.8 illustre l'exécution des Ricobjs : on voit un terminal (en noir) qui lance ROS, un Workspace qui exécute les Ricobjs (zone avec des Icones) et un Ricobj qui implémente un terminal qui exécute un Rsh. Le terminal contient trois sections organisées horizontalement : 1) la première (qui se trouve tout en haut) est la zone de capture des commandes, 2) la deuxième zone permet de générer des événements valués (l'événement et la valeur sont des Strings), et 3) la troisième zone, qui se trouve au fond, permet de stopper ou résumer

la machine réactive (clock), de faire réagir la machine une fois lorsqu'elle stoppée (step), de créer un nouveau terminal (Rsh), de visualiser le nombre d'instructions exécutées (inst=2119), de visualiser la hauteur de l'arbre (height=35), et de visualiser le nombre de Rejos présents dans le système (rejos=14).

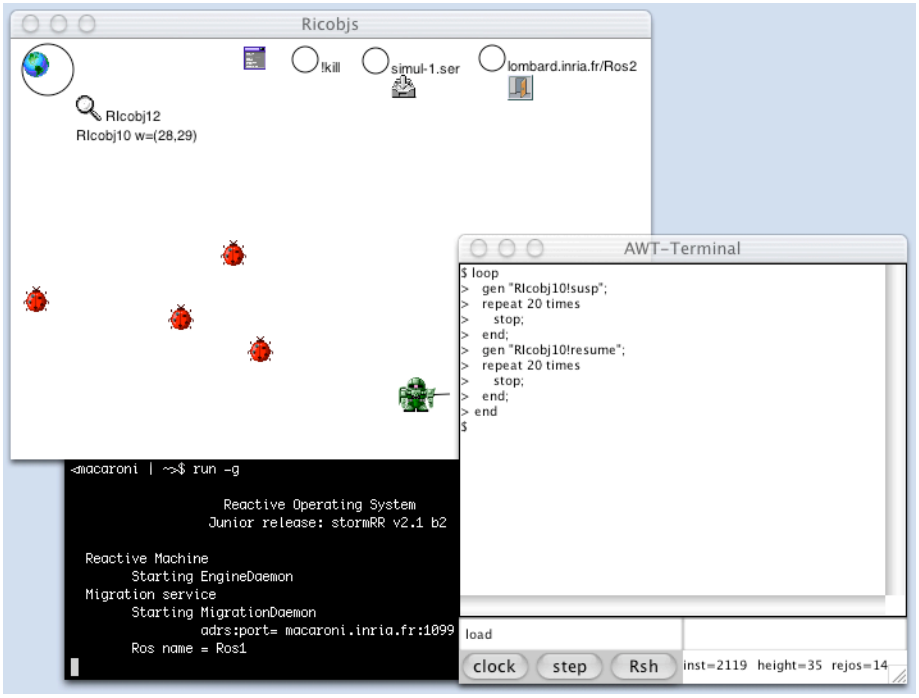

FIG. 7.8: Les Ricobj (Icobjs code en Rejo)

Quatre des Ricobjs qui se trouvent dans cette figure sont définis pour faciliter l'utilisation de l'apport de Rejo/ROS :

Lorsqu'on active ce Ricobj, il détecte tous les Ricobjs qui sont dans sa zone d'influence et génère leurs événements de contrôle. Par défaut l'événement de contrôle généré est l'événement pour suspendre le comportement mais on peut changer l'événement de contrôle avec les touches : **k** pour tuer (kill), **r** pour résumer et **s** pour suspendre. La touche **c** sert à mettre le Ricobj dans un état dit continu dans lequel le Rciobj génère à chaque instant l'événement de contrôle.

Lorsqu'on active ce Ricobj il détecte tous les Ricobjs qui sont dans sa zone d'influence et génère leurs événements de migration. L'événement de migration est généré avec l'adresse affichée (lombard.inria.fr/Ros2). Ce Ricobj écoute l'événement valué <"adrs", "siteRos"> pour modifier l'adresse de migration utilisée. La touche **c** sert à mettre le Ricobj dans l'état continu dans lequel le Ricobj génère à chaque instant l'événement de migration.

Lorsqu'on active ce Ricobj il détecte tous les Ricobjs qui sont dans sa zone d'influence et génère leurs événements de sérialisation. L'événement de sérialisation est généré avec la valeur affichée (simul-1.ser) qui est utilisée pour sauver les Ricobjs dans un fichier qui porte ce nom. Ce Ricobj réagit aux touches suivantes : la touche **c** sert à effacer le fichier de sauvegarde ; la touche **r** met le Ricobj en mode lecture (read), lorsqu'on active le Ricobj il lit le fichier de sauvegarde et génère une instance des Ricobjs stockés ; enfin, la touche **w** met le Ricobj en mode écriture (write) dans lequel il génère l'événement de

sérialisation. La sérialisation consiste à détecter les Ricobjs dans la zone d'influence et à les ajouter à la fin du fichier de sauvegarde (qui doit se trouver en $ROS/lib/class/) ; lorsqu'on exécute les Ricobjs dans un browser, cette fonctionnalité n'est pas toujours disponible à cause des limitations dans les droits d'accès.

Le Ricobj 

est un Ricobj expérimental qui affiche les noms de Ricobjs qui se trouvent dans sa zone d'influence. Pou chacun, il affiche aussi sa hauteur (28) et son poids (29).

Les autres icônes sont des Ricobjs normaux qui implémentent les comportements réactifs suivants :

 Lorsqu'on active ce Ricobj il crée un terminal qui exécute un shell réactif (Rsh) comme celui que l'on voit à droite dans la figure 7.8.

 Ce Ricobj exerce une force de gravité sur tous les Ricobjs qui se trouvent dans le Workspace. La force de gravité est dirigée vers la partie inférieure de la fenêtre.

 Ce Ricobj implémente un comportement de combat. Il est capable de se déplacer et de lancer de bombes. Il est catalogué comme un Ricobj prédateur.

 Ce Ricobj implémente un comportement de proie. Lorsqu'il est près d'un Ricobj identifié comme prédateur, il fuit. Plus il est près du prédateur, plus la force de répulsion est grande.

La figure 7.9 illustre l'exécution d'une application, construite avec les Ricobjs décrits précédemment, qui implémente une sorte de Ping Pong, c'est-à-dire, une entité qui passe son temps à migrer entre deux sites. La figure 7.9 contient cinq fenêtres : les deux fenêtres noires qui se trouvent en haut et qui lancent chacune un site Ros, à gauche le site s'appelle Ros1 (nom par défaut) et à droite le site s'appelle Ros2 (nom donné avec l'option -n) ; ensuite, il y a deux fenêtre qui représentent les Workspaces de chaque site Ros, et finalement une fenêtre noire à droite au fond, qui lance le Rmiregistry (un serveur de noms utilisé pour localiser les objets distants) auquel chaque site Ros enregistre son service de migration.

Une fois lancé, chaque site contient quatre Ricobjs : une terre, un robot, une coccinelle, et un Ricobj de migration (qui est en état continu et correctement configuré pour migrer vers l'autre site) ; les coccinelles se mettent à migrer d'un site à l'autre à cause des différentes forces qui influent sur elles (la gravité et la répulsion des robots, dues au positionnement initial des coccinelles, au-dessus des robots).

En conclusion, on a défini une notion de Ricobj avec les caractéristiques suivantes :

- Il n'y a pas de syntaxe des Ricobjs ; ils ne sont manipulés que graphiquement.
- Il n'y a pas d'intermédiaire de traduction : un Ricobj peut être exécuté dés sa création (il n'y a rien à compiler, l'approche est interprétée).
- Les Ricobjs se combinent en séquence, en parallèle ou en boucle de manière naturelle : un seul Ricobj réalise en fait l'ensemble de ces fonctions. Les Ricobjs sont modulaires et réutilisables sans contrainte.
- La diffusion d'événements pour contrôler les Ricobjs est possible, ce qui fournit un moyen puissant de communication entre Ricobjs.
- Les Ricobj possèdent une sémantique claire et immédiate sous la forme d'un programme Rejo.
- La migration de Ricobjs à travers le réseau est possible et simple à réaliser.

Un ensemble de démos est fourni pour montrer l'intérêt de l'approche.

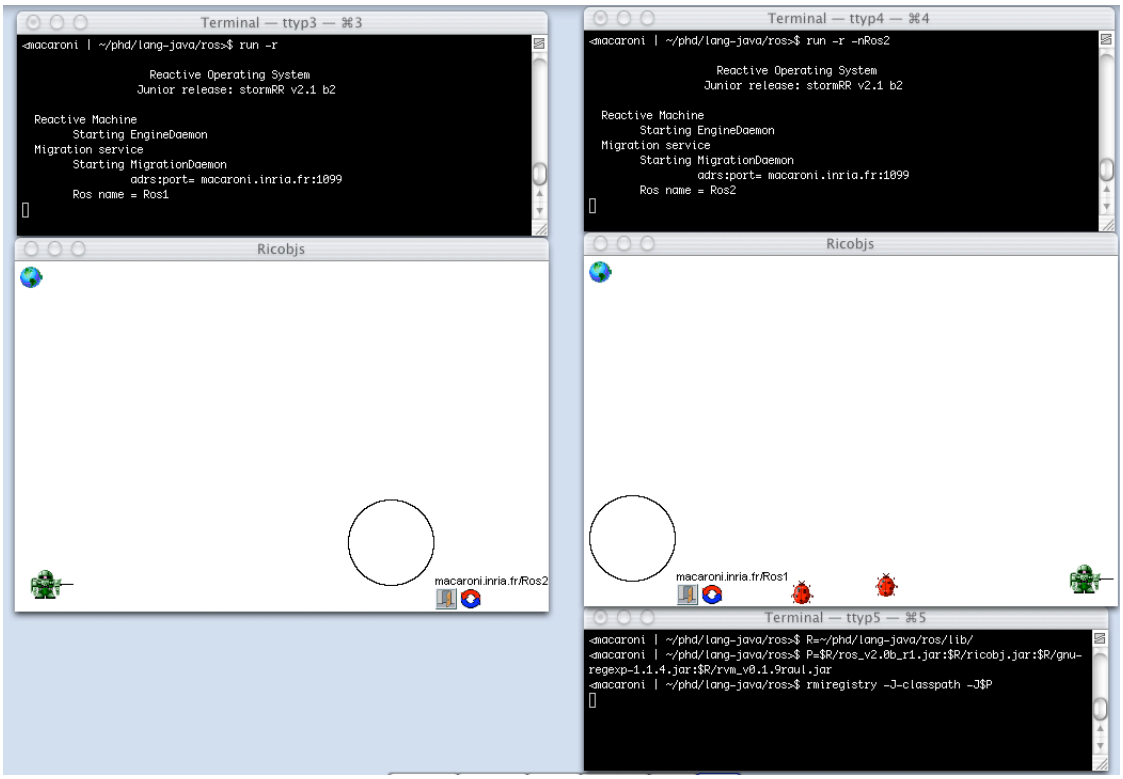

FIG. 7.9: L'exemple de Ping Pong avec les Ricobjs

7.5 Travaux similaires

RAMA

RAMA est un système d'agents réactifs migrants qui a été conçu par Navid Nikaein [NIK 99]. RAMA a été implémenté en Java et utilise les SugarCubes pour exprimer les comportements réactifs et faire communiquer les agents à l'aide de la diffusion. Les propriétés de RAMA sont les mêmes que celles de Rejo/ROS, c'est-à-dire que le système offre aux agents de la concurrence, de l'interactivité (communication par diffusion), de la réactivité (instructions réactives), de la mobilité, du dynamisme et une exécution multi-plate-forme grâce à Java.

Les caractéristique de RAMA en tant que SAM sont :

Langage de programmation Pour créer des agents réactifs, RAMA avait besoin d'un langage de programmation qui fournisse la réactivité entre les agents, ainsi que d'autres propriétés comme la portabilité, la robustesse et la sécurité. Le langage Java fournit la plupart de ces propriétés, sauf la réactivité ; les SugarCubes ont été utilisés pour combler ce manque.

Migration RAMA utilise le même mécanisme de migration que Rejo/ROS, c'est-à-dire, l'instruction freezable et un ensemble d'objets qui implémentent la migration en utilisant RMI. Autrement dit, RAMA implémente un mécanisme de mobilité forte.

Routage RAMA a été conçu avec un algorithme de routage simple qui cache l'information de routage dans chaque site, en établissant un anneau bidirectionnel virtuel entre les différents sites. Grâce à l'anneau, les agents peuvent parcourir tous les sites (sans connaître leurs adresse) à l'aide de primitives du type `MigrateToNextSite`.

Communication RAMA a été conçu pour permettre la communication intra-groupe locale[2]. Grâce au mécanisme de diffusion instantanée qu'offre l'approche réactive, ce type de communication est obtenu de façon naturelle. La communication intra-groupe globale est obtenue facilement en migrant parmi les différents sites qui forment l'anneau. La communication par diffusion en SugarCubes permet d'avoir une communication synchrone transparente car tous les agents reçoivent la même information à la fin de l'instant. En plus, RAMA offre une communication point à point (*unicast*) au cas où l'agent en a besoin.

Services RAMA est un prototype qui offre principalement la notion de groupe et les mécanismes mentionés précédemment.

Comme résultat du mélange de Java et SugarCubes, les agents mobiles ont tout le pouvoir expressif de Java et des SugarCubes ; toutefois RAMA présente aussi quelques désavantages :

- RAMA n'offre pas une interface claire qui définit la façon d'utiliser le système et de le protéger ; la seule chose existante, c'est la protection des données et des méthodes importantes. L'absence d'une interface conduit à révéler des détails de l'implémentation de RAMA qui ne sont pas importants ou qui simplement peuvent gêner le programmeur.
- L'architecture de RAMA est monolithique et ne permet pas d'ajouter et d'enlever de nouveaux services, ni de modifier les services existants. Le mécanisme de routage est difficile à modifier et donc restreint au routage en anneau ou au point-à-point.
- La programmation des agents est faite à la SugarCubes, c'est-à-dire qu'elle est de bas niveau et donc fastidieuse et peu modifiable.

Moorea

Moorea [MOOR 00](*MObile Objects, REactive Agents*) est un système d'agents mobiles réactifs créé en Java. Autrement dit, Moorea est proche du système ROS mais avec de grosses différences dans le modèle d'objet réactif. Il offre par ailleurs un plus grand nombre de services et d'APIs que ROS.

Moorea est un projet de grande envergure de France Télécom dans lequel un grand nombre de personnes ont travaillé dans les différents couches de la plate-forme. Moorea est constitué de :

1. Un moteur réactif et un langage de programmation réactive. Le moteur réactif est Junior et le langage de programmation est Rhum.
2. Une plate-forme d'objets distribués. La plate-forme s'appelle Jonathan [Jona] et elle offre plusieurs services et leurs interfaces (appelées *personalities*). Les programmeurs ont à leurs disposition un ensemble de composants (protocoles, marshallers, stub factories, etc.) pour construire des nouveaux services. Jonathan offre deux personnalités : David qui implémente un ORB (*Object Request Broker*) CORBA, et Jeremie qui implémente l'API RMI de Java.
3. Une spécification de MASIF [MASIF](*Mobile Agent System Interoperability Facilities*). MASIF de l'OMG (l'organisme qui gère la norme CORBA) est l'effort de plusieurs entreprises pour standardiser un ensemble de caractéristiques des systèmes d'agents mobiles avec l'objectif d'assurer leurs interopérabilité.

Si on centre l'analyse de Moorea et Ros sur leurs points communs, c'est-à-dire sur le modèle réactif utilisé, on voit que Moorea présente deux différences : 1) Moorea offre des primitives spéciales pour cacher la mobilité d'un agent, et 2) le modèle de programmation réactive est celui de Rhum.

Moorea cache la migration des agents grâce à un système de nommage global qui permet d'envoyer des événement à un agent là il se trouve, et de garder une référence sur un agent qui reste valide même si l'agent migre.

[2]La communication entre agents qui appartiennent au même groupe et qui sont dans la même machine; la diffusion d'un message n'atteint pas les membres du groupe qui sont sur autres machines.

Le système de nommage est implémenté avec la combinaison de deux techniques pour améliorer les performances : le *forwarding* (ou *reference chain*) et le *relocator*. Le *relocator* consiste à associer le nom d'un agent à l'adresse du site où il se trouve ; l'association est mise-à-jour à chaque fois que l'agent migre. Le *forwarding* consiste à garder un objet de référence dans chaque site où l'agent est passé. Pour trouver un agent il suffit de parcourir les objets de référence. Pour améliorer les performances le *relocator* est le premier à être utilisé, avant le *forwarding* si l'agent a migré. L'information du *relocator* est mise-à-jour lorsqu'on fait plus d'un certain nombre de sauts.

Ecomobile

Ecomobile [ROS 02] est un système d'agents mobiles qui a été créé avec l'objectif d'avoir une plate-forme pour construire des applications réparties dans le domaine de la gestion des réseaux, par exemple l'allocation de ressources, le routage par contraintes ou le *monitoring*. Le système a été construit en Java et utilise Junior pour programmer les comportement réactifs des agents. Le système a été conçu pour résoudre les problèmes liés à l'utilisation des réseaux optiques mais aussi les problèmes liés aux comportements des agents, comme par exemple atteindre ses objectifs et de quelle façon les objectifs influencent l'agent (faut-il migrer ou pas ? quand et vers quel site ?, etc.).

Le modèle de programmation par objectifs est un aspect très important de la plate-forme car il influence la performance du système. Les agents sont constitués de comportements (de coordination et navigation) et d'objectifs ; les objectifs ne sont pas attachés de façon permanente aux agents. Ce découplage entre comportements et objectifs permet de 1) stocker les objectifs dans un site pour qu'un autre agent (avec un objectif similaire) puisse les reprendre et 2) créer des nouveaux agents qui reprendront en charge les objectifs.

Une bonne partie de la conception du système a été dédiée à la mise au point de toutes ces notions pour garantir la stabilité du système (par exemple la taille de la population) dans des topologies différentes.

Systèmes d'agents avec des comportements réactifs

D'autres systèmes d'agents mobiles utilisent aussi la notion de réaction (pas celle de Junior qui est liée à la notion d'instant) pour programmer des agents mobiles ou une partie de la plate-forme de migration. Un exemple de ces systèmes est MARS [CAB 98]; MARS utilise l'approche réactive avec un paradigme de programmation différent pour mettre à jour des données des agents en utilisant des événements. Les données utilisées sont de type *Tuple Space* comme celui de Linda.

7.6 Conclusions

On a présenté dans ce chapitre une nouvelle plate-forme pour l'exécution d'objets réactifs appelée ROS. ROS, de l'anglais *Reactive Operating System*, est une plate-forme qui a les caractéristiques suivantes :

- on peut exécuter des objet réactifs programmés avec Rejo. Les Rejos sont des entités qui ont un nom et une structure de contrôle pour les suspendre, les résumer, les tuer et les modifier en ajoutant des nouveaux comportements en parallèle.
- on peut sérialiser et migrer des objets réactifs. Ces deux opérations sont le point de départ pour implémenter des services plus complexes, par exemple la persistance et la gestion de réseaux. La migration des objets réactifs est une migration forte qui peut être réactive ou proactive grâce au mécanisme de diffusion d'événements.
- les systèmes réactifs construits montrent que le parallélisme de l'approche réactive est une alternative réelle à l'utilisation traditionnelle de threads.
- l'architecture de ROS est une architecture modulaire et minimale (562 lignes de code) analogue à celle d'un système d'exploitation à laquelle on peut ajouter et enlever des modules selon les besoins. Deux applications ont été construites pour atteindre les fonctionnalités du système : un shell réactif (Rsh, 1452 lignes de code) et une interface graphique (les Ricobjs, 1197 lignes de code).

En résumé, ROS ressemble à un système d'exploitation réparti car il possède une interface graphique (Ricobjs), une interface en mode texte pour l'utilisateur (Rsh), une interface de programmation (API), un ensemble des services (sérialisation, migration, etc.), et un micro-noyau modulaire. Le langage utilisé pour programmer les applications en ROS est Rejo.

Chapitre 8

Conclusion et perspectives

Les travaux présentés dans cette thèse ont été effectués au sein du Centre de Mathématiques Appliquées (CMA) de l'Ecole de Mines de Paris, localisé à l'INRIA. Ils s'inscrivent dans la ligne des travaux effectués, principalement, par Frédéric BOUSSINOT, Laurent HAZARD et Jean-Ferdinand SUSINI sur l'***Approche Réactive Synchrone***.

L'approche réactive synchrone reprend plusieurs notions de l'***Approche Synchrone*** comme la notion d'instant et la diffusion d'événements. Néanmoins, l'approche réactive se distingue de l'approche synchrone par la durée des instants; pour l'approche réactive cette durée est non-nulle. Les conséquences de la durée non-nulle sont : 1) la possibilité des construire de systèmes dynamiques dans lesquels le nombre d'instructions et d'événements utilisés varie à l'exécution, 2) la réaction à l'absence d'un événement est reportée à l'instant suivant et, 3) la préemption forte est éliminée, seule la préemption faible restant possible.
Parce que la réaction à l'absence est reportée à l'instant d'après, l'approche réactive synchrone ne présente pas de problèmes de causalité, trait caractéristique des langages issus de l'approche synchrone. L'approche synchrone a été conçue pour construire des systèmes de contrôle ; ces systèmes sont par nature des systèmes réactifs et de ce fait le nom d'approche réactive synchrone est "mal" choisi. Néanmoins, on continue à utiliser ce nom pour des raisons historiques et chaque fois qu'il risque d'y avoir confusion on dit que l'on utilise l'approche réactive comme l'a définie Frédéric BOUSSINOT.

Les travaux sur l'approche réactive ont démarré à la fin des années quatre-vingt avec la création de ***Reactive-C***. Ce formalisme a engendré un grand nombre d'expérimentations (voir la figure 8.1) qui continuent jusqu'à nos jours. Avec l'apparition des langages basés sur des machines abstraites et en particulier avec le succès de Java, deux variantes de Reactive-C ont été créées en Java: Junior et SugarCubes.

La programmation en Reactive-C, Junior et SugarCubes est une programmation de bas niveau. Deux langages de haut niveau qui facilitent leur programmation ont été proposés : le langage interprété ***Reactive Scripts*** et le langage compilé ***Rhum***. Rhum possède des mécanismes de distribution ainsi que d'autres formalismes : Plusieurs applications réparties qui ont été construites, ***DRM*** (Distributed Reactive Machine) qui est une implémentation répartie du moteur réactif de SugarCubes et ***RAMA*** (Reactive Autonomous Mobile Agents), qui est un système d'agents mobiles. D'autres projets ont utilisé Junior dans la construction de systèmes d'agents mobiles [MOOR 00, ROS 02].

Plusieurs de ces formalismes ont été utilisés dans la création de l'environnement graphique des ***Icobjs***. Les Icobjs introduisent un nouveau style de programmation basé sur l'utilisation de la souris pour programmer (*drag-and-click*) des comportements réactifs. Les Icobjs offrent une interface très agréable et puissante qui facilite la construction et le debuggage d'applications. Les Icobjs ont été utilisé dans la plate-forme PING et dans la simulation de phénomènes physiques [SAM 02].

Pour ma part, j'ai travaillé sur plusieurs des formalismes que l'on vient de décrire. Ces travaux ont été publiés en [ACO 00b], [ACO 02] et [ACO 03], et utilisés dans la plate-forme PING et dans la simulation de vie artificielle [LIM 01]. Une page web [WEB RejoRos] regroupe toute l'information (papiers, logiciels, etc.). Voici une description plus détaillée des travaux menés :

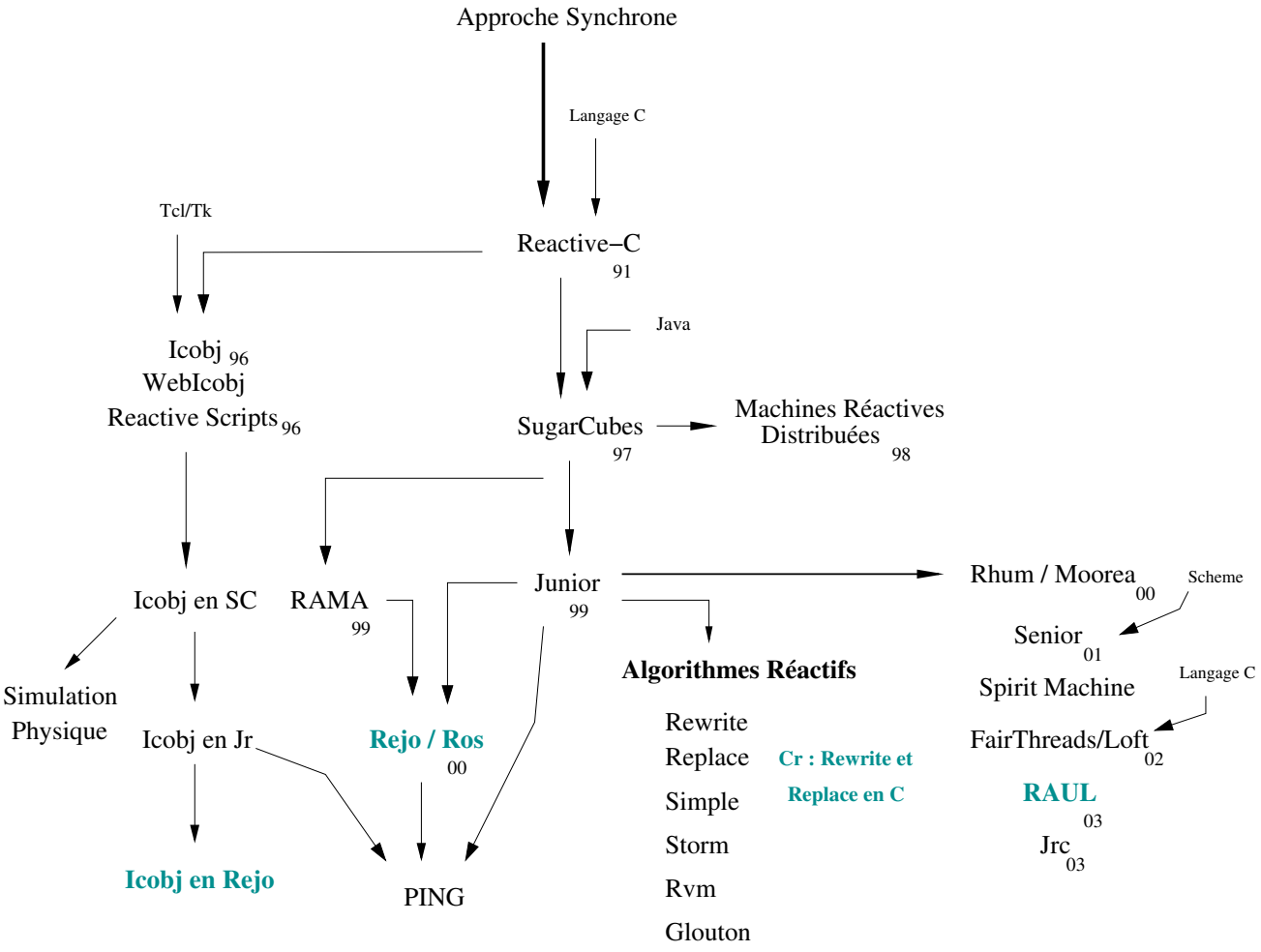

FIG. 8.1: Evolution de l'Approche Réactive Synchrone

Junior

Le travail central de cette thèse a consisté en la création d'un nouveau langage de programmation réactive, Rejo. La construction de Rejo a été fortement influencée par les SugarCubes mais surtout par Junior; Rejo est implémenté avec Junior[1]. C'est pour cette raison que l'on a présenté Junior dans les chapitres 2 et 3. Dans ces chapitres, on a vu le modèle d'exécution de Junior, son API de programmation, quelques exemples et on a fait une analyse des différentes implémentations de Junior et des instructions réactives.

Plusieurs des traits caractéristiques de Junior se retrouvent en Rejo, d'où l'importance d'analyser la programmation et les implémentations de Junior. L'analyse de Junior et son utilisation en Rejo nous ont permis de détecter quelques problèmes, d'améliorer plusieurs instructions et d'en proposer d'autres. Le travail en Junior s'est concentré sur quatre aspects :

1. *Analyse des instructions* : on a analysé les instructions réactives pour mieux les comprendre et, dans certains cas, les améliorer. En particulier, on a trouvé que :

   (a) L'instruction Control, qui n'accepte aucun type de configuration, peut être étendue pour utiliser les configurations And et Or. Par contre, l'instruction Freezable nécessite un mécanisme auxiliaire pour les utiliser.

[1]Il existe une version expérimentale qui génère du code en SugarCubes.

(b) L'instruction Until a une sémantique compliquée ; le cas plus gênant est l'exécution d'une instruction Stop qui, lorsqu'il y a préemption, ne stoppe pas l'exécution de la branche qui la contient.

(c) Il faut faire attention à la terminaison instantanée qui peut générer des problèmes, par exemple la préemption instantanée et la boucle instantanée.

(d) Il manque une partie `handler` dans l'instruction Freezable.

2. *Workbench* : on a fait une série de tests pour évaluer les différentes implémentations de Junior. Dans le workbench réalisé, on a testé 6 implémentations en Java (Rewrite, Replace, Storm, Simple, Glouton, Rvm) et 4 implémentations en C (Cr, Jrc, Loft, Spirit Machine). Ce workbench nous a permis de :

   (a) trouver que les instructions binaires de séquence et de parallélisme ont un impact important sur le temps d'exécution et sur la taille des programmes que l'on peut exécuter. Ces instructions imposent des limites à la taille de programmes réactifs aussi bien à la compilation (toutes les implémentation partent d'un arbre) qu'à l'exécution (surtout pour les implémentations qui gardent la structure d'arbre pour exécuter le programme réactif).

   (b) analyser le problème de l'attente active des événements. L'attente active se présente dans les réactions qui s'enchaînent, comme dans la cascade inverse, ou dans les instructions événementielle qui effectuent des attentes inter-instant. Les premières versions de Junior (Rewrite et Replace) ne considèrent pas ces problèmes ; les versions qui ont été créées après souffrent du manque de sémantique (Simple) ou du traitement de certains cas (en Storm l'attente inter-instant n'est pas traitée). Les nouvelles versions (Glouton) commencent à concilier ces deux facteurs.

   (c) constater le coût élevé qui résulte de la construction d'un comportement réactif constitué par : 1) des composants exécutés en parallèle, et 2) des événements auxiliaires utilisés pour les synchroniser. Lorsque le comportement réactif est utilisé souvent (comme la structure de contrôle qui suspend, résume et tue un programme réactif), il vaux mieux définir et implémenter une instruction réactive ad-hoc de façon à améliorer les performances et à avoir une sémantique claire et compacte.

3. *Propositions* : on a proposé quelques nouvelles instructions comme :

   (a) L'instruction Dynapar. Cette instruction est inspirée de l'instruction Shell de SugarCubes ; elle permet de mettre en parallèle des nouvelles instructions de façon dynamique et sélective. A la différence du Shell, Dynapar utilise un événement valué pour l'ajout des instructions.

   (b) L'instruction Run. Cette instructions est née du besoin qui existait en Rejo d'exécuter dynamiquement des nouvelles instructions sans passer par les événements et sans retarder d'un instant l'exécution. L'instruction Run définit ainsi un mécanisme de modularité très dynamique. Cependant, ce mécanisme est potentiellement dangereux, par exemple lorsqu'il est combiné avec la récursion.

   (c) L'instruction Try. Cette instruction a été créée pour pouvoir récupérer le contrôle lorsque des exceptions Java sont levées, principalement, lors de l'exécution des wrappers et des atomes. L'instruction reste dans un état expérimental car il reste encore trois aspects à considérer : 1) la définition d'une sémantique formelle qui prenne en compte l'environnement Java, 2) la définition d'exceptions réactives, par exemple une exception levée lorsqu'une boucle instantanée est rencontrée, et 3) étudier les liens avec la préemption forte.

   (d) Des instruction n-aires de séquence et de parallélisme. Ces instructions rendent l'exécution plus rapide mais aussi permettent d'exécuter des programmes réactifs de plus grande taille.

   On a donné une sémantique formelle à ces instructions et on les a implémenté dans une version expérimentale utilisée en Rejo.

4. *Pistes* : on a exploré quelques instructions introduisant du *pattern-matching*, de la QoS et introduisant des événements qui n'ont pas une porté locale. La plupart de ces proposition nécessitent une étude plus approfondie (des sujets de recherche à poursuivre) car certains aspects sont non convaincants : 1) les primitives pour faire de la QoS introduisent, dans certains cas, la préemption forte, 2) le mécanisme de *pattern-matching* fonctionne mais ne passe pas à l'échelle.

Dans les chapitres qui présentent Junior, on a montré les avantages et les désavantages suivants :

Avantages :

1. Sémantique formelle. Junior dispose d'une sémantique formelle à base de règles de réécriture qui permet de valider les implémentations et de comprendre facilement et sans ambiguïté les programmes réactifs.
2. Programmation concurrente. Junior définit un opérateur de parallélisme logique qui se présente comme une alternative à la programmation concurrente par threads. Les threads, dans la plupart des systèmes, posent un problème de passage à l'échelle et ont une sémantique difficile à manipuler. L'opérateur de parallélisme de Junior a au contraire une sémantique claire et il existe une analogie forte avec les systèmes coopératifs.
3. Vitesse d'exécution. La vitesse d'exécution a été beaucoup améliorée de sorte que l'on peut maintenant commencer à s'intéresser aux systèmes temps réel ou embarqués. Parmi les optimisations faites, on peut mentionner une meilleure gestion des événements et un meilleur traitement des instructions de composition : remplacement des instructions binaires et pour certaines implémentations élimination des instructions de parallélisme ou de séquence, c'est-à-dire rejet de l'interprétation de code.

Désavantages :

1. Syntaxe difficile. La syntaxe de Junior est semblable à celle des langages fonctionnels (une paire de parenthèses par opérateur) ; cette syntaxe se complique à cause des opérateurs binaires de composition.
2. Programmation de bas niveau. Le programmeur doit définir des wrappers et des actions atomiques à chaque fois qu'il est nécessaire d'exécuter du code Java.
3. Pas de programmation par objets. L'instruction Link a été proposée pour l'implémenter mais le mécanisme est de bas niveau car il faut : 1) au moins un cast pour traiter l'objet correctement, 2) prendre en compte les exceptions liées aux casts, et 3) utiliser un préfixe (l'objet lié) à chaque fois que l'on utilise un champ de l'objet ou une de ses méthodes.
4. Sémantique compliquée de quelques instructions. En particulier, tout ce qui concerne la terminaison instantanée (préemption instantanée, loop instantanée, dé-synchronisation).
5. Deux types d'exécution. L'exécution est en deux phases : tout d'abord une phase de construction d'objets puis l'exécution réactive proprement dite. Par exemple, la création d'un objet réactif doit prendre en compte le fait que certaines données ne sont disponibles qu'à l'exécution, en particulier celles relatives à l'environnement réactif.
6. Certains éléments du langage n'ont pas de sémantique formelle car l'exécution du code Java n'est pas modélisée formellement. C'est le cas par exemple pour : les méthodes `currentValues` et `previousValues`, l'exécution des wrappers, des atomes et de quelques instructions réactives (Scanner, If, Try-catch).
7. Instructions à état. Le fait que les instructions réactives aient un état pose des problèmes pour les exécuter dans des systèmes qui utilisent uniquement de la mémoire ROM.

Il existe quelques caractéristiques de Junior qui peuvent être vues à la fois comme avantages ou désavantages selon les caractéristiques du système que l'on veut construire :

1. Systèmes coopératifs vs Systèmes préemptifs. Les systèmes coopératifs sont contraints de coopérer pour évoluer ce qui rend leur utilisation difficile dans les systèmes multiprocesseurs. D'autre part, l'effort pour les porter sur les systèmes préemptifs est important (il faut détecter les variables partagées pour les protéger). D'un autre coté, les systèmes coopératifs peuvent être programmés plus efficacement que les systèmes préemptifs en évitant de faire des changements de contexte inutiles. Les systèmes coopératifs sont aussi plus simples à débugger car le non-déterminisme introduit par la préemption n'existe plus.

De l'autre côté, les systèmes préemptifs sont plus robustes (le manque de coopération ne fait pas planter le système entier), et le mécanisme de scheduling est mieux adapté aux systèmes qui font de gros calculs. Le problème des systèmes préemptifs se trouve dans les régions critiques issues de la concurrence "sauvage" ; les différentes techniques qui ont été proposées pour résoudre les conflits d'accès à une ressource partagée (verrous, sémaphores, monitors) sont coûteuses et pas toujours modulaires.

Pour les systèmes à forte interaction que l'on veut construire, l'approche coopérative offre plus d'avantages que d'inconvénients.

2. Déterminisme vs Non-déterminisme. Le choix entre ces deux types de systèmes n'est pas simple. Les programmes Junior sont généralement non-déterministes. Cependant, les premières versions de Junior incluaient un opérateur de parallélisme déterministe ; Rewrite et Replace (comme SugarCubes) implémentent d'ailleurs encore cet opérateur considéré comme un cas particulier de l'opérateur de parallélisme non déterministe de Junior. L'enjeu entre déterminisme et non-déterminisme se situe entre deux aspects : 1) La vitesse. Les systèmes non-déterministes peuvent être exécutés plus rapidement que les systèmes déterministes, et 2) La vérification. Les systèmes déterministes sont plus simples à debugger et il permettent de vérifier des propriétés.

   Les systèmes que l'on a construit ne sont pas de systèmes critiques pour lesquels il est indispensable de vérifier des propriétés ; je pense que c'est cela qui justifie l'introduction du non-déterminisme en Junior. Dans mon expérience, je mettais au point mes programmes en utilisant une version déterministe (mes programmes ne supposaient pas le déterminisme) et une fois corrigés, j'utilisais la version non déterministe pour les exécuter plus rapidement.

3. Systèmes synchrones vs Systèmes réactifs synchrones. Par rapport aux systèmes synchrones, Junior offre deux qualités très importantes, le dynamisme et l'absence des problèmes de causalité. Par contre, la vérification de propriétés est fortement limitée car l'ajout de nouveaux comportements réactifs oblige à une vérification dynamique qui pourrait être coûteuse. En ce qui concerne la vitesse entre ces deux type des systèmes, on ne peut rien dire car il n'y a pas eu d'études. Moi, je pense que les systèmes synchrones sont plus rapides que les systèmes réactifs synchrones (à cause des optimisations statiques), mais ceci reste à vérifier.

Rejo

Rejo est un langage de haut niveau pour la programmation réactive. Il s'agit en fait d'une extension de Java pour créer des objets réactifs en Java (Rejo signifie *REactive Java Objects*). Les instructions de Rejo sont traduites en un langage de bas niveau (Junior), cachant ainsi la complexité de celui-ci.

Les travaux en Rejo se sont concentrés sur les points suivants :

1. Programmation de haut niveau. Rejo rend l'utilisation des wrappers, des configurations, des atomes et des instructions binaires transparentes. Dans le cas des wrappers, le programmeur ne les définit que lorsqu'il utilise un type particulier d'événement (ni des chaînes ni des entiers).

2. Notion d'objet réactif. Rejo définit un modèle d'objets réactifs au-dessus des modèles d'exécution de Java et de Junior. Ce modèle définit :

   (a) Méthode réactive. Rejo définit 18 instructions réactives, 16 explicites et 2 implicites (la séquence et l'instruction Nothing). Ces instructions sont groupées dans des méthode dites réactives qui peuvent être exécutées statiquement avec Inline, ou dynamiquement avec Call. La plupart des instructions réactives en Rejo sont primitives et il n'y a que quelques macros.

   (b) Variables réactives. Il existe 3 types de variables en Rejo : les variables globales à l'objet, les variables locales aux méthodes Java et les variables locales aux méthodes réactives. Les variables locales aux méthodes réactives existent tout au long de l'existence du Rejo ; cette existence est nécessaire pour pouvoir les utiliser dans la suite d'instants pendant lesquels la méthode est exécutée (comme les variables statiques d'une fonction en C).
   Par ailleurs, on peut aussi parler des variables réactives, des variables (globales ou locales aux

méthodes réactives) qui sont exécutées par des instructions réactives. Ces variables réactives, implémentées par des wrappers (par exemple les IntegerWrappers de Junior), ne sont lues qu'à des points spécifiques de l'exécution de l'instruction réactive qui les utilise ; en général, elles sont lues lors de la première activation comme dans les instructions Repeat et When.

(c) Héritage et polymorphisme. L'intégration de Junior et Java est faite de telle façon en Rejo que l'héritage et le polymorphisme sont préservés. En particulier on peut parler d'héritage de comportements réactifs.

Le problème central de la notion d'objet est la propriété de dynamisme que l'on voudrait exprimer sous la forme de nouveaux champs (variables et méthodes réactives) dans l'objet. Le programmeur peut ajouter des nouveaux comportements réactifs à un Rejo, mais ceux-ci ne sont pas vus par les entités externes, c'est-à-dire qu'un autre Rejo ne peut pas faire référence aux champs ajoutés à partir d'une référence de l'objet, par exemple `obj.method_added()`. Le programmeur doit implémenter son propre mécanisme d'ajout de champs comme celui qui existe dans les Icobjs (basé sur une table de hash).

3. Optimisation et vérification du code. Le compilateur de Rejo, Rejoc (construit avec JavaCC) a la capacité d'optimiser les actions atomiques. Ce compilateur réalise deux phases, une pour compiler le code réactif en code Java, et une autre pour compiler le code Java. Pour réduire les erreurs de traduction , on a créé une batterie de tests qui vérifie la syntaxe et la sémantique des programmes Rejo. Cette batterie (enrichie par la batterie de tests de Junior) nous a permis de créer des nouvelles versions de Rejoc avec un haut degré de confiance et, dans certains cas, de détecter des bugs dans Junior.

4. Expérimentation en C. On a construit une variante de Rejo en C appelée RAUL (*Reactive Algorithmes Unified Language*). RAUL nous a permis d'expérimenter avec les optimisations liées à la construction d'un arbre de parsing et de confronter les problèmes liés à la création d'un langage impératif réactif qui unifie les différentes langages créés (Reactive Scripts, Rhum, Rejo, Lotf, Jrc); le *garbage collector*, l'utilisation des pointeurs et la définition d'un modèle à objets sont les problèmes les plus importants dans la conception d'un langage qui ne dépend pas de Java ou d'un langage à la C. Le développement de RAUL est un des sujets de recherche à suivre.

5. Applications. Au début de la création de Rejo, sa conception fut influencée par son utilisation dans la construction de la plate-forme PING; Rejo a été utilisé dans la construction de comportements réactifs. Rejo a été utilisé aussi dans la réalisation de composants d'une application de simulation de vie artificielle [LIM 01] s'appuyant sur une plate-forme d'environnements virtuels partagés pour : 1) coder des règles de métabolisme et de comportements de survie, 2) développer le simulateur et vérifier la viabilité du système, et 3) porter l'application sous l'environnement partagé (Continuum).

En résumé, Rejo est un langage qui a les caractéristiques suivantes :

- *Concurrence* : les objets réactifs de Rejo s'exécutent en concurrence (dans un mode coopératif). Comme Junior, Rejo constitue donc une alternative aux threads Java ;
- *Interactivité* : les objets Rejo communiquent en utilisant des événements diffusés instantanément ;
- *Réactivité* : les comportements des Rejos sont construits à partir de primitives permettant de réagir à la présence et à l'absence des événements diffusés ;
- Dynamicité : les Rejos peuvent être modifiés, ajoutés et enlevés du système dynamiquement, en cours d'exécution ;
- Multi-plates-formes : la portabilité est la conséquence de l'utilisation du langage Java (byte code Java).

ROS

Dans le chapitre 7 on a présenté une nouvelle plate-forme, appelée ROS (*Reactive Operating System*, pour l'exécution de programmes réactifs. ROS ressemble à un système d'exploitation car il est constitué par une

interface graphique, un shell, une interface de programmation, un ensemble des services et un micro-noyau modulaire. Le langage utilisé pour programmer les applications en ROS est Rejo.

ROS est une plate-forme qui a les caractéristiques suivantes :

- SAM. ROS est un Système d'Agents Mobiles (SAM); les agents de ROS sont des objets réactifs programmés avec Rejo. Les agents de ROS sont constitués d'un nom et d'une structure de contrôle pour les suspendre, les résumer, les tuer et les modifier en ajoutant des nouveaux comportements en parallèle. La structure de contrôle des agents est un élément crucial dans le système, elle doit être implémentée efficacement et sans ambiguïté. Une première structure de contrôle d'agents a été programmée en Rejo; puis une autre a été créée en définissant une nouvelle instruction réactive appelée thread réactif. Le thread réactif dispose d'une sémantique formelle et d'une implémentation efficace.
- Services. ROS offre fondamentalement deux services aux agents : la sérialisation et la migration. Ces deux opérations sont le point de départ pour implémenter des services plus complexes, comme la persistance ou la gestion de réseaux. La migration des agents est une migration forte qui peut être réactive ou proactive [2] grâce au mécanisme de diffusion d'événements.
- Architecture. La philosophie que l'on a adoptée dans la conception de ROS est d'avoir un système minimal que l'on puisse entretenir et modifier facilement. ROS a une architecture modulaire analogue à celle d'un système d'exploitation à laquelle on peut ajouter et enlever des modules selon les besoins. ROS ressemble à un système d'exploitation avec un micro-noyau car il offre juste la notion de processus (les agents) et peut démarrer sans aucun des autres modules (service de migration, shell, etc.).
- Applications. Deux applications ont été construites pour étendre les fonctionnalités du système: un shell réactif (Rsh) et une interface graphique (les Ricobjs). Rsh permet de tester interactivement les applications réactives construites avec Rejo ; les Ricobjs offrent le moyen d'animer les applications en utilisant en ensemble de routines graphiques et de simulation de comportements physiques. Rsh et les Ricobjs sont construits avec Rejo.

Le système ROS a montré qu'il existe trois notions de base indispensables dans les systèmes réactifs : une structure de contrôle, un système de nommage pour réaliser les opérations de contrôle et une plate-forme qui gère les deux notions précédentes. Ces éléments se retrouvent dans les Cubes, les Icobjs, les agents de RAMA et bien entendu dans les agents de ROS. ROS a montré aussi que le parallélisme de l'approche réactive est une alternative réelle à l'utilisation traditionnelle de threads car l'opérateur de parallélisme passe mieux à l'échelle et a une sémantique claire.

Perspectives

Plusieurs axes de recherche peuvent être suivis :

1. Continuer l'étude du modèle d'objet réactif. Comme on l'a déjà mentionné, le dynamisme pose des problèmes avec la notion d'objet réactif. L'idée de pouvoir ajouter des champs dans l'objet pour ensuite faire des opérations sur l'objet est une propriété souhaitable. Les instructions Dynapar et Run sont un des premiers pas dans cette voie. Un autre aspect de cette étude est la relation entre les objets réactifs et la machine réactive. A l'heure actuelle, le moteur réactif ne connaît pas l'existence des objets réactifs. On pourrait construire un moteur spécialisé qui effectuerait des opérations sur les objets réactifs et non uniquement sur les instructions et les événements.
2. Cr dans le noyau de Linux. Un des sujets qui m'attire est l'utilisation du réactif dans la construction d'un module en Linux. Le dynamisme de l'approche réactive en fait un bon candidat pour ce genre d'opérations. D'autre part, les systèmes d'exploitation peuvent certainement profiter des atouts de l'approche réactive. L'application de Cr dans la construction des systèmes embarqués est une autre voie possible.

[2] Parfois connues comme migration objective ou subjective, respectivement.

3. Les SugarCubes ont eu un grand impact dans le développement de Rejo qui reprend les principales structures de contrôle et le mécanisme de notification. D'autres aspects comme les variables réactives, ou l'exécution des sous-horloges, restent des sujets à étudier. On a commencé l'étude de la relation entre Rejo et les SugarCubes par la construction d'une version expérimentale (utilisée dans les *workbenchs*) du compilateur de Rejo qui génère du code en SugarCubes.

4. D'autres sujets de recherche plus généraux sont possibles : 1) la création de systèmes ROS synchronisés (comme dans le cas des machines réactives distribuées), 2) la création (comme dans les FairThreads) de programmes réactifs qui ont des composants qui ne sont pas réactifs et qui, dans certaines conditions, peuvent se détacher de la notion d'instant pour s'exécuter en parallélisme asynchrone avec les instructions réactives.

Annexe A

Grammaire formelle de Rejo

Cette annexe présente la grammaire du langage Rejo en utilisant la syntaxe BNF définie dans [AHO 86]. Cette grammaire est utilisée comme entrée pour le compilateur de grammaires JavaCC [WEB JavaCC] qui produit du code Java [JavaDocs]. Les mots en majuscules représentent les tokens en provenance de l'analyseur lexical. La grammaire est présentée dans une fonte normale, cursive, pour distinguer quatre sections : la structure des programmes Java, les expressions, les structures de contrôle et les méthodes réactives.

```
1  CompilationUnit ::=      ( PackageDeclaration )? ( ImportDeclaration )*
2                           ( TypeDeclaration )* <EOF>
3
4  PackageDeclaration ::=   <PACKAGE> Name <SEMICOLON>
5
6  ImportDeclaration ::=    <IMPORT> Name ( <DOT> <STAR> )? <SEMICOLON>
7
8  TypeDeclaration ::=        ClassDeclaration
9                           | InterfaceDeclaration
10                          | <SEMICOLON>
11
12 ClassDeclaration ::=     (<ABSTRACT>|<FINAL>|<PUBLIC>)* UnmodifiedClassDeclaration
13
14 UnmodifiedClassDeclaration ::= <CLASS> <IDENTIFIER> ( <EXTENDS> Name )?
15                          ( <IMPLEMENTS> NameList )? ClassBody
16
17 ClassBody ::=            <LBRACE> ( ClassBodyDeclaration )* <RBRACE>
18
19 NestedClassDeclaration ::= (<STATIC>|<ABSTRACT>|<FINAL>|<PUBLIC>|<PROTECTED>
20                             |<PRIVATE>)* UnmodifiedClassDeclaration
21
22 ClassBodyDeclaration ::=   Initializer
23                          | NestedClassDeclaration
24                          | NestedInterfaceDeclaration
25                          | ConstructorDeclaration
26                          | MethodDeclaration
27                          | RMethodDeclaration
28                          | FieldDeclaration
29
30 MethodDeclarationLookahead ::= ("public" |"protected" |"private" |"static" |
31                                "abstract" |"final" |"native" |"synchronized")*
32                                ResultType ( <IDENTIFIER> | RTokens ) "("
33
34 RTokens ::=         <IF>    | <CALL> | <WHEN> | <UNTIL>  | <REPEAT>
```

```
35                  | <PAR>  | <STOP> | <LINK> | <FREEZ>  | <INLINE>
36                  | <GEN>  | <CTRL> | <WAIT> | <LOCAL>  | <PRNATM>
37                         | <HAND> | <LOOP> | <INOUT>  | <PRNLATM>
38                         | <ATOM> | <REAC> |             <GENERATE>
39
40 RMethodDeclarationLookahead ::= ("public"|"protected"|"private"|"static"
41                                |"abstract"|"final"|"native"|"synchronized")*
42                                <REAC>
43
44 InterfaceDeclaration ::= ( <ABSTRACT> | <PUBLIC> )* UnmodifiedInterfaceDeclaration
45
46 NestedInterfaceDeclaration ::= (<STATIC> | <ABSTRACT> | <FINAL> | <PUBLIC>
47                                | <PROTECTED> |<PRIVATE>)*
48                                UnmodifiedInterfaceDeclaration
49
50 UnmodifiedInterfaceDeclaration ::=
51                         <INTERFACE> <IDENTIFIER> ( "extends" NameList )?
52                         <LBRACE> ( InterfaceMemberDeclaration )* <RBRACE>
53
54 InterfaceMemberDeclaration ::=    NestedClassDeclaration
55                                | NestedInterfaceDeclaration
56                                | MethodDeclaration
57                                | FieldDeclaration
58
59 FieldDeclaration ::=     ( <PUBLIC> | <PROTECTED> | <PRIVATE> | <STATIC> | <FINAL>
60                         | <TRANSIENT> | <VOLATILE> )* Type VariableDeclarator
61                         ( <COMMA> VariableDeclarator )* <SEMICOLON>
62
63 VariableDeclarator ::=  VariableDeclaratorId ( <ASSIGN> VariableInitializer )?
64
65 VariableDeclaratorId ::= ( <IDENTIFIER> | RTokens ) ( <LBRACKET> <RBRACKET> )*
66 VariableInitializer ::=   ArrayInitializer
67                         | Expression
68
69 ArrayInitializer ::=    <LBRACE> (VariableInitializer ("," VariableInitializer )*)?
70                         ( <COMMA> )? <RBRACE>
71
72 MethodDeclaration ::=    ( <PUBLIC> | <PROTECTED> | <PRIVATE> | <STATIC> | <ABSTRACT>
73                         | <FINAL> | <NATIVE> | <SYNCHRONIZED> )* ResultType
74                         MethodDeclarator (<THROWS> NameList )? (Block |<SEMICOLON>)
75
76 MethodDeclarator ::=     ( <IDENTIFIER> | RTokens ) FormalParameters
77                         ( <LBRACKET> <RBRACKET> )*
78
79 FormalParameters ::=     <LPAREN> ( FormalParameter (<COMMA> FormalParameter )*)?
80                         <RPAREN>
81
82 FormalParameter ::=      ( <FINAL> )? Type VariableDeclaratorId
83
84 ConstructorDeclaration ::= (( <PRIVATE> | <PROTECTED> | <PUBLIC> ))? <IDENTIFIER>
85                            FormalParameters ( "throws" NameList )? <LBRACE>
86                            ( ExplicitConstructorInvocation )?
87                            ( BlockStatement )* <RBRACE>
88
89 ExplicitConstructorInvocation ::= "this" Arguments ";"
90                         | ( PrimaryExpression "." )? "super" Arguments ";"
91
92 Initializer ::=          ( <STATIC> )? Block
93
```

```
94  Type ::=                ( PrimitiveType | Name ) ( <LBRACKET> <RBRACKET> )*
95
96  PrimitiveType ::=       ( <BOOLEAN> | <CHAR> | <BYTE> | <SHORT> | <INT>
97                          | <LONG> | <FLOAT> | <DOUBLE>)
98
99  ResultType ::=          <VOID> | Type
100
101 Name ::=                ( <IDENTIFIER> | <STOP> | <ATOM> )
102                         ( <DOT> ( <IDENTIFIER> | RTokens ) )*
103
104 NameList ::=            Name ( <COMMA> Name )*

105 Expression ::=           CondExpression ( AssignmentOperator Expression )?
106
107 AssignmentOperator ::= ( "=" | "*=" | "/=" | "\%=" | "+=" | "-=" | "<<=" | ">>="
108                         | ">>>=" | "&=" | "^=" | "|=" )
109
110 CondExpression ::= CondOrExpression ( <HOOK> Expression <COLON>
111                           CondExpression )?
112
113 CondOrExpression ::= CondAndExpression ( <SC_OR> CondAndExpression )*
114
115 CondAndExpression ::= InclusiveOrExpression ( <SC_AND> InclusiveOrExpression )*
116
117 InclusiveOrExpression ::= ExclusiveOrExpression ( <BIT_OR> ExclusiveOrExpression )*
118
119 ExclusiveOrExpression ::= AndExpression ( <XOR> AndExpression )*
120
121 AndExpression ::= EqualityExpression ( <BIT_AND> EqualityExpression )*
122
123 EqualityExpression ::= InstanceOfExpression ((<EQ> | <NE>) InstanceOfExpression )*
124
125 InstanceOfExpression ::= RelationalExpression ( <INSTANCEOF> Type )?
126
127 RelationalExpression ::= ShiftExpression ((<LT>|<GT>|<LE>|<GE>) ShiftExpression)*
128
129 ShiftExpression ::= AdditiveExpression ( ( <LSHIFT> | <RSIGNEDSHIFT> |
130                       <RUNSIGNEDSHIFT> ) AdditiveExpression )*
131
132 AdditiveExpression ::= MultiplicativeExpression (( <PLUS> | <MINUS>)
133                         MultiplicativeExpression )*
134
135 MultiplicativeExpression ::= UnaryExpression ( ( <STAR> | <SLASH> | <REM> )
136                                 UnaryExpression )*
137
138 UnaryExpression ::=       ( <PLUS> | <MINUS> ) UnaryExpression
139                         | PreIncrementExpression
140                         | PreDecrementExpression
141                         | UnaryExpressionNotPlusMinus
142
143 PreIncrementExpression ::= <INCR> PrimaryExpression
144
145 PreDecrementExpression ::= <DECR> PrimaryExpression
146
147 UnaryExpressionNotPlusMinus ::= ( <TILDE> | <BANG> ) UnaryExpression
148                         | CastExpression
149                         | PostfixExpression
150
151 CastLookahead ::=         "(" PrimitiveType
```

```
152                             | "(" Name "[" "]"
153                             | "(" Name ")" ( "~" | "!" | "(" | <IDENTIFIER> | "this"
154                                             | "super" | "new" | Literal )
155
156  PostfixExpression ::=      PrimaryExpression ( <INCR> | <DECR> )?
157
158  CastExpression ::=           <LPAREN> Type <RPAREN> UnaryExpression
159                             | <LPAREN> Type <RPAREN> UnaryExpressionNotPlusMinus
160
161  PrimaryExpression ::=      PrimaryPrefix ( PrimarySuffix )*
162
163  PrimaryPrefix ::=            Literal
164                             | <THIS>
165                             | <SUPER> <DOT> <IDENTIFIER>
166                             | <LPAREN> Expression <RPAREN>
167                             | AllocationExpression
168                             | ResultType <DOT> <CLASS>
169                             | Name
170
171  PrimarySuffix ::=            <DOT> <THIS>
172                             | "." AllocationExpression
173                             | <LBRACKET> Expression <RBRACKET>
174                             | <DOT> <IDENTIFIER>
175                             | Arguments
176
177  Literal ::=                ( <INTEGER_LITERAL>
178                             | <FLOATING_POINT_LITERAL>
179                             | <CHARACTER_LITERAL>
180                             | <STRING_LITERAL>
181                             | BooleanLiteral
182                             | NullLiteral )
183
184  BooleanLiteral ::=         "true" | "false"
185  NullLiteral ::=            "null"
186
187  Arguments ::=              <LPAREN> ( ArgumentList )? <RPAREN>
188
189  ArgumentList ::=           Expression ( <COMMA> Expression )*
190
191  AllocationExpression ::=  <NEW> PrimitiveType ArrayDimsAndInits
192                             | <NEW> Name (ArrayDimsAndInits | Arguments (ClassBody)?)
193
194  ArrayDimsAndInits ::=     (<LBRACKET> Expression <RBRACKET> )+ (<LBRACKET>
195                            <RBRACKET>)* | (<LBRACKET> <RBRACKET>)+ ArrayInitializer

196  Statement ::=                LabeledStatement
197                             | Block
198                             | EmptyStatement
199                             | SwitchStatement
200                             | ifStatement
201                             | StatementExpression <SEMICOLON>
202                             | WhileStatement
203                             | DoStatement
204                             | ForStatement
205                             | BreakStatement
206                             | ContinueStatement
207                             | ReturnStatement
208                             | ThrowStatement
209                             | SynchronizedStatement
```

```
210                          | TryStatement
211
212 LabeledStatement ::=     <IDENTIFIER> <COLON> Statement
213
214 Block ::=                 <LBRACE> ( BlockStatement )* <RBRACE>
215
216 BlockStatement ::=          LocalVariableDeclaration <SEMICOLON>
217                          | Statement
218                          | UnmodifiedClassDeclaration
219                          | UnmodifiedInterfaceDeclaration
220
221 LocalVariableDeclaration ::= ("final")? Type VariableDeclarator
222                          (","  VariableDeclarator)*
223
224 EmptyStatement ::=       <SEMICOLON>
225
226 StatementExpression ::= PreIncrementExpression
227                          | PreDecrementExpression
228                          | PrimaryExpression ( <INCR> | <DECR> | AssignmentOperator
229                            Expression )?
230
231 SwitchStatement ::=      <SWITCH> <LPAREN> Expression <RPAREN> <LBRACE> (SwitchLabel
232                          ( BlockStatement )* )* <RBRACE>
233
234 SwitchLabel ::=            <CASE> Expression <COLON>
235                          | <_DEFAULT> <COLON>
236
237 ifStatement ::=          <IF> <LPAREN> Expression <RPAREN> Statement
238                          ( <ELSE> Statement )?
239
240 WhileStatement ::=       <WHILE> <LPAREN> Expression <RPAREN> Statement
241
242 DoStatement ::=          <DO> Statement <WHILE> <LPAREN> Expression <RPAREN>
243                          <SEMICOLON>
244
245 ForStatement ::=         <FOR> <LPAREN> ( ForInit )? <SEMICOLON> ( Expression )?
246                          <SEMICOLON> ( ForUpdate )? <RPAREN> Statement
247
248 ForInit ::=              LocalVariableDeclaration | StatementExpressionList
249
250 StatementExpressionList ::= StatementExpression ( <COMMA> StatementExpression )*
251
252 ForUpdate ::=            StatementExpressionList
253
254 BreakStatement ::=       <BREAK> ( <IDENTIFIER> )? <SEMICOLON>
255
256 ContinueStatement ::=    <CONTINUE> ( <IDENTIFIER> )? <SEMICOLON>
257
258 ReturnStatement ::=      <RETURN> ( Expression )? <SEMICOLON>
259
260 ThrowStatement ::=       <THROW> ( Expression )? <SEMICOLON>
261
262 SynchronizedStatement ::= <SYNCHRONIZED> <LPAREN> Expression <RPAREN> Block
263
264 TryStatement ::=         <TRY> Block ( <CATCH> <LPAREN> FormalParameter
265                                  <RPAREN> Block )*( <FINALLY> Block )?

266 RMethodDeclaration ::= ( <PUBLIC> | <PROTECTED> | <PRIVATE> | <STATIC>
267                        | <ABSTRACT> | <FINAL> | <NATIVE> | <SYNCHRONIZED> )*
268                       <REAC> RMethodDeclarator RMBlock
```

```
269
270 RMethodDeclarator ::=    <IDENTIFIER> RFormalParameters (<LBRACKET> <RBRACKET>)*
271
272 RFormalParameters ::=    <LPAREN> (RFormalParameter (<COMMA>
273                          RFormalParameter )*)?  <RPAREN>
274
275 RFormalParameter ::=     RType ( <FINAL> )? RVariableDeclaratorId
276
277 RType ::=                ( RPrimitiveType | RName ) ( <LBRACKET> <RBRACKET> )*
278
279 RPrimitiveType ::=       ( <BOOLEAN> | <CHAR> | <BYTE> | <SHORT> | <INT>
280                          | <LONG> | <FLOAT> | <DOUBLE>)
281
282 RName ::=                <IDENTIFIER> ( <DOT> <IDENTIFIER> )*
283
284 RVariableDeclaratorId ::= <IDENTIFIER> ( <LBRACKET> <RBRACKET> )*
285
286 RMBlock ::=              <LBRACE> SeqStatements <RBRACE>
287
288 RStatementS ::=          RStatement | <LBRACE> SeqStatements <RBRACE>
289
290 SeqStatements ::=        ( RStatement )*
291
292 RStatement ::=           SetOfRStatements | AtomStatement
293
294 SetOfRStatements ::=       If    | Call  | When    | Repeat  | Generate
295                          | Par   | Stop  | Until   | Inline  | Variables
296                          | Try   | Wait  | Local   | Nothing | CtrlFreez
297                                  | Loop  | Print   | Scanner | SuffixEvent
298                                  | Link  | Dynapar
299
300 AtomStatement ::=        <ATOM> <LBRACE>
301                            ( StatementExpression <SEMICOLON> | ifStatement
302                            | DoStatement | ForStatement | BreakStatement
303                            | WhileStatement | SwitchStatement
304                            | ContinueStatement | LocalVariableDeclaration
305                           <SEMICOLON> )*
306                          <RBRACE>
307                          | StatementExpression <SEMICOLON>
308
309 Variables ::=            RLocalVariableDeclaration <SEMICOLON>
310
311 RLocalVariableDeclaration ::= RType RVariableDeclarator
312                          (<COMMA> RVariableDeclarator)*
313
314 RVariableDeclarator ::= RVariableDeclaratorId (<ASSIGN> VariableInitializer)?
315
316 Nothing ::=      <SEMICOLON>
317
318 Stop ::=         <STOP> <SEMICOLON>
319
320 Print ::=        ( <PRNATM> | <PRNLATM> ) <LPAREN> SubSetOfExpression
321                  <RPAREN> <SEMICOLON>
322
323 Call ::=         <CALL> MyExpression <SEMICOLON>
324
325 Inline ::=       <INLINE> MyExpression <SEMICOLON>
326
327 Generate ::=     ( <GEN> | <GENERATE> ) SubSetOfExpression
328                  ( <COMMA> MyExpression )? <SEMICOLON>
329
330 Wait ::=         <WAIT> CondExp ( <COMMA> SubSetOfExpression )? <SEMICOLON>
331
332 SuffixEvent ::= <STRING_LITERAL> ( <BANG> | <HOOK> ) <SEMICOLON>
333
334 When ::=         <WHEN> <LPAREN> CondExp <RPAREN>
335                          RStatementS
336                  ( <ELSE> RStatementS )?
```

```
337
338 If ::=             <IF> <LPAREN> MyExpression <RPAREN>
339                         RStatementS
340                   ( <ELSE> RStatementS )?
341
342 Try ::=           <TRY> RStatementS <CATCH> RStatementS
343
344 Loop ::=          <LOOP> RStatementS
345
346 Par ::=           <PAR> <LBRACE> SeqStatements ( <SC_OR>
347                   SeqStatements )* <RBRACE>
348
349 Scanner ::=       <SCAN> <LPAREN> SubSetOfExpression <RPAREN>
350                   ( <LBRACE> ( StatementExpression <SEMICOLON> | ifStatement
351                              | DoStatement | ForStatement | BreakStatement
352                              | WhileStatement | SwitchStatement
353                              | ContinueStatement | LocalVariableDeclaration
354                              <SEMICOLON> )*
355                     <RBRACE> | AtomStatement )
356
357 Local ::=         <LOCAL> <LPAREN> ( <STRING_LITERAL> | <INTEGER_LITERAL> )
358                   (<COMMA>(<STRING_LITERAL>|<INTEGER_LITERAL>))* <RPAREN>
359                         RStatementS
360
361 CtrlFreez ::=     ( <CTRL> | <FREEZ> ) <LPAREN> SubSetOfExpression <RPAREN>
362                         RStatementS
363
364 Link ::=  <LINK> <LPAREN> SubSetOfExpression <RPAREN> RStatementS
365
366 Repeat ::=  <REPEAT> <LPAREN> MyExpression <RPAREN> RStatementS
367
368 Until ::=         <UNTIL> <LPAREN> CondExp <RPAREN>
369                   RStatementS ( <HAND> RStatementS )?
370
371 Dynapar ::= <DPAR> SubSetOfExpression <RPAREN> RStatementS
372
373 CondExp ::=       AND_Exp
374
375 AND_Exp ::=       OR_Exp ( <SC_AND> AND_Exp )?
376
377 OR_Exp ::=        Not_Exp ( <SC_OR> OR_Exp )?
378
379 Not_Exp ::=         <BANG> Parent_Exp
380                   | Parent_Exp
381
382 Parent_Exp ::=      <LPAREN> AND_Exp <RPAREN>
383                   | <ALL>
384                   | SubSetOfExpression
385
386 SubSetOfExpression ::= AdditiveExpression
387
388 MyExpression ::= Expression
```

Bibliographie

[ACM 87] *Lustre, a declarative language for programming synchronous systems*. In 14th ACM Symposium on Principles of programming Languages, POPL'87, Munchen, January 1987.

[ACO 00a] *REJO/ROS*, Raúl Acosta-Bermejo, Rapport de DEA RSD à l'ESSI, Sophia Antipolis France, Septembre 2000.

[ACO 00b] *Reactive Operating System, REactive Java Objects*, Raúl Acosta-Bermejo, Notere'2000, ENST Paris, Novembre 2000.

[ACO 02] *La programmation en Rejo*, Raúl Acosta-Bermejo, Les intergiciels, développements récents dans CORBA, Java RMI et les agents mobiles, Publications Hermes Sience Lavoisier 2002, http://www.hermes-science.com.

[ACO 03] *Quelle méthode et quel langage choisir?*, Raúl Acosta-Bermejo, L'informatique professionnelle, Publications Gartner, Dossier développement, Numéro 210, janvier 2002,

[AHO 86] *Compilers : Principles, Techniques, and Tools*, A.V. Aho, R. Sethi, J.D. Ullman, Addison-Wesley, 1986.

[AND 95] *SyncCharts: a visual representation of reactive behaviors*, C. André, Technical Report RR 95-52, I3S, Sophia-Antipolis, France, October 1995.

[AND 96] *Representation and Analysis of Reactive Behaviors: A Synchronous Approach*, C. André, CESA'96, IEEE-SMC, Lille(F), July 9-12, 1996.

[ARI 82] *Principles of Concurrent and Distributed Programming*, M. Ben-Ari, C.A.R. Hoare Prentice Hall International Series in Computer Science, Series Editor 1982.

[AUG 99a] *Objets à contrôle réactif: Un modèle de conception pour applications concurrentes*, M. Augeraud, F. Bertrand, Technique et science informatiques, janvier 1999.

[AUG 99b] *BDL: A Specialized Language for per-Object Reactive Control*, M. Augeraud, F. Bertrand, IEEE Transactions on Software Engineering, a paraître, 1999.

[BER 87] *Programmation synchrone des systèmes réactifs: le langage Esterel*, G. Berry, P. Couronné, et G. Gonthier. Techniques et Sciences Informatiques, 6(4), 1987.

[BER 92] *The Esterel Synchronous Language: Design, Semantics Implementation*, G. Berry, G. Gonthier. Science of Computer Programming, 19(2), 1992.

[BER 96] *Un modèle de contrôle réactif pour les langages à objets concurrents*, F. Bertrand, Thèse de doctorat, Université de La Rochelle, L3I, Avenue Marillac, 17042 La Rochelle Cedex, janvier 1996.

[BOU 91] *Reactive-C: An extention of C to Program Reactive Systems*, F. Boussinot, Software Practice and Experience, 21(4), 401-428, 1991.

[BOU 96a] *Reactive Scripts*, F. Boussinot, L. Hazard, Proc. International Conference on Real-Time Computing Systems and Applications, RTCSA, Seoul, IEEE, October 1996.

[BOU 96b] *Reactive Scripts*, F. Boussinot, L. Hazard, INRIA Research Report 2868, May 1996.

[BOU 96c] *Icobj programming*, F. Boussinot, INRIA Research Report 3028, October 1996, ou le rapport 2796 en français.

[BOU 96d] *Reactive objects*, F. Boussinot, G. Doumenc, et J.B. Stefani. INRIA Research Report 2664, October 1995. Ou dans Annales des Télécommunications, 51(9-10), 1996.

[BOU 96e] *La programmation réactive : Applications aux systèmes communicants*, F. Boussinot, Masson, Collection Technique et Scientifique des Télécommunications, 1996.

[BOU 96f] *The SL Synchronous Language*, F. Boussinot, and R. de Simone, IEEE Transactions on Software Engineering, vol. 22(4), p. 256-266, April 1996.

[BOU 97] *The SugarCubes Tool Box - Rsi-Java implementation*, F. Boussinot, J-F. Susini, INRIA Research Report 3247, 1997. http://www-sop.inria.fr/mimosa/rp/RapportsRecherche/RR-3247.pdf

[BOU 98a] *Distributed Reactive Machines*, F. Boussinot, J.F. Susini, L. Hazard, INRIA Research Report 3376, March 1998.

[BOU 98b] *SugarCubes Implementation of Causality*, F. Boussinot, Research Report 3487, September 1998.

[BOU 98c] *The SugarCubes Tool Box: A Reactive Java Framework*, F. Boussinot, J-F. Susini, Software Practice and Experience, 28(14), 1531-1550, 1998.

[BOU 00a] *Java threads and SugarCubes*, F. Boussinot, J-F. Susini, Software Practice and Experience, 30(5), 545-566, 2000.

[BOU 00b] *Junior Automata*, F. Boussinot, Research Report RR-4031 Octobre 2000, at http://www-sop.inria.fr/mimosa/rp/ et and http://www.inria.fr/rrrt/rr-4031.html.

[BOU 00c] *Pickling threads state in the Java system*, S. Bouchenak, D. Hagimont, Proceedings of 33rd International Conference on Technology of Object-Oriented Languages (TOOLS Europe/2000), Mont-Saint-Michel, France, juin 2000. http://sirac.inrialpes.fr/~bouchena.

[BOU 00d] *Junior semantics*, F. Boussinot, J-F. Susini, Draft of Research Report October 2000, founded at http://www-sop.inria.fr/mimosa/rp/.

[BOU 02] *FairThreads*, F. Boussinot, http://www-sop.inria.fr/mimosa/rp/.

[BRA 93] *An Efficient Object-Oriented Variation of the Statecharts Formalism for Distributed Real-Time Systems*, Bran Selic, Computer Hardware Description Languages and their Applications (CHDL) 1993: 335-344.

[BRI 96] *Objets, parallélisme et répartition*, J. Briot, R. Guerraoui, Technique et science informatiques, vol. 15, no 6, p. 765-800, 1996.

[BRU 01] *Rapport de DEA : Etude de la cohérence dans les jeux en réseaux distribués* Christian Brunette Rapport de DEA RSD à l'ESSI, Sophia Antipolis France, juin 2001.

[BRU 02] *A Visual Reactive Framework for Dynamic Behavior Creation*, Christian Brunette, OOPSLA Second Workshop on Domain Specific Visual Languages October, 2002.

[CAB 98] *Reactive Tuple Spaces for Mobile Agent Coordination*, G. Cabri, L. Leonardi, F. Zambonelli, Lecture Notes in Computer Science, vol. 1477, 1998.

[CAS 87] *A functional extention to Lustre*. Paul. Caspi, Marc Pouzet, In 8th International Symposium on Languages for Intensional Programming, Sydney, May 1995.

[CAS 95] *Compilation of the ELECTRE reactive language into finite transition systems*, F. Cassez, O. Roux, Theoretical Computer Science, vol. 146, juillet 1995.

[CAS 96] *SynchronousKahn networks*, P. Caspi, M.Pouzet. In Proceedings of the International Conference on Functional Programming. ACM Press, 1996.

[CHE 95] *Itinerant Agents for Mobile Computing*, Chess, David, et al., IBM T.J. Watson Research Center, Yorktown Heights, New York, 1995.

[COL 98] *Design and implementation of Triveni: A process-algebraic API for threads + events.* C. Colby, L. J. Jagadeesan, et al., In Proceedings of the 1998 IEEE International Conference on Computer Languages. IEEE Computer Press, 1998. To appear.

[COL 99] *Semantics of Triveni : A process-algebraic API for threads + events*, C. Colby, et al., Electronic Notes in Theoretical Computer Science, 14, 1999.

[Coroutines] *Coroutines in C* Simon Tatham, http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html.

[DEM 01] *Programmation réactive fonctionnelle avec Senior*, Julien Demaria, Rapport de DEA Informatique à l'ESSI, Sophia Antipolis France, September 2001. http://www.inria.fr/mimosa/rp/Senior

[DER 92] *Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design*, Derek Coleman, Fiona Hayes, Stephen Bear, IEEE Transactions on Software Engineering (TSE), Volume 18, number 1, pages 9-18, 1992.

[DIJ 65] *Solution of a Problem in Concurrent Programming Control*, Edsger W. Dijkstra, Communications of the ACM, 8(9):569, September 1965.

[ELL 97] *Functional reactive animation* C. Elliott and P. Hudak.. In Proceedings of the International Conference on Functional Programming. ACM Press, 1997.

[FEL 87] *A Reduction Semantics for Imperative Higher-Order Languages*, In Proc. Conf. on Parallel Architecture and Languages Europe, pages 206-223. Lecture Notes 259 in Computer Science. Springer-Verlag, 1987.

[FRA 92] *Program Verification*, Nissim Francez, Addison-Wesley, Reading, MA, 1992.

[FRA 96] *Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents*, S. Franklin, A. Graesser, Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996, http://www.msci.memphis.edu/~franklin/AgentProg.html.

[FUG 98] *Understanding Code Mobility*, A. Fugette, G-P. Picco, G. Vigna, IEEE Transactions on Software Engineering, vol. 24, No 5, 1998, pp.342-361.

[FUN 98] *Transparent Migration of Java-based Mobile Agents(Capturing and Reestablishing the State of Java Programs)* S. Fünfrocken, Proceedings of Second International Workshop Mobile Agents 98 (MA'98), Stuttgart, Allemagne, septembre 1998. http://www.informatik.tu-darmstadt.de/~fuenf.

[FUR 95] *Parallélisme et distribution en C++ : une revue des langages existants*, N. Furmenton, Y. Roudier, G. Siegel, Rapport technique RR 95-02, I3S, Sophia-Antipolis, FRANCE, 1995.

[GAM 95] *Design Patterns: Elements of Reusable Object Oriented Software*, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995.

[GUE 91] *Programming realtime applications with SIGNAL*, P. Le Guernic, T. Gautier, M. Le Borgne, et M. Le Maire. Rapport Technique 1446, INRIA-Rennes, 1991 apparu aussi en Proceedings of the IEEE, 79, 9, p 1321-1336.

[GUE 96] *Un environnement opérationnel de conception et de réalisation de systèmes multi-agents*, Z. Guessoum, Thèse de doctorat, LAFORIA, Université Paris VI, mai 1996.

[GON 79] *Graphes et algorithmes*, M. Gondran, M. Minoux, Eyrolles, 1979. Collection de la Direction des Études et Recherches d?Électricité de France.

[HAL 91] *The Synchronous dataflow Programming Language Lustre*. N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud. Proceedings f the IEEE, 79(9):1305-1320, September 1991.

[HAL 93a] *Synchronous programming of reactive systems*. N. Halbwachs, Kluwer Academic Publishers, 1993.

[HAL 93b] *The Synchronous languages Twelve Years later* A. Benveniste et al. Proceedings of the , VOL. 91, NO. 1, January 2003.

[HAL 02] *A tutorial of Lustre*. N. Halbwachs, P. Raymond, January 2002.

[HAR 85] *On the Development of Reactive Systems*. D. Harel, A. Pnueli. In Logic and Models of Concurrent Systems, NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems, Springer Verlag, Vol. 13, pp 477-498, New York, 1985.

[HAR 87] *Statecharts: A visual formalism for complex systems.*, D.Harel. Science of Computer programming, 8:231-274, 1987.

[HAR 96] *The Statemate Semantics of Statecharts*, D. Harel, A. Naamad, ACM Transactions on Software Engineering and Methodology, 5(4):292-333, 1996.

[HAZ 99] *The Junior reactive kernel*, L. Hazard, J-F. Susini, F. Boussinot, INRIA Research Report 3732, July 1999.

[HAZ 00a] *Programming with Junior*, L. Hazard, J-F. Susini, F. Boussinot, INRIA Research Report 4027, July 2000.

[HOR 73] *An Axiomatic Definition of the Programming Language Pascal*, C. A. R. Hoare and Niklaus Wirth, Acta Informatica, 2, 1973, pp. 335?355.

[HPC 03] *Cours et TD de compilation*, Henri-Pierre Charles http://www.prism.uvsq.fr/~hpc/Enseignement/Compilation/Cours/.

[GRI 96] *The Icon Programming Language*, R. Griswold and M. Griswold. Peer-to-Peer Communications, Inc., third edition, 1996.

[JavaDocs] http://java.sun.com/docs/.

[JavaEmbed] *Design and Specification of Embedded Systems in Java using Succesive Formal Refinement*, J.S. Young, et al. In Design Automation Conference, 1998. http://java.sun.com/products/embeddedjava/.

[JavaLang] http://java.sun.com/docs/books/jls/.

[JavaThreads] *Why JavaSoft is Deprecating Thread.stop, Thread.suspend, Thread.resume,* JavaSoft Documentation, http://java.sun.com/products/, http://java.sun.com/products/jdk/1.2/docs/guide/misc/threadsPrimitiveDeprecation.html.

[Jona] *Jonathan* Dans le site http://www.objectweb.org/jonathan/download/index.html se trouve toute l'information.

[KAN 87] *Natural Semantics*, Giles Kahn, In Fourth Annual Symposium on Theoretical Aspects of Computer Science, edited by F. Bandenburg, G. VidalNaquet, and M. Wirsing, Lecture Notes in Computer Science, 247, Springer-Verlang, Berlin, 1987, pp. 22?39.

[LAU] Pour l'instant il n'y a aucun document officiel sur Rhum, néanmoins le lecteur intéressé peut consulter http://users.info.unicaen.fr/~eoulaoun/Projets/Corba/Moorea.htm, ou contacter Laurent Hazard par e-mail mailto:laurent.hazard@francetelecom.com.

[LAV 99] *ECL : A Specification Environment for System-Level Design*, Luciano Lavagno, Ellen Senrovich, apparu en DAC 99.

[LIM 01] Projet de fin d'études réalisé par Benjamin LIM chez France Télécom R&D. Pour plus d'information consulter http://membres.lycos.fr/benjaminlim/cv.html ou Anne Gerodolle mailto:anne.gerodolle@rd.francetelecom.fr.

[MAN 02] *Aspects dynamiques dans les langages synchrones: le cas des SugarCubes*, Louis Mandel, Rapport de DEA Programmation : Sémantique, Preuves et Langages, Laboratoire d'Informatique Paris 6, Septembre 2002.

[MAR 89] *Argos: an Automaton-Based Synchronous Language*, F. Maraninchi, Y. Remond, Computer Languages, Elsevier, No. 27, pags.61-92, 2001.

[MASIF] *Mobile Agent System Interoperability Facility* Dans le site http://www.fokus.gmd.de/research/cc/ecco/masif/ se trouve toute l'information.

[MEY 93] *Introduction to the Theory of Programming Languages*, Bertrand Meyer, Prentice Hall, Hemel Hempstead, UK, 1990.

[MEY 93] *Systematic concurrent object-oriented programming*, B. Meter, Communications of the ACM, vol. 36, no 9, p. 56-80, 1993.

[MOOR 00] *Moorea, a Service Execution Environment for Telecommunication Application*, B. Dillenseger, A-M Tagant, H. Tran-Viet, L. Hazard. http://users.info.unicaen.fr/~eoulaoun/Projets/Corba/Index.htm.

[MOS 99] *Foundations of Modular SOS*, Peter D. Mosses, BRICS Report Series RS-99-54, December 1999.

[NIK 99] *RAMA: Reactive Autonomous Mobile Agent*, Navid Nikaein, Rapport de DEA RSD à l'ESSI, Sophia Antipolis France, Septembre 1999.

[PAR 03] *Programmation réactive sur systèmes embarqués* Olivier Parra, Rapport de DEA Informatique à l'ESSI, Sophia Antipolis France, juillet 2003.

[PLO 81] *A Structural Approach to Structural Semantics*, G. Plotkin, Rapport Technique DAIMI FN-19, Université de Aarhus,1981.

[PNU 91] *What is a step: On the semantics of Statecharts*, A. Pnueli, M Shalev. Lecture Notes in Computer Science, 525, Springer Berlin 1991 pp244-464.

[POT 96] *Etude et prototypage en ESTEREL de la gestion de processus d'un micronoyau de système d'exploitation réparti avec garantie de service*, Olivier Potonniée, Thèse faite à l'université Paris VI, Spécialité en informatique, avril 1996.

[PUC 98] *Reactive Programming in Standard ML*, Riccardo R. Pucella. IEEE International Conference on Computer Language 1998. Vous pouvez trouver plus d'information sur sa page web: http://www.cs.cornell.edu/riccardo/.

[REP 95] *The SL Synchronous Language,* INRIA Research Report 2510, March 1995.

[RIC 01] *Description de comportements d'agents autonomes évoluant dans des mondes virtuels*, Nadine Richard, Thèse de doctorat, École Nationale Supérieure des Télécommunications, octobre 2001. http://www.inviwo.org, http://www.infres.enst.fr/~richard/Recherche/These/.

[ROS 02] *ECOMOBILE: A network Agent Ecosystem for Distributed Network Management* D. Rossier, R. Scheurer, in Proc. of ECUNM'02, 2nd European Conference on Universal Multiservice Networks, pp. 141-149, Colmar, France, April 8-10, 2002.

[SAK 00] *Bytecode Transformation for Portable Thread Migration in Java* T. Sakamoto, T. Sekiguchi, A. Yonezawa, Proceedings of Second International Workshop Mobile Agents 2000 (MA'2000), Zurich, Suisse, septembre 2000.

[SAM 02] *Application de la programmation réactive à la modélisation physique* Alexander Samarin Rapport de DEA RSD à l'ESSI, Sophia Antipolis France, juille 2002.

[SCH 88] *Denotational Semantics: A Methodology for Language Development*, David Schmidt, Wm. C. Brown Publishers, Dubuque, IA, 1988.

[SCH 00] *A verified hardware synthesis for Esterel* K. Schneider.. In F. Rammig, editor, International IFIP Workshop on Distributed and Parallel Embedded Systems, pages 205-214, Schlo Ehringerfeld, Germany, 2000. Kluwer Academic Publishers.

[SCH 01] *A new method for compiling schizophrenic synchronous programs*, K. Schneider, M. Wenz, International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), (Atlanta, Georgia), Nov. 2001.

[STO 77] *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, Joseph Stoy, MIT Press, Cambridge, MA, 1977.

[SUS 96] J.F. Susini, Rapport de DEA RSD à l'ESSI, Sophia Antipolis France, Juin 1996.

[SUS 01] *L'Approche Réactive au dessus de Java: sémantique et implémentation des SugarCubes et de Junior*, Jean-Ferdinand Susini, Thèse faite à l'Ecole de Mines de Paris, Spécialité en Informatique temps-réel Robotique Automatique, septembre, 2001.

[TAN 95] *Distributed Operating Systems* A.S. Tanenbaum, Prentice Hall 1995.

[TRU 00] *Portable support for transparent thread migration in Java*, E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, P. Verbaeten, Proc. ASA/MA 2000, Lecture Notes in Computer Science 1882, Springer, pp. 29-43.

[WEB Agents] http://www.cetus-links.org/oo_mobile_agents.html, http://agents.umbc.edu/.

[WEB AgentX] http://www.ietf.org/html.charters/OLD/agentx-charter.html.

[WEB Aglets] http://aglets.sourceforge.net/.

[WEB AgMail] http://www.mobility-list.org/.

[WEB AgSoc] http://www.agent.org/.

[WEB AgWeb] http://www.agentweb.net/, http://agents.umbc.edu/.

[WEB Concordia] http://www.merl.com/projects/concordia/WWW/.

[WEB FIPA] http://leonardo.telecomitalialab.com/fipa/, http://www.fipa.org/

[WEB Icobj] *Introduction to Icobjs*, http://www-sop.inria.fr/mimosa/rp/Icobjs/IntroToIcobjs/index.html.

[WEB JavaCC] *JavaCC*, Distributed by the WebGain company. http://www.webgain.com/products/java_cc/.

[WEB Jester] http://www.parades.rm.cnr.it/projects/jester/jester.html.

[WEB Mole] http://mole.informatik.uni-stuttgart.de/

[WEB RejoRos] Page web sur Rejo/Ros : http://www-sop.inria.fr/mimosa/rp/ROS/.

[Web Synch] http://www.inrialpes.fr/synalp/.

[Web SynEifel] http://ais.gmd.de/EES/synchronie/Doc/lang.html.

[Web synERJY] http://www.ais.fhg.de/~ap/sE/.

[WEB Voyager] http://www.recursionsw.com/products/voyager/.

[WEB W3C] http://www.w3.org/TR/2002/REC-UAAG10-20021217/, http://www.w3.org/WAI/UA/.

[WEL 96] *A Taxonomy for Distributed Object-Oriented Real-Time Systems*, L. Welch, D.K. Hammer, O. Van Roosmalen, OOPS Messenger, vol. 7, no 1, p. 78-85, janvier 1996.

[WHI 97] *Telescript Technology: Mobile Agents* White, J., In Bradshaw, J. (ed.) Software Agents. MIT Press, 1997.

Résumé :

Cette thèse présente Rejo un nouveau langage de haut niveau pour programmer des systèmes réactifs. Rejo, de l'anglais *REactive Java Objets*, est une extension à Java qui permet de définir des ***objets réactifs***, c'est-à-dire des objets qui encapsulent des données et des mélanges d'instructions Java et d'instructions réactives. Le modèle d'exécution est celui de l'approche réactive synchrone, dans lequel les instructions Java sont exécutées d'une façon atomique. Le compilateur de Rejo génère du code 100% Java où les instructions réactives sont implémentées avec la librairie Junior. L'influence de Junior sur Rejo est très importante, et les performances, les avantages et les inconvénients de Rejo découlent du modèle et du moteur d'exécution de Junior. Une étude approfondie de la programmation et des différentes implémentations de Junior est réalisée.

Les objets réactifs de Rejo peuvent être considérés comme des ***agents mobiles*** car ils ont la capacité de migrer en utilisant une plate-forme, appelée ROS, qui offre les fonctionnalités nécessaires. ROS, de l'anglais *Reactive Operating System*, est un Système d'Agents Mobiles qui ressemble à un système d'exploitation car il est constitué d'un micro-noyau modulaire et d'un ensemble des services autour desquels on trouve une interface graphique (les Ricobjs), un shell (Rsh) et une interface de programmation.

Mots clés : Réactif, Synchrone, Java, Junior.

Abstract:

This thesis presents Rejo a new high level language for programming reactive systems. Rejo, which means REactive Java Objets, is an extension of Java that creates ***Reactive Objets***, i.e., objects that encapsulate data and a mix of Java instructions and reactive instructions. The execution model is the one of the Reactive Synchronous Approach which executes the Java instructions in an atomic way. The Rejo compiler generates 100% Java code where reactive instructions are implemented with the Junior library. The influence of Junior on Rejo is important, and performance, advantages, and drawbacks are herited from the model and implementation of Junior. Junior programming and several implementations of it are studied in the thesis .

The Rejo reactive objects may be considered as ***mobile agents*** because they can migrate using a platform, called ROS, that provides the functionalities they need. ROS, which means Reactive Operating System, is a Mobile Agent System similar to an operating system because it is based on a modular micro-kernel and on a set of services, and it provides a graphical interface (Ricobjs), a shell (Rsh), and an API.

Keywords: Reactive, Synchronous, Java, Junior.