Université de Nice – Sophia Antipolis *UFR Sciences*

Ecole Doctorale
Sciences et Technologies de l'Information et de la Communication

Thèse de Doctorat

Préparée pour obtenir le titre de
Docteur en Sciences de l'Université de Nice-Sophia Antipolis

Spécialité : Informatique

Préparée conjointement à l'Ecole des Mines de Paris -
Centre de Mathématiques Appliquées de Sophia Antipolis
et à l'INRIA Sophia Antipolis - équipe MIMOSA

par

Christian Brunette

Construction et simulation graphiques de comportements : le modèle des *Icobjs*

Directeur de thèse : Frédéric Boussinot

Soutenue publiquement le mercredi 20 octobre 2004
à l'Ecole Supérieure en Science Informatique
de Sophia Antipolis

devant le jury composé de :

MM.	Charles André	Président	UNSA-I3S
	Marc Pouzet	Rapporteur	Université Paris 6
	Jean-Pierre Talpin	Rapporteur	INRIA Rennes
	Jean-Paul Rigault	Examinateur	ESSI Sophia-Antipolis
	Frédéric Boussinot	Directeur de thèse	Ecole des Mines de Paris

à mes parents
et à tous mes amis

Je tiens à remercier les personnes suivantes qui m'ont permis d'effectuer ce travail :

- tout d'abord mon directeur de thèse, Frédéric Boussinot, de m'avoir convaincu de faire une thèse. Il a été présent à chaque moment où j'en avais besoin et a fait preuve d'une très grande patience tout au long de ces 3 ans de thèse et surtout durant les mois de rédaction. Toutes les discussions que nous avons pu avoir dans le cadre du travail ou non ont été très enrichissantes ;
- Marc Pouzet et Jean-Pierre Talpin qui ont accepté la lourde tâche de lire cette thèse et de la rapporter ;
- Charles André d'avoir accepté de présider mon jury ;
- Jean-Paul Rigault d'avoir accepté de participer à mon jury ;
- toutes les personnes avec qui j'ai pu travailler durant ces 3 ans et principalement tous les membres des équipes MIMOSA, TICK et du CMA ;
- enfin, plus généralement toutes les personnes que j'ai rencontrées au cours de ces années, que ce soit à l'INRIA ou à l'extérieur.

Table des matières

Chapitre 1

Introduction

Depuis le paléolithique l'homme a représenté graphiquement des scènes de la vie. Ces représentations vont des peintures rupestres des Grottes de Chauvet, datant d'environ 30.000 ans avant notre ère, aux jeux vidéo actuels. L'avantage que l'on a actuellement, c'est que nous pouvons animer ces représentations et montrer ainsi le comportement de chacun des "dessins". Cette thèse ne mettra pas en avant des techniques de conservation des peintures rupestres mais elle présentera plutôt les moyens que nous avons mis en place pour créer des simulations interactives en axant le propos sur la définition des comportements des entités simulées.

Revenons à notre époque. Intuitivement, une simulation graphique peut être vue comme un espace borné dans lequel plusieurs entités évoluent en parallèle. Elles ont chacune un comportement et c'est l'interaction entre leur comportements qui définit le scénario (comportement global) de la simulation. Il paraît naturel d'utiliser des langages de programmation concurrente pour ce genre de simulation. On distingue deux types d'ordonnancement de fils d'exécution concurrents :

- le plus fréquent est l'ordonnancement *préemptif*. Dans ce cas, tous les fils d'exécution sont actifs en même temps et c'est l'ordonnanceur qui alloue à un fil d'exécution, pour une portion de temps, l'utilisation du processeur. Il est appelé *préemptif* car l'ordonnanceur n'attend pas que le fil d'exécution lui rende la main. Ce genre d'ordonnancement se trouve principalement dans les systèmes d'exploitation, comme par exemple la librairie des *PThreads* [55] sous *Linux*. Les langages possédant des mécanismes de concurrence utilisent le plus souvent ce type d'ordonnancement, c'est par exemple le cas de *Java*. On peut noter qu'il existe une extension des spécifications du langage *Java* et de sa machine virtuelle pour utiliser des threads temps réel [63]. On trouve une comparaison entre les différentes solutions proposées dans [41] ; Il faut noter que ces solutions sont toujours à base de threads préemptifs.
- l'autre type est l'ordonnancement *coopératif*. Dans ce cas, l'ordonnanceur donne la main à un fil d'exécution en particulier et c'est ce dernier qui rendra explicitement la main à l'ordonnanceur soit quand il aura terminé son exécution, soit quand il se suspendra. Ce type d'ordonnancement peut être trouvé dans un langage comme *Smalltalk* [31].

Cependant, même si les entités de la simulation donnent l'impression d'évoluer en parallèle et de pouvoir être programmées dans des langages concurrents, elles sont codées la plupart du temps sous forme de programmes séquentiels et cela pour plusieurs raisons que nous allons détailler maintenant.

La première raison est qu'il est difficile de programmer des simulations multi-threadées dans un environnement préemptif. S'il est facile de suivre l'exécution d'un seul thread, il est très compliqué de suivre celle de plusieurs threads, surtout si l'on ne connaît pas l'ordre dans lequel ils sont exécutés. Cette difficulté de compréhension qui augmente avec le nombre de threads rend difficile la recherche d'erreurs dans les programmes.

Une seconde raison est l'inefficacité de l'utilisation des threads. En effet, dans les mécanismes d'ordonnancement préemptif, le passage de l'exécution d'un thread à l'autre est coûteux en terme de mémoire et d'utilisation du processeur. À chaque fois que l'ordonnanceur donne la main à un nouveau thread, il effectue un changement de contexte, c'est-à-dire qu'il doit d'abord enregistrer l'état du thread qui s'exécute pour pouvoir reprendre plus tard son exécution à l'endroit où elle a été interrompue, et ensuite charger l'état du thread à exécuter. Si le nombre de threads est important, le temps passé dans les changements de contexte peut devenir très coûteux.

Une troisième raison vient de la difficulté de programmer les simulations dans un environnement préemptif où tous les objets sont partagés. En effet, puisque le thread qui s'exécute ne décide pas du moment où il rend la main, il est possible que celui-ci soit en train de modifier l'état d'une donnée au moment du changement de contexte. Dans le cas où plusieurs threads veulent accéder et/ou modifier la même donnée de manière concurrente, il est nécessaire de mettre en place des mécanismes de synchronisation pour protéger cette ressource. Pour cela, un verrou est posé sur une donnée par un thread et cette donnée ne peut être accédée ou modifiée par un autre thread avant que le premier ne le relâche. Cependant, de tels mécanismes amènent de nouveaux problèmes dont le *deadlock*. Ce problème est fondamentalement celui du conflit entre plusieurs threads qui veulent chacun accéder à une ressource verrouillée par un autre thread.

Les problèmes précédents concernent principalement les systèmes préemptifs. Dans cette thèse, nous nous proposons d'étudier les langages issus de l'approche réactive synchrone dont les caractéristiques les rapprochent plutôt des systèmes coopératifs. Plus précisément, nous proposons d'utiliser l'approche réactive synchrone dans le cadre des simulations graphiques. Nous sommes plus intéressés par l'aspect comportemental des entités du système que par les possibilités de rendu graphique qui peuvent être prises en compte par d'autres environnements dédiés à la modélisation graphique. Notre objectif est de créer un environnement permettant de décrire et modifier dynamiquement les comportements des entités simulés en profitant des spécificités des langages réactifs synchrones. Pour cela, nous sommes partis du modèle des *Icobjs* [12] défini par F. Boussinot. L'idée principale des *Icobjs* est d'associer à chaque entité graphique un comportement réactif qui peut être réutilisé dans le cadre d'un système de construction graphique.

La section 1.1 introduit le modèle d'exécution défini par l'approche synchrone avant de présenter une liste non exhaustive de langages basés sur cette approche. La section 1.2 présentera quelques exemples outils permettant la construction de simulations graphiques. Enfin, avant de présenter le plan de ce document dans la section 1.4, nous détaillerons, dans la section 1.3, les objectifs et les réalisations de cette thèse.

1.1 L'approche synchrone

D. Harel et A. Pnueli ont classifié les systèmes informatiques en deux grandes catégories [36] : les systèmes transformationnels et les systèmes réactifs. Ils représentent les premiers

comme des boîtes noires qui reçoivent des entrées qu'elles manipulent et qui, en terminant, retournent des sorties. Les systèmes réactifs sont représentés, selon l'expression de D. Harel, comme des *cactus noirs*, c'est-à-dire des systèmes qui interagissent, en continu et rapidement, avec leur environnement. Ce sont ces systèmes réactifs qui ont donné naissance au modèle synchrone.

Une des caractéristiques du modèle synchrone est que l'on passe de systèmes en temps continu à des systèmes en temps discret. L'utilisation d'une horloge globale permet d'échantillonner les modifications apportées à l'environnement. Le passage des systèmes en temps continu à des systèmes en temps discret est assuré par un mécanisme d'activation/réaction. À chaque fois que le système est activé, il réagit le plus rapidement possible à toutes les modifications de son environnement (les entrées) en produisant ses propres modifications de l'environnement (les sorties). De tels systèmes utilisent souvent des mécanismes de parallélisme ou de concurrence pour permettre à plusieurs composants parallèles du système de réagir en même temps aux modifications de l'environnement. On peut distinguer les systèmes réactifs en fonction de l'entité qui contrôle les réactions :

- soit le rythme est donné par l'environnement. Dans ce cas, le système réagit immédiatement à chaque modification de l'environnement.
- soit le rythme est donné par le système lui-même qui réagira donc immédiatement à toute modification de l'environnement effectuée depuis la dernière réaction. Il est important que le rythme imposé par le système soit alors assez soutenu pour répondre rapidement aux modifications de l'environnement. Nous allons nous intéresser plus particulièrement à ces systèmes.

L'approche synchrone repose sur une hypothèse fondamentale selon laquelle la *durée logique des réactions est nulle*. Plus précisément, on suppose que le système réagit suffisamment vite par rapport aux modifications de l'environnement pour ne pas perdre d'événements. Cela entraîne que les sorties et les entrées sont logiquement émises simultanément. La conséquence est que l'environnement est parfaitement déterminé et stable lors d'une réaction, puisque de façon instantanée les entrées et les sorties sont connues et constituent l'échantillon de l'environnement.

Du modèle synchrone découlent différents langages de programmation qui permettent d'exprimer les réactions du système à une suite d'activations. Une contrainte de ces langages est qu'ils doivent assurer l'absence d'incohérence dans ce qui est produit par les composants parallèles. En effet, au cours d'une réaction, tous les composants parallèles doivent avoir la même vision de l'environnement. En général, la puissance expressive des langages synchrones permet d'écrire des systèmes incohérents et on utilise une analyse statique des programmes pour rejeter ceux qui ne sont pas corrects.

Une autre caractéristique importante est que *les formalismes synchrones sont déterministes*. Cela signifie que les sorties doivent être définies de manière unique en fonction des entrées à chaque réaction. L'analyse statique des programmes doit donc aussi rejeter les programmes qui peuvent retourner pour une même suite d'entrées plusieurs sorties possibles.

Parmi les langages issus de l'approche synchrone, on distingue deux grandes familles de formalismes : ceux orientés flots de données et ceux orientés contrôle d'exécution. Des formalismes graphiques ont vu le jour à partir principalement des formalismes orientés contrôle d'exécution. Enfin, à partir de l'approche synchrone est aussi apparue une approche dite *réactive synchrone*. Nous allons maintenant présenter ces formalismes en les illustrant par une liste non exhaustive de langages les utilisant.

1.1.1 Formalismes orientés flots de données

Dans les formalismes orientés flots de données, l'environnement d'exécution est constitué de flots de données, c'est-à-dire de suites de valeurs définies en fonction du temps. Le programme réactif est représenté par un ensemble d'équations qui calculent les flots de sortie par rapport aux flots d'entrée. Parmi les langages construits sur ces formalismes, on trouve :

- LUSTRE [23] est un langage déclaratif qui permet de décrire des programmes réactifs sous la forme d'un ensemble d'équations (différentielles, booléennes...) qui doivent être satisfaites par les variables du programmes à chaque réaction. Ces variables sont des fonctions dépendant du temps, c'est-à-dire que chaque variable est associée à une horloge qui définit la séquence des réactions auxquelles la variable contient une valeur. Toute les fonctions exprimées dans le langage doivent satisfaire les propriétés de causalité (une sortie ne doit dépendre que des entrées reçues au cours des réactions précédentes et de la réaction courante) et d'exécution en mémoire bornée (les flots de données ne doivent pas s'accumuler en mémoire).
  LUSTRE dispose d'un compilateur qui produit du code séquentiel sous forme d'un automate fini étendu, d'un outil pour effectuer la distribution de code séquentiel, d'un outil de vérification appelé *LESAR* et d'un outil de génération de tests appelé *Lurette*. Il est possible d'appeler des fonctions externes codées en C. Au niveau de la vérification, LUSTRE permet d'exprimer des propriétés de sûreté et de décrire le programme et sa spécification dans le même langage. LUSTRE est le langage à la base de l'environnement industriel SCADE qui est utilisé dans le monde de l'avionique.
- SIGNAL [33] est un langage déclaratif pour la programmation temps réel utilisant des systèmes d'équations portant sur des signaux. Ces signaux sont des suites de valeurs associées à des horloges qui peuvent dépendre de données locales ou d'événements externes. L'horloge d'un programme SIGNAL est alors la borne supérieure de toutes les horloges des différents signaux du programme (les réactions du programme sont les réactions de l'un au moins de ces signaux).
  Le langage SIGNAL est construit sur un petit nombre de primitives dont la sémantique est donnée en terme de processus. Les autres opérateurs de SIGNAL sont définis à l'aide de ces primitives (relations étendues aux suites, retard, extraction sur condition booléenne, mélange avec priorité...), et le langage complet fournit les constructions adéquates pour une programmation modulaire.
  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, et de produire un code exécutable en C ou en FORTRAN si le programme est correct. 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, ainsi qu'une synthèse de l'ordonnancement global des calculs spécifiés.
- *Lucid Synchrone* [24] est un langage issu de *Lucid* [85] et qui combine les fonctionnalités de LUSTRE et des langages *à la* ML. *Lucid Synchrone* est un langage fonctionnel d'ordre supérieur avec un typage fort construit au-dessus d'OBJECTIVE CAML. Il utilise comme valeurs primitives des séquences infinies (*streams*) qui représentent les signaux d'entrées et de sorties des systèmes réactifs, et sont combinées par l'intermédiaire d'un ensemble d'opérateurs *à la* LUSTRE. *Lucid Synchrone* fournit aussi plusieurs types d'analyses statiques (inférence de type, calcul d'horloge, causalité,...). *Lucid Synchrone* utilise des horloges qui spécifient les vitesses d'exécution de chacun des composants du

programme. Le programme doit cependant vérifier certaines règles sur les horloges pour assurer la réactivité. Enfin, *Lucid Synchrone* dispose d'un système de modules pour utiliser les valeurs d'Objective Caml ou d'autres modules synchrones.

1.1.2 Formalismes orientés contrôle d'exécution

L'objectif de ce type de formalisme est le contrôle d'exécution, c'est-à-dire de faciliter, par l'intermédiaire de primitives, l'ordonnancement et le contrôle des tâches à exécuter par le programme. L'exécution d'un programme est contrôlée par l'intermédiaire de signaux dont la caractéristique principale est la présence ou l'absence dans l'environnement d'exécution durant une réaction. Le principal langage basé sur ces principes est le langage Esterel [9]. Ce langage est l'un des premiers langages synchrones qui permet de construire des systèmes réactifs de contrôle (systèmes embarqués, protocoles de communication,...). C'est un langage impératif composé d'un ensemble de primitives de contrôle (séquence, parallélisme, génération et tests de présence d'événements, préemption forte, boucle...) qui ont été chacune décrite rigoureusement par une sémantique formelle [10]. Esterel dispose d'un large éventail d'outils : un compilateur, un simulateur graphique, un système de vérification (explicite ou par *BDD* [20] [83]) et des outils d'optimisations. Le compilateur a la particularité de pouvoir générer des programmes exécutables en C ou des circuits (schéma de connexion de portes logiques).

L'analyse statique doit assurer qu'à chaque instant les signaux de sortie sont définis de façon unique (solution déterministe) à partir des signaux d'entrée, c'est-à-dire que toutes les instructions voient les signaux de la même manière (présent ou absent). Les problèmes rencontrés concernent principalement la causalité [7], les boucles instantanées [71] et la schizophrénie [72].

Les formalismes synchrones s'intéressent principalement à des systèmes statiques, totalement déterminés, sur lesquels il est possible d'appliquer des raisonnements formels en vue de prouver certaines propriétés. À partir de l'approche synchrone orientée contrôle d'exécution, une nouvelle approche appelée *approche réactive synchrone* a été développée par F. Boussinot [36]. La principale différence de cette approche par rapport à celle d'Esterel est l'introduction de la réaction retardée à l'absence d'un signal. La notion d'instant correspond à des *réactions de durée non nulle*. L'intérêt principal de cette approche est que les problèmes de causalité disparaissent, ce qui permet une plus grande modularité et un plus grand dynamisme. Nous présenterons cette approche plus en détails dans le chapitre 2. Les langages développés autour de cette approche sont :

- *Reactive-C* [11] est une extension du langage C. *Reactive-C* introduit les procédures réactives, un mécanisme de gestion d'exceptions et des primitives comme *par* pour exécuter en parallèle plusieurs procédures réactives, *stop* pour terminer la réaction courante, *loop, repeat, every* pour les boucles, *watching* pour effectuer une préemption sur test booléen, etc. Au niveau implémentation, *Reactive-C* dispose d'un pré-processeur générant du C pur. Il est également possible de générer des automates.
- *SL* [17] est une restriction d'Esterel implémentée avec *Reactive-C*. La restriction est qu'il n'est pas possible de faire d'hypothèse sur la présence ou sur l'absence d'un signal. Un événement n'est présent qu'à partir du moment où il a été généré et n'est absent que si, à la fin de l'instant, il n'a pas encore été généré. Cette restriction évite les problèmes de causalité, mais elle empêche d'utiliser la préemption forte présente dans Esterel, c'est-à-dire on ne peut préempter un programme à l'instant où l'événement

de préemption est généré. Même si *Reactive-C* permet d'ajouter dynamiquement des programmes, *SL* l'empêche comme en ESTEREL.

- *SugarCubes* [18][69] est une implémentation du modèle réactif au-dessus de *Java*. *SugarCubes* n'est pas un langage à part entière car un programme est une suite d'instructions réactives qui sont des objets *Java*. C'est pourquoi on caractérise les *SugarCubes* comme une API de programmation. La notion particulière introduite dans les *SugarCubes* est la notion de *Cube* [70]. Un Cube est un modèle d'objet associant un objet *Java* classique à un programme réactif.
- *Junior* [40] est issu de la formalisation par des règles de réécritures des instructions réactives présentes dans *SugarCubes*. Il faut noter que le formalisme des *SugarCubes* (depuis la version 3) dispose aussi d'une sémantique formelle conçue par J-F. Susini. Il existe actuellement des implémentations de *Junior* réalisées en *Scheme*, *Senior* [26], et en C pour les systèmes embarqués, *Jrc* [60]. Nous décrirons *Junior* de façon plus détaillée dans le chapitre 2.
- *REJO* [1] [2] est une extension de *Java* pour programmer des objets réactifs. *REJO* dispose d'un compilateur qui génère du code *Java* et qui utilise *Junior* pour les instructions réactives. 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*. Cette plate-forme, issue d'une première version appelée RAMA [56], est un *Système d'Agents Mobiles* (SAM) constitué d'un micro-noyau modulaire et d'un ensemble de services autour desquels on trouve une interface graphique, un shell et une interface de programmation.
- *FairThreads* [14] est une API définissant une programmation concurrente à base de threads reposant sur le modèle réactif. Sa principale caractéristique est l'utilisation de *fair threads* qui peuvent basculer du mode préemptif au mode coopératif et inversement. En fait, ces threads peuvent dynamiquement s'enregistrer ou se désenregistrer d'un ordonnanceur coopératif que l'on peut considérer comme un serveur de synchronisation. S'il s'enregistre à l'ordonnanceur, alors le *fair thread* sera exécuté de manière coopérative et déterministe en utilisant des événements réactifs pour communiquer. S'il se désenregistre de l'ordonnanceur coopératif, alors il sera exécuté par le système d'exploitation de façon préemptive. Il existe trois implémentations des *FairThreads* : *Java*, *Scheme* et C.
- *LOFT* [76] signifie *Language Over Fair Thread*. C'est un langage de programmation concurrente basé sur l'utilisation de threads en C. L'objectif de *LOFT* est de simplifier la programmation des *fair threads*, d'avoir des implémentations efficaces pour des programmes composés d'un grand nombre d'entités concurrentes, de disposer d'une implémentation légère pour des systèmes embarqués où les ressources sont limitées, et de pouvoir bénéficier de l'exécution sur des machines à plusieurs processeurs utilisant le SMP (*Symetric Muti-Processing*). Les programmes écrits avec *LOFT* peuvent être soit traduits en code C utilisant les *FairThreads*, soit traduits en automates pour être utilisés sur des systèmes embarqués à faible ressource.

1.1.3 Formalismes graphiques

Un certain nombre de formalismes graphiques (principalement orientés contrôle) ont également été proposés pour les systèmes synchrones :

- Les *StateCharts* [37], créés par D. Harel, permettent la spécification graphique de

systèmes réactifs à base de boîtes qui décrivent les états et de flèches qui décrivent les transitions. De nombreuses sémantiques ont été proposées pour les *StateCharts*. L'article [35] présente la première sémantique exécutable. Le but de ce formalisme était d'étendre le modèle de Machine à États Finis (*Finite State Machine ou FSM*) pour décrire des comportements complexes. Les *StateCharts* étendent le modèle des FSM de trois manières. Ils ajoutent :
  – l'orthogonalité. Le modèle des FSM ne dispose d'aucune construction pour représenter la concurrence.
  – la hiérarchie. Une des limitations du modèle de FSM est que la complexité des diagrammes augmente de façon dramatique avec le nombre d'états. Les *StateCharts* permettent d'imbriquer des états (boîtes) les uns dans les autres, ce qui rend le diagramme plus compréhensible.
  – la diffusion. Ce mécanisme permet à plusieurs états orthogonaux de communiquer par des événements.
- *Argos*[34] [49], développé par F. Maraninchi reprend les principes des *StateCharts* en proposant une sémantique strictement définie. *Argos* offre à la fois une syntaxe graphique et textuelle. Les principales différences avec les *StateCharts* sont l'utilisation d'un véritable opérateur de composition hiérarchique qui supprime toutes les transitions entre les différents niveaux hiérarchiques et l'application d'un synchronisme strict. *Argos* dispose d'un environnement appelé *Argonaute* qui fournit un compilateur et des outils de vérification. Le compilateur peut produire soit des fichiers au format *oc* pour la génération de code et la simulation, soit un automate pour la vérification en utilisant *Aldebaran* [29], soit un automate temporisé pour la vérification avec *Kronos* [47]. L'extension aux formalismes synchrones orientés flots de données et en particulier à LUSTRE est également couverte par ce formalisme.
- Les *SyncCharts*[4][5], développés par C. André, s'inspirent des *StateCharts* et d'*Argos* pour spécifier et programmer des systèmes réactifs. Un *syncCharts* qui est une instance du modèle peut être traduit en un programme ESTEREL (v5), ce qui permet de profiter de l'environnement d'ESTEREL. La principale caractéristique des *SyncCharts* par rapport aux *StateCharts* et à *Argos* est l'introduction de la préemption dans le formalisme graphique. Il y a trois types de flèches qui peuvent sortir d'un état : une pour représenter une terminaison normale, une autre pour représenter le changement d'état sur préemption forte et une troisième pour un changement d'état sur préemption faible. Dans les *SyncCharts*, on manipule les notions de *Star* qui représente un état (boîte) et tous les moyens pour sortir de cet état (flèches sortantes), de *Constellation* qui représente une interconnexion de *Star* et de *MacroState* qui représente une composition parallèle de plusieurs *Constellation*. De plus, à chaque *Star*, *Constellation* ou *MacroState* sont associés trois ensembles de signaux : un pour les signaux d'entrée, un pour les signaux de sortie et un pour les signaux locaux. Toute communication utilisant ces signaux est faite par diffusion instantanée.

1.2 Outils de construction et simulation graphique

Dans cette section, nous allons présenter une liste non-exhaustive d'outils permettant la construction de simulations graphiques et d'animations. Nous détaillerons ces outils en présentant leurs caractéristiques concernant la manière d'exécuter le comportement des entités

et les différentes approches utilisées.

Commençons tout d'abord par introduire *Squeak* [42]. *Squeak* est une implémentation portable de *SmallTalk-80* dont la machine virtuelle est entièrement écrite en *SmallTalk* la rendant ainsi facile à déboguer et à modifier. *Squeak* dispose de fonctionnalités utiles dans le cadre des simulations graphiques comme la gestion du son et la gestion d'image (rotation, zoom, anti-aliasing...). De plus, *Squeak* dispose d'un objet générique appelé *Morphs* qui comporte une représentation visuelle, et qui doit :

- réagir aux manipulations de l'utilisateur. Un morph peut être sélectionné par l'utilisateur afin de lui appliquer des modifications, comme par exemple, le déplacer.
- pouvoir être composé avec d'autres morphs. Chaque morph peut devenir le conteneur d'un ou plusieurs autres morphs (sub-morphs) et être lui-même contenu dans un morph. Un morph contenant d'autres morphs est manipulé comme une seule entité dans le cas des déplacements, des copies et des retraits.
- exécuter des actions à intervalle régulier. Chaque morph dispose d'une méthode qui par défaut est appelé chaque seconde (la valeur du minuteur peut être changée et un morph peut disposer de plusieurs minuteurs). Ces actions appelées régulièrement permettent de construire des animations (un morph qui change d'apparence périodiquement) sans utiliser de threads. Il faut noter qu'à l'inverse de *SmallTalk* où les threads sont coopératifs, *Squeak* propose une implémentation préemptive des threads.
- contrôler le positionnement et la taille de tous les morphs qu'il contient.

Squeak dispose aussi d'un moteur 3D appelé *Squeak-Alice* construit autour des idées de l'outil *Alice* [74]. L'objectif d'*Alice* est de permettre à des utilisateurs sans expérience en programmation de construire des mondes virtuels 3D interactifs.

L'approche de *Squeak* est très intéressante car il est possible à chaque moment d'introspecter tous les objets graphiques et de reprogrammer dynamiquement leur comportement. De plus, la notion de *morphs* se rapproche beaucoup de la notion d'icobj qui lie une représentation graphique à un comportement.

De nombreux outils pour construire des simulations ont été créés pour représenter les phénomènes sociaux complexes [6] en décrivant le comportement de chaque entité et leurs interactions avec les autres. Dans ce domaine, on trouve par exemple *StarLogo* [73], *Swarm* [25] [67], *RePast* [78] ou *Ascape* [58]. Plus récemment, on trouve des outils comme :

- Breve [45] est un environnement qui permet de créer des simulations 3D de vie artificielle. Les utilisateurs définissent le comportement de chaque agent dans un environnement 3D en temps continu. Breve contient un moteur physique et des mécanismes de détection de collisions pour permettre de simuler des comportements réalistes. L'implémentation de Breve est faite en C et la partie graphique utilise *OpenGL* [32]. Breve propose un langage interprété de type impératif appelé *Steve* pour programmer les comportements des agents. Pour programmer le comportement des agents, il faut définir des méthodes spécifiques qui sont appelées en fonction des événements générés. L'agent contient une méthode qui est appelée à chaque itération du moteur et dans laquelle l'utilisateur doit définir les comportements cycliques de l'agent. *Steve* donne également le moyen de spécifier une date à laquelle certaines méthodes de l'agent doivent être exécutées. Il est aussi possible de définir des événements sur lesquels l'agent doit réagir, comme la collision ou les événements provenant de l'utilisateur. Breve permet de rajouter dynamiquement de nouveaux objets à l'intérieur de la simulation. Cependant,

il ne permet pas de modifier dynamiquement les comportements comme *Squeak* par exemple. De plus, la notion d'événement et l'ordonnancement des tâches ne sont pas clairement définis.

- *Mason* [46] [77] est une librairie en *Java* servant de base de travail pour la construction de simulations multi-agents. *Mason* propose un modèle qui sépare le moteur d'exécution de comportements des outils de visualisation 2D et 3D. L'objectif de *Mason* est de servir de noyau pour construire des systèmes multi-agents spécifiques à certains domaines. Son moteur d'exécution est exécuté par un seul thread. Pour l'ordonnancement des tâches, *Mason* donne accès à certaines primitives qui permettent d'exécuter une liste de tâches soit en séquence (en utilisant toujours le même ordre ou en réordonnançant la liste de manière aléatoire à chaque itération) soit en parallèle où chaque tâche dans la liste est exécutée par un thread. Chaque tâche peut être stoppée explicitement. Ces primitives ne disposent pas d'une sémantique claire (formalisée). De plus, pour pouvoir utiliser la primitive de parallélisation, l'utilisateur doit être sûr de l'indépendance entre les tâches exécutées.

Concernant les mondes virtuels, beaucoup d'outils existent permettant de créer les mondes virtuels multi-utilisateurs centrés sur l'interaction avec les autres utilisateurs. Par contre, peu d'outils permettent de décrire facilement les comportements des entités autonomes. F. Harrouet [38] et N. Richard [65] ont fourni des outils pour décrire ces comportements :

- F. Harrouet a développé un langage appelé *oRis*. Il s'agit d'un langage interprété orienté objet fortement typé dont la grammaire se rapproche de *C++* et *Java*. Pour favoriser la conception la plus décentralisée possible, le langage ne permet pas l'accès à des variables globales ou statiques. L'activité globale de l'application n'est que le résultat des interactions des objets actifs qui la constituent. *oRis* permet trois modes d'ordonnancement (coopératif, préemptif et profondément parallèle) pour exécuter le comportement des agents. Il introduit aussi la notion de cycle de simulation permettant d'assurer le fait qu'aucun flot d'exécution ne peut prendre du retard ou de l'avance sur les autres. Ces flots peuvent être exécutés dans l'ordre ou de manière aléatoire pour éviter que l'ordre d'exécution n'influe sur le comportement des agents. Les entités autonomes communiquent par différents moyens (synchrones ou asynchrones). Un point important dans *oRis* est la possibilité de modifier dynamiquement le comportement des agents par le langage. Il est possible de déclencher de nouveaux traitements, d'introduire de nouvelles classes (ou fonctions) et de modifier les classes (fonctions) existantes à l'exécution. Ces modifications peuvent être locales à une instance d'une classe ou se généraliser à toutes les instances. En cas d'erreur, la branche d'où provient l'erreur est retirée et le système continue l'exécution des autres branches. La plate-forme *AReVi* [64] propose un environnement 3D pour exécuter les agents décrits avec *oRis*.
- N. Richard a proposé une alternative pour programmer les comportements d'agents virtuels à mi-chemin entre l'approche réactive et l'approche à base de threads en réalisant la plate-forme *InViWo* (Intuitive Virtual Worlds). Tout objet d'un monde *InViWo* est ainsi un agent capable de réagir aux modifications de son environnement et de modifier le monde dans lequel il évolue en fonction de son état interne. La structure d'un agent est composée d'attributs, de capteurs, d'un organe de décision et d'effecteurs. La structure d'un agent InViWo est entièrement dynamique. En effet, il est possible d'ajouter ou de retirer dynamiquement chacun de ces composants. L'organe de décision a pour rôle de choisir l'action à effectuer (effecteurs). Pour cela, il utilise un mécanisme

d'arbitrage permettant de combiner les décisions prises par chacun des modules comportementaux réactifs qui composent le comportement de l'agent. Pour programmer ces modules, *InViWo* dispose d'un langage synchrone, MARVIN, dont la syntaxe est proche de celle d'ESTEREL. En fait, chaque agent peut être considéré comme une machine réactive qui exécute son comportement dans un cadre synchrone. Par contre, chaque agent est exécuté par un thread *Java* distinct. Il n'y a donc pas de synchronisation forte entre les agents. Chacun dispose d'une horloge interne gérant son thread d'exécution et déclenchant les réactions successives selon un pas de temps de durée fixe. Le rendu du monde 3D est découplé du moteur. Ainsi, il est possible de changer d'affichage [66] en réutilisant les mêmes comportements.

L'approche réactive synchrone a également été utilisée dans FRAN (Functional Reactive Animation)[27]. À l'opposé des formalismes de l'approche réactive synchrone, FRAN exécute ses programmes en temps continu. FRAN est implémenté avec *Hugs*, une implémentation d'*Haskell* [43]. FRAN fournit un ensemble de types et de fonctions pour créer des animations. FRAN manipule deux notions importantes : les comportements et les événements. Un comportement est une valeur qui varie au cours du temps alors que les événements regroupent les interactions avec le monde extérieur (souris, clavier) et des conditions booléennes basées sur des paramètres de l'animation. FRAN fournit une sémantique formelle dénotationelle incluant la notion de temps réel. La manière de programmer avec FRAN est relativement similaire de celle avec SIGNAL.

Les outils que nous venons de détailler permettent de construire des simulations et des comportements avec des langages particuliers ou par des API de programmation. D'autres outils proposent de construire graphiquement les comportements. On peut par exemple citer les outils commerciaux comme *Simulink* [50] qui permet de concevoir, simuler, mettre en œuvre et tester avec précision des systèmes de contrôle, de traitement du signal ou de communication, ou comme *virtools* [84] qui fournit des outils pour des développer des jeux ou des simulations virtuelles. Il existe également des outils libres comme *Bean Builder* [51] pour construire des interfaces graphiques. Les principes de construction graphique de ces différents outils sont les mêmes. Ils fournissent des librairies de comportements élémentaires pouvant être étendues par chaque utilisateur. Les comportements sont programmés en général en *C++* ou en *Java*. Les comportements élémentaires sont ensuite chargés dans une interface de construction graphique. Chacun d'entre eux dispose d'entrées et de sorties spécifiques. La construction graphique consiste alors à relier les entrées et les sorties de différentes boîtes et à les paramétrer par l'intermédiaire de boîtes de dialogue. De plus, certains outils proposent des langages de scripts pour compléter le comportement. Chaque nouveau comportement créé ainsi peut alors être réutilisé dans une nouvelle construction.

1.3 Objectifs et réalisation

Notre objectif est de créer un outil dont les caractéristiques sont :

- de construire des simulations multi-agents où le scénario d'une simulation est la résultante de l'exécution des agents qu'elle contient. À l'inverse de *InViWo*, les comportements des agents seront exécutés dans le même environnement synchrone.
- de disposer, comme dans *Squeak*, d'un modèle d'objet graphique générique aisément extensible. Pour cela, nous développons la notion d'*icobj* qui associe un comportement

à un objet graphique.
- de disposer d'un mécanisme de construction graphique permettant à des non-spécialistes en programmation de pouvoir construire des comportements complexes donnant lieu à des animations de manière simple et intuitive. Pour cela, nous réutiliserons le mécanisme de construction des *Icobjs* proposant une technique original permettant de réutiliser les comportements des entités existantes pour les combiner en y rajoutant des primitives de contrôle.
- de disposer d'un moteur réactif efficace fondé sur une sémantique claire. Nous souhaitons réaliser un moteur proche de *Junior*, mais dédié à l'exécution des icobjs.
- de permettre d'inspecter et de modifier dynamiquement les paramètres d'exéxcution et les comportements de chaque agent. Ainsi, il serait possible de tester rapidement des scénarios de simulation et de les modifier dynamiquement.

Pour répondre à ces objectifs, nous avons réalisé une API facilement extensible dont chaque objet peut être inspecté et modifié dynamiquement à partir du modèle réactif synchrone. L'apport de cette thèse a été de :
- réaliser une API claire des *Icobjs* permettant à la fois d'utiliser de manière simple et intuitive le système de construction graphique qui est à l'origine de la création des *Icobjs* et de réaliser des simulations virtuelles où il est possible d'intervenir dynamiquement sur le comportement de chaque entité.
- réaliser un moteur réactif dédié à l'utilisation des *Icobjs*. Ce moteur, appelé *Reflex*, est une variante de *Junior* et dispose d'une sémantique claire. On parle de variante de *Junior* car nous avons profité de certaines caractéristiques des *Icobjs* pour retirer certaines contraintes de *Junior*, ajouter certaines instructions réactives et modifier la sémantique d'instructions existantes. Par exemple, pour rendre les constructeurs graphiques plus intuitifs, nous avons modifié certaines instructions réactives dont le comportement n'était pas *régulier*. La régularité des instructions sera discutée plus tard dans la section 2.3.
- réaliser plusieurs expérimentations dont des simulations distribuées, des simulations physiques et des simulations multi-horloges.
- réaliser un site Web [81] pour expliquer comment utiliser l'API des *Icobjs* que ce soit à travers un tutorial présentant les mécanismes de construction graphique et d'extension de l'API des *Icobjs*.

1.4 Structure du document

Ce document est découpé en deux grandes parties :
– La première partie concerne les travaux menés sur la réalisation d'un moteur réactif dédié aux *Icobjs*, *Reflex*, qui est une variante de *Junior*.
Le chapitre 2 présente d'abord les caractéristiques de l'approche réactive synchrone (dans la suite du document, nous l'appellerons approche réactive). Il décrit ensuite le modèle d'exécution de *Junior* en détaillant les différentes instructions réactives offertes par ce langage. Enfin, ce chapitre se termine en présentant les modifications apportées à *Junior* pour réaliser *Reflex*.

Le chapitre 3 décrit en détails la sémantique des instructions réactives de *Reflex*. Cette sémantique est présentée par l'intermédiaire de règles de réécritures selon le formalisme introduit par Plotkin [61].

Le chapitre 4 est dédié aux implémentations de *Junior* et de *Reflex*. Il commence par la présentation des implémentations *Rewrite*, *Replace*, *Simple*, *Storm* et *Glouton* avant de détailler les modifications apportées à la version *Storm* qui est à la base de *Reflex*.

– La seconde partie est dédiée à la description du modèle des *Icobjs*, à son implémentation et à l'implémentation de son environnement.
Le chapitre 5 présente le mécanisme de construction graphique offert par les *Icobjs* en détaillant l'utilisation de chacun des *constructeurs graphiques*. Ce chapitre décrit aussi comment manipuler l'environnement des *Icobjs* pour agir dynamiquement sur les simulations.

Le chapitre 6 est dédié à l'implémentation de l'API des *Icobjs* et de l'environnement construit autour de ce modèle. On y évoque, entre autres, la structure de données minimale nécessaire à l'utilisation des *Icobjs* et les problèmes liés à l'utilisation d'un modèle synchrone dans un environnement asynchrone.

Le chapitre 7 décrit trois expérimentations faites autour du modèle des *Icobjs* : l'utilisation des *Icobjs* dans le projet IST-PING [82] et les problèmes liés aux systèmes distribués ; la réalisation d'un moteur dédié à la simulation physique selon le modèle d'Alexander Samarin [68] ; enfin, la réalisation de simulations multi-horloges.

Première partie

Moteur réactif

Chapitre 2

Modèle réactif

Junior est le fruit des travaux menés autour des *SugarCubes* [13]. *Junior* reprend les principales instructions définies dans le formalisme des *SugarCubes* en donnant à celles-ci une sémantique formelle. La sémantique initiale de *Junior*, à base de règles de réécritures, est définie dans [39]. Pour permettre différentes implémentations basées sur ce même modèle opérationnel, une interface de programmation (API) générique a été créée. Cette API définit un certain nombre d'instructions pour créer des programmes réactifs.

Dans le cadre de l'utilisation d'un modèle réactif pour un environnement graphique comme celui des *Icobjs* (cf. section 5), nous proposons certaines simplifications ou ajouts au modèle existant pour tenir compte des caractéristiques et des besoins spécifiques de ce type d'environnement.

La section 2.1 présentera les différentes notions qui caractérisent l'approche réactive et le modèle d'exécution de *Junior*. À la suite de cela, nous détaillerons l'API de *Junior* (section 2.2) et nous donnerons quelques exemples pour illustrer son utilisation. Pour finir, nous décrirons les modifications apportées à l'API de *Junior* pour réaliser celle de *Reflex* (section 2.3). Ces modifications sont introduites dans la perspective de définir un moteur réactif dédié aux environnements graphiques. Elles se manifestent principalement par des ajouts et des retraits d'instructions et par des modifications apportées aux instructions existantes.

2.1 Approche réactive

La principale caractéristique d'un système réactif [36] est de réagir, **rapidement** et en **continu**, aux activations et aux modifications de l'environnement. À l'inverse des programmes transformationnels que l'on exécute en utilisant des paramètres, et qui retournent un résultat au moment où ils se terminent, les systèmes réactifs ne se terminent généralement pas. Ils sont exécutés en continu et, en réponse à chaque activation, ils réagissent en modifiant leur environnement. L'approche réactive considérée dans ce document s'articule autour de quatre notions principales.

Approche Réactive = instant + concurrence + diffusion d'événements + dynamisme

2.1.1 Définitions

La notion d'instant

Un instant représente la réaction du système à une activation. L'instant peut être vu comme une unité de temps dans l'évolution discrète du système. À la différence des systèmes dits temps réel où l'on se base sur un temps physique, l'instant représente une unité de temps logique. Contrairement à ESTEREL et aux modèles synchrones où la durée d'un instant est considérée comme nulle, dans l'approche réactive, aucune hypothèse n'est faîte sur cette durée (cf. figure 2.1). La durée d'un instant est le temps entre l'activation du système et le retour du système à un état stable, c'est-à-dire lorsqu'il ne peut plus évoluer dans l'instant.

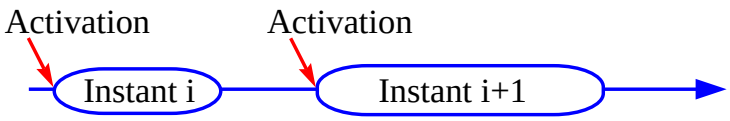


FIG. 2.1 – Notion d'instant

La durée d'un instant étant non nulle, le système évolue donc progressivement vers un état stable à chaque instant. Il est possible de découper l'exécution au cours d'un instant en plusieurs micro-étapes. Chacune de ces micro-étapes reflète une partie de l'évolution du système pour arriver à l'état stable. Les instructions que nous allons présenter plus bas ont une sémantique qui se base sur cette notion de micro-étape.

La concurrence

Tout comme le modèle synchrone (cf. ESTEREL), le modèle réactif définit un moyen pour exprimer l'exécution en parallèle de plusieurs composants. Un instant se termine au moment où chacun des composants du système réactif a atteint un état stable (cf. figure 2.2).

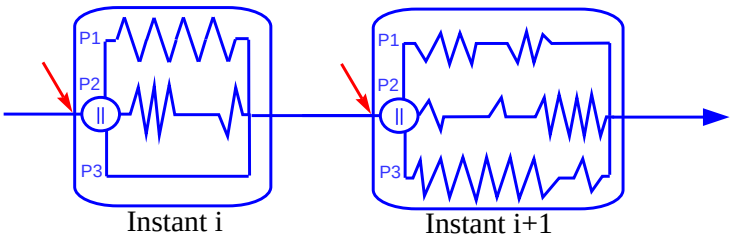


FIG. 2.2 – Concurrence

Dans la figure 2.2, P1, P2 et P3 représentent chacun un des composants qui sont exécutés en parallèle dans le système réactif. Chacun des composants évolue à sa manière aux modifications apportées à l'environnement d'exécution du système. La découpe en micro-étapes peut aussi être appliquée à l'exécution de chacun des composants exécutés en parallèle.

La diffusion d'événements

Dans le modèle réactif, les composants concurrents utilisent des événements pour communiquer entre eux. Ces événements permettent également à différents composants du système

en attente sur un même événement de se synchroniser. Les événements sont diffusés, c'est-à-dire qu'ils sont transmis à tous les composants. La notion d'instant permet de définir la notion de diffusion instantanée qui signifie qu'au cours d'un instant, un événement est vu de la même manière par tous les composants. Si un événement est vu présent par un composant du système, il ne peut pas être vu absent par un autre composant au cours du même instant, et inversement. Ce mécanisme est celui du langage ESTEREL [8]. Un événement diffusé au cours d'un instant donné n'est présent qu'au cours de celui-ci. À l'instant suivant, il sera considéré comme absent, excepté s'il est de nouveau diffusé. Il faut noter que la diffusion est instantanée. Cela signifie qu'au cours d'un instant donné, tous les composants verront l'événement généré de la même manière. Par contre, les composants n'auront pas nécessairement une vue cohérente sur le statut d'un événement au cours d'une micro-étape.

Pour maintenir l'état de cohérence, on ne peut faire aucune hypothèse quant à l'absence d'un événement pendant un instant et on ne peut être sûr qu'un événement est absent qu'à la fin de l'instant. Donc, on ne peut réagir instantanément à une absence d'événement et cette réaction doit être retardée à l'instant suivant. Cela constitue une des différences majeures entre l'approche réactive et l'approche synchrone. Cela évite en même temps les cycles de causalité [7] que l'on peut avoir avec ESTEREL. Principalement, les cycles de causalité sont dus à l'invalidation d'hypothèses faites au cours de l'exécution. Voici un exemple de cycle de causalité :

```
signal S in
    present S else emit S end
end
```

Ce programme teste la présence du signal local `S` et s'il est absent, alors il émet le signal dans le même instant. Si l'hypothèse est faite que `S` est absent pour l'instant courant alors la réaction est d'exécuter la branche `else` et d'émettre `S` pour ce même instant, ce qui est en contradiction avec l'hypothèse initiale. Inversement, si on fait l'hypothèse que `S` est présent alors il n'est pas émis, ce qui est également une contradiction. En reportant la réaction à l'absence à l'instant suivant, il est impossible de rencontrer de tels problèmes.

La réaction retardée à l'absence donne une plus grande modularité à la programmation. En effet, la mise en parallèle de plusieurs composants ne peut plus générer de problème de causalité.

Le dynamisme

Le dynamisme concerne la possibilité d'ajout et/ou de retrait de composants en cours d'exécution. Les ajouts et les retraits de composants ne sont pas des opérations instantanées. La notion d'instant permettant de garantir la stabilité du système à la fin de celui-ci, ces opérations sont uniquement effectuées entre deux instants. Comme nous l'avons fait remarquer dans le paragraphe précédent, aucun problème de causalité ne peut apparaître dans la combinaison de plusieurs composants en parallèle, donc dans l'ajout de composants.

Le retrait d'un composant peut être considéré comme une préemption de ce composant. À la différence d'ESTEREL, le modèle réactif ne permet pas la préemption forte, c'est-à-dire qu'il n'est pas possible d'interrompre définitivement l'exécution d'un comportement qui n'a pas atteint un état stable au moment où un événement est généré. Le modèle réactif permet uniquement la préemption faible et ne permet pas de stopper immédiatement l'exécution d'un

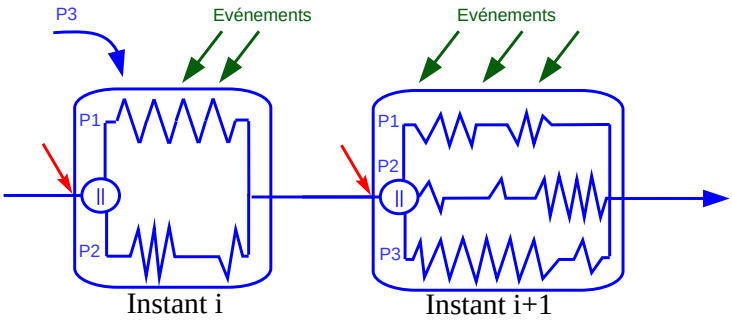


FIG. 2.3 – Dynamisme

programme au moment où un événement est généré. Le programme doit atteindre un état stable pour l'instant courant avant d'être préempté, c'est-à-dire qu'il doit avoir fini l'exécution de son comportement pour l'instant courant. Le modèle réactif propose deux mécanismes de retrait de composant :

- une simple préemption (primitive `Until`)
- une préemption dans laquelle on garde l'état du composant préempté pour pouvoir le réutiliser dans un autre environnement d'exécution (primitive `Freezable`).

Ces opérations sont effectuées par l'intermédiaire d'instructions décrites dans la section 2.2.

2.1.2 Modèle d'exécution

Nous venons de décrire un système réactif comme un ensemble de composants exécutés en parallèle dans un environnement commun et répondant aux activations. En fait, chacun de ces composants est décrit à l'aide d'instructions réactives que nous détaillerons dans l'API de *Junior* (section 2.2). Ces instructions réactives sont enregistrées et exécutées par une machine réactive.

Les instructions

Les instructions réactives en *Junior* sont des objets *Java* avec un certain nombre de méthodes déclarées. L'exécution d'une instruction réactive retourne un statut qui indique l'état de l'instruction. Les trois statuts retournés sont *TERM*, *STOP* et *SUSP*. Dans la version *Storm*, le statut *WAIT* a été rajouté. C'est cette version que l'on considère ici.

- *TERM* indique que l'instruction a complètement terminé son exécution pour l'instant courant et les suivants. Si l'on réactive une instruction qui a renvoyé le statut *TERM*, il ne se passera plus rien et elle renverra toujours le statut *TERM*.
- *STOP* indique que l'instruction a atteint un état stable et a terminé son exécution pour l'instant courant, mais qu'il y aura encore quelque chose à effectuer aux instants suivants. L'exécution reprendra à l'endroit où l'instruction s'est stoppée.
- *SUSP* indique que l'instruction n'est pas encore dans un état stable et doit donc être ré-exécutée au cours du même instant.
- *WAIT* a été rajouté dans la version *Storm* de *Junior* pour des questions d'efficacité. Ce statut est renvoyé par les instructions événementielles pour indiquer qu'elles sont en attente d'un événement et qu'il n'est pas nécessaire de les réactiver tant que l'événement

n'est pas présent ou tant que la fin d'instant n'a pas été déclarée. Cela évite de repasser à chaque micro-étape sur l'ensemble des instructions en attente d'un événement, et d'exécuter seulement celles dont l'événement déclencheur a été généré.

Le statut *WAIT* n'apparaît pas directement dans la spécification de *Junior*. Nous l'utiliserons tout de même dans la suite des explications, car les modifications que nous avons apportées sont basées sur la sémantique et l'implémentation de *Storm*.

La machine réactive

La machine réactive exécute les instructions réactives à chacune des ses activations. Plus précisément, elle contient une référence vers la racine de l'arbre de programmes et elle demande sa réécriture. C'est la machine réactive qui gère les différentes notions explicitées dans la section 2.1.1. Pour préciser le fonctionnement de la machine, nous décrirons le fonctionnement de la machine dans la version *Storm*.

Après une activation de la machine, seule celle-ci peut déclarer la fin de l'instant. Pour cela, elle exécute le programme qu'elle contient et, tant que cette exécution renvoie le statut *SUSP*, elle continue cette exécution. Nous pouvons remarquer ici la notion de micro-étapes. Une micro-étape correspond à une exécution du programme contenu par la machine. Si l'exécution renvoie le statut *WAIT*, cela signifie qu'il ne reste que des programmes bloqués en attente d'événements et qu'aucun programme ne peut progresser (plus aucun événement ne peut être généré). Dans ce cas, la fin d'instant est déclarée. La fin d'instant équivaut à une levée de drapeau dans l'environnement d'exécution à la suite de quoi la machine exécute une dernière fois son programme pour que toutes les instructions en attente d'un événement puissent déclarer son absence, se préparer à réagir à l'instant suivant et retourner le statut *STOP*. Pour finir un instant, il faut que l'exécution du programme renvoie *STOP* ou *TERM*.

La diffusion d'événement est assurée par l'intermédiaire d'un environnement dans lequel le programme de la machine réactive est exécuté. Cet environnement évolue grâce aux événements générés au cours de l'instant et est réinitialisé par la machine réactive à la fin de chaque instant.

Enfin, la concurrence est exprimée à l'aide d'une instruction réactive (`Par`). La notion de dynamisme est caractérisée par la possibilité d'ajouter dynamiquement des programmes à la machine réactive en cours d'exécution. Ces programmes sont mis en parallèle avec le programme déjà exécuté par la machine en utilisant l'instruction de concurrence. Cette mise en parallèle est effectuée pour l'instant suivant celui de l'ajout du programme à la machine et ceci pour garantir une fin d'instant dans tous les cas. En effet, on considère qu'un instant a une certaine durée et, que, pour déclarer la fin d'instant, il faut que tous les composants parallèles aient fini leur exécution pour l'instant concerné. Donc, si des programmes étaient rajoutés en permanence durant l'instant courant, il y aurait un risque qu'ils n'atteignent jamais un état stable et que la fin d'instant ne puisse jamais être déclarée.

2.2 API de *Junior*

Junior dispose d'une interface de programmation, appelée `Jr`, qui fournit un ensemble de méthodes pour accéder directement aux instructions réactives. Cela permet aussi de masquer les différences potentielles entre diverses implémentations de *Junior*. Dans cette partie, nous allons détailler les instructions de l'API standard de *Junior*. Ces instructions ne sont pas simplement des mots clés comme dans d'autres langages, mais sont représentées par des objets

Java qui implémentent les règles de réécritures. Des facilités syntaxiques ont été ajoutées à `Jr` et à son extension `Jre`, mais nous ne les détaillerons pas ici.

Nous distinguons plusieurs types d'instructions réactives :

- celles qui exécutent le code *Java* défini par l'utilisateur
- les instructions d'ordonnancement de base, c'est-à-dire la séquence, la composition parallèle, les boucles, les tests booléens.
- les instructions événementielles qui permettent de générer des événements ou de réagir en fonction de la présence ou de l'absence d'un ou plusieurs événements.

La sémantique des instructions que nous allons décrire est donnée dans [39].

2.2.1 Interfaçage avec Java

Les Wrappers

Les wrappers sont des pointeurs vers des données qui ne sont pas définies au moment de la création du programme. Cela permet aux instructions réactives d'utiliser des valeurs qui ne seront disponibles qu'après certains calculs. Les wrappers sont des objets *Java* que l'utilisateur doit implémenter et qui disposent d'une méthode appelée `evaluate(Environment env)`. L'appel à cette méthode va évaluer et retourner la donnée dont l'instruction réactive a besoin pour s'exécuter. Il y a 4 types de wrappers utilisés par les instructions de l'API standard :

- `IdentifierWrapper` qui renvoie un objet *Java* qui implémente l'interface `Identifier`. Cette interface `Identifier` est utilisée pour désigner les événements.
- `IntegerWrapper` qui renvoie un entier.
- `BooleanWrapper` qui renvoie un booléen.
- `ObjectWrapper` qui renvoie un objet *Java*.

Les instructions

- `Jr.Atom(Action a)`
  Cette instruction exécute un atome et termine immédiatement en renvoyant *TERM*. Les atomes (ou actions atomiques) permettent d'exécuter du code *Java* et donc d'interagir avec l'environnement *Java* de façon atomique. En effet, aucun autre code *Java* exécuté par la machine réactive ne peut interrompre l'exécution de l'atome. Il n'est donc pas nécessaire de protéger les objets manipulés par des verrous. Par contre, aucune garantie n'est donnée quant au partage, entre plusieurs threads *Java*, des objets manipulés par les actions atomiques. La propriété d'atomicité n'a donc de sens que dans le contexte de la machine réactive.
  Cette méthode prend en paramètre un objet qui implémente l'interface `Action` et qui contient une méthode `void execute(Environment env)`. C'est dans cette méthode que doit se situer le code *Java* à exécuter.

- `Jr.Link(ObjectWrapper objWrap, Program body)`
  Cette instruction associe un objet *Java* à l'exécution d'un programme réactif. L'objet *Java* associé est obtenu après évaluation du wrapper d'objet. En fait, cette instruction place l'objet *Java* dans l'environnement d'exécution pour que le programme réactif exécuté puisse interagir avec lui.

2.2.2 Les instructions de base

- `Jr.Nothing()`
  Cette instruction ne fait rien et termine immédiatement, ce qui correspond à retourner *TERM* à chaque activation.

- `Jr.Stop()`
  Cette instruction termine l'exécution d'une séquence d'instructions pour l'instant courant. Cela correspond à retourner *STOP* à la première activation et *TERM* à toutes les suivantes.

- `Jr.Seq(Program first, Program second)`
  Cette instruction binaire exécute séquentiellement le programme `first` puis le programme `second`. Le programme `second` ne pourra être exécuté qu'à partir du moment où l'exécution du premier renverra *TERM*. L'instruction `Seq` est terminée quand le programme `second` est terminé.

- `Jr.Par(Program left, Program right)`
  Cette instruction binaire exécute deux programmes `left` et `right` en parallèle. Cela ne signifie pas que les deux programmes vont être exécutés en même temps, mais simplement qu'ils sont activés à chaque instant d'exécution de l'instruction `Par` tant qu'ils ne sont pas terminés. L'instruction `Par` termine uniquement quand `left` et `right` sont terminés. Le tableau suivant détermine le statut retourné à chaque activation de l'instruction `Par` en fonction du statut des deux programmes `left` et `right`.

`left\right`	*TERM*	*STOP*	*WAIT*	*SUSP*
TERM	*TERM*	*STOP*	*WAIT*	*SUSP*
STOP	*STOP*	*STOP*	*WAIT*	*SUSP*
WAIT	*WAIT*	*WAIT*	*WAIT*	*SUSP*
SUSP	*SUSP*	*SUSP*	*SUSP*	*SUSP*

TAB. 2.1 – Statut retourné par l'instruction *Par*

  À la différence des *SugarCubes* où l'opérateur de concurrence, nommé `Merge`, est déterministe, dans *Junior*, l'ordre dans lequel les deux programmes sont appelés n'est pas déterminé. Cette différence permet une plus grande diversité dans l'implémentation. Cependant, les versions *Rewrite*, *Replace* et *Storm* implémentent l'instruction `Par` comme un `Merge` où l'on exécute d'abord `left` puis `right`.

- `Jr.Loop(Program body)`
  Cette instruction exécute en boucle le programme `body`. Dès que `body` se termine, il est réinitialisé puis immédiatement ré-exécuté. Cela signifie que les wrappers qui avaient été evalués le seront à nouveau à la prochaine exécution. Ceci est une optimisation du système car il est moins coûteux de réinitialiser les instructions entre deux exécutions plutôt que de recréer, à chaque itération, une nouvelle instance du programme.
  L'utilisation de cette instruction peut cependant poser le problème de boucle instantanée. En effet, si l'exécution de `body` est instantanée, l'instruction `Loop` ne peut converger vers un état stable en temps fini, puisque `body` sera immédiatement ré-exécuté.

Une boucle instantanée est un très gros problème pour un système réactif, puisqu'elle empêche le système de parvenir à un état stable, ce qui empêche l'instant de se finir. Certaines versions de *Junior* et des *SugarCubes* utilisent des heuristiques [70] pour détecter les boucles instantanées et les stopper.

- `Jr.Repeat(IntegerWrapper num, Program body)`
  Cette instruction exécute un nombre fini de fois le programme `body`. Ce nombre est défini à la première activation de `Repeat` en évaluant le wrapper d'entier. Comme pour `Loop`, à la fin de chaque exécution, `body` est réinitialisé. Par contre, le problème de boucle instantanée ne se pose pas ici, puisque le nombre d'itérations est fini, ce qui implique que le programme converge toujours vers un état stable en temps fini.

- `Jr.If(BooleanWrapper cond, Program thenInst, Program elseInst)`
  Cette instruction exécute le programme `thenInst` ou le programme `elseInst` en fonction du résultat retourné par l'évaluation du wrapper. Si l'évaluation du wrapper retourne *true* alors `thenInst` est exécuté. Dans le cas contraire, c'est le programme `elseInst` qui est exécuté. Cette évaluation n'a lieu qu'à la première activation de l'instruction `If`. En effet, l'exécution du programme `thenInst` ou `elseInst` peut durer plusieurs instants et, au cours des instants suivants, le wrapper n'est pas réévalué. Le résultat obtenu lors de la première activation est réutilisé.

2.2.3 Événements et configurations événementielles

Les événements

Comme nous l'avons vu, l'approche réactive dispose d'un moyen de communication puissant : la diffusion d'événement. Un événement en *Junior* est un objet *Java* qui implémente l'interface `Identifier` pour laquelle il faut redéfinir deux méthodes présentes dans la classe `Object` :

- `boolean equals(Object obj)` Cette méthode permet de vérifier que deux objets *Java* sont bien des identificateurs du même événement.
- `int hashCode()` Les événements étant stockés dans une table de hachage, il faut que deux identificateurs représentant le même événement retournent la même clé de hachage.

Les configurations événementielles

Un événement peut être présent ou absent pour un instant donné. Le statut de présence d'un événement est donc défini par un booléen. Une configuration événementielle est une expression booléenne qui permet de tester la présence d'un ou plusieurs événements. En *Junior*, une configuration événementielle implémente l'interface `Configuration` et s'exprime à partir des 4 constructions suivantes :

- `Jr.Presence(IdentifierWrapper idWrap)`
  Cette configuration teste la présence d'un événement dont l'identificateur est obtenu à partir de l'évaluation du wrapper d'événement. L'évaluation du wrapper est effectuée au moment du premier test de présence de l'événement.

- `Jr.And(Configuration c1, Configuration c2)`
  Cette configuration exprime la conjonction de deux configurations événementielles.

- `Jr.Or(Configuration c1, Configuration c2)`
  Cette configuration exprime la disjonction de deux configurations événementielles.

- `Jr.Not(Configuration c)`
  Cette configuration exprime la négation d'une configuration événementielle. L'introduction de cette configuration donne la possibilité aux instructions événementielles de réagir à l'absence d'un ou plusieurs événements.

Les instructions événementielles

Les instructions événementielles n'utilisent pas toutes des configurations, mais parfois un événement seul. Nous allons maintenant détailler les différentes instructions événementielles. Tous les `IdentifierWrapper` composant une configuration événementielle sont évalués une seule fois avant le premier test de satisfaction de la configuration.

- `Jr.Generate(IdentifierWrapper idWrap)`
  `Jr.Generate(IdentifierWrapper idWrap, ObjectWrapper objWrap)`
  Cette instruction génère un événement qui est alors considéré comme présent dans l'environnement d'exécution pour l'instant courant. L'identificateur d'événement est obtenu après évaluation du wrapper d'événement. La seconde version de l'instruction permet de générer un événement et de lui associer une valeur. Cette valeur est obtenue après évaluation du wrapper d'objet *Java*. Cette instruction termine immédiatement après avoir modifié l'environnement d'exécution.

- `Jr.Await(Configuration c)`
  Cette instruction bloque l'exécution d'une séquence tant que la configuration événementielle n'a pas été satisfaite. Dès que celle-ci est satisfaite, l'instruction termine. Il se peut que cette configuration ne soit satisfaite qu'à la fin de l'instant, si celle-ci est en attente d'une absence d'événement. Dans ce cas, l'instruction termine uniquement au prochain instant.

- `Jr.When(Configuration c, Program thenInst, Program elseInst)`
  Cette instruction correspond à l'instruction `If`, excepté qu'au lieu d'évaluer un `Boolean Wrapper`, une configuration événementielle est évaluée. Si cette configuration événementielle est satisfaite, alors le programme `thenInst` est exécuté, sinon c'est le programme `elseInst`. L'exécution de `thenInst` ou `elseInst` peut être reportée à l'instant suivant en fonction de la configuration événementielle. En effet, la décision de satisfaction ou de non-satisfaction de la configuration événementielle peut être reportée à la fin de l'instant. L'évaluation de la configuration événementielle n'est faite qu'au premier instant où l'instruction est exécutée.

- `Jr.Until(Configuration c, Program body, Program handler)`
  Cette instruction préempte le programme `body` dès que la configuration événementielle est satisfaite. Tant que la configuration n'est pas satisfaite, `body` continue son exécution. À l'instant où la configuration est satisfaite, on attend que l'exécution de `body` soit finie pour l'instant courant avant de le préempter. Si la préemption a lieu avant que la fin d'instant est déclarée, le programme `handler` est immédiatement exécuté, sinon l'exécution de `handler` est reportée à l'instant suivant. La configuration événementielle est testée à chaque instant d'exécution de `body`.

- `Jr.Freezable(IdentifierWrapper idWrap, Program body)`
  Cette instruction effectue également une préemption sur le programme `body`. Cependant, contrairement à `Until`, la préemption se fait sur la présence d'un événement obtenu après évaluation du wrapper à la première exécution de l'instruction. Comme `Until`, c'est une préemption faible. À l'instant où l'événement est généré, `Freezable` attend que `body` termine pour l'instant courant avant de le préempter et il stocke ensuite le résidu de `body` dans l'environnement d'exécution en le mettant en parallèle avec les autres programmes gelés sur le même événement. L'instruction termine au moment où le programme `body` a été préempté.

- `Jr.Control(IdentifierWrapper idWrap, Program body)`
  Cette instruction conditionne l'exécution de `body`, c'est-à-dire que `body` ne sera exécuté qu'aux instants où un événement sera présent. Cet événement est obtenu après évaluation du wrapper, à la première exécution de `Control`.

- `Jr.Local(Identifier id, Program body)`
  Cette instruction définit un événement dont la portée est le programme `body`. Cela signifie que si l'événement est généré en dehors de `body`, alors il ne sera pas vu comme présent dans l'exécution de `body` et réciproquement.

2.2.4 Machine et environnement d'exécution

La machine réactive est l'entité qui exécute les instructions réactives. L'environnement d'exécution et la machine réactive sont, en *Junior*, deux entités distinctes mais totalement dépendantes l'une de l'autre. L'environnement d'exécution gère l'ensemble des événements qui ont été générés au cours de l'instant et des programmes qui ont été gelés à l'instant précédent. De son côté, la machine réactive gère l'exécution des instructions, les ajouts de programmes et déclare les fins d'instants et réinitialise l'environnement.

La machine réactive

Junior propose deux types de machines réactives : la *safe-machine* et l'*unsafe-machine*. La différence entre ces deux types de machines est que la *safe-machine* est conçue pour fonctionner dans un environnement multi-threadé. Les événements générés dans une *unsafe-machine* sont perdus à partir du moment où la fin d'instant est déclarée. Dans une *safe-machine*, on a la garantie de traiter tous les événements générés même si plusieurs threads *Java* accèdent et/ou modifient la machine en même temps. Ces deux types de machine sont accessibles à travers l'API par les deux méthodes suivantes :

- `Jr.Machine(Program p)`
  Cette méthode crée une *unsafe-machine* exécutant le programme `p`.

- `Jr.SafeMachine(Program p)`
  Cette méthode crée une *safe-machine* exécutant le programme `p`.

Pour interagir avec la machine, l'utilisateur dispose des méthodes suivantes :

- `void add(Program p)`
  Cette méthode ajoute un programme en parallèle à ceux qui sont déjà chargés dans la machine. Cet ajout est pris en compte à l'instant suivant. La *safe-machine* n'ajoute pas directement le programme, mais une copie de celui-ci.

- `boolean react()`
  Cette méthode fait progresser d'un instant le programme chargé dans la machine. La méthode retourne un booléen qui indique s'il reste encore quelque chose à faire à l'instant suivant. La *safe-machine* bloque les appels à cette méthode provenant d'autres threads quand un instant est en train d'être exécuté. Elle empêche aussi les appels ré-entrants, c'est-à-dire que si une action atomique appelle la méthode `react` sur la machine qui l'exécute, cette dernière renverra immédiatement *false* sans rien exécuter.

- `void generate(Identifier id)`
  `void generate(Identifier id, Object obj)`
  Ces deux méthodes génèrent, respectivement, un événement sans ou avec une valeur dans la machine sans passer par l'instruction `Generate`. Une génération dans la *safe-machine* se fait à l'instant courant si la fin d'instant n'a pas été déclarée. Si elle a été déclarée, la génération est prise en compte à l'instant suivant. À l'inverse, pour l'*unsafe-machine*, l'événement est perdu si la génération est faite après que la fin d'instant soit déclarée.

- `Program getFrozen(Identifier id)`
  Cette méthode retourne les programmes qui ont été gelés à l'instant précédent par l'instruction `Freezable` sur génération de l'événement `id`. Pour un événement donné, on ne peut récupérer qu'une seule fois ces programmes.

L'environnement d'exécution

De même qu'il y a des *safe-machine* et des *unsafe-machine*, l'API dispose aussi de *safe-environnement* et de *unsafe-environnement*. Dans l'exécution des actions atomiques et dans l'évaluation des wrappers, l'environnement d'exécution est passé en référence, ce qui permet d'accéder à certaines informations concernant l'environnement par l'intermédiaire des méthodes suivantes :

- `Object[] currentValues(Identifier id)`
  Cette méthode retourne un tableau regroupant les valeurs associées aux générations de l'événement `id` pour l'instant courant. Les valeurs obtenues sont celles qui ont déjà été générées. Il est donc possible qu'il y ait encore d'autres générations valuées dans le même instant.

- `Object[] previousValues(Identifier id)`
  Cette méthode retourne un tableau regroupant toutes les valeurs associées aux générations de l'événement `id` pour l'instant précédent. En utilisant cette méthode, on a la certitude d'avoir toutes les valeurs générées au cours de l'instant précédent.

- `Program getFrozen(Identifier id)`
  Cette méthode retourne les programmes qui ont été gelés à l'instant précédent par l'instruction `Freezable` sur génération de l'événement `id`. Pour un événement donné, on ne peut récupérer qu'une seule fois ces programmes.

- `Object linkedObject()`
  Cette méthode retourne l'objet qui est lié au programme en train d'être exécuté.

2.2.5 Exemple d'utilisation de *Junior*

Nous allons maintenant présenter un exemple qui illustre la façon de créer un programme avec *Junior* et de l'exécuter. Dans le programme suivant, nous allons montrer comment simuler un thread *Java* dont on peut démarrer, stopper, suspendre et reprendre l'exécution. Nous utiliserons l'interface `Jre` qui est une extension de l'interface `Jr`. Cette extension offre des facilités syntaxiques en permettant, par exemple, de définir des événements par l'intermédiaire de chaînes de caractères.

```
import jre.Jre;
import junior.Jr;
import junior.Machine;
import junior.Program;

public class Thread
{
   public static void main(String[] args)
   {
      Program toExecute = Jr.Loop(Jr.Seq(Jr.Print("stepped-"), Jr.Stop()));

      Program control =
         Jr.Seq(
            Jr.Seq(Jre.Await("start"), Jr.Print("started-")),
            Jre.Until("stop",
               Jre.Local("step",
                  Jr.Par(
                     Jr.Loop(
                        Jre.Until("suspend",
                           Jr.Loop(Jr.Seq(Jre.Generate("step"), Jr.Stop())),
                           Jr.Seq(
                              Jr.Print("suspended-"),
                              Jre.When("resume",
                                 Jr.Seq(Jr.Print("resumed-"), Jr.Stop()),
                                 Jr.Seq(Jre.Await("resume"), Jr.Print("resumed-")))))),
                     Jre.Control("step", toExecute))),
               Jr.Print("stopped")));

      Machine machine = Jr.Machine(control);

      for (int i = 1; i <= 12; i++)
      {
         System.out.print(i + ":");
         switch (i)
         {
            case 2 : machine.add(Jre.Generate("start"));
                     break;
            case 5 : machine.generate(Jre.StringIdentifier("suspend"));
                     break;
            case 7 : machine.add(Jre.Generate("resume"));
                     break;
            case 9 : machine.generate(Jre.StringIdentifier("stop"));
                     break;
         }
         machine.react();
         System.out.println();
      }
   }
}
```

TAB. 2.2 – Exemple en *Junior*

Pour afficher les traces de l'exécution, nous utilisons la méthode `Print("message")` présente dans l'interface `Jr`. Cet appel de méthode équivaut à l'utilisation de l'action atomique `Jr.Action(new Print("message"))` qui se charge de l'affichage.

Le programme que doit exécuter le thread est contenu dans la variable `toExecute` et, dans notre exemple, il affiche, à chaque instant d'exécution, le message `"stepped-"`. Le programme `control` contient la structure qui va contrôler l'exécution du thread. Cette structure de contrôle peut être manipulée par l'intermédiaire de quatre événements :

- `start` qui démarre l'exécution du thread. Tant que cet événement n'est pas généré, le thread est bloqué.
- `stop` qui arrête définitivement l'exécution du thread. Dès l'instant suivant la génération de cet événement, le programme ne peut plus être exécuté, quels que soient les événements qui sont générés.
- `suspend` qui stoppe momentanément l'exécution du thread. La génération de cet événement préempte la génération de l'événement local `step` ce qui implique que le programme `toExecute` sera suspendu à partir de l'instant suivant.
- `resume` qui reprend une exécution après suspension. La génération de cet événement permet de redémarrer la boucle qui génère l'événement local `step` à chaque instant. Si l'événement `resume` est généré au même instant que l'événement `suspend`, l'exécution continuera normalement à l'instant suivant.

L'événement local `step` sert d'intermédiaire. En effet, si la préemption était directement appliquée à `toExecute`, l'exécution du thread devrait être redémarrée complètement à chaque reprise de l'exécution. Il est donc préférable de préempter uniquement la génération de l'événement qui contrôle l'exécution du thread. Dans notre exemple, cela n'aurait cependant pas vraiment eu d'importance puisque le thread exécute cycliquement un programme qui ne dure qu'un instant.

Une fois le programme réalisé, il faut créer une machine réactive et le charger. Ensuite, il ne reste qu'à appeler *n fois* la méthode `react()` pour exécuter *n instants* du programme. À plusieurs reprises, la machine est modifiée entre deux instants soit par des ajouts de programmes qui génèrent les événements souhaités, soit par la génération externe des événements. L'exécution de ce programme retourne la trace suivante :

```
1:
2:started-stepped-
3:stepped-
4:suspended-stepped-
5:
6:resumed-stepped-
7:stepped-
8:stepped-
9:stopped
10:
11:
12:
```

Nous pouvons observer que les messages `suspended` et `stepped` sont affichés au même instant. En effet, l'instruction `Until` permettant uniquement d'effectuer une préemption faible, le programme est quand même exécuté à l'instant de génération de l'événement `suspend`.

2.3 Modèle d'exécution de *Reflex*

Dans le cadre de la construction d'un système comme celui des *Icobjs*, nous proposons certaines modifications de l'API standard de *Junior* qui concernent l'ajout, le retrait ou la modification de la sémantique de certaines instructions. Ces propositions sont faites pour améliorer l'efficacité du moteur et, surtout, pour obtenir une plus grande régularité dans le comportement des instructions événementielles, cette régularité étant nécessaire dans le cadre des compositions graphiques de comportements (cf. section 5.2). Pour implémenter ces modifications, nous avons créé un nouveau moteur dédié aux *Icobjs*. La nouvelle API appelée `Ic`, exclusivement destinée à l'utilisation des *Icobjs*, reste cependant assez proche de celle de *Junior*.

L'API de *Junior* a été modifiée sur plusieurs aspects que nous détaillons dans cette partie. Le premier aspect qui concerne l'ensemble des instructions est de permettre la sauvegarde et la migration des programmes en rendant tous les objets de l'API sérialisables. Le second aspect concerne la régularité des instructions de *Junior*. Cela se caractérise par le retrait de la configuration `Not` et par l'introduction de `Kill` comme instruction de préemption régulière. De plus, le retrait de `Not` permet d'étendre l'instruction `Control` aux configurations événementielles. Le troisième aspect concerne la mise en place d'un modèle d'objets réactifs (introduction de l'instruction `IcobjThread` et modifications apportées à la machine réactive) et traite des conséquences de l'utilisation de ce modèle (retrait des instructions `Freezable` et `Link`). Enfin, le dernier aspect que nous évoquons est l'introduction des instructions `Scanner` et `Run` qui permettent de définir des comportements plus modulaires.

2.3.1 Sérialisation des objets de l'API

Toutes les instructions sont déclarées comme étant de type `Serializable` pour permettre leur enregistrement dans un fichier et la mise en place du mécanisme de migration. De même, tout objet manipulé par l'environnement, comme les valeurs associées aux générations d'événements, sont également sérialisables. Ainsi, les wrappers d'objets *Java* ne retournent plus un objet de type `Object`, mais de type `Serializable`.

2.3.2 Disparition de la configuration Not

Le retrait de la configuration événementielle `Not` est l'une des principales modifications apportées à l'API. Les raisons de ce retrait sont multiples. La première raison est qu'il permet à toutes les instructions événementielles d'être plus régulières dans leur fonctionnement. En effet, nous voulons pouvoir déterminer de façon statique sur chacune des instructions quelle branche est exécutée à quel instant. Par exemple, dans le cas de l'instruction `When`, nous voulons pouvoir garantir que la branche *then* est toujours exécutée à l'instant courant si la configuration est satisfaite et, si ce n'est pas le cas, que la branche *else* est toujours exécutée à l'instant suivant. Ce n'est pas le cas en utilisant la configuration `Not` qui retarde obligatoirement la réaction à l'instant suivant. Prenons l'exemple du programme de la table 2.3. Ce programme affiche le message `thenBranch` soit à l'instant courant si l'événement `A` est présent, soit à l'instant suivant si `B` est absent. Par contre, le message `elseBranch` ne peut être affiché qu'à l'instant suivant, puisque pour cela, il faut que `A` soit absent et `B` présent alors que la détection de l'absence de `A` prend nécessairement un instant. On a donc ici un comportement dissymétrique qui complique la programmation.

```
When( "A" Or Not("B"))
   then print "thenBranch";
   else print "elseBranch";
```

TAB. 2.3 – Exemple d'irrégularité du comportement de `When`

La seconde raison du retrait de `Not` est liée à la gestion événementielle en général. Dans les langages comme *Java*, ou dans des langages dédiés à la création de mondes virtuels et d'animations comme *VRML-X3D*[79], on ne réagit en fait jamais à l'absence d'un événement. L'événement sert uniquement de déclencheur à une réaction. Plus généralement, la notion d'absence d'un événement ne prend véritablement son sens que par rapport à la notion d'instant. En effet, on ne peut déterminer l'absence d'un événement que si on considère une durée relative durant laquelle un événement devrait être présent. Dans *Junior*, on peut distinguer deux types d'instructions événementielles : celles qui s'intéressent uniquement à la satisfaction d'une configuration et qui attendent de façon bloquante cette satisfaction (`Await, Control`) et celles qui s'intéressent, à la fois, à la satisfaction et à la non-satisfaction d'une configuration (`When, Until`). Dans le contexte des *Icobjs*, nous avons choisi de nous passer de l'attente bloquante sur des absences d'événements et donc de la configuration `Not`.

Nous pouvons noter que `Await` n'est pas une instruction primitive. En effet, le comportement de celle-ci peut être simulé par l'intermédiaire d'autres instructions de l'API comme cela est décrit dans la table 2.4. Cette traduction est possible car `Until` permet de préempter des programmes et d'exécuter le `handler` dans le même instant (ce qui n'est pas le cas avec l'instruction `Kill` considérée dans la section 2.3.3).

```
Jre.Local("S",
   Jre.Until("S",
      Jr.Loop(
         Jr.Seq(
            Jre.When(C,
               Jr.Seq(Jre.Generate("S"), Jr.Stop()),
               Jr.Nothing())))))
```

TAB. 2.4 – Programme simulant le programme `Jr.Await(C)`

Enfin, la dernière raison pour laquelle on veut retirer `Not` est liée à l'implémentation. Le fait d'avoir des instructions avec un comportement plus régulier permet de définir des algorithmes plus simples et donc plus efficaces. En particulier, les instructions événementielles bloquantes ne peuvent être réveillées que par la génération de l'événement attendu. Sans `Not`, il n'est plus nécessaire de prévoir des mécanismes supplémentaires pour réveiller, à la fin de l'instant, toutes les instructions qui attendent des absences d'événements.

La disparition de `Not` entraîne une perte d'expressivité par rapport à *Junior*. Cependant, il reste toujours possible d'exprimer des configurations événementielles en attente de l'absence d'événement, par l'intermédiaire de l'instruction `When`. Par exemple, dans le cas de l'instruction `Await`, on peut simuler l'attente d'absence d'événement en utilisant la même idée que celle du programme décrit dans la table 2.4. La table 2.5 montre des exemples simulant des attentes d'absence d'événement.

`Jr.Await(Jr.Not(A))`	`Jre.Local("S",`
	`  Jre.Until("S",`
	`  Jr.Loop(`
	`  Jre.When("A",`
	`  Jr.Stop(),`
	`  Jr.Seq(Jre.Generate("S"), Jr.Stop()))))))`

`Jr.Await(Jr.And(A, Jr.Not(B)))`	`Jre.Local("S",`
	`  Jre.Until("S",`
	`  Jr.Loop(`
	`  Jr.Seq(`
	`  Jre.Await("A"),`
	`  Jre.When("B",`
	`  Jr.Stop(),`
	`  Jr.Seq(Jre.Generate("S"), Jr.Stop())))))) `

`Jr.Await(Jr.Or(A, Jr.Not(B)))`	`Jre.Local("S",`
	`  Jre.Until("S",`
	`  Jr.Par(`
	`  Jr.Seq(`
	`  Jre.Await(Jre.Or("A","S")),`
	`  Jre.Generate("S")),`
	`  Jr.Loop(`
	`  Jre.When(Jre.Or("B","S"),`
	`  Jr.Stop(),`
	`  Jr.Seq(Jre.Generate("S"), Jr.Stop()))))))`

TAB. 2.5 – Attente sur absence d'événement

Finalement, la seule limitation réelle qu'entraîne cette perte d'expressivité concerne la préemption. La traduction de la préemption sur absence d'événement n'est pas possible directement. En effet, en utilisant, dans le cadre d'une préemption, une instruction `When` pour détecter l'absence des événements (cf. table 2.4), on exécute obligatoirement un instant supplémentaire du comportement à préempter. En l'absence de `Not`, la seule instruction de l'API permettant de détecter l'absence d'un événement est `When`. La réaction à cette absence est, selon les règles de réécritures, reportée à l'instant suivant. Or, en exécutant un instant supplémentaire, le corps de l'instruction `Until` sera obligatoirement exécuté à nouveau et il faudra attendre que celui-ci atteigne un état stable avant de pouvoir le préempter. Cela tient dans le fait que `Until` est une instruction primitive et que l'on ne dispose pas de mécanisme de préemption forte comme en ESTEREL.

2.3.3 Préemption *régulière*

Nous avons vu dans la section 2.2.3 l'instruction de préemption fournie par *Junior* : `Until`. Le problème de cette instruction est qu'elle n'est pas *régulière* dans le sens où l'instant auquel le handler est exécuté dépend du moment (pendant ou à la fin de l'instant) où le corps de l'instruction atteint un état stable. Si le programme préempté atteint un état stable avant la fin d'instant, alors le `handler` est immédiatement exécuté, sinon il est exécuté à l'instant suivant. On a donc une situation dissymétrique source d'irrégularité.

Prenons l'exemple du programme suivant :

```
Jr.Par(
    Jre.Generate("preempt"),
    Jre.Until("preempt",
        Jr.Seq(Jre.Await("event"),Jr.Stop()),
        Jr.Print("preempted")))
```

L'exécution de ce programme, au premier instant, diffère selon que l'événement `event` est présent ou non. Si l'événement est présent, alors le programme à préempter atteint un état stable en arrêtant son exécution sur l'instruction `Stop`. Puisque l'événement de préemption est présent, alors le `handler`, c'est-à-dire l'affichage de `preempted`, est immédiatement exécuté. Si, au contraire, l'événement `event` est absent, alors il faut attendre la fin d'instant pour que le programme atteigne un état stable. La fin d'instant ayant été déclarée, le `handler` doit attendre l'instant suivant pour être exécuté. Dans ce cas, l'affichage de `preempted` n'aura lieu qu'à l'instant suivant.

Initialement, l'instruction `Until` a été introduite pour exécuter le maximum d'instructions avant de se stabiliser. Cela ne nous semble pas être une raison suffisante dans le cadre des *Icobjs*. En effet, dans ce cadre, la préemption est surtout utilisée pour changer de mode de comportement, c'est-à-dire stopper un comportement cyclique et en exécuter un nouveau. Plus généralement dans le cadre de la programmation réactive graphique (cf. section 5.2), il nous semble préférable d'avoir une instruction de préemption au comportement prévisible, donc *régulier*, dépendant le moins possible des spécificités du comportement à préempter. Nous voulons connaître exactement le comportement de l'instruction de préemption quand elle exécute sa préemption.

C'est pourquoi, comme dans le cas de *Glouton*[48], l'instruction `Kill`, reprise au langage *SL* [17], a été rajoutée à l'API de *Reflex*. Comme `Until`, `Kill` est une instruction de préemption faible, mais elle a un comportement plus régulier au sens où le `handler` est toujours exécuté à l'instant suivant celui de la préemption. La sémantique de cette instruction est décrite dans la section 3.9.1.

2.3.4 Control avec configuration événementielle

Dans *Junior*, l'instruction `Control` ne considère qu'un événement et non une configuration événementielle. La raison de cette restriction est d'empêcher l'utilisation de la configuration `Not`. En effet, par définition, l'instruction `Control` n'exécute un programme réactif qu'aux instants où un événement est présent. L'utilisation de `Control` sur absence d'un événement serait ainsi en contradiction avec la définition de l'instruction. De plus, elle serait, comme toute réaction à l'absence d'un événement, retardée jusqu'à l'instant suivant. Or, `Control` a pour but de réagir immédiatement aux événements. Ce retard entrerait également en contradiction avec la définition de `Control`.

Dans le cas de *Reflex*, puisque nous nous passons de la configuration `Not`, nous pouvons étendre l'instruction `Control` en permettant l'utilisation d'une configuration événementielle. En effet, nous avons la garantie qu'en l'absence de `Not`, une configuration événementielle, si elle est satisfaite, l'est toujours avant la fin de l'instant et donc que le programme filtré peut être immédiatement exécuté.

2.3.5 Retrait de l'instruction Freezable

Initialement, l'instruction `Freezable` a été introduite pour coder des agents migrants. Quand un agent doit partir d'un site, il gèle son comportement et en récupère le résidu, puis migre vers un autre site où il exécute le résidu.

Le gel de programme sur génération d'un événement, réalisé par cette instruction, correspond, pour la partie préemption, à l'instruction `Until`. Comme nous l'avons signalé dans

la section 2.3.3, `Until` est une instruction au comportement irrégulier. De plus, l'instruction `Freezable` pose d'autres problèmes que nous allons détailler maintenant.

Le premier problème de l'instruction `Freezable` vient de ce qu'elle peut transformer deux programmes placés en séquence en une composition parallèle de ces deux programmes. En effet, il est possible de terminer l'exécution d'une instruction `Freezable` et de passer directement à l'exécution d'autres instructions qui sont en séquence et qui peuvent être gelées sur le même événement, au même instant. L'exemple suivant illustre ce point.

```
Jr.Seq(
   Jre.Freezable("freeze",
      Jre.Repeat(5,
         Jr.Seq(Jre.Generate("in"), Jr.Stop()))),
   Jre.Freezable("freeze"),
      Jr.Seq(Jre.Await("out"), Jr.Stop()))
```

Le comportement défini dans cet exemple est de générer, pendant 5 instants, l'événement `"in"` et ensuite d'attendre la génération de l'événement `"out"`. Cet exemple n'est pas réaliste, mais il révèle le problème. En effet, si l'événement de gel `"freeze"` est généré par exemple au 3ème instant, la boucle finie de génération de l'événement `"in"` est gelée avant la fin de l'instant. Ainsi, l'attente de l'événement `"out"` est exécutée et sera gelée à la fin d'instant si l'événement est absent. Les résidus des deux programmes gelés sont, au moment du gel, mis en parallèle et enregistrés dans l'environnement. Le programme gelé va donc générer l'événement `"in"` tout en attendant `"out"`, ce que ne faisait pas le programme initial.

Le second problème de l'instruction `Freezable` est lié à l'enregistrement du résidu du programme gelé dans l'environnement. En effet, on veut que les agents migrants puissent partir dès la fin de l'instant. L'utilisation de `Freezable` ne rend pas cela possible. En effet, on ne peut récupérer un programme gelé qu'à l'instant suivant le gel, pour récupérer tous les comportements gelés sur le même événement. Pendant le changement d'instant, il peut se passer différentes choses (enregistrement dans un fichier, migration de code...) qui vont interférer avec le bon fonctionnement des *Icobjs*. Par exemple, si on enregistre l'état d'un icobj à la fin de l'instant alors que celui-ci a gelé une partie de son comportement qui est stocké dans l'environnement, l'enregistrement ne tiendra pas compte de cette partie. Prenons par exemple le cas de l'instruction `DynaPar` introduit par R. Acosta dans [2] dont on peut trouver une traduction dans la table 2.6 en *REJO* en utilisant les instructions `Freezable` et `Run`. L'objectif de `DynaPar` est d'ajouter dynamiquement à une instruction `Par` de nouvelles branches en générant un événement dont la valeur contient le programme à ajouter. Pour cela, le programme initialement enregistré dans l'instruction `DynaPar` est encapsulé par une instruction `Freezable` qui gèle le programme sur présence de l'événement d'ajout. Lorsque cet événement est généré, le programme est gelé au plus tard à la fin de l'instant. À l'instant suivant, le programme gelé à l'instant précédent est récupéré dans l'environnement et mis en parallèle avec l'ensemble des programmes associés à l'événement d'ajout généré à l'instant précédent. Le problème vient de la possibilité d'imbrication de plusieurs instructions `Freezable`. Considérons le cas d'un agent (comme un *REJO*) qui exécute une instruction `DynaPar`. Si l'événement demandant la migration de l'agent et l'événement d'ajout d'un nouveau comportement à l'instruction `DynaPar` sont générés au même instant, l'instruction `DynaPar` et l'instruction `Freezable` qui encapsule le comportement de l'agent gèlent chacune le programme qu'elles encapsulent. `DynaPar` enregistre le résidu du comportement dans l'environnement pour le récupérer et le recharger à l'instant suivant en y ajoutant tous les programmes associés à l'événement d'ajout. De son

```
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);
   }
}
```

TAB. 2.6 – Traduction de l'instruction DynaPar introduite en REJO

côté, l'agent enregistre aussi le résidu de son comportement complet, y compris l'instruction `DynaPar`, dans l'environnement pour pouvoir le récupérer à l'instant suivant et le faire migrer sur le nouveau site. Quand l'agent exécutera le résidu de son comportement sur le site distant, l'instruction `DynaPar` demandera à récupérer son comportement sur son nouveau site alors que celui-ci est stocké sur le site précédent, ce qui sera incorrect.

Dans le cas de l'utilisation d'agents et plus généralement d'objets réactifs que l'on veut pouvoir sauvegarder ou migrer, il faut qu'à la fin de chaque instant le programme attaché à l'agent ou à l'objet réactif soit complet. C'est pourquoi nous avons décidé de retirer l'instruction `Freezable`. Pour remplir le rôle de cette instruction dans le cas des *Icobjs*, nous avons ajouté l'instruction `IcobjThread` que nous allons présenter dans la section 2.3.7. Il faut noter que le retrait de `Freezable` entraîne également celui de la méthode `getFrozen` de la machine réactive.

2.3.6 Retrait de l'instruction Link

Nous avons défini un modèle strict d'objets réactifs qui est réalisé au niveau de la machine réactive par l'instruction `IcobjThread`. Cette instruction gérant elle-même le lien entre l'objet et son programme, il n'est plus nécessaire de conserver l'instruction `Link` dans l'API de *Reflex*.

Le retrait de cette instruction entraîne celui de l'objet `linkedObject` et celui de la méthode qui permet d'y accéder `Object linkedObject()`.

2.3.7 Instruction IcobjThread

Nous avons choisi de spécialiser notre moteur aux objets réactifs que sont les icobjs. Ainsi, tout comportement sera toujours rattaché à un icobj. Pour effectuer cette spécialisation, nous avons ajouté l'instruction interne `IcobjThread`. Tout icobj est automatiquement encapsulé dans cette structure. Cette instruction est directement utilisée par le moteur. Nous allons détailler son rôle :

- elle exécute le programme associé à un icobj dans le contexte de celui-ci. Comme l'instruction `Link`, elle place dans l'environnement une référence vers l'icobj qui exécute son comportement. Pour accéder à cette référence, nous avons ajouté à l'environnement la méthode `Icobj This()`.
- elle gèle le programme associé lorsque cela lui est demandé et stocke le résidu dans un champ prédéfini de l'icobj. Cette requête de gel n'est pas événementielle comme pour l'instruction `Freezable`. Il faut demander à la machine réactive le retrait de l'objet attaché à ce programme, ce qui entraînera le gel du programme à la fin de l'instant. Cependant, si l'on souhaite geler le programme sur présence d'un événement, il est toujours possible d'écrire un programme qui attend l'événement de gel et qui exécute en séquence une action atomique demandant le gel du comportement de l'icobj qui est en train d'être exécuté.
- elle permet d'ajouter dynamiquement de nouveaux programmes à un objet réactif qui est déjà en train d'être exécuté dans la machine. Ces nouveaux programmes seront pris en compte au début de l'instant suivant. Le fait de regrouper tous les programmes associés à un même objet accélère l'exécution. En particulier, il n'est plus nécessaire de parcourir tous les programmes contenus dans la machine pour récupérer ceux associés à l'objet migrant. De plus, pour un objet donné, on évite d'avoir plusieurs programmes dispersés dans l'arbre de syntaxe, exécutant chacun une instruction `Link` et une instruction `Freezable`. Il faut noter qu'il est toujours possible d'ajouter un programme à un icobj tant que celui-ci n'est pas explicitement retiré de la machine. Cela signifie que même si les programmes attachés à l'icobj ont tous terminé leur exécution, l'objet `IcobjThread` continue d'exister.

2.3.8 Machine réactive

Junior permet l'utilisation de deux types de machines réactives : la *unsafe-machine* ou la *safe-machine*. Pour la gestion des *Icobjs*, on se trouve nécessairement dans un cadre multi-threadé. Des événements sont par exemple générés depuis l'extérieur de la machine réactive, comme ceux provenant de la gestion du clavier ou de la souris. L'utilisation d'une *unsafe-machine* ne donne aucune garantie quant à une gestion correcte des événements provenant de l'extérieur. Par exemple, une *unsafe-machine* ne garantit pas que des événements soient toujours pris en compte.

C'est pourquoi, l'API de *Reflex* ne donne accès qu'à un seul type de machine réactive, la `MachineIcobj`. Cette machine a presque les mêmes caractéristiques qu'une `safe-machine`, excepté la méthode `add`. En effet, dans le cas de notre `MachineIcobj`, tout programme ajouté à la machine est obligatoirement attaché à un objet `Icobj` par l'intermédiaire de la méthode

`void add(Icobj ic, Program p)`. Si l'icobj est déjà enregistré dans la machine, alors le programme sera ajouté en parallèle dans l'objet `IcobjThread` en charge de l'icobj. Si l'icobj n'est pas encore présent dans la machine, alors la machine crée un nouvel objet `IcobjThread` pour se charger de cet icobj et l'ajoutera à l'instant suivant à l'instruction `Par` de plus haut niveau.

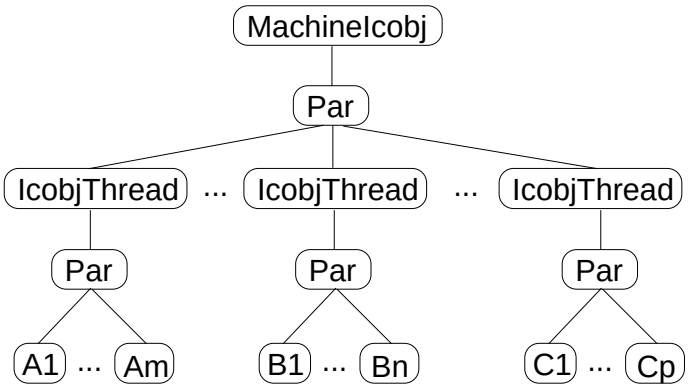


FIG. 2.4 – Structure interne de MachineIcobj

D'autres méthodes ont été ajoutées à la machine réactive. Nous les détaillerons dans la section 4.2.3 concernant l'implémentation de la machine.

La figure 2.4 montre la structure interne de notre machine réactive et l'utilisation interne de l'instruction `IcobjThread`. Au plus haut niveau de la machine, nous avons une instruction `Par` dont chaque fils est un `IcobjThread`. Cette instruction `Par` est une instruction n-aire et non binaire. Dans le cadre de notre moteur, les instructions `Par` et `Seq` sont en fait toujours des instructions n-aires. Nous détaillerons cela en présentant la sémantique et l'implémentation du moteur dans la section 4.2.1.

2.3.9 Instruction Scanner

Junior offre la possibilité de générer des événements avec valeurs, mais la seule possibilité de traiter l'ensemble des valeurs générées au cours d'un instant est d'attendre l'instant suivant et de récupérer ces valeurs par l'intermédiaire de la méthode `previousValues` de l'environnement. *SugarCubes* permet de traiter ces valeurs générées au cours de l'instant par l'intermédiaire d'une instruction appelée `Callback`. Une instruction identique `Scanner` a été ajoutée dans certaines implémentations de *Junior*. Nous rajoutons de même cette instruction à l'API de *Reflex*. Le format est le suivant :

`Ic.Scanner(IdentifierWrapper id, ScanAction scan)`

Cette instruction exécute, à chaque occurrence d'un événement valué, une action atomique traitant la valeur associée à l'événement. L'événement est obtenu après évaluation du wrapper d'événement. L'action atomique est de type `ScanAction` et non de la classe `Action`. Voilà la signature des méthodes de chacun des deux types d'actions atomiques :

- `void execute(Environment env, Icobj self, Serializable value)` pour une action atomique `ScanAction`
- `void execute(Environment env, Icobj self)` pour une action atomique classique

Nous pouvons noter, qu'à la différence de *Junior* et pour plus de commodités, nous ajoutons en argument de ces méthodes une référence à l'icobj qui est exécuté.

L'instruction `Scanner` est très utile dans le cadre graphique. En particulier, elle est indispensable pour traiter la totalité des événements envoyés par l'utilisateur dans l'instant de leur génération, comme par exemple, des événements externes provenant du clavier ou de la souris.

La `ScanAction` ne doit pas générer l'événement sur lequel l'instruction `Scanner` réagit car, si elle le génère, une boucle infinie instantanée apparaît alors et empêche la fin de l'instant. De même, il faut aussi faire attention à l'utilisation de la valeur de l'événement par l'action atomique. En effet, toute valeur générée n'est qu'une référence à un objet *Java* et non une copie stricte. Un objet `ScanAction` peut donc modifier l'état de cette valeur. À cause de ces effets de bords, on ne peut donc pas garantir que toutes les `ScanAction` verront la valeur générée dans le même état.

2.3.10 Instruction Run

En utilisant les wrappers d'événements, *Junior* offre la possibilité de déterminer l'identificateur d'un événement à l'exécution du programme et non à la création de celui-ci. Il existe également des wrappers pour les objets *Java*, des wrappers pour les entiers utilisés par l'instruction `Repeat` et des wrappers pour les valeurs booléennes utilisés par l'instruction `If`.

De la même manière, nous avons ajouté le moyen d'exécuter un programme qui n'est pas connu au chargement dans la machine réactive. Ce moyen est l'instruction `Run` qui prend en paramètre un wrapper de programmes. Cette instruction existe déjà en *REJO*[1][2] où elle est appelée `call`. Le format de cette instruction est le suivant :

```
Ic.Run(ProgramWrapper prog)
```

Au moment de son exécution, l'instruction `Run` évalue le wrapper de programme qui lui retourne alors un programme réactif qui sera exécuté immédiatement. Pour cela, la classe `ProgramWrapper` dispose de la méthode `Program evaluate(Environment env)` qui permet de récupérer le programme à exécuter.

2.4 Bilan

Nous venons de présenter les principales modifications apportées à l'API standard de *Junior* pour réaliser celle de *Reflex*. Ces modifications mettent en évidence la création d'un modèle d'objets réactifs stricts dans lequel tout comportement est obligatoirement associé à un objet. Nous appliquons directement ce modèle aux *Icobjs*, mais une telle approche pourrait être généralisée à d'autres objets pourvu qu'ils disposent d'une méthode pour enregistrer le résidu de leur comportement lorsqu'ils sont retirés de la machine. Ce modèle permet de faire migrer des objets dès la fin de l'instant. Ceci est aussi vrai pour l'enregistrement de l'état d'un icobj dans un fichier car on peut considérer un enregistrement comme une migration vers un fichier.

Une autre modification importante est le retrait de la configuration `Not` qui rend l'utilisation des instructions événementielles plus régulière.

Le chapitre 3 va maintenant détailler la sémantique de chacune des instructions de *Reflex* et le chapitre 4 présentera l'implémentation de ce moteur en détaillant les modifications qui ont été apportées à *Storm*.

Chapitre 3

Sémantique de *Reflex*

Dans cette partie, nous allons présenter la sémantique formelle de *Reflex*. Cette sémantique est, à la fois, proche et éloignée de l'implémentation du moteur. Elle est proche au sens où elle présente des optimisations du moteur, par exemple par l'ajout d'un nouveau statut qui évite les attentes actives inter-instants, de même que *Storm* l'avait fait en rajoutant le statut *WAIT* pour les attentes actives dans l'instant. De plus, cette sémantique contrairement aux précédentes introduit les wrappers et montre le moment de leur évaluation. Cette sémantique est aussi assez proche de l'implémentation car elle présente, par l'intermédiaire de nombreuses fonctions intermédiaires, le fonctionnement exact des différentes instructions. Nous représenterons également les moments où des effets de bords peuvent apparaître. Cependant, nous ne ferons référence qu'aux effets de bords relatifs aux instructions réactives et nous ne tiendrons absolument pas compte des effets de bords dus à l'exécution de programmes concurrents dans d'autres threads *Java*. Le moteur de *Junior* a la possibilité de recevoir des générations externes d'événements ; nous ne tiendrons pas compte non plus de ces générations dans cette sémantique.

Par contre, cette sémantique sera assez éloignée de l'implémentation au sens où nous n'avons pas souhaité la compliquer par des considérations qui ont déjà été abordés dans la sémantique de *Replace* ou de *Storm*. Par exemple, nous continuons à représenter le fonctionnement de l'instruction *Loop* comme en *Rewrite*, c'est-à-dire comme si on recréait les objets, alors qu'au final, nous utilisons dans l'implémentation la méthode définie dans *Replace* qui réinitialise les instructions. Nous ne présenterons pas la sémantique de l'instruction interne *IcobjThread* car il nous faudrait représenter, en plus de ses réécritures, les effets de bords dus aux appels de méthode pour l'ajout de programmes et le retrait de l'instruction. De plus, elle modifie un pointeur dans l'environnement qui n'est pas directement réutilisé par les autres instructions et qui complexifierait la représentation de l'environnement.

3.1 Règles de sémantique

Pour formaliser *Reflex*, nous utiliserons une sémantique opérationnelle [44] et, plus particulièrement, nous décrirons l'ensemble des instructions réactives en utilisant des règles SOS, formalisme introduit par Plotkin ([61]). Ces règles sont de la forme suivante :

$$\frac{condition_1 \quad \dots \quad condition_n}{t, E \xrightarrow{<\alpha, A>} t', E'}$$

Cette règle signifie que, sous les conditions 1 à n, l'exécution du terme t dans l'environnement E se réécrit en un terme t' dans l'environnement E'. Cette exécution retourne un couple $< \alpha,\ A >$, α représentant le statut de terminaison résultant de l'exécution de l'instruction et A étant l'ensemble des configurations événementielles (cf. section 3.3) sur lesquelles le terme t' est en attente.

Il est possible qu'il n'y ait aucune condition préalable pour l'application d'une règle, auquel cas, la règle sera simplifiée en $t, E \xrightarrow{<\alpha,A>} t', E'$.

3.1.1 Notations

Commençons d'abord par préciser les différentes notations utilisées dans ces règles.

- On note t, u et v les termes utilisés. Un terme correspond à une instruction réactive.
- Nous désignons par $< v_1, \ldots, v_n >$ des n-uplets. Le remplacement d'un des éléments de ce n-uplet par un $-$ signifie qu'on ne tient pas compte de la valeur de cet élément.
- On note $a.b$ la liste composée dans l'ordre de l'élément a suivi de l'élément b. Si a et/ou b sont des listes, alors la liste résultante sera une concaténation de deux listes ou l'ajout d'un élément au début de la liste (si b désigne la liste) ou à la fin de la liste (si a désigne la liste).
- On note ε la liste vide.

3.1.2 Statuts de terminaison

Les quatre statuts de terminaison possibles dans *Reflex* sont *TERM*, *STOP*, *WAIT* et *LONGWAIT*. À la différence des autres versions de *Junior*, le statut *SUSP* a disparu et il a été remplacé par les statuts *WAIT* et *LONGWAIT* pour des raisons d'optimisation. Voici la signification de ces quatre statuts de terminaison :

- ***TERM*** : l'exécution du terme t est définitivement terminée et il n'est plus nécessaire de l'activer.
- ***STOP*** : l'exécution du terme est terminée pour l'instant courant, mais il devra être réactivé à l'instant suivant.
- ***WAIT*** : l'exécution du terme t est bloquée en attente d'un événement ou en attente de la fin d'instant. Le terme devra obligatoirement être ré-exécuté une fois que la fin d'instant aura été déclarée, si l'événement n'a pas été généré au cours de l'instant.
- ***LONGWAIT*** : ce nouveau statut, assez proche du statut *WAIT*, est retourné lorsque l'exécution d'un terme est bloquée sur la présence d'un événement. Mais, contrairement à *WAIT*, il est inutile de le réveiller à la fin de l'instant. Le terme qui renvoie ce statut ne sera pas ré-exécuté tant que l'événement sur lequel il est bloqué n'a pas été généré.

Le statut de terminaison est retourné accompagné d'un ensemble de configurations événementielles sur lesquelles la branche est en attente. Par définition, une branche qui renvoie *TERM* ou *STOP* n'est pas en attente d'événements, donc l'ensemble renvoyé est vide.

3.1.3 Environnement d'exécution

L'environnement d'exécution, noté E, contient les informations nécessaires à une exécution correcte d'un programme. Cet environnement englobe l'environnement d'exécution du langage hôte auquel est ajouté :

- une table qui associe à chaque identifiant d'événement, les informations concernant cet événement pour l'instant courant, c'est-à-dire son statut de présence et une liste constituée de toutes les valeurs associées aux générations de cet événement au cours de l'instant.
- un booléen *eoi* qui signale la fin d'instant. Ce booléen est à *true* si la fin d'instant est déclarée, *false* sinon. La déclaration de fin d'instant est faite par la machine réactive (cf. section 3.8).

Dans le cadre des règles de réécritures, nous allons utiliser les notations suivantes pour représenter les interactions avec l'environnement :

$$\begin{aligned} E(eoi) &\equiv eoi \\ E[eoi \setminus b] &\equiv E' \text{ semblable à } E \text{ excepté que } E'(eoi) = b \\ E(S) &\equiv < b, V > \text{ avec } b \text{ représentant la présence de l'événement } S \\ &\quad \text{et } V \text{ la liste des valeurs associées à la génération de } S \\ E[S \setminus < b, V >] &\equiv E' \text{ semblable à } E \text{ excepté que } E'(S) =< b, V > \end{aligned}$$

3.2 Wrappers

En *Junior*, il existe plusieurs types de wrappers qui sont utilisés par les instructions réactives :

- le wrapper d'événement X. Son évaluation retourne un identifiant d'événement.
- le wrapper d'entier I. Son évaluation retourne une valeur entière.
- le wrapper de booléen B. Son évaluation retourne *true* ou *false*.
- le wrapper d'objet O. Son évaluation renvoie un objet quelconque[1] du langage hôte.
- le wrapper de programme T. Son évaluation renvoie un terme.

Nous considérons l'exécution d'une action atomique comme une évaluation d'un programme du langage hôte que l'on désigne par a. Nous introduisons une fonction *eval* qui prend un wrapper (ou un programme du langage hôte) et l'environnement d'exécution en paramètre et qui renvoie la valeur retournée par l'évaluation et le nouvel environnement. Nous ne spécifions pas les modifications de l'environnement d'exécution du langage hôte. La seule chose garantie est que l'évaluation du wrapper (ou du programme du langage hôte) modifie uniquement l'environnement du langage hôte contenu dans E (il ne modifie ni *eoi*, ni la table des événements).

$$\begin{aligned} eval(X, E) &\equiv < S, E' > \\ eval(I, E) &\equiv < n, E' > \mid n \in \mathbb{N} \\ eval(B, E) &\equiv < b, E' > \mid b \in \{true, false\} \\ eval(O, E) &\equiv < o, E' > \mid o \text{ une valeur} \\ eval(T, E) &\equiv < t, E' > \mid t \text{ un terme} \\ eval(a, E) &\equiv < -, E' > \end{aligned}$$

$$\text{avec } E(eoi) = E'(eoi) \text{et } \forall S \; E(S) = E'(S)$$

[1] On suppose que l'on se place dans le cadre d'un langage hôte objet, ce qui est le cas de *Java*.

Remarque : Pour la plupart des instructions utilisant les wrappers, nous introduisons une forme auxiliaire désignée par le nom de l'instruction suivi de *. Ces formes auxiliaires sont équivalentes aux formes normales excepté qu'elles manipulent des identifiants d'événements au lieu de wrappers d'événements qu'elles n'ont plus à évaluer.

3.3 Configurations événementielles

Une configuration événementielle permet d'utiliser des combinaisons de présences d'événements. Ces combinaisons s'expriment à l'aide d'expressions booléennes. Nous différencierons deux types de configurations : la configuration de wrappers que l'on va noter CX et la configuration d'événements que l'on va noter C. Dans chacun de ces types, nous considérons trois formes de configurations possibles :

$$\begin{array}{rcl} CX & \equiv & X : \text{un wrapper d'événement} \\ & | & CX_1 \ And \ CX_2 : \text{une conjonction de deux configurations de wrappers} \\ & | & CX_1 \ Or \ CX_2 : \text{une disjontion de deux configurations de wrappers} \end{array}$$

$$\begin{array}{rcl} C & \equiv & S : \text{un événement} \\ & | & C_1 \ And \ C_2 : \text{une conjonction de deux configurations d'événements} \\ & | & C_1 \ Or \ C_2 : \text{une disjontion de deux configurations d'événements} \end{array}$$

On a besoin d'étendre la fonction *eval* pour qu'elle tienne compte des configurations de wrappers. On ajoute juste ces deux définitions :

$$\begin{array}{rcl} eval(CX_1 \ And \ CX_2, E) & \equiv & < C_1 \ And \ C_2, E'' > \\ eval(CX_1 \ Or \ CX_2, E) & \equiv & < C_1 \ Or \ C_2, E'' > \\ & & \text{où } < C_1, E' >= eval(CX_1, E) \\ & & \text{et } < C_2, E'' >= eval(CX_2, E') \end{array}$$

Pour juger de la présence ou non d'un événement, et plus généralement de la satisfaction d'une configuration d'événement à un instant donné, il faut s'assurer que l'on dispose de suffisamment d'information sur les événements entrants dans la composition d'une configuration. Pour simplifier les règles de réécritures, nous introduisons la fonction *sat* qui retourne le statut d'une configuration. Il y a donc trois cas possibles :

- soit la configuration est satisfaite et *sat* retourne *true*,
- soit elle ne l'est pas et *sat* retourne *false*,
- soit il manque des informations pour juger et *sat* retourne $\perp$.

Pour définir clairement *sat*, nous définirons d'abord deux fonctions auxiliaires, *present* et *fixed*. La première, *present*, évalue l'expression booléenne formée par la configuration d'événements avec les informations contenues dans l'environnement au moment de l'appel de

present.

$$\begin{aligned} present(S,E) &\equiv E(S) =< true, - > \\ present(C_1\ And\ C_2,E) &\equiv present(C_1,E) \text{ and } present(C_2,E) \\ present(C_1\ Or\ C_2,E) &\equiv present(C_1,E) \text{ or } present(C_2,E) \end{aligned}$$

La fonction *fixed* permet de juger si, au moment où elle est appelée, on dispose de suffisamment d'information sur les événements de la configuration pour pouvoir juger de la satisfaction. Pour cela, il suffit de vérifier si un nombre suffisant d'événements composant la configuration a été généré ou si on peut être sûr que ces événements ne le seront pas, ce qui est le cas à la fin de l'instant.

$$\begin{aligned} fixed(S,E) &\equiv E(S) =< true, - > \text{ or } E(eoi) = true \\ fixed(C_1\ And\ C_2,E) &\equiv fixed(C_1,E) \text{ and } fixed(C_2,E) \\ &\quad \text{or } fixed(C_1,E) \text{ and not } present(C_1,E) \\ &\quad \text{or } fixed(C_2,E) \text{ and not } present(C_2,E) \\ fixed(C_1\ Or\ C_2,E) &\equiv fixed(C_1,E) \text{ and } fixed(C_2,E) \\ &\quad \text{or } fixed(C_1,E) \text{ and } present(C_1,E) \\ &\quad \text{or } fixed(C_2,E) \text{ and } present(C_2,E) \end{aligned}$$

Enfin, voilà *sat* que l'on peut maintenant définir à partir de *present* et de *fixed*.

$$\begin{aligned} sat(C,E) &\equiv present(C,E) \text{ si } fixed(C,E) \\ &\quad \perp \text{ sinon} \end{aligned}$$

Nous introduisons directement une deuxième définition qui prend en compte le nombre de générations valuées sur un événement. Cette définition ne va être utilisée que dans le cas de l'instruction *Scanner* (cf. section 3.9.2).

$$\begin{aligned} S_n &\equiv \text{la } n\text{-ième génération valuée de l'événement } S \\ sat(S_n,E) &\equiv true \text{ si } E(S) =< true, o_1 \ldots o_n.V > \\ &\quad false \text{ sinon} \end{aligned}$$

Enfin, on introduit la fonction *check* qui renvoie *true* si au moins un des éléments contenus dans un ensemble est satisfait. Cet ensemble est composé de configurations d'événements C et/ou d'événements dont on attend une certaine occurrence de génération valuée S_n.

$$check(A,E) = true \equiv \exists x \in A | sat(x,E) = true$$

Par rapport à *Junior*, la configuration *Not* a disparu. Donc, on est sûr qu'une configuration :

- est nécessairement satisfaite avant la fin d'instant ($E(eoi) = false$). On ne teste que des présences d'événements ou des conjonctions/disjonctions de présences et un événement ne peut être généré pour un instant courant qu'avant la fin de celui-ci.
- ne peut être insatisfaite qu'à la fin de l'instant ($E(eoi) = true$). Pour les mêmes raisons, on ne peut décider de l'absence d'un événement qu'à la fin de l'instant et pour qu'une configuration ne soit pas satisfaite, il faut qu'il y ait au moins un événement absent.

3.4 Syntaxe abstraite

Dans cette partie, nous allons donner la traduction de la syntaxe concrète des instructions de l'API vers la syntaxe abstraite qui est utilisée dans les règles de réécritures.

Syntaxe concrète	Syntaxe abstraite
Ic.Nothing()	*Nothing*
Ic.Stop()	*Stop*
Ic.Atom(Action a)	$Atom(a)$
Ic.Seq(Program[] t)	$Seq(t_1.t_2.\ldots.t_n)$
Ic.Loop(Program t)	$Loop(t)$
Ic.Repeat(IntegerWrapper I, Program t)	$Repeat(I, t)$
Ic.If(BooleanWrapper B, Program t, Program u)	$If(B, t, u)$
Ic.Generate(IdentifierWrapper X)	$Generate(X)$
Ic.Generate(IdentifierWrapper X, ObjectWrapper O)	$Generate(X, O)$
Ic.Await(Configuration CX)	$Await(CX)$
Ic.Control(Configuration CX, Program t)	$Control(CX, t)$
Ic.When(Configuration CX, Program t, Program u)	$When(CX, t, u)$
Ic.Until(Configuration CX, Program t, Program u)	$Until(CX, t, u)$
Ic.Local(Identifier S, Program t)	$Local(S, false, \varepsilon, t)$
Ic.Par(Program[] t)	$Par(t_1.t_2.\ldots.t_n, \varepsilon, \varepsilon, \varepsilon)$
Ic.Kill(Configuration CX, Program t, Program u)	$Kill(CX, t, u)$
Ic.Scanner(IdentifierWrapper X, ScanAction a)	$Scanner(X, a)$
Ic.Run(ProgramWrapper T)	$Run(T)$

TAB. 3.1 – Traduction de la syntaxe concrète vers la syntaxe abstraite

Pour l'instruction *Local*, nous introduisons, en plus de l'identifiant d'événement et du terme à exécuter :

- un booléen indiquant l'état de présence de l'événement. Ce booléen est initialisé à *false*.
- une liste de valeurs associées aux générations de cet événement. Cette liste est initialement vide ε.

Pour l'instruction *Par*, nous ajoutons trois listes initialement vide (ε) que nous définirons dans la section 3.7.

3.5 Instructions de base

3.5.1 Nothing

L'instruction *Nothing* ne fait rien et se termine instantanément.

$$Nothing, E \xrightarrow{<TERM,\emptyset>} Nothing, E \tag{3.1}$$

3.5.2 Stop

L'instruction *Stop* permet d'arrêter l'exécution d'une séquence d'instructions pour l'instant courant. À l'instant suivant, l'exécution de cette séquence reprendra à partir de ce point d'arrêt.

$$Stop, E \xrightarrow{<STOP,\emptyset>} Nothing, E \tag{3.2}$$

3.5.3 Actions atomiques

Une action atomique est une instruction qui exécute un programme du langage hôte sans être interrompue par une autre action concurrente. Cette instruction se termine instantanément.

$$\frac{eval(a, E) =< -, E' >}{Atom(a), E \xrightarrow{<TERM,\emptyset>} Nothing, E'} \tag{3.3}$$

Remarque : On suppose que l'exécution du programme du langage hôte termine nécessairement. Si ce n'était pas le cas, il n'y aurait pas de réécriture possible.

3.5.4 Séquence N-aire

L'instruction *Seq* est une instruction N-aire et traite donc une liste d'instructions. *Seq* exécute la première instruction de la liste. Dès que celle-ci termine, l'instruction *Seq* exécute immédiatement l'instruction suivante dans la liste, et ceci jusqu'à ce que cette liste soit vide. Si c'est le cas, l'instruction termine.

Si la liste est vide, alors on suit cette règle :

$$Seq(\varepsilon), E \xrightarrow{<TERM,\emptyset>} Nothing, E \tag{3.4}$$

S'il y a au moins un programme dans la liste et que l'exécution de ce programme ne termine pas dans l'instant, alors on exécute cette règle :

$$\frac{t, E \xrightarrow{<\alpha,A>} t', E' \quad \alpha \neq TERM}{Seq(t.L), E \xrightarrow{<\alpha,A>} Seq(t'.L), E'} \tag{3.5}$$

Si, par contre, la première instruction de la liste termine immédiatement, alors elle est enlevée de la liste et l'exécution de la séquence se poursuit récursivement sur le reste de la liste jusqu'à ce qu'une des deux règles précédentes s'applique.

$$\frac{t, E \xrightarrow{<TERM,\emptyset>} t', E' \quad Seq(L), E' \xrightarrow{<\alpha,A>} u, E''}{Seq(t.L), E \xrightarrow{<\alpha,A>} u, E''} \tag{3.6}$$

3.5.5 Boucle infinie

L'instruction *Loop* ré-exécute son corps dès que celui-ci termine. Si le corps ne termine pas au cours de l'instant, alors l'instruction *Loop* se réécrit en une séquence, ce qui est traduit par cette règle :

$$\frac{t, E \xrightarrow{<\alpha,A>} t', E' \quad \alpha \neq TERM}{Loop(t), E \xrightarrow{<\alpha,A>} Seq(t'.Loop(t)), E'} \tag{3.7}$$

Si le corps de la boucle termine instantanément, alors il est immédiatement ré-exécuté.

$$\frac{t, E \xrightarrow{<TERM,\emptyset>} t', E' \quad Loop(t), E \xrightarrow{<\alpha,A>} u, E''}{Loop(t), E \xrightarrow{<\alpha,A>} u, E''} \tag{3.8}$$

Remarque : si l'exécution du corps renvoie toujours *TERM*, alors nous obtenons une boucle instantanée. Dans ce cas, il n'y a pas de réécriture possible.

3.5.6 Boucle finie

L'instruction *Repeat* permet de boucler un nombre fini de fois sur un programme. Dans ce cas, il n'y a pas besoin de se protéger d'une boucle instantanée puisque l'exécution d'un *Repeat* terminera au bout d'un temps fini. D'abord, *Repeat* commence par évaluer son wrapper qui lui renvoie un entier, puis continue son exécution en utilisant sa forme auxiliaire $Repeat^*$.

$$\frac{eval(I, E) =< n, E' > \quad Repeat^*(n, t), E' \xrightarrow{<\alpha,A>} u, E''}{Repeat(I, t), E \xrightarrow{<\alpha,A>} u, E''} \tag{3.9}$$

Si l'entier retourné est inférieur à 0, alors $Repeat^*$ termine immédiatement.

$$\frac{n \leq 0}{Repeat^*(n, t), E \xrightarrow{<TERM,\emptyset>} Nothing, E} \tag{3.10}$$

Si au contraire cet entier est strictement positif, alors on exécute en séquence le corps de l'instruction suivi de l'exécution de $Repeat^*$ obtenue en décrémentant son compteur.

$$\frac{n > 0 \quad Seq(t.Repeat^*(n-1, t)), E \xrightarrow{<\alpha,A>} u, E'}{Repeat^*(n, t), E \xrightarrow{<\alpha,A>} u, E'} \tag{3.11}$$

3.5.7 Contrôle booléen

L'instruction *If* permet d'exécuter un programme (le terme t) ou un autre programme (le terme u) en fonction de l'évaluation d'un wrapper de booléen. Si l'évaluation renvoie *true*, alors *If* se comporte comme le terme t :

$$\frac{eval(B,E) =< true, E' > \quad t, E' \xrightarrow{<\alpha,A>} t', E''}{If(B,t,u), E \xrightarrow{<\alpha,A>} t', E''} \tag{3.12}$$

Dans le cas contraire, *If* se comporte comme le terme u.

$$\frac{eval(B,E) =< false, E' > \quad u, E' \xrightarrow{<\alpha,A>} u', E''}{If(B,t,u), E \xrightarrow{<\alpha,A>} u', E''} \tag{3.13}$$

3.6 Instructions événementielles

3.6.1 Génération d'événement

L'instruction *Generate* ajoute un événement à l'environnement pour l'instant courant en mettant son booléen de présence à *true* et se termine instantanément. Cet événement est fonction de l'évaluation d'un wrapper d'événement.

$$\frac{eval(X,E) =< S, E' > \quad E'(S) =< -, V >}{Generate(X), E \xrightarrow{<TERM,\emptyset>} Nothing, E'[S \setminus < true, V >]} \tag{3.14}$$

L'instruction *Generate* peut aussi générer un événement valué. La valeur est placée dans l'environnement associé à l'événement S. Cette valeur est obtenue par l'intermédiaire d'un wrapper d'objet.

$$\frac{eval(X,E) =< S, E' > \quad eval(O,E') =< o, E'' > \quad E''(S) =< -, V >}{Generate(X,O), E \xrightarrow{<TERM,\emptyset>} Nothing, E''[S' \setminus < true, V.o >]} \tag{3.15}$$

Remarque : il faut noter que l'on ne fait pas de copie profonde de l'objet ajouté à la liste des valeurs. Donc cet objet est soumis aux effets de bord du langage hôte.

3.6.2 Attente d'événements

L'instruction *Await* bloque l'exécution d'une séquence d'instructions jusqu'à satisfaction de sa configuration événementielle. *Await* évalue d'abord sa configuration de wrappers puis continue son exécution en utilisant sa forme auxiliaire $Await^*$.

$$\frac{eval(CX, E) =< C, E' > \qquad Await^*(C), E' \xrightarrow{<\alpha,A>} u, E''}{Await(CX), E \xrightarrow{<\alpha,A>} u, E''} \tag{3.16}$$

Puisqu'on ne teste que des présences d'événements, si la configuration est satisfaite, cela signifie que la fin d'instant n'a pas été déclarée, donc $Await^*$ termine instantanément en renvoyant *TERM*.

$$\frac{sat(C, E) = true}{Await^*(C), E \xrightarrow{<TERM,\emptyset>} Nothing, E} \tag{3.17}$$

Si, par contre, la configuration n'est pas satisfaite ou si elle n'est pas encore déterminée, alors $Await^*$ se met en attente de la satisfaction de cette configuration en renvoyant le statut *LONGWAIT* et sa configuration d'événements. Cette attente peut durer plusieurs instants.

$$\frac{sat(C, E) \neq true}{Await^*(C), E \xrightarrow{<LONGWAIT,\{C\}>} Await^*(C), E} \tag{3.18}$$

3.6.3 Contrôle événementiel

L'instruction *Control* exécute son corps uniquement aux instants où la configuration de contrôle est satisfaite. L'instruction commence d'abord par évaluer sa configuration de wrappers et utilise sa forme auxiliaire $Control^*$.

$$\frac{eval(CX, E) =< C, E' > \qquad Control^*(C, t), E' \xrightarrow{<\alpha,A>} u, E''}{Control(CX, t), E \xrightarrow{<\alpha,A>} u, E''} \tag{3.19}$$

Si la configuration d'événements est satisfaite, alors $Control^*$ exécute son corps.

$$\frac{sat(C, E) = true \quad t, E \xrightarrow{<\alpha,A>} t', E'}{Control^*(C, t), E \xrightarrow{<\alpha,A>} Control^*(C, t'), E'} \tag{3.20}$$

Par contre, si la configuration d'événements n'est pas satisfaite, on se met en attente inter-instant sur la configuration C en renvoyant le statut *LONGWAIT*. De la même manière que pour $Await^*$, cette attente peut durer plusieurs instants.

$$\frac{sat(C, E) \neq true}{Control^*(C, t), E \xrightarrow{<LONGWAIT,\{C\}>} Control^*(C, t), E} \tag{3.21}$$

3.6.4 Condition événementielle

À la différence des deux instructions précédentes, *When* s'intéresse à la présence et à l'absence des événements. Cette instruction s'apparente à l'instruction *If* à la différence qu'une configuration événementielle est testée. Le test n'est pas forcément immédiat, car il faut peut être attendre la fin d'instant pour prendre la décision. *When* évalue d'abord sa configuration de wrappers puis utilise sa forme auxiliaire $When^*$.

$$\frac{eval(CX, E) =< C, E' > \qquad When^*(C, t, u), E' \xrightarrow{<\alpha,A>} v, E''}{When(CX, t, u), E \xrightarrow{<\alpha,A>} v, E''} \tag{3.22}$$

Si la configuration d'événements est satisfaite, alors $When^*$ se comporte comme t.

$$\frac{sat(C, E) = true \quad t, E \xrightarrow{<\alpha,A>} t', E'}{When^*(C, t, u), E \xrightarrow{<\alpha,A>} t', E'} \tag{3.23}$$

Si, par contre la configuration n'est pas satisfaite, alors $When^*$ renvoie $STOP$, car on est sûr d'être à la fin de l'instant. À l'instant suivant, le terme u sera exécuté.

$$\frac{sat(C, E) = false}{When^*(C, t, u), E \xrightarrow{<STOP,\emptyset>} u, E} \tag{3.24}$$

Enfin, tant que la configuration d'événement n'est pas fixée, $When^*$ renvoie $WAIT$ et la configuration C. On renvoie $WAIT$ et non $LONGWAIT$, car $When^*$ devra être ré-exécuté au plus tard à la fin de l'instant pour décider de l'exécution du terme t ou du terme u.

$$\frac{sat(C, E) = \bot}{When^*(C, t, u), E \xrightarrow{<WAIT,\{C\}>} When^*(C, t, u), E} \tag{3.25}$$

3.6.5 Préemption

L'instruction *Until* permet de préempter un programme sur satisfaction d'une configuration événementielle. C'est une préemption faible, donc, si la configuration est satisfaite, il faut attendre la fin de l'exécution, pour l'instant courant, du programme à préempter.

Nous différencions ici *Until* qui gère une configuration de wrappers de $Until^*$ qui gère une configuration d'événements. Dans tous les cas, *Until* comme $Until^*$ se comportent comme le corps de l'instruction si celui-ci renvoie $TERM$ ou $WAIT$.

$$\frac{t, E \xrightarrow{<\alpha,A>} t', E' \quad \alpha \in \{TERM, WAIT\}}{Until(CX, t, u), E \xrightarrow{<\alpha,A>} Until(CX, t', u), E'} \tag{3.26}$$

$$\frac{t, E \xrightarrow{<\alpha, A>} t', E' \quad \alpha \in \{TERM, WAIT\}}{Until^*(C, t, u), E \xrightarrow{<\alpha, A>} Until^*(C, t', u), E'} \tag{3.27}$$

Par contre, si l'exécution du corps renvoie $STOP$ ou $LONGWAIT$, alors une différence entre $Until$ et $Until^*$ apparaît. À la différence de $Until^*$, $Until$ commence d'abord par évaluer sa configuration de wrappers. Ensuite, dans les deux cas, l'exécution continue en utilisant soit $Until^{STOP}$, soit $Until^{LONGWAIT}$ selon le statut retourné par le corps.

$$\frac{t, E \xrightarrow{<\alpha, A>} t', E' \quad \alpha \in \{STOP, LONGWAIT\} \quad eval(CX, E') =< C, E'' > \quad Until^{\alpha}(C, t', u, A), E'' \xrightarrow{<\beta, A'>} v, E'''}{Until(CX, t, u), E \xrightarrow{<\beta, A'>} v, E'''} \tag{3.28}$$

$$\frac{t, E \xrightarrow{<\alpha, A>} t', E' \quad \alpha \in \{STOP, LONGWAIT\} \quad Until^{\alpha}(C, t', u, A), E' \xrightarrow{<\beta, A'>} v, E''}{Until^*(C, t, u), E \xrightarrow{<\beta, A'>} v, E''} \tag{3.29}$$

Commençons d'abord par analyser les règles pour $Until^{STOP}$. $Until$ (ou $Until^*$) utilise cette forme auxiliaire quand l'exécution du corps est terminée pour l'instant courant. Elle permet de tester si la configuration événementielle est satisfaite. Si c'est le cas, $Until^{STOP}$ se comporte comme le handler de l'instruction.

$$\frac{sat(C, E) = true \quad u, E \xrightarrow{<\alpha, A'>} u', E'}{Until^{STOP}(C, t, u, A), E \xrightarrow{<\alpha, A'>} u', E'} \tag{3.30}$$

Si, la configuration n'est pas satisfaite et que la fin d'instant a été déclarée, alors l'instruction utilise la forme $Until^*$ pour reprendre l'exécution du corps à l'instant suivant.

$$\frac{sat(C, E) = false}{Until^{STOP}(C, t, u, A), E \xrightarrow{<STOP, \emptyset>} Until^*(C, t, u), E} \tag{3.31}$$

Enfin, tant que la configuration d'événements n'est pas fixée, l'instruction se met en attente jusqu'à la fin de l'instant en renvoyant $WAIT$ et sa configuration.

$$\frac{sat(C, E) = \bot}{Until^{STOP}(C, t, u, A), E \xrightarrow{<WAIT, \{C\}>} Until^{STOP}(C, t, u, A), E} \tag{3.32}$$

Remarque : on est sûr que l'ensemble A dans toutes les règles de $Until^{STOP}$ est vide, car on ne peut être dans $Until^{STOP}$ que si le corps a renvoyé un statut $STOP$, c'est-à-dire que son exécution est finie pour l'instant courant et donc qu'on n'est plus en attente d'événements.

Voyons maintenant la forme auxiliaire $Until^{LONGWAIT}$. *Until* (ou $Until^*$) utilise cette forme auxiliaire quand l'exécution du corps renvoie le statut $LONGWAIT$, lorsque le corps est en attente inter-instant. Il faut alors un moyen pour vérifier si la configuration événementielle est satisfaite tout en laissant la possibilité au corps de se ré-exécuter si un événement sur lequel ce corps est en attente arrive.

Si la fin d'instant a été déclarée et que la configuration est satisfaite, l'instruction peut préempter son corps puisqu'à ce moment, aucun événement ne peut plus être généré. À l'instant suivant, on passera à l'exécution du handler u.

$$\frac{check(A,E) = false \quad sat(C,E) = true \quad E(eoi) = true}{Until^{LONGWAIT}(C,t,u,A), E \xrightarrow{<STOP,\emptyset>} u, E} \tag{3.33}$$

Par contre, si la configuration est satisfaite et que la fin d'instant n'a pas été déclarée, il faut attendre la fin d'instant pour pouvoir préempter le corps, donc $Until^{LONGWAIT}$ se met en attente sur les configurations bloquant l'exécution de son corps jusqu'à la fin de l'instant.

$$\frac{check(A,E) = false \quad sat(C,E) = true \quad E(eoi) = false}{Until^{LONGWAIT}(C,t,u,A), E \xrightarrow{<WAIT,A>} Until^{LONGWAIT}(C,t,u,A), E} \tag{3.34}$$

Si la configuration n'est pas satisfaite, alors $Until^{LONGWAIT}$ peut se mettre en attente inter-instant, sur l'ensemble des configurations événementielles bloquant son corps ainsi que sur sa configuration événementielle.

$$\frac{check(A,E) = false \quad sat(C,E) \neq true}{Until^{LONGWAIT}(C,t,u,A), E \xrightarrow{<LONGWAIT,A\cup\{C\}>} Until^{LONGWAIT}(C,t,u,A), E} \tag{3.35}$$

Enfin, $Until^{LONGWAIT}$ se comporte comme $Until^*$ si une des configurations bloquant l'exécution du corps est satisfaite.

$$\frac{check(A,E) = true \quad Until^*(C,t,u), E \xrightarrow{<\alpha,A'>} v, E'}{Until^{LONGWAIT}(C,t,u,A), E \xrightarrow{<\alpha,A'>} v, E'} \tag{3.36}$$

Remarque : il est possible de passer plusieurs fois au cours du même instant de l'état $Until^{LONGWAIT}$ à l'état $Until^*$ et vice-versa.

3.6.6 Événements locaux

L'instruction *Local* déclare un événement local S pour l'exécution du corps de l'instruction. Elle consiste à :

- remplacer les données de S contenues dans l'environnement par celles de l'événement S local,
- exécuter le corps de l'instruction dans ce nouvel environnement,
- récupérer les données de l'événement local et les remplacer par celles qui étaient initialement dans l'environnement.

Nous introduisons une fonction appelée *swap* qui effectue le "changement de contexte" pour l'événement S définie par :

$$\begin{aligned} swap(S,b,V,E) \quad \equiv \quad & < b', V', E' > \\ & \text{avec } E(S) =< b', V' > \\ & \text{et } E' = E[S' \setminus < b, V >] \end{aligned}$$

Si l'exécution du corps de *Local* est terminée, définitivement ou pour l'instant, alors on réinitialise le statut de l'événement local S pour l'instant suivant et le booléen de présence est mis à *false*.

$$\frac{\begin{array}{c} swap(S,b,V,E) =< b', V', E' > \quad t, E' \xrightarrow{<\alpha,A>} t', E'' \quad \alpha \in \{TERM, STOP\} \\ swap(S,b',V',E'') =< b'', V'', E''' > \quad E(eoi) = false \end{array}}{Local(S,b,V,t), E \xrightarrow{<\alpha,\emptyset>} Local(S, false, \varepsilon, t'), E'''} \tag{3.37}$$

Si, par contre, le corps se met en attente d'événement, alors l'instruction *Local* conserve l'état de l'événement et renvoie *WAIT* pour être ré-exécutée au plus tard à la fin de l'instant.

$$\frac{\begin{array}{c} swap(S,b,V,E) =< b', V', E' > \quad t, E' \xrightarrow{<\alpha,A>} t', E'' \quad \alpha \in \{WAIT, LONGWAIT\} \\ swap(S,b',V',E'') =< b'', V'', E''' > \quad E(eoi) = false \end{array}}{Local(S,b,V,t), E \xrightarrow{<WAIT, A\setminus\{S\}>} Local(S, b'', V'', t'), E'''} \tag{3.38}$$

Enfin, si la fin d'instant a été déclarée, alors on réinitialise dans tous les cas le statut de l'événement local pour l'instant suivant.

$$\frac{\begin{array}{c} swap(S,b,V,E) =< b', V', E' > \quad t, E' \xrightarrow{<\alpha,A>} t', E'' \\ swap(S,b',V',E'') =< b'', V'', E''' > \quad E(eoi) = true \end{array}}{Local(S,b,V,t), E \xrightarrow{<\alpha, A\setminus\{S\}>} Local(S, false, t'), E'''} \tag{3.39}$$

Remarque : Si, à la fin de l'instant, l'exécution du corps de l'instruction *Local* renvoie $< LONGWAIT, \{S\} >$, alors l'instruction ne sera plus exécutée puisqu'elle sera en attente inter-instant sur un ensemble vide d'événement. On peut alors considérer l'instruction *Local* comme un code mort.

3.7 Ordonnancement : Par N-aire

À la différence de *Rewrite*, *Replace* et *Storm*, l'instruction *Par*, en *Reflex*, est N-aire et non binaire. Cela permet d'accéder directement aux instructions qui ont besoin de l'être et de ne pas traiter les instructions en attente d'événements. Elle est constituée de 4 files :

- R, la file des programmes à exécuter au cours de l'instant courant
- W, la file des programmes dont l'exécution a retourné le statut *WAIT* au cours de l'instant courant ; tous les éléments de cette file sont des couples de la forme $< t,\ A >$ avec t le programme et A l'ensemble des événements sur lequel le programme est bloqué
- LW, la file des programmes dont l'exécution a retourné le statut *LONGWAIT*. Ces éléments sont de la même forme que ceux de la file W
- S, la file des programmes dont l'exécution a retourné le statut *STOP* au cours de l'instant courant

3.7.1 Fonctions auxiliaires

Nous avons ajouté des fonctions auxiliaires pour gérer l'ensemble des déplacements entre ces files.

La fonction *extract*$(E,\ L)$ gère les files d'attente sur événement. Elle récupère les instructions en attente d'événements contenues dans L qui peuvent être ré-exécutées. Elle crée, à partir d'une file de type $< t,\ A >$, 2 files :

- la première est une file de termes extraite de la file initiale. Elle contient tous les termes dont une des configurations sur lesquels le terme était en attente est satisfaite.
- la seconde contient le complémentaire, c'est-à-dire les éléments de la file initiale qui n'ont pas été extraits car aucune de leurs configurations attendues n'a été satisfaite.

$$\begin{aligned} \mathit{extract}(E,\ \varepsilon) &\equiv < \varepsilon,\ \varepsilon > \\ \mathit{extract}(E, < t, A > .L) &\equiv < L_1, < t, A > .L_2 >, \text{si } \mathit{check}(A, E) = \mathit{false} \\ &\quad < t.L_1, L_2 >, \text{si } \mathit{check}(A, E) = \mathit{true} \\ &\quad \text{avec } < L_1,\ L_2 >= \mathit{extract}(E, L) \end{aligned}$$

La fonction suivante *free_waitings* applique la fonction *extract* sur les 2 files des instructions en attente d'événements W et LW et récupère les instructions bloquées qui peuvent à nouveau être exécutées. Ces instructions seront ajoutées à la fin de la file des instructions à exécuter au cours de l'instant.

$$\begin{aligned} \mathit{free\_waitings}(E, R, W, LW) &\equiv < R.freed_1.freed_2, blocked_1, blocked_2 > \\ &\quad \text{avec } < freed_1, blocked_1 >= \mathit{extract}(E, W) \\ &\quad\quad < freed_2, blocked_2 >= \mathit{extract}(E, LW) \end{aligned}$$

La fonction *update* modifie les files de l'instruction *Par* en fonction des modifications apportées par l'exécution d'un terme. En paramètre de cette fonction, on a le résultat de l'exécution du terme avec son statut de retour, l'ensemble des configurations bloquant l'exécution du terme, l'environnement résultant de cette exécution et les quatre files de l'instruction parallèle. Cette fonction utilise *free_waitings* pour libérer les instructions en attente d'événements qui ont été générées pendant l'exécution du terme, puis ajoute le résultat de

l'exécution d'un terme dans la file appropriée en fonction de son statut de retour.

$$\begin{aligned} update(t,\alpha,A,E,R,W,LW,S) \equiv\ & Par(R',W'.<t,\ A>,LW',S), \text{ si } \alpha = WAIT \\ & Par(R',W',LW'.<t,\ A>,S), \text{ si } \alpha = LONGWAIT \\ & Par(R',W',LW',S.t), \text{ si } \alpha = STOP \\ & Par(R',W',LW',S), \text{ si } \alpha = TERM \\ & \quad \text{avec } <R',W',LW'> = free\_waitings(E,R,W,LW) \end{aligned}$$

Enfin, la fonction *set* récupère l'ensemble des configurations bloquant l'exécution des instructions contenues dans les files W ou LW.

$$\begin{aligned} set(\varepsilon) &\equiv \emptyset \\ set(<t,\ A>.L) &\equiv A \cup set(L) \end{aligned}$$

3.7.2 Les règles

Le principe général de l'instruction *Par* est :

- quand $E(eoi) = false$, d'exécuter les instructions de la file R tant qu'elle n'est pas vide.
- quand $E(eoi) = true$, d'exécuter les instructions de la file W tant qu'elle n'est pas vide.

Détaillons les différents cas de figures possibles. Dans le cas où toutes les files sont vides, l'instruction n'a plus rien à exécuter, termine instantanément et renvoie *TERM*.

$$Par(\varepsilon,\varepsilon,\varepsilon,\varepsilon),E \xrightarrow{<TERM,\emptyset>} Nothing,E \tag{3.40}$$

Des éléments uniquement dans la file S

S'il y a que des éléments dans la file des instructions terminées pour l'instant courant, *Par* renvoie le statut *STOP* et place toutes les instructions de S dans la file R des instructions à exécuter à l'instant suivant.

$$Par(\varepsilon,\varepsilon,\varepsilon,t.S),E \xrightarrow{<STOP,\emptyset>} Par(t.S,\varepsilon,\varepsilon,\varepsilon),E \tag{3.41}$$

Des éléments uniquement dans la file LW

Dans ce cas, il y a deux règles. Si aucune des configurations bloquant les instructions de LW n'est satisfaite, alors l'exécution de *Par* renvoie le statut *LONGWAIT* et l'ensemble des configurations bloquant l'exécution des instructions de *Par*.

$$\frac{free\_waitings(E,\varepsilon,\varepsilon,<t,A>.LW) =< R',-,LW' > \qquad R' = \varepsilon}{Par(\varepsilon,\varepsilon,<t,A>.LW,\varepsilon),E \xrightarrow{<LONGWAIT,set(LW')>} Par(\varepsilon,\varepsilon,LW',\varepsilon),E} \tag{3.42}$$

Si, au contraire, certaines configurations sont satisfaites, alors on libère les instructions dont les configurations sont satisfaites, on les place dans la file R des instructions à exécuter au cours de l'instant et on reprend en exécutant immédiatement les instructions de la file R.

$$\frac{\mathit{free\_waitings}(E, \varepsilon, \varepsilon, < t, A > .LW) =< R', W', LW' > \quad R' \neq \varepsilon \quad Par(R', W', LW', \varepsilon), E \xrightarrow{<\alpha, A'>} u, E'}{Par(\varepsilon, \varepsilon, < t, A > .LW, \varepsilon), E \xrightarrow{<\alpha, A'>} u, E'} \tag{3.43}$$

Des éléments uniquement dans les files LW et S

Dans ce cas de figure, il y trois règles applicables. La première s'applique lorsque la fin d'instant a été déclarée. Il n'est donc plus possible aux instructions contenues dans la file des attentes inter-instants de pouvoir progresser, car si elles pouvaient progresser, cela aurait été fait avant la fin de l'instant. La réécriture de Par consiste donc en un déplacement des éléments de la file S vers la file R, suivi d'un renvoi du statut $STOP$, car des instructions devront être exécutées à l'instant suivant.

$$\frac{E(eoi) = true}{Par(\varepsilon, \varepsilon, < t, A > .LW, u.S), E \xrightarrow{<STOP, \emptyset>} Par(u.S, \varepsilon, < t, A > .LW, \varepsilon), E} \tag{3.44}$$

Dans le cas où la fin d'instant n'a pas encore été déclarée, on vérifie qu'aucune des instructions bloquées dans la file LW ne peut être libérée. Si c'est le cas, alors on renvoie le statut $WAIT$, car il faudra revenir au plus tard à la fin de l'instant pour mettre les éléments de la file S dans la file R.

$$\frac{E(eoi) = false \quad \mathit{free\_waitings}(E, \varepsilon, \varepsilon, < t, A > .LW) =< R', -, LW' > \quad R' = \varepsilon}{Par(\varepsilon, \varepsilon, < t, A > .LW, u.S), E \xrightarrow{<WAIT, set(LW')>} Par(\varepsilon, \varepsilon, LW', u.S), E} \tag{3.45}$$

Si des instructions bloquées dans LW peuvent être libérées, on les place dans la file R et on les exécute immédiatement.

$$\frac{E(eoi) = false \quad \mathit{free\_waitings}(E, \varepsilon, \varepsilon, < t, A > .LW) =< R', -, LW' > \quad R' \neq \varepsilon \quad Par(R', \varepsilon, LW', u.S), E \xrightarrow{<\alpha, A'>} v, E'}{Par(\varepsilon, \varepsilon, < t, A > .LW, u.S), E \xrightarrow{<\alpha, A'>} v, E'} \tag{3.46}$$

Des éléments dans la file W et pas dans la file R

Dans ce cas, il y a de nouveau trois règles applicables. La première s'applique lorsque la fin d'instant a été déclarée. Il faut alors ré-exécuter l'ensemble des instructions de la file W

qui étaient en attente soit sur certaines configurations, soit sur la fin d'instant. Cette règle est appliquée jusqu'à ce que la file W soit vide.

$$\frac{E(eoi) = true \quad t, E \xrightarrow{<\beta,A'>} t', E' \quad update(t', \beta, A', E', R, W, LW, S), E' \xrightarrow{<\alpha,A''>} u, E''}{Par(\varepsilon, < t, \ A > .W, LW, S), E \xrightarrow{<\alpha,A''>} u, E''} \tag{3.47}$$

Si la fin d'instant n'a pas été déclarée, alors il faut tester si des instructions de la file W ou LW peuvent être libérées. Si aucune ne peut être libérée, alors l'exécution de *Par* renvoie *WAIT* et l'ensemble des configurations attendues par les instructions des files W et LW.

$$\frac{E(eoi) = false \quad free\_waitings(E, \varepsilon, < t, A > .W, LW) =< R', W', LW' > \quad R' = \varepsilon}{Par(\varepsilon, < t, A > .W, LW, S), E \xrightarrow{<WAIT, set(W') \cup set(LW')>} Par(\varepsilon, W', LW', S), E} \tag{3.48}$$

Si, par contre, des instructions peuvent être libérées, alors on les place dans la file R et elles sont immédiatement exécutées.

$$\frac{E(eoi) = false \quad free\_waitings(E, \varepsilon, < t, A > .W, LW) =< R', W', LW' > \quad R' \neq \varepsilon \quad Par(R', W', LW', S), E \xrightarrow{<\alpha,A'>} u, E'}{Par(\varepsilon, < t, A > .W, LW, S), E \xrightarrow{<\alpha,A'>} u, E'} \tag{3.49}$$

Des éléments dans la file R

Dans ce cas, les instructions de la première file sont exécutées jusqu'à ce que cette file soit vide. L'instruction *Par* étant non déterministe, nous ne précisons pas l'ordre dans lequel ces instructions sont exécutées. Cela se traduit en prenant au hasard une des instructions de la file R (décomposée en $R_1.t.R_2$).

$$\frac{E(eoi) = false \quad t, E \xrightarrow{<\beta,A>} t', E' \quad update(t', \beta, A, E', R_1.R_2, W, LW, S), E' \xrightarrow{<\alpha,A'>} u, E''}{Par(R_1.t.R_2, W, LW, S), E \xrightarrow{<\alpha,A'>} u, E''} \tag{3.50}$$

3.8 Machine réactive

À l'inverse des règles précédentes, les règles pour *Machine* ne renvoient ni statut de terminaison, ni ensemble de configurations.

Il y a deux règles applicables pour *Machine*. Si l'exécution du corps renvoie un statut différent de *WAIT*, alors l'instant est terminé. Dans le cas d'un statut *TERM* ou *STOP*, cela signifie qu'il n'y a plus aucune instruction à faire progresser pour l'instant courant. Dans le

cas d'un statut *LONGWAIT*, cela signifie que la seule manière de progresser est de générer un des événements attendus. Puisque aucune autre instruction ne peut être exécutée, l'instant est donc terminé.

$$\frac{t, E[eoi \setminus false] \xrightarrow{<\alpha,->} t', E' \quad \alpha \neq WAIT}{Machine(t), E \longrightarrow Machine(t'), E'[eoi \setminus false]} \tag{3.51}$$

Remarque : si le statut renvoyé est *LONGWAIT*, le terme t' ne pourra pas non plus progresser à l'instant suivant, sauf si de nouvelles instructions sont ajoutées ou si l'on considère des générations externes d'événements.

Par contre, si l'exécution du corps renvoie *WAIT*, alors *Machine* déclare la fin d'instant et ré-exécute le terme t'. Cette activation supplémentaire permet aux instructions qui ont renvoyé *WAIT* de décider de l'absence d'un événement et donc de ce qu'elles vont exécuter au prochain instant. Il ne peut plus y avoir de changement dans l'environnement.

$$\frac{t, E[eoi \setminus false] \xrightarrow{<WAIT,->} t', E' \quad t', E'[eoi \setminus false] \xrightarrow{<-,->} u, E''}{Machine(t), E \longrightarrow u, E''} \tag{3.52}$$

3.9 Ajout pour Icobjs

Dans cette partie, nous allons présenter la sémantique d'instructions nécessaires à l'utilisation des *Icobjs*.

3.9.1 Préemption *régulière*

Kill garantit que le handler de l'instruction sera toujours exécuté à l'instant suivant celui de la préemption. Les règles de sémantique de *Kill* sont les mêmes que celles d'*Until* à l'exception de la règle 3.30 qui est remplacée par celle-ci.

$$\frac{sat(C, E) = true}{Kill^{STOP}(C, t, u, A), E \xrightarrow{<STOP,\emptyset>} u, E} \tag{3.53}$$

3.9.2 Scanner

L'instruction *Scanner* exécute, pour chaque génération valuée d'un événement, un programme du langage hôte en lui passant en paramètre la valeur associée à l'événement. D'abord *Scanner* évalue le wrapper d'événement et continue son exécution en utilisant la forme auxiliaire $Scanner^*$. Pour cette forme auxiliaire, un compteur de la prochaine génération valuée de l'événement attendue est ajouté. Initialement, il est mis à 1.

$$\frac{eval(X, E) =< S, E' > \quad Scanner^*(S, a, 1), E' \xrightarrow{<\alpha,A>} u, E''}{Scanner(X, a), E \xrightarrow{<\alpha,A>} u, E''} \tag{3.54}$$

Il y a trois règles applicables à la forme auxiliaire *Scanner**. Dans le premier cas, si la fin d'instant a été déclarée, alors *Scanner** renvoie *STOP* et à l'instant suivant, il n'y a plus rien à faire.

$$\frac{E(eoi) = true}{Scanner^*(S,a,n), E \xrightarrow{<STOP,\emptyset>} Nothing, E} \tag{3.55}$$

Considérons maintenant les cas où la fin d'instant n'a pas encore été déclarée. Si la n-ième génération valuée de l'événement n'a pas encore eu lieu, alors *Scanner** se met en attente sur cette génération en renvoyant *WAIT*.

$$\frac{E(eoi) = false \quad sat(S_n, E) = false}{Scanner^*(S,a,n), E \xrightarrow{<WAIT,\{S_n\}>} Scanner^*(S,a,n), E} \tag{3.56}$$

Enfin, si cette n-ième génération a déjà eu lieu, alors on exécute un programme du langage hôte en lui passant en paramètre la valeur associée à cette n-ième génération. L'exécution de *Scanner** continue alors en incrémentant le compteur de la prochaine génération valuée de l'événement attendu.

$$\frac{E(eoi) = false \quad E(S) =< true, o_1 \dots o_n.V > \quad eval(a(o_n), E) =< -, E' > \quad Scanner^*(S,a,n+1), E' \xrightarrow{<\alpha,A>} u, E''}{Scanner^*(S,a,n), E \xrightarrow{<\alpha,A>} u, E''} \tag{3.57}$$

3.9.3 Run

L'instruction *Run* permet d'exécuter, à un niveau spécifique de l'arbre d'un programme, un terme non défini au moment de la création de ce programme. L'instruction *Run* évalue le wrapper de programme et exécute immédiatement le terme retourné.

$$\frac{eval(T,E) =< t, E' > \quad t, E' \xrightarrow{<\alpha,A>} t', E''}{Run(T), E \xrightarrow{<\alpha,A>} t', E''} \tag{3.58}$$

Il faut être très précautionneux dans l'utilisation de cette instruction qui exécute immédiatement un programme. Si, par exemple, le terme retourné est lui-même une instruction *Run* utilisant le même wrapper, on obtient alors une boucle infinie instantanée.

3.10 Bilan

Nous avons obtenu une sémantique à 58 règles. Le nombre de règles est assez important, mais cette sémantique définit les règles strictes du fonctionnement de *Reflex* en se rapprochant au plus près de l'implémentation. Par exemple, notre sémantique prend en compte l'évaluation

des wrappers. Cette sémantique synthétise en fait les deux systèmes de règles de *Storm* définis dans [70] : celui pour l'exécution descendante des instructions réactives et celui pour libérer les chemins en remontant dans l'arbre. Les principales caractéristiques de notre sémantique sont :

- la configuration événementielle *Not* a disparu. On ne peut réagir à l'absence d'un événement autrement qu'avec l'instruction *When*.
- l'instruction *Freezable* et l'instruction *Link* n'existent pas dans notre moteur. Elles sont remplacées par une instruction interne. L'environnement en est simplifié puisqu'il n'est plus nécessaire d'y stocker les résidus de comportements gelés par l'instruction *Freezable*.
- l'instruction *Par* est une instruction n-aire totalement non déterministe. La sémantique de cette instruction permet de n'exécuter que les instructions qui doivent l'être. De plus, à la différence des versions précédentes de *Junior*, la sémantique de cette instruction fait que celle-ci ne retourne un statut d'exécution que si elle n'a plus aucune branche à exécuter. À l'inverse de la sémantique d'origine qui stipule d'abord l'exécution d'une branche puis l'exécution de l'autre branche avant de retourner un statut en fonction du statut de chacune des branches, dans la sémantique de *Reflex*, tant qu'une branche peut encore être exécutée, elle le sera. Cela équivaut en *SugarCubes* à un programme `Close(Par(...))`. L'utilisation d'un tel algorithme associé à une instruction n-aire permet d'accéder directement, avec une complexité en O(1), à l'instruction qui a besoin d'être exécutée.
- le statut *LONGWAIT* a été rajouté à la sémantique et permet ainsi d'éviter les attentes actives d'événements inter-instants. De plus, le statut *SUSP* a été enlevé de la sémantique. En effet, ce dernier ne servait qu'à indiquer les instructions qui ont besoin d'être exécutées dans l'instant, ce qui était le cas des instructions événementielles. Or, maintenant, les instructions événementielles retournent soit le statut *WAIT*, soit le statut *LONGWAIT* si la configuration événementielle en jeu n'est pas satisfaite. Ce statut permettait aussi, dans *Storm*, de libérer des chemins dans les instructions *Par*. Or, dans notre sémantique, cela équivaut à un changement de liste dans cette même instruction.
- l'instruction *Control* utilise une configuration événementielle et non un événement seul.

L'efficacité d'une implémentation directe de notre sémantique est importante mais elle peut être améliorée. Comme nous l'avons annoncé dans l'introduction, nous avons simplifié notre sémantique en décrivant des règles comme en *Rewrite* où les instructions se réécrivent en de nouvelles instructions et non comme en *Replace* où les instructions conservent des états internes et peuvent être réinitialisées à la fin de l'exécution dans le cas des boucles. Dans l'implémentation, nous avons évidemment utilisé la méthode définie dans *Replace* comme par exemple, pour l'instruction *Stop* :

$$\begin{array}{rcl} Stop(false), E & \xrightarrow{<STOP,\emptyset>} & Stop(true), E \\ Stop(true), E & \xrightarrow{<TERM,\emptyset>} & Stop(true), E \\ reset(Stop), E & \equiv & Stop(false), E \end{array}$$

Outre le fait de passer de règles type *Rewrite* à des règles type *Replace*, le principe de tester l'ensemble des événements au niveau de l'instruction *Par* peut largement être optimisé par des méthodes de précurseurs comme il en existe en *Storm*. En effet, l'utilisation de ces

tests dans *Par* et le fait de remonter les configurations bloquant une branche d'exécution permet de représenter le système de règles de libération de chemins défini dans *Storm*.

Nous pouvons aussi noter qu'il peut y avoir des problèmes dans l'évaluation de wrappers ou dans l'exécution d'actions atomiques. Si, par exemple, l'action exécutée ne termine pas ou si une exception est générée pendant son exécution, alors il n'y a plus de réécritures possibles. Nous ne tenons pas compte de ces problèmes ici. Dans [2], R. Acosta propose, pour pallier à ce genre de problèmes, quelques pistes en introduisant de la qualité de service (*QoS*) dans *Junior*. Il propose par exemple l'introduction d'une nouvelle primitive *Try* pour intercepter les exceptions, ou des méthodes de limitation en temps pour l'exécution d'un programme. Ces méthodes font principalement appel à de la préemption forte, ce qui casse le modèle et rend donc le système instable. C'est la raison pour laquelle nous avons décidé de ne pas les introduire ici.

Chapitre 4

Implémentations

Dans ce chapitre, nous allons nous intéresser aux implémentations de *Junior*. Nous commencerons d'abord par présenter, dans la section 4.1, les caractéristiques de chacune des versions de *Junior* existantes à ce jour.

Pour réaliser l'implémentation d'un moteur correspondant à la sémantique présentée dans le chapitre précédent, nous nous sommes basés sur *Storm*, l'implémentation de *Junior* réalisée par *J-F. Susini* [70]. La version *Storm* était considérée comme étant moins rapide que *Simple*, mais avait l'avantage de posséder une sémantique formelle claire. En effet, la complexité des algorithmes utilisés dans *Simple* ne permet pas de garantir une concordance avec la sémantique de *Junior*. La section 4.2 présentera les diverses modifications apportées à l'implémentation de *Storm* pour implémenter notre sémantique, dont, entres autres, les modifications au niveau de la gestion événementielle et au niveau de l'interfaçage avec les *Icobjs*.

Enfin, dans la section 4.3, nous détaillerons les différences de performance entre *Reflex* et les versions *Storm*, *Simple* et *Glouton* de *Junior*.

4.1 Implémentations existantes

Dans cette partie, nous détaillerons plus particulièrement les versions *Rewrite*, *Replace* et *Storm* qui sont *de la même famille*. En effet, *Replace* est une amélioration de *Rewrite* en ce qui concerne la gestion des instructions et *Storm* est une amélioration de *Replace* au niveau de la gestion des événements. Concernant *Simple* et *Glouton*, nous présenterons uniquement les principes de base.

4.1.1 *Rewrite*

Rewrite est la première implémentation de *Junior*. Le but de cette version était d'abord d'obtenir une implémentation de référence pour vérifier rapidement l'exactitude des règles de réécritures de *Junior*. Pour cela, ces règles [39] ont été implémentées le plus fidèlement possible, c'est-à-dire que la structure des règles est reproduite le plus fidèlement possible dans les classes *Java*.

Chaque instruction est implémentée par une classe *Java* qui étend la classe `Instruction`. Cette classe ne définit qu'une méthode `rewrite` qui prend en paramètre l'environnement d'exécution. Cette méthode retourne un objet de type `MicroState` qui est une structure qui contient uniquement le statut retourné par l'exécution et le résidu de l'instruction exécutée.

```
abstract public class Instruction implements Program, java.io.Serializable
{
    abstract public MicroState rewrite(EnvironnementImpl env);
}
```

TAB. 4.1 – Structure d'une instruction en *Rewrite*

L'exemple de l'instruction `Control` présenté dans la table 4.2 permet de constater la fidèle concordance entre les règles de réécritures et l'implémentation.

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

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

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

```
public class Control extends UnaryInstruction
{
   ...

   public Control(Presence presence, Program body)
   { ... }

   public MicroState rewrite(EnvironmentImpl env)
   {
      if (presence.sat(env))
      {
         MicroState s = body.rewrite(env);
         return new MicroState(s.flag,
                     new Control(presence, s.term));
      }
      if(config.unsat(env))
         return new MicroState(Flags.STOP, this);
      return new MicroState(Flags.SUSP, this);
   }
}
```

TAB. 4.2 – Sémantique/Implémentation de l'instruction *Control* pour *Rewrite*

La classe `Control` n'étend pas directement `Instruction`, mais `UnaryInstruction` qui correspond à toutes les instructions qui contiennent un sous-terme. De la même manière, il existe une classe `BinaryInstruction` qui correspond à toutes les classes qui contiennent deux sous-termes comme, par exemple, l'instruction `If`.

L'environnement d'exécution est composé de :

- un booléen qui indique si la fin de l'instant a été déclarée.
- un booléen qui indique s'il y a eu des générations d'événements pendant une micro-étape.
- un entier qui représente l'instant courant. En effet, au lieu de vider à chaque début d'instant l'environnement, un compteur date l'instant. Lorsqu'un événement est généré à un instant donné, il est daté en utilisant la valeur de ce compteur.
- une référence vers un l'objet *Java* lié au programme en cours d'exécution.
- une table de hachage pour stocker l'ensemble des événements. Cette table associe, à chaque objet de type `Identifier`, un objet de type `EventData`. Un `EventData` contient un compteur indiquant le dernier instant où il a été mis à jour, un compteur indiquant le dernier instant auquel l'événement correspondant a été généré et deux tableaux contenant, l'ensemble des objets associées aux générations valuées de l'instant courant et de l'instant précédent. En fait, ces deux tableaux contiennent les valeurs correspondant

au dernier instant de mise à jour et à l'instant précédent celui-ci. Cela évite de devoir mettre à jour les tableaux de tous les événements enregistrés dans l'environnement à chaque instant.
- deux tables de hachage qui vont associer, à chaque événement qui a gelé un programme, le résidu du programme gelé. Il y a deux tables : l'une pour l'instant courant dans laquelle est stocké l'ensemble des programmes gelés dans l'instant et l'autre qui contient les programmes gelés à l'instant précédent. À chaque nouvel instant, ces deux tables sont interverties et celle devant contenir les programmes gelés pour l'instant suivant est vidée.

Les avantages d'une telle implémentation tiennent surtout dans le fait de pouvoir exécuter fidèlement les règles de sémantique. De plus, il est facile d'opérer des modifications pour tester rapidement des changements de sémantique. Cependant, le problème majeur de cette implémentation est son inefficacité :
- à chaque changement d'état d'une instruction, un nouvel objet est créé. Par exemple, dans l'instruction `Control` (cf. table 4.2), à chaque instant où l'événement est présent et donc où le corps de l'instruction est exécuté, un nouvel objet de type `Control` contenant le résidu de l'exécution du corps est créé. Il faut noter que le corps de l'instruction peut être exécuté plusieurs fois au cours du même instant et qu'un nouvel objet de type `Control` sera créé à chaque fois. La création d'objets intermédiaires est encore plus flagrante pour l'instruction `Loop` où le corps de la boucle est entièrement dupliqué pour le mettre en séquence avec la boucle elle-même.
- puisqu'il y a beaucoup de créations d'objets, il faut aussi tenir compte des destructions d'objets. En effet, une perte d'efficacité peut être constatée suite au traitement effectué à chaque passage du *Garbage Collector*.
- l'efficacité d'une exécution dépend également de la façon d'écrire le programme. Ceci sera présenté par l'intermédiaire de l'exemple de la cascade inverse (cf. figure 4.1).
- les instructions événementielles font de l'attente active sur les événements. À chaque micro-étape, tout l'arbre de programme est à nouveau entièrement parcouru et les instructions événementielles testent à chaque fois la satisfaction ou non de leur configuration événementielle. À chaque test, pour chaque événement attendu, l'instruction recherche dans la table de hachage si les événements composant sa configuration ont été générés pour l'instant courant.
- la table de hachage contenant les événements n'est jamais vidée. En terme de mémoire, il y a donc un clair gaspillage pour stocker des événements qui sont peut être *obsolètes*, c'est-à-dire qui ne seront plus utilisés par aucun programme.

Cascade inverse

L'exécution d'un programme en *Rewrite* est plus ou moins efficace selon la manière dont ce programme est écrit. L'exemple le plus flagrant de cette dépendance est celui de la cascade inverse. La figure 4.1 représente la version à trois branches de cet exemple. L'instruction `Par` de *Rewrite* qui correspond à l'instruction `Merge` des *SugarCubes* est une instruction binaire exécutant dans l'ordre sa branche gauche puis sa branche droite. La cascade inverse est un programme où chaque composant parallèle est une séquence qui attend un événement émis par le composant suivant, suivie d'une génération de l'événement qui bloque la branche précédente.

```
Jr.Par(
   Jr.Seq(Jr.Await(e3),Jr.Print("fin")),
   Jr.Par(
      Jr.Seq(Jr.Await(e2),Jr.Generate(e3)),
      Jr.Par(
        Jr.Seq(Jr.Await(e1),Jr.Generate(e2)),
        Jr.Generate(e1))))
```

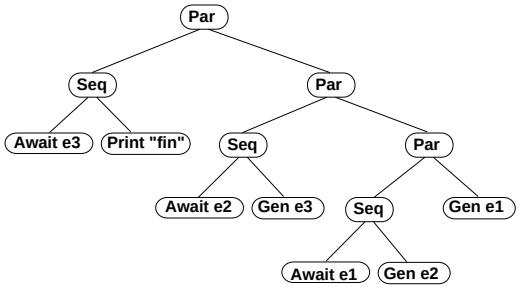


FIG. 4.1 – Cascade Inverse à 3 branches

L'exemple de la cascade inverse de la figure 4.1 nécessite quatre micro-étapes en utilisant *Rewrite*. Au cours des 3 premières, un événement `eN` est généré et à la dernière, le message `"fin"` est affiché. À chaque micro-étape, seule la dernière branche non-terminée est exécutée, mais par contre, toutes les configurations événementielles non-satisfaite sont testées. Chacun de ces tests correspond à une recherche supplémentaire dans la table de hachage. De plus, par construction, cette branche est toujours celle dont la profondeur est la plus grande. Cet exemple met en évidence l'inefficacité de *Rewrite*. Pour résoudre cette cascade inverse à 3 branches, *Rewrite* teste 4 fois la présence de l'événement `e3`, 3 fois celle de l'événement `e2` et deux fois celle de l'événement `e1`. Si les branches parallèles avaient été ordonnées dans le sens inverse (en cascade directe), *Rewrite* l'aurait résolu en une micro-étape où la présence de chaque événement n'aurait été testée qu'une seule fois. Cela met en évidence la dépendance de *Rewrite* envers la manière dont le code est écrit. Dans la section 4.3, nous présenterons, d'autres résultats concernant cet exemple.

4.1.2 *Replace*

L'objectif de la version *Replace* était, à partir de *Rewrite*, d'éliminer les créations inutiles d'objets à chaque exécution. L'état d'une instruction est désormais contenu dans l'instruction elle-même. La classe `Instruction` est alors la suivante :

```
abstract public class Instruction implements Program, java.io.Serializable
{
   public EnvironmentImpl env;

   abstract public void bind(EnvironmentImpl env);
   abstract public byte rewrite();
   abstract public void reset();
   abstract public Instruction residual();

   ...
}
```

TAB. 4.3 – Structure d'une instruction en *Replace*

Pour illustrer les modifications apportées, nous allons reprendre l'exemple de l'instruction `Control` dont la nouvelle classe est présentée dans la table 4.4. Les modifications sont les suivantes :

- Chaque instruction dispose d'une référence à l'environnement. Au moment où l'instruction est ajoutée à une machine réactive, cette référence est mise à jour par l'intermédiaire de la méthode `bind` qui propage cette référence à toutes les sous-instructions constituant le programme. Il n'est donc plus nécessaire de passer en paramètre l'environnement d'exécution à chaque appel à `rewrite`.
- La méthode `rewrite` ne renvoie plus la structure `MicroState` comprenant le statut retourné par l'exécution et le résidu du programme résultant de cette exécution, mais uniquement le statut. Les instructions ne sont pas recrées pour refléter leur état après une réécriture. Chaque instruction conserve son propre état. Ainsi, pour l'instruction `Control`, si l'événement est présent, l'instruction `Control` exécute son corps et retourne le statut retourné par cette exécution sans avoir à créer un nouvel objet `Control`. Autre exemple, un booléen a été ajouté à la classe de l'instruction binaire `Seq` signalant si la première branche a déjà terminée son exécution.
- Un mécanisme a été ajouté pour éviter à `Repeat` et à `Loop` de recréer, à chaque itération, les instructions composant leur corps, comme c'est le cas en *Rewrite*. À chaque fois que l'exécution du corps retourne *TERM*, les instructions composant le corps sont réinitialisées par l'appel à leur méthode `reset` avant d'être à nouveau exécutées. Par exemple, dans le cas de l'instruction `Control`, la méthode `reset` propage la réinitialisation à son corps et réinitialise sa propre configuration. La réinitialisation d'une configuration entraîne qu'au prochain test de satisfaction, tous les wrappers d'événements composant la configuration seront à nouveau évalués.
- Pour récupérer le résidu d'un programme gelé, la méthode `residual` a été ajoutée à chaque instruction. En *Rewrite*, le programme se simplifiait au fur et à mesure de l'exécution et de la terminaison des instructions. Donc, il suffisait de récupérer directement les instructions composant ce programme. En *Replace*, la structure d'un programme n'est pas modifiée durant l'exécution. Seul l'état de chaque instruction change. Or, il est inutile de récupérer des instructions définitivement terminées dans le résidu du programme. La méthode `residual` se charge de retourner, au moment du gel, une copie du programme qu'il reste à exécuter. Cette méthode retourne une copie du programme et non le programme lui-même car l'instruction de gel peut être encapsulée dans une boucle qui peut ré-exécuter son corps et donc ré-exécuter les instructions gelées.

Il résulte de ces modifications que *Replace* est bien plus efficace que *Rewrite*. En effet, toute la phase de création et de destruction d'objets inutiles est totalement éliminée. Par contre, *Replace* gaspille plus de mémoire que *Rewrite*. En effet, puisque la structure du programme chargé dans la machine n'est plus simplifiée durant l'exécution, des instructions définitivement terminées sont conservées. Tout comme en *Rewrite*, les problèmes de gestion d'événements (attente active, événements *obsolètes*,...) et les problèmes de dépendance envers la façon dont les programmes sont écrits sont toujours présent dans *Replace*.

4.1.3 *Simple*

La version *Simple*, au contraire de ce qu'indique son nom, est une implémentation basée sur des algorithmes bien plus complexes que ceux de *Rewrite* et *Replace*. Le but de cette implémentation, réalisée par Laurent Hazard de France Telecom R&D, était de pouvoir gérer, de manière efficace, un très grand nombre de composants parallèles et d'événements. Voilà les grandes lignes de son algorithme :

- la machine réactive est chargée avec un programme initial structuré en arbre d'instruc-

```
public class Control extends UnaryInstruction
{
   ...

   public void bind(EnvironmentImpl env)
   {
      this.env = env;
      body.bind(env);
   }
   public void reset()
   {
      super.reset();
      presence.reset();
   }
   public byte rewrite()
   {
      if (presence.sat(env))
         return body.rewrite();
      if(presence.unsat(env))
         return Flags.STOP;
      return Flags.SUSP;
   }
   public Instruction residual()
   {
      return new Control((Presence)presence.copy(), body.residual());
   }
   ...
}
```

TAB. 4.4 – Implémentation de l'instruction *Control* en *Replace*

tions comme avec *Replace* et *Rewrite*.

- la première activation, c'est-à-dire la première micro-étape, se déroule comme en *Replace*. Le programme est exécuté en descendant le long de l'arbre de syntaxe pour activer les instructions.
  - quand l'activation d'une instruction renvoie le statut *STOP*, l'instruction est enregistrée dans une file d'attente d'instructions à réactiver à l'instant suivant.
  - quand l'instruction se suspend sur l'attente d'un ou plusieurs événements, l'instruction est enregistrée dans des files d'attente correspondant chacune à un événement attendu. Au moment où un des événements est généré, toutes les instructions de la file correspondante, dont la configuration est satisfaite, sont placées dans une autre file dont tous les éléments seront exécutées dans l'instant. Si une instruction est en attente d'une absence d'événement, alors l'instruction est aussi placée dans une file dont tous les éléments seront réactivés à la fin de l'instant pour vérifier si leur configuration est satisfaite. Si c'est le cas, alors ces instructions sont placées dans la file des instructions à exécuter à l'instant suivant.
- après cette première activation, il ne reste que des listes d'instructions dans des files d'attente. Toutes les instructions `Par` de plus haut niveau, ont disparu. Pour exécuter un nouvel instant, il suffit de repartir des instructions contenues dans la file des instructions stoppées à l'instant précédent.
- il faut noter que si l'instruction enregistrée dans la liste est une feuille de l'arbre de programme, la réactivation de cette instruction consiste à remonter à son instruction parente et à continuer l'exécution de celle-ci. On a donc à la fois une exécution montante

et descendante.

Simple est basée sur des algorithmes efficaces qui n'exécutent que les instructions qui ont besoin de l'être. Il faut aussi noter que *Simple* a une politique pour retirer de la table d'événements ceux qui sont devenus *obsolètes*. La politique de nettoyage est assez simple, mais très coûteuse. En effet, la table d'événements est parcourue périodiquement dans son intégralité pour chercher et retirer tous les événements dont la file d'attente est vide.

Simple est une implémentation pragmatique de *Junior* et non une implémentation qui reprend la structure de la sémantique, ce qui la rend complexe à comprendre et surtout à modifier. De plus, le fait de casser la structure de l'arbre de syntaxe et d'exécuter les programmes à la fois de façon montante et descendante ne permet pas de garantir une correspondance à la sémantique de *Junior*. Par exemple, la sémantique d'une instruction `Par` spécifie que l'instruction est terminée dès que ses sous-instructions sont terminées. Puisque la structure de l'arbre est cassée, un mécanisme de synchronisation a dû être mis en place pour récupérer la fin de chacune des branches et ainsi terminer l'instruction `Par` qui les a exécuté. *Simple* dispose de plusieurs mécanismes pour coller au mieux à la sémantique, mais cela ne la rend que plus complexe à analyser, à comprendre et à modifier.

4.1.4 *Storm*

La version *Storm* est une étape intermédiaire entre *Replace* et *Simple*. En effet, elle garde la structuration en arbre du programme tout au long de l'exécution, comme *Replace*, mais elle utilise les mécanismes de file d'attente pour gérer les événements, comme *Simple*.

Voici les principales caractéristiques de *Storm* :

- l'exécution d'un programme est uniquement descendante comme en *Replace* et il n'y a aucune exécution remontante comme en *Simple*. *Storm* a sa propre sémantique dans laquelle sont introduites les différentes optimisations décrites dans cette partie.
- tout comme en *Rewrite* et *Replace*, l'instruction `Par` de *Storm* est l'instruction `Merge` des *SugarCubes*. Pour connaître l'état d'une branche parallèle, l'instruction `Par` dispose d'une variable qui conserve le statut de la dernière exécution de la branche concernée.
- une instruction qui est en attente d'un événement absent s'enregistre dans une file d'attente comme en *Simple* et retourne le statut *WAIT*, nouveau statut introduit dans *Storm*. Ce statut, déjà décrit dans la section 2.1.2, est une optimisation évitant de tester une configuration événementielle à chaque micro-étape durant un instant.
- au moment où un événement est généré, toutes les instructions enregistrées dans la file d'attente de l'événement sont réveillées par un mécanisme appelé *précurseur*. Il est chargé de libérer un chemin entre l'instruction réveillée et la racine de l'arbre. L'exécution d'une branche est bloquée par une ou plusieurs instructions `Par`. La libération d'un chemin entre l'instruction réveillée et la racine de l'arbre consiste donc à modifier l'état des instructions `Par` bloquant l'exécution de l'instruction en passant le statut de *WAIT* à *SUSP*.

Nous allons détailler les modifications qui ont été apportées à l'implémentation de *Replace*. La première modification est la mise en place du mécanisme de file d'attente. Pour cela, dans chaque objet de type `EventData`, une file de précurseurs (objet qui est de type `Zappable`) est rajoutée. Les précurseurs possibles sont en fait les instructions événementielles.

Ensuite, l'enregistrement d'un précurseur dans la file d'attente d'un événement doit être faite après avoir testé si l'événement concerné n'est pas présent. Ainsi, la méthode **`boolean fixed()`** définie dans la classe `Presence` est modifiée pour enregistrer, par l'intermédiaire

de la méthode `postPrecursor`, l'instruction qui effectue ce test dans la file correspondant à l'événement. Pour cela, les configurations événementielles sont chaînées à leur instruction en utilisant la méthode `bind` comme pour le chaînage des instructions entre elles lorsque le programme est ajouté à la machine. Chaque objet de type `Presence` d'une configuration connaît ainsi l'instruction à laquelle il est lié.

Enfin, pour réveiller les précurseurs, à chaque génération dans l'objet `EventData`, la file des instructions bloquées par l'événement concerné est parcourue pour réveiller celles dont la configuration événementielle est fixée. En effet, puisqu'il est possible que la configuration soit constituée de plusieurs événements, il n'est donc pas nécessaire de réveiller une instruction si sa configuration n'est pas complètement fixée. Pour effectuer ce test, la méthode `Config getConfig()` ajoutée par l'interface `Zappable` est utilisée. Au cas où la configuration est fixée, il faut pouvoir réveiller l'instruction et libérer le chemin à travers l'arbre du programme. C'est le rôle de la méthode `zap`, dont le code est montré dans la table 4.5, que toutes les instructions doivent définir.

```
abstract public class Instruction implements Program, java.io.Serializable
{
   public Environment env;

   abstract public void bind(Environment env);
   abstract public byte rewrite();
   abstract public void reset();
   abstract public Instruction residual();
   abstract public void zap(Instruction son);

   ...
}
```

TAB. 4.5 – Structure d'une instruction en *Storm*

La méthode `zap` est appelée de nœud en nœud en remontant le long de l'arbre jusqu'à arriver sur une instruction `Par` dont le statut de la deuxième branche est *SUSP*, c'est-à-dire exécutable dans l'instant. Durant la remontée, la seule modification apportée par cette méthode est de changer le statut de toutes les instructions `Par` rencontrées de *WAIT* à *SUSP*. Nous pouvons voir, dans la table 4.6, les modifications de la sémantique et du code de l'instruction `Control`.

Nous pouvons également remarquer que la sémantique de l'instruction `Control` n'a guère changé entre *Rewrite* et *Storm*. La seule différence est la troisième règle qui retourne le statut *WAIT* à la place du statut *SUSP*. Le paramètre `son` de l'instruction `zap` permet à l'instruction réveillée de vérifier que l'instruction qui l'a réveillée est toujours en attente de l'événement concerné. En effet, il n'y a pas de mécanisme de désabonnement d'événements en *Storm*. Seul un booléen dans l'objet `Presence` indique si l'instruction a déjà été enregistré.

Un autre mécanisme mis en place dans *Storm* est le balancement d'arbre qui modifie l'ensemble des instructions `Par` et `Seq`. Cette modification limite la profondeur maximale et rend l'exécution du programme moins dépendante de la façon dont il est écrit. La figure 4.2 montre l'effet du balancement de l'arbre de programme consistant en la combinaison parallèle des programmes `P1`, `P2`, `P3` et `P4`. Si les 3 premiers programmes étaient suspendus en attente d'un événement et que seul le quatrième peut être exécuté, l'exécution sera plus longue que

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

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

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

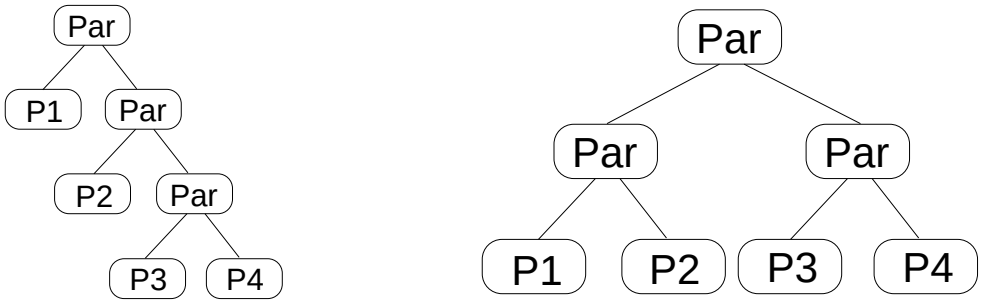
```
public class Control extends UnaryInstruction
                     implements Zappable
{
   ...

   public byte rewrite()
   {
      if (presence.sat())
         return body.rewrite();
      if (presence.unsat())
         return STOP;
      return WAIT;
   }
   public Config getConfig()
   {
      return presence;
   }
   public void zap(Instruction son)
   {
      parent.zap(this);
   }

   ...
}
```

TAB. 4.6 – Sémantique/Implémentation de l'instruction *Control* pour *Storm*

si l'on considérait le programme `P1` exécutable et les 3 autres en attente. Il en est de même pour les séquences. De plus, il faut noter que plus la profondeur d'une branche en attente d'un événement est grande, plus le mécanisme de remontée du précurseur est coûteux car le chemin à travers les instructions `Par` est potentiellement plus grand. Le balancement de l'arbre est effectué après chaque ajout de programme à la machine réactive.



(a) Avant balancement

(b) Après balancement

FIG. 4.2 – Effet du balancement

Les inconvénients de *Storm* sont une attente active inter-instant et le problème récurrent de dépendance de la façon dont les branches parallèles sont initialement ordonnancées. En effet, *Storm* évite de tester plusieurs fois la configuration d'une même instruction au cours d'un instant. Mais pour une instruction comme `Await` qui est bloquante tant que la configuration n'est pas satisfaite, ce qui peut durer plusieurs instants, il sera tout de même nécessaire de tester cette configuration à chaque instant. De plus, une configuration est testée au minimum

deux fois par instant si elle n'est pas satisfaite : au début et à la fin de l'instant.

Storm est moins dépendant de la façon dont le code est écrit que *Replace* et *Rewrite* à cause du balancement de l'arbre. Par contre, tout comme *Replace* et *Rewrite*, *Storm* utilise une instruction `Par` binaire totalement déterministe, ce qui empêche d'accéder immédiatement à une instruction qui a besoin de l'être. Pour l'exemple de la cascade inverse à 3 branches de la figure 4.1, *Storm* prend toujours 4 micro-étapes pour finir l'exécution du programme, mais excepté à la première micro-étape où toutes les branches parallèles sont parcourues, toutes les suivantes n'ont qu'une seule branche à exécuter. Le balancement de l'arbre permet de limiter sa profondeur, mais l'ordonnancement de l'exécution des branches parallèles est toujours le même.

4.1.5 *Glouton*

L'objectif de *Glouton*, réalisée par Louis Mandel [48], était de formaliser les mécanismes mis en place dans *Simple* en une sémantique claire et d'implémenter le plus directement possible cette sémantique. Les principales caractéristiques de *Glouton* sont :

- l'utilisation de deux classes par instruction. Initialement, le programme donné à la machine n'est pas exécutable, c'est un arbre de syntaxe abstraite (AST) qui décrit le comportement du programme. Quand le programme est chargé dans la machine, il est immédiatement traduit en instruction exécutable. À la différence de *Simple* qui casse la structure du programme au fur et à mesure de son activation, *Glouton* le fait au moment du chargement du programme.
- l'introduction de la notion de *groupe*. Un groupe est une structure qui contrôle un ensemble d'instructions et qui est l'interface avec l'environnement. Les nœuds de l'arbre du programme exécutable sont des groupes et non des instructions. Il y a plusieurs types de groupes :
  - le `Groupe` qui est la structure de base d'un groupe. Une séquence `Seq(p1,p2)` crée un groupe qui contient les instructions du programme `p1` et signale que la continuation est le programme exécutable retourné par `p2`. Une boucle est une séquence dont la continuation est son propre groupe. Un groupe est créé pour chacune des branches des instructions binaires (`Kill, If, When`).
  - le `GroupeKill` qui regroupe toutes les instructions et sous-groupes à préempter. Ce groupe enregistre sa configuration événementielle si elle n'est pas satisfaite. Dès que cette configuration est satisfaite, le groupe est enregistré dans celui de plus haut niveau pour être rappelé à la fin de l'instant et ainsi préempter son contenu et mettre le `handler` dans les instructions à exécuter à l'instant suivant.
  - le `GroupeLoc` qui regroupe toutes les instructions qui sont à exécuter dans le contexte d'un événement local. Une génération d'événement est effectuée directement dans son groupe qui génère à son tour l'événement dans le groupe local le plus proche du sien. Si l'événement généré ne correspond pas à l'événement local du groupe, la génération remonte le long de l'arbre, de groupe local en groupe local, jusqu'au groupe de plus haut niveau.
  - le `GroupeLink` qui regroupe toutes les instructions dont l'exécution doit être associée à un objet. Comme pour les événements, quand une instruction veut connaître l'objet qui lui est associé, elle interroge son groupe qui lui indique le `GroupeLink` le plus proche et ce dernier lui retourne l'objet concerné.
  - le `GroupeTop` qui est le groupe de plus haut niveau dans la machine réactive. Ce

groupe est particulier car il correspond aussi à l'environnement d'exécution. Il contient la table des événements et le booléen indiquant la fin de l'instant.

- la modification de la sémantique de `Until` pour en faire une instruction `Kill`. Après la traduction de l'AST en programme exécutable, cette instruction donne naissance à un groupe soumis à la préemption. Dès que sa configuration est satisfaite, l'instruction s'enregistre dans une file du groupe racine dédiée aux préemptions, pour être réveillée à la fin de l'instant pour réaliser cette préemption.
- la disparition de l'instruction `Control` dans le programme exécutable. Pendant la traduction de l'AST vers le programme exécutable, l'instruction `Control` est remplacée par un `Await` sur l'événement de contrôle. Ensuite, l'événement de contrôle est propagé dans toutes les branches du programme contrôlé pour modifier tous les points d'arrêt potentiels, c'est-à-dire les instructions `Stop` et les instructions événementielles. Toutes les instructions `Stop` sont automatiquement suivies par une instruction `Await` utilisant l'événement de contrôle. De même, les configurations événementielles des instructions soumises au contrôle sont remplacées par une conjonction de cette configuration et de l'événement de contrôle. Il y a une exception à cela, c'est l'instruction `When` dont l'évaluation se fait dans l'instant de son exécution. `When` est dans ce cas précédée par une instruction `Await` sur l'événement de contrôle.
- la fin de l'instant est déclarée à la fois par un booléen et par l'intermédiaire de la génération de l'événement `"eoi"`. Les configurations de toutes les instructions événementielles qui dépendent de la fin de l'instant, c'est-à-dire celles concernées par une absence d'événement et celles des instructions `When` sont modifiées. Elles sont remplacées par une conjonction composée de leur configuration initiale et d'une disjonction de l'événement `"eoi"` et d'un événement `"true"` généré à chaque début d'instant. Puisque la fin de l'instant est considérée comme un événement, *Glouton* dispose aussi d'une file dans laquelle toutes les instructions qui dépendent de la fin de l'instant seront enregistrées. Ces dernières seront réveillées pour être évaluées à la fin de l'instant.

L'implémentation actuelle de *Glouton* donne des performances comparables à celle de *Simple*. L'inconvénient de cette implémentation par rapport aux autres versions de *Junior* est qu'elle ne dispose pas encore de l'instruction `Freezable`. Un des problèmes posé par de cette instruction est le fait que le programme est modifié pour faire disparaître l'instruction de contrôle. Dans le cas d'une préemption simple, cela ne pose pas de problème, puisqu'il n'est pas nécessaire de récupérer le résidu du programme préempté. Par contre, dans le cas de `Freezable`, le résidu du programme gelé ne doit pas contenir les modifications liées à une instruction `Control` qui ne serait pas gelée. Un autre inconvénient est que *Glouton* ne dispose pas de mécanisme de nettoyage de la table d'événements.

Enfin, nous pouvons noter que, en *Glouton*, les générations externes d'événements dans une *safe-machine* sont uniquement prises en compte à l'instant suivant, contrairement aux autres implémentations de *Junior* qui prennent en compte ces générations au cours de l'instant si l'événement a été généré avant la fin de celui-ci.

4.2 Implémentation de *Reflex*

Pour réaliser l'implémentation de notre sémantique, nous partons de l'implémentation de *Storm* en lui appliquant des modifications qui se caractérisent par une réduction de la profondeur de l'arbre due au remplacement du format binaire des instructions `Par` et `Seq` par

leur format N-aire, par une meilleure gestion événementielle et par l'ajout de l'attente passive inter-instant.

4.2.1 Instructions N-aire

Dans les implémentations présentées précédemment, les instructions `Par` et `Seq` sont des instructions binaires. Pour les versions *Simple* et *Glouton*, il existe des instructions supplémentaires appelées `MultPar` et `MultSeq`. Les instructions `Par` disparaissent durant la première activation pour *Simple* ou durant l'ajout à la machine pour *Glouton* pour remplir les différentes files d'attente du système. En *Rewrite*, *Replace* et *Storm*, `Par` et `Seq` sont conservées pour garder la structure du programme, *Rewrite* se permettant de simplifier l'arbre au fur et à mesure que les différentes branches se terminent.

En *Reflex*, nous n'avons pas rajouté d'instructions supplémentaires comme *Simple* et *Glouton*, mais les instructions `Par` et `Seq` sont directement N-aires et manipulent des listes d'instructions comme le montre les règles de réécritures (cf. sections 3.7 et 3.5.4). Pour permettre de manipuler aisément ces listes, chaque instruction dispose d'un pointeur vers la liste qui la contient et de deux pointeurs vers les instructions précédentes et suivantes dans la liste, comme le montre la classe abstraite `Instruction` de la table 4.7.

```
abstract public class Instruction implements Program, Cloneable, Serializable
{
   protected transient InternEnvironment env;
   protected transient Instruction parent;

   protected Instruction next = null;
   protected Instruction previous = null;
   protected InstructionList list = null;

   abstract protected void bind(InternEnvironment env, Instruction parent, boolean removable);
   abstract protected byte rewrite();
   abstract protected void reset();

   abstract protected Instruction residual(boolean unregisterEvent);
   abstract protected void zap(Instruction son);

   ...
}
```

TAB. 4.7 – Structure d'une instruction en *Reflex*

La manière de construire des programmes reste exactement la même qu'en *Junior*. L'interface de programmation `Ic` dispose des mêmes méthodes (`Program Seq(Program p1, Program p2)` ou `Program Seq(Program[] p)`). Au moment de la construction, l'objet `Seq` va tester si les programmes qui lui sont passés sont eux-mêmes des objets de type `Seq`. Si c'est le cas, les listes de l'instruction `Seq` parent et des sous-instructions `Seq` sont concaténées. Il en est de même pour l'instruction `Par` avec des sous-instructions de type `Par`.

Le résultat du programme décrit dans la table 4.8 est une instruction `Par` unique contenant les programmes `P1, P2, P3, P4`. Si un de ces programmes débute par une instruction `Par`, la liste des branches de cette sous-instruction `Par` est absorbée par le `Par` de plus haut niveau. Il faut tout de même noter qu'en fusionnant les listes des instructions `Par` imbriquées,

```
Program p = Ic.Par(P1,
                   Ic.Par(P2,
                      Ic.Par(P3, P4)));
```

TAB. 4.8 – Programme écrit avec des instructions Par imbriquées

la sémantique du programme est légèrement modifiée (ce n'est pas le cas pour l'instruction `Seq`). Si l'on considère la sémantique de l'instruction `Par` (cf. section 3.7) de façon rigoureuse, la remontée des sous-instructions d'une instruction `Par` imbriquée dans l'instruction `Par` de niveau supérieur modifie l'ordonnancement des tâches. D'après la sémantique, une instruction `Par` ne retourne un statut que si toutes ses sous-instructions sont dans un état qui ne peut plus évoluer. En suivant la sémantique, l'exemple du code précédent exécute d'abord le programme `P1` ou la composition parallèle des programmes `P2` et la composition parallèle de `P3` et `P4`. Or, la remontée de ces sous-instructions les place au même niveau que les autres déjà contenues dans l'instruction `Par`. Il n'est alors plus nécessaire d'attendre qu'un sous-groupe d'instructions ne puisse plus évoluer pour exécuter les autres branches du `Par` de haut niveau. En faisant remonter ces quatre programmes, l'exécution des branches peut être ordonnancé dans n'importe quel ordre, comme, par exemple, `P2`, `P4`, `P1` et `P3`.

Séquence

La classe de l'instruction `Seq` contient la liste de tous les sous-programmes qui ne sont donc plus des séquences et `current` un pointeur sur la première instruction de la liste. Lorsque `Seq` est exécuté, l'instruction pointée par `current` est exécutée. Dès que le statut retourné est *TERM*, `current` est déplacé sur l'instruction suivante dans la liste. L'instruction `Seq` retourne *TERM* dès que `current` pointe sur **`null`**. Notons cependant que, à la différence de la sémantique, la liste ne se vide pas au fur et à mesure de la terminaison des programmes.

Parallèle

L'implémentation de l'instruction `Par` n'utilise pas quatre listes (R, W, LW et S) comme dans la sémantique (cf. section 3.7) mais cinq. Une liste supplémentaire permet de stocker les instructions qui ont terminées leur exécution. La liste n'était pas présente dans la sémantique, puisque nous avons décrit celle-ci comme fonctionnant "à la *Rewrite*" pour des raisons de simplification (les boucles sont représentées, de façon récursive, comme une séquence entre le corps de la boucle et la boucle elle-même, sans méthode de réinitialisation). Comme nous l'avons vu, la sémantique de notre instruction `Par` équivaut à un programme `Close(Par(...))` en *SugarCubes*, le but étant de finir localement l'exécution des sous-instructions de l'instruction `Par` avant de retourner le statut. Ceci permet d'effectuer beaucoup moins de micro-étapes globales et donc moins de parcours de l'arbre. L'algorithme défini dans la méthode **`byte`** `rewrite()` consiste alors en ceci :

- si la fin de l'instant n'a pas été déclarée, les instructions de la liste des instructions à exécuter au cours de l'instant sont exécutées tant que la liste n'est pas vide, dans l'ordre dans lequel elles sont enregistrées dans la liste (les instructions ne sont pas prises au hasard comme dans la sémantique). Selon le statut retourné par leur exécution, les instructions sont déplacées vers les listes correspondant au statut de retour. La

disparition du statut *SUSP* garantit qu'il n'est pas possible d'exécuter en boucle la même instruction.

- si la fin de l'instant a été déclarée, on fait de même sur la liste des instructions ayant retourné *WAIT*. Cette exécution ne retourne que des statuts *STOP* ou *LONGWAIT*, ce qui garantit que nous n'allons pas boucler indéfiniment.
- à la fin de cette dernière exécution, il ne peut y avoir des éléments que dans les listes des instructions ayant retourné *TERM*, *STOP* ou *LONGWAIT*. Il ne reste donc plus qu'à préparer l'instruction pour l'instant suivante c'est-à-dire de remplir la liste des instructions à exécuter avec les instructions de la liste des *STOP*. Pour cela, nous intervertissons simplement les références vers ces deux listes.

Le non-déterminisme de cette implémentation vient du fait que les instructions ne sont pas insérées dans les listes dans l'ordre dans lequel elles étaient enregistrées au départ. Pour la gestion des événements, nous n'implémentons pas directement la sémantique, ce qui serait trop coûteux, mais nous utilisons le mécanisme de *précurseur* défini dans *Storm*. L'appel à la méthode `zap` de l'instruction `Par` reçoit en paramètre l'instruction à réveiller. Puisque chaque instruction dispose d'un pointeur vers la liste dans laquelle elle est enregistrée, nous n'avons pas besoin de parcourir la liste des *WAIT* et la liste des *LONGWAIT* pour la retrouver. Ce pointeur nous permet de l'enlever immédiatement de la liste dans laquelle elle se trouve pour la placer à la fin de la liste des instructions à exécuter dans l'instant, comme nous le montre le code de la table 4.9.

```
protected void zap(Instruction son)
{
   if (son.list == waited)
   {
      waited.remove(son);
      if (suspended.first == null && parent != null)
         parent.zap(this);
      suspended.add(son);
   }
   else if (son.list == longwaited)
   {
      longwaited.remove(son);
      if (suspended.first == null && parent != null)
         parent.zap(this);
      suspended.add(son);
   }
}
```

TAB. 4.9 – Méthode zap de l'instruction *Par* en *Reflex*

Comme en *Rewrite*, il est intéressant de pouvoir réduire l'arbre au fur et à mesure que les différentes branches se terminent, ce qui permet de libérer la mémoire inutilement utilisée. Par contre, comme nous l'avons vu, quand ces instructions se trouvent dans une boucle, il est plus intéressant de les réutiliser en les réinitialisant simplement. Nous proposons ici un intermédiaire entre les deux solutions dans lequel nous simplifions l'arbre de programme uniquement dans les instructions `Par` si celles-ci ne sont pas contenues dans une instruction `Loop` ou `Repeat`. Pour cela, nous avons ajouté un paramètre supplémentaire à la méthode `bind` qui indique si l'instruction est contenue dans une instruction de boucle. Ce booléen est stocké par l'instruction `Par`. Quand l'exécution d'une branche d'une instruction `Par` retourne

TERM, cette branche est mise dans la file des programmes terminés selon la valeur du booléen. Cette méthode évite de garder, comme c'est le cas en *Replace* et *Storm*, tous les programmes qui ont déjà fini leur exécution.

Il faut aussi noter qu'une réinitialisation de l'instruction `Par` apparaissant dans une boucle ne remet pas l'instruction `Par` dans son état initial. En effet, elle ne conserve pas l'ordre initial des instructions, les instructions étant replacées dans l'ordre dans lequel elles se sont terminées. Pour précisément, les instructions sont replacées, au moment de la réinitialisation, dans la liste à exécuter au cours de l'instant dans l'ordre suivant :

- les instructions déjà terminées sont placées au début de la liste dans l'ordre dans lequel elles se sont terminées,
- si, avant la réinitialisation, il y avait encore des instructions à exécuter, elles sont concaténées à la suite des instructions terminées,
- enfin, s'il reste des instructions en attente d'événements dans la liste *LONGWAIT*, elles sont placées à la fin.

Après une réinitialisation, le fait d'ordonner les instructions dans l'ordre dans lequel elles se sont terminées, permet, d'une certaine manière, de les positionner dans un ordre qui peut être en général plus favorable à la prochaine exécution. Dans l'exemple de la cascade inverse, les branches parallèles de ce programme terminent dans l'ordre inverse dans lequel elles sont enregistrées dans l'instruction `Par`. Si ce programme est mis dans une boucle, après la première exécution, la réinitialisation le transformera en une cascade directe dont l'exécution est beaucoup moins complexe en terme de déplacement de programmes dans les différentes listes de l'instruction. Le cas de la cascade inverse est évidemment un cas extrême dont il ne faut pas faire une généralité et ce réordonnancement n'est pas nécessairement toujours le plus favorable. Il suffit, par exemple, que les branches parallèles doivent être ré-exécutées sur plusieurs instants pour que le bénéfice de l'ordonnancement des branches soit perdu.

L'utilisation d'instructions n-aire évite le problème de dépendance envers la façon dont le code est écrit. En effet, toutes les instructions, décrites comme devant être placées à la même profondeur dans l'arbre, le sont réellement. Le mécanisme de balancement de l'arbre de *Storm* devient alors inutile. L'arbre de programme obtenu est naturellement équilibré puisque tous les nœuds sont au même niveau. Cette caractéristique accélère les libérations de chemin dans l'arbre puisqu'il y a moins d'instructions à parcourir pour libérer un chemin. Elle accélère également les exécutions des instructions en évitant de perdre du temps dans le parcours de l'arbre.

Le fait d'avoir une instruction `Par` n-aire et indéterministe permet d'obtenir une complexité en O(1) pour l'accès aux branches parallèles qui ont besoin d'être exécutées. Le réveil d'instructions bloquées sur des attentes d'événements a aussi une complexité en O(1). On accède en effet directement à l'instruction bloquée sans avoir à parcourir les deux listes (*WAIT* et *LONGWAIT*) des instructions bloquées. Ainsi, le changement de liste peut être effectué immédiatement. Puisque l'ordre des instructions n'est pas conservé, il n'est pas nécessaire de faire des insertions coûteuses, et l'instruction peut être placée en fin de liste.

4.2.2 Gestion des événements

La gestion des événements est un point essentiel pour réaliser une implémentation efficace de *Junior*. En effet, pour améliorer l'efficacité, il faut, entre autres, diminuer le nombre de fois qu'une configuration est testée. Dans *Storm*, cela a conduit à l'ajout du statut *WAIT*.

Ce nouveau statut combiné à la remontée des précurseurs évite d'évaluer les configurations événementielles à chaque micro-étape, mais uniquement quand la configuration est satisfaite ou bien à la fin de l'instant. Ce mécanisme a permis à *Storm* d'être bien plus efficace que *Replace* et *Rewrite* et de ne pas réaliser d'attente active au cours d'un instant.

```
public class Control extends Zappable
{
   ...

   public Control(Configuration config, Program body){ ... }
   protected void reset()
   {
      body.reset();
      config.reset();
   }
   protected byte rewrite()
   {
      if (config.sat())
      {
         byte res = body.rewrite();
         if (res == Flags.TERM)
            config.reset();
         return res;
      }
      return Flags.LONGWAIT;
   }
   protected Instruction residual(boolean unregisterEvent)
   {
      if(unregisterEvent)
         config.reset();
      return new Control(config.copy(), body.residual());
   }
   protected void bind(InternEnvironment env, Instruction aParent, boolean removable)
   {
      super.bind(env, aParent, removable);
      body.bind(env, this, removable);
      config.bind(env, this);
   }
   ...
}
```

TAB. 4.10 – Implémentation de l'instruction *Control* pour *Reflex*

L'utilisation du statut *WAIT* est utile pour des instructions, comme par exemple `Scanner` ou `When`, qui sont intéressées à la fois par la présence et par l'absence des événements. Par contre, pour des instructions comme `Await` et `Control`, ce mécanisme est inefficace. En effet, ces deux instructions sont bloquantes tant que l'événement n'est pas présent, ce qui veut dire qu'en l'absence de l'événement attendu, chacune d'elles est évaluée deux fois par instant. Pour optimiser cela, nous avons rajouté un nouveau statut d'exécution, le statut *LONGWAIT* qui permet d'éviter une attente active inter-instant.

L'instruction `Control` (cf. table 4.10) utilise ce nouveau statut. De même, nous pouvons remarquer que `Control` manipule désormais une configuration événementielle et non une présence unique. De plus, l'instruction n'est plus intéressée par l'absence de l'événement ; soit la configuration est satisfaite et dans ce cas, nous sommes certain d'exécuter le corps immédiatement, soit le statut *LONGWAIT* est retourné.

Mécanisme d'abonnement et désabonnement des précurseurs

L'une des raisons pour lesquelles *Storm* faisait de l'attente active inter-instant est l'absence de mécanisme de désabonnement des précurseurs en attente d'un ou plusieurs événements. Nous allons d'abord commencer par décrire le mécanisme d'abonnement de *Reflex* qui est sensiblement le même que celui de *Storm*. L'ajout de précurseur est effectué par la méthode `fixed` de l'objet `Presence` (cf. table 4.11) si l'événement n'est pas présent ou s'il n'a pas déjà été enregistré. À la différence de *Storm*, au moment de l'enregistrement, l'objet de type `Presence` dans *Reflex* conserve une référence vers la cellule enregistrée dans la liste des précurseurs de l'événement concerné. Cette cellule permet d'accéder directement à la liste dans laquelle elle se trouve. Le retrait de cette liste est donc fait avec une complexité en O(1).

```
public class Presence extends Configuration
{
   ...
   private transient EventData eventData;
   transient PrecursorCell cell;

   protected void reset()
   {
      if (cell != null)
      {
         cell.remove();
         cell = null;
      }
      evaluated = false;
      eventData = null;
   }
   protected boolean fixed()
   {
      if (evaluated == false)
      {
         event = wrapper.evaluate(env);
         evaluated = true;
      }
      if (eventData == null || eventData.id == null || eventData.id.equals(event) == false)
         eventData = env.getEventData(event);
        boolean res = eventData.isGenerated(env.getInstant());
      if (!(res || cell != null))
         cell = eventData.postPrecursor(precursor, this);
      return res || env.eoi();
   }
   protected boolean eval()
   {
      if (eventData == null)
         return false;
      return eventData.isGenerated(env.getInstant());
   }
   ...
}
```

TAB. 4.11 – Implémentation de la configuration *Presence* en *Reflex*

Nous pouvons noter que nous avons ajouté deux champs à l'objet `Presence` : `cell` qui pointe vers la cellule de la liste des précurseurs et `eventData` qui pointe vers l'objet de type `EventData` concernant l'événement attendu. Ce deuxième pointeur permet d'être plus efficace en évitant de rechercher continuellement dans la table des événements.

Pour le désabonnement, il faut d'abord noter qu'une cellule contient une référence vers l'instruction à réveiller et une référence vers l'objet de type `Presence` à partir duquel cette instruction a été enregistrée. Ainsi, nous disposons d'un double chaînage qui permet à l'objet de type `Presence` d'avoir un lien vers la liste des précurseurs et à la cellule d'avoir un lien vers l'objet de type `Presence`. Pour se désabonner, il y a donc deux moyens :

- soit l'événement est généré et dans ce cas la liste des précurseurs est parcourue pour retirer toutes les cellules dont le précurseur peut être réveillé, c'est-à-dire dont la configuration événementielle est satisfaite.
- soit le précurseur n'a plus besoin d'être exécuté pour diverses raisons, comme une préemption du précurseur ou un gel, et dans ce cas le précurseur doit lui-même se désabonner. Pour désabonner un précurseur, il suffit d'appeler la méthode `reset` sur sa configuration événementielle. Cette opération retire, pour chaque événement de la configuration, la cellule contenant le précurseur de la liste des précurseurs. Quand une instruction événementielle se termine, elle réinitialise immédiatement sa configuration événementielle comme le montre le code de l'instruction `Control` dans la table 4.10. Si l'instruction a été préemptée, l'instruction de préemption, `Kill` ou `Until`, appelle la méthode `reset` sur le programme préempté ce qui le réinitialise, ainsi que toutes ses configurations. Par contre, dans le cas d'un gel d'instruction, il ne faut pas réinitialiser le programme puisque son résidu doit être récupéré. Nous avons donc modifié la méthode `residual` pour que chacune des instructions événementielles réinitialise elle-même sa configuration avant de retourner son résidu.

Pour l'instruction `Scanner`, la gestion est entièrement effectuée par l'instruction elle-même. Celle-ci s'abonne lors de sa première exécution dans l'instant et, dès que la fin de l'instant a été déclarée, elle est réveillée pour se désabonner.

Pour éviter les pertes de temps dues à la création de nouvelles cellules et celles dues au passage du *Garbage Collector*, nous conservons les cellules des listes de précurseurs inutilisées dans un *pool* de cellules pour les réutiliser.

Nettoyage de la table d'événements

Dans toutes les implémentations de *Junior*, un ajout dans l'environnement crée une nouvelle entrée dans une table de hachage ainsi que la création d'un objet de type *EventData*. Le problème qui peut survenir ici est une augmentation constante de la taille mémoire utilisée due à l'apparition de nouveaux événements. Un autre problème est que le temps de recherche dans la table de hachage dépend en partie du nombre d'événements que cette table contient. Ni *Rewrite*, *Replace*, *Storm* ou *Glouton* ne disposent d'un mécanisme permettant de *nettoyer* l'environnement des événements devenus *obsolètes*. Quant à *Simple*, il propose un mécanisme qui parcourt l'ensemble de la table des événements pour retirer les événements *obsolètes*, mais qui peut devenir très coûteux si le nombre d'événements est grand. De plus, ce mécanisme n'est pas automatisé et c'est à l'utilisateur de paramétrer l'environnement pour indiquer la fréquence (en nombre d'instants) de son passage.

Dans le cadre des *Icobjs*, chaque entité a de nombreux événements qui lui sont propres, comme par exemple les événements de souris et de clavier. Ces événements sont fonction de l'identifiant de l'icobj qui est unique et quand un icobj disparaît d'une simulation, il est inutile de les conserver.

Pour pouvoir mettre en place le mécanisme de nettoyage, il faut d'abord définir le moment où un événement peut être déclaré *obsolète*. La première approche est de dire qu'un événement

est devenu *obsolète* quand sa liste de précurseurs est vide. Le problème est que, dans le cadre de *Junior*, le système doit conserver les valeurs associées à un événement pour l'instant courant et l'instant précédent. On considère donc qu'*un événement est obsolète et peut être enlevé de l'environnement dès que sa liste de précurseurs est vide et que sa dernière génération date d'au moins deux instants.*

Pour être efficace, nous procédons de la même manière que pour la gestion des programmes gelés dans *Storm*. Dans l'environnement, nous rajoutons deux listes pouvant contenir des objets de type `EventData`. La première liste contient les objets `EventData` des événements dont la liste de précurseurs est devenu vide au cours de l'instant et la deuxième liste contient les objets `EventData` qui pourront être retirés de la table d'événements à la fin de l'instant courant. Le mécanisme consiste donc, à la fin de l'instant, à retirer les éléments de la deuxième liste de la table des événements et à échanger les deux listes (la liste parcourue qui a été vidée devient la liste des objets `EventData` dont la liste de précurseurs est vide au cours du nouvel instant). De plus, si, au cours d'un instant, il y a une génération d'un événement dont l'objet `EventData` est enregistré dans une des listes de retrait ou si une nouvelle instruction se met en attente de cet événement, celui-ci est automatiquement retiré de cette liste. Comme pour les précurseurs, il y a un double chaînage entre les objets de type `EventData` et les listes qui contiennent ces objets. Ce double chaînage permet d'avoir un retrait de cette liste avec une complexité en O(1).

L'instruction `Local`

Les événements locaux posent un problème particulier au mécanisme de nettoyage. En effet, il ne faut pas que les événements locaux s'enregistrent dans les différentes tables de nettoyage car un événement global utilisant le même identificateur pourrait être retiré alors qu'il est toujours utilisé. Pour éviter ce problème, l'objet `EventData` de l'instruction `Local` est créé sans référence à l'environnement, ce qui l'empêche de s'enregistrer.

```
protected void bind(InternEnvironment env, Instruction aParent, boolean removable)
{
   super.bind(env, aParent, removable);
   body.bind(env, this, removable);

   /* Pour eviter les problemes a la migration */
   if (localEvent.generated == localEvent.lastActualization)
      localEvent.lastActualization = localEvent.generated = env.getInstant();
   else if ((localEvent.generated + 1) == localEvent.lastActualization)
   {
      localEvent.lastActualization = env.getInstant();
      localEvent.generated = localEvent.lastActualization - 1;
   }
}
```

TAB. 4.12 – Méthode *bind* de l'instruction *Local* en *Reflex*

Pour savoir si un événement est généré, la génération est datée en utilisant le compteur d'instant au moment de cette génération. Dans le cadre d'une migration d'un programme contenant des événements locaux, la structure `EventData` de l'événement local migre aussi. Mais, en changeant de machine réactive, le compteur d'instant n'a pas forcément la même valeur. Ainsi, pour mettre à jour les informations contenues par l'objet `EventData`, nous

avons modifié la méthode `bind` de l'instruction `Local` pour que celle-ci assure la cohérence de ces informations en arrivant sur une nouvelle machine. Le résultat de cette modification est présenté dans la table 4.12. La date de génération est contenue dans le champ nommé `generated` et le champ `lastActualization` indique la date de dernière mise à jour de l'objet `EventData`. Par l'intermédiaire de ce mécanisme, si une valeur a été générée au cours de l'instant où le programme a été gelé, nous avons la garantie de pouvoir traiter cette valeur sur le nouveau site.

4.2.3 Machine réactive

Nous allons maintenant présenter les principales caractéristiques de la machine réactive de *Reflex*. `MachineIcobj` dispose de champs supplémentaires par rapport à l'implémentation de la *safe-machine* dans *Storm* :

- une table de hachage associant les identifiants des chacun des icobjs enregistrés à leur objet `IcobjThread`. Cette table permet un accès direct aux `IcobjThread` pour effectuer les ajouts de programmes, leur retrait ou leur réinitialisation.
- une liste contenant tous les événements externes générés durant une micro-étape. Le stockage de ces événements dans une liste intermédiaire permet d'éviter les problèmes de concurrence des threads *Java* (celui qui se charge de l'exécution de la machine réactive et ceux générant les événements externes).

`MachineIcobj` dispose également de nouvelles méthodes :

- `void add(Icobj icobj, Program behav)`
  Cette méthode ajoute un programme dans la machine réactive à l'instant suivant. Ce programme est associé à l'icobj passé en argument. Si l'icobj est déjà enregistré dans la machine réactive, c'est-à-dire si un objet `IcobjThread` est en charge de lui, alors on appelle la méthode `add` de cet objet `IcobjThread`. Au contraire, si l'icobj n'a pas encore été enregistré, alors un nouvel `IcobjThread` est créé. Cet objet `IcobjThread` est ajouté dans une liste contenant l'ensemble des objets `IcobjThread` à rajouter au début de l'instant suivant. Comme pour une *safe-machine*, on ajoute en fait une copie du programme et non le programme lui-même. Pour savoir si l'icobj a déjà été enregistré dans la machine réactive, tous les objets `IcobjThread` sont enregistrés dans une table de hachage dont la clé est l'identifiant de l'icobj qui est unique. Les accès et les modifications de cette table de hachage sont protégés par une section critique.

- `void reset(String identifier)`
  Cette méthode préempte, à la fin de l'instant, les programmes associés à l'icobj dont l'identifiant est passé en argument. L'objet `IcobjThread` en charge de cet icobj est récupéré dans la table de hachage et la méthode `reset` est appelée sur cet objet.

- `boolean remove(String identifier)`
  Cette méthode demande le gel, à la fin de l'instant, des programmes associés à l'icobj dont l'identifiant est passé en argument. L'objet `IcobjThread` correspondant à l'identifiant est récupéré dans la table de hachage et sa méthode `remove` est appelée. Le programme gelé sera alors stocké dans un des champs prédéfinis de l'icobj. Un appel à `remove` est le seul moyen de retirer un objet `IcobjThread` de la machine.

```
public synchronized boolean react()
{
   // Protection contre les appels reentrants
   if (reacting) return false;
   reacting = true;
   // Generation des evenements externes
   synchronized (toGenerate)
   { performGenerations(); }
   // Ajout des nouveaux IcobjThreads
   synchronized (icobjs)
   { performAddings(); }
   byte s = Flags.WAIT;
   // Tant que le statut WAIT est retourne, on continue a executer
   // le programme.
   while (s == Flags.WAIT)
   {
      s = programs.rewrite();
      synchronized (toGenerate)
      {
         // On genere les evenements externes s'il y en a
         // Sinon, la fin d'instant peutê tre édcaree
         if (toGenerate.isEmpty() == false)
         {
            performGenerations();
            if (s == Flags.LONGWAIT)
               s = Flags.WAIT;
         }
         else
            env.setEoi();
      }
   }
   env.newInstant();
   reacting = false;
   return (s == Flags.TERM);
}
```

TAB. 4.13 – Méthode *react* de *MachineIcobj* en *Reflex*

- `Program residualBehaviorOf(String identifier)`
  Cette méthode retourne le résidu des programmes associés à l'icobj dont l'identifiant est passé en argument. L'appel à cette méthode ne peut être fait qu'entre deux réactions, sinon une exception est générée. Nous avons ajouté cette méthode pour permettre de récupérer le résidu des programmes en vue d'un enregistrement dans un fichier, tout en permettant à ces derniers de continuer à s'exécuter à la prochaine réaction. Pour différencier la récupération du résidu des programmes de celle effectuée par l'instruction `IcobjThread`, nous avons ajouté un booléen en argument de la méthode `residual`. Ce booléen indique si les différentes configurations événementielles doivent être réinitialisées, et si les précurseurs doivent être désabonnés. Il est mis à *true* dans le cas où l'instruction `IcobjThread` l'utilise.

- `void generate(Identifier event)`
  `void generate(Identifier event, Serializable obj)`
  Ces méthodes génèrent des événements externes. En *Junior*, les générations externes sont immédiatement effectuées si la fin de l'instant n'est pas déclarée. Par contre cela n'est pas possible dans notre cas, comme dans *Storm*. En effet, si nous autorisons la

génération directe des événements, il y a alors des accès et des modifications concurrentes entre l'exécution descendante des instructions et la libération de chemin en remontant dans l'arbre, ce qui peut provoquer des incohérences. C'est pourquoi, toutes les générations externes sont placées dans une liste intermédiaire et sont uniquement effectuées entre deux réécritures du parallèle de plus haut niveau, comme décrit dans la table 4.13.

- `boolean react()`
  Cette méthode fait réagir l'instruction `Par` de plus haut niveau, comme le montre le code de la table 4.13. Nous avons déplacé le comportement de l'instruction `Instant` de *Storm*, *Rewrite* et *Replace* pour permettre d'effectuer les générations externes faîtes sur la machine entre les exécutions de l'instruction `Par`.

4.3 Performances

Nous allons exécuter quelques tests de performances sur *Reflex* et *Storm* dont *Reflex* est issu. Nous comparons aussi *Reflex* à *Glouton* et *Simple* qui sont les deux implémentations les plus efficaces de *Junior*. Les comparatifs prendront en compte le temps d'exécution dans le cas des quatre premiers tests et l'espace mémoire occupée dans le cas du dernier. Nous avons utilisé la version 3.1 des *SugarCubes* qui implémente les algorithmes de *Storm* et la version 2.6 de *Glouton*. Nous utilisons l'implémentation des *SugarCubes* pour tester les algorithmes de *Storm* car l'algorithme de balancement de l'arbre de l'implémentation *Junior* ne fonctionne pas. Il faut noter que *SugarCubes* dispose d'une interface qui permet d'exécuter des programmes écrits en *Junior*.

Tous les programmes de tests que nous allons présenter sont exécutés sur la configuration suivante :

Intel(R) Pentium(R) 4 CPU 2.60GHz - 512Mo de RAM
Linux version 2.6.8 (version gcc 3.3.4 (Debian 1 :3.3.4-9))
Java(TM) 2 Runtime Environment, Standard Edition (build Blackdown-1.4.1-beta)

Nous allons d'abord présenter quatre tests standard qui permettent de représenter le coût de l'attente d'un très grand nombre d'événements et le coût de l'ordonnancement des tâches à exécuter. Ces tests sont la cascade directe instantanée, la cascade inverse instantanée, la cascade directe inter-instant et la cascade inverse inter-instant. Ces tests servent de référence pour juger de l'efficacité de chacune des implémentations. La cascade directe instantanée permet de juger du temps nécessaire pour exécuter de nombreuses tâches parallèles sans que celles-ci soient contraintes par l'ordonnancement des branches parallèles. La cascade inverse instantanée permet d'ajouter des contraintes d'ordonnancement entre les générations et les attentes d'événements des différentes branches. Les exemples inter-instants permettent d'observer le coût d'attentes inter-instants.

Pour chacun de ces tests, nous présentons le temps d'exécution en fonction du nombre de branches parallèles. Le temps mesuré comprend la moyenne de la durée totale de l'instant et la moyenne de la somme des durées de chaque instant pour les cascades inter-instants. Résoudre une cascade (directe ou inverse) signifie exécuter l'ensemble des instructions jusqu'à ce que l'instruction `Par` qui les contient retourne *TERM*. Nous présentons, pour chacun des exemples, un graphique montrant le temps de résolution de la première résolution de ces programmes et un graphique montrant la moyenne des temps des résolutions suivantes. Dans le cas de la cascade instantanée, la résolution est faite en un instant, et celle de la cascade inter-instant

est faite en autant d'instant qu'il y a de branches parallèles. Nous faisons la distinction entre la première résolution et les suivantes car, à la fois, l'initialisation des mécanismes proposés par chacune des implémentations et le lancement de la machine virtuelle influe sur la première exécution. Par contre, durant les résolutions suivantes, il est intéressant de voir les performances sur des programmes cycliques en attente d'événement, ce qui correspond aux comportements des icobjs.

Dans ces tests, nous observerons le temps de façon globale, chaque version de *Junior* disposant de son propre mécanisme de mise en place du système lors du premier instant. Dans le cas de *Simple* et de *Glouton*, il s'agit de la mise en place des programmes dans chacune des listes et dans le cas de *Storm* du balancement de l'arbre. De plus, le premier instant est toujours plus coûteux puisqu'il enregistre l'ensemble des événements. Nous avons également voulu représenter le coût du mécanisme de nettoyage. Il est automatique dans le cas de *Reflex*. *Storm* et *Glouton* ne dispose pas de mécanisme de nettoyage. *Simple* en a un, mais il n'est pas activé par défaut. Ce mécanisme parcourt entièrement la table des événements tous les n instants, n devant être paramétré par l'utilisateur en fonction des applications. Dans le cadre de nos tests, nous avons activé le mécanisme de *Simple* tous les instants (ce que nous nommerons *Simple1*) pour le comparer aux mécanismes de *Reflex* qui est actif à chaque instant, et nous avons également effectué ces tests en activant le mécanisme tous les 50 instants (ce que nous nommerons *Simple50*).

La cascade directe instantanée

Le programme exécuté est présenté sur la table 4.14. Ce programme sert de point de référence pour pouvoir le comparer aux suivants. Les attentes sont placées de telle sorte que la génération de l'événement correspondant ait déjà eu lieu dans l'instant. Ainsi lorsque l'instruction `Await` teste la présence de l'événement, celui-ci est toujours présent. La résolution de la cascade directe est faite en un seul instant. Ce programme est placé dans une boucle pour différencier la première exécution des suivantes.

```
Program[] par = new Program[num];
for(int i = 0; i < (num-1); i++)
   par[i] = Jre.Seq(Jre.Await("s" + i), Jre.Generate("s" + (i + 1)));
par[num-1] = Jre.Await("s" + (num-1));

Program x = Jre.Loop(Jre.Seq(Jre.Generate("s0"), Jre.Seq(Jre.Par(par), Jre.Stop())));
Machine machine = Jre.SafeMachine(x);
```

TAB. 4.14 – Programme de la cascade directe instantanée

La figure 4.3 représente le coût du premier instant d'exécution. Nous pouvons constater sur ce graphique que le coût de démarrage pour *Glouton* est bien plus important que pour les trois autres implémentations. Ceci s'explique principalement par la traduction de l'AST en instructions exécutables et la création d'événements et de files d'attente attachées à ces événements. *Storm* et *Reflex* ont des coûts relativement proches. Il faut tout de même rappeler qu'à la différence des trois autres implémentations, *Storm* ne dispose que d'un `Par` binaire.

La figure 4.4 traduit le coût de ce même programme durant les ré-exécutions dues à l'instruction `Loop`. On peut voir que toutes les versions ont un temps d'exécution linéaire et que *Reflex* et *Glouton* sont relativement proches. *Simple* est légèrement plus coûteux, ce

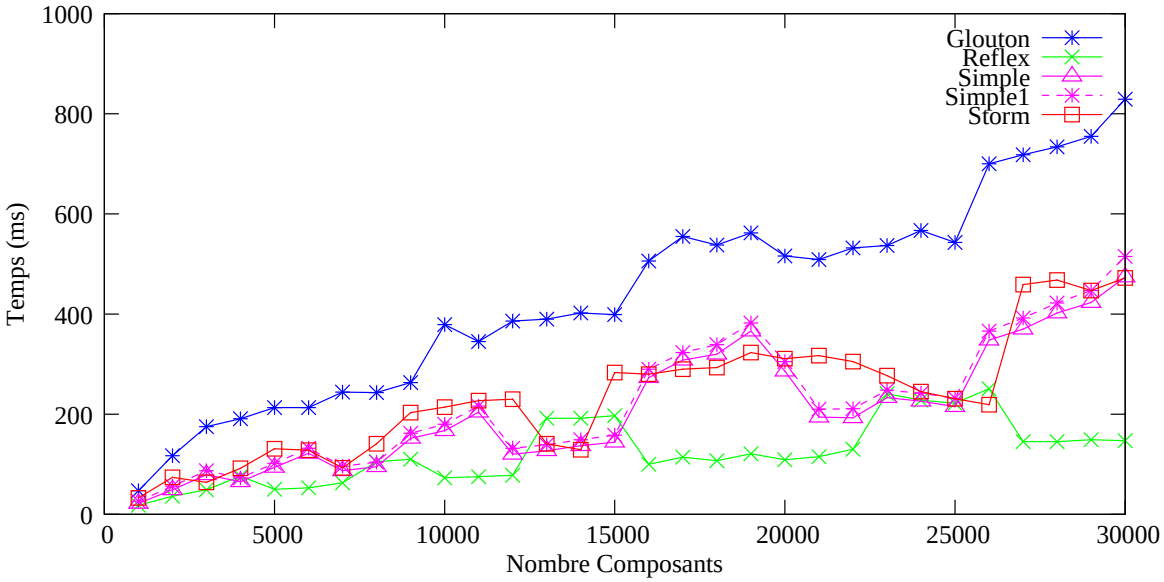


FIG. 4.3 – Cascade directe instantanée (première exécution)

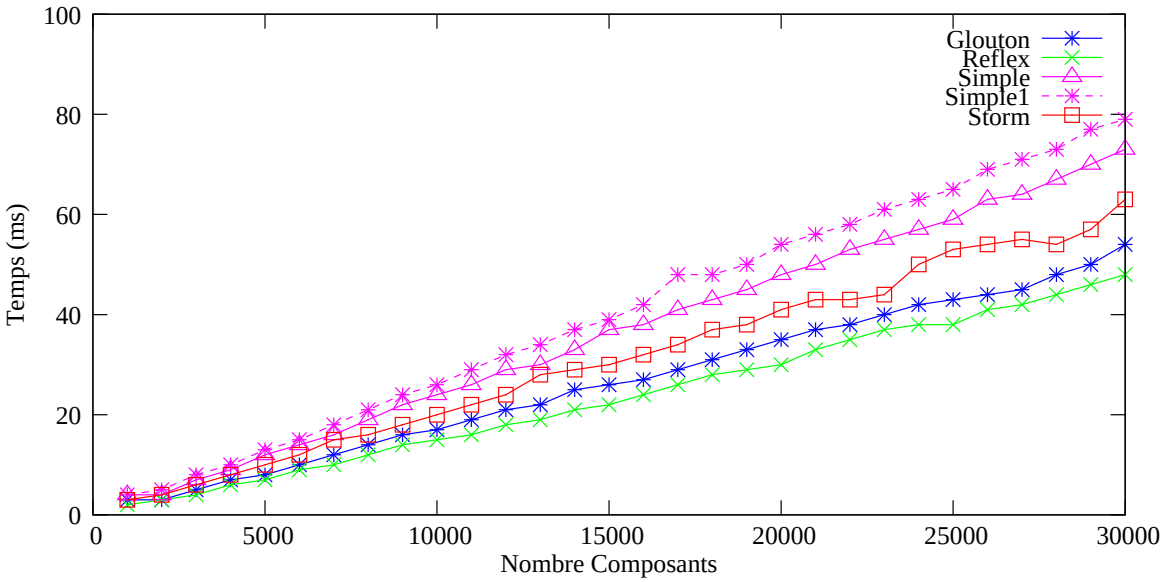


FIG. 4.4 – Cascade directe instantanée

qui est sûrement dû à un mécanisme de gestion de files d'attente assez lourd. La différence entre *Simple* et *Simple1* montre le temps pris par le mécanisme de nettoyage de la table d'événements. Evidemment, plus le nombre d'événements est important, plus le temps de nettoyage est grand. De plus, la différence de temps ne traduit que le parcours de la table d'événements et non un retrait de la table de hachage puisque les événements sont réutilisés d'un instant sur l'autre. Nous n'avons pas représenté ici *Simple50* puisque la résolution de la cascade instantanée est effectuée en un instant.

Ce cas est le plus favorable pour les versions, comme *Storm*, qui ont une instruction

`Par` déterministe dans laquelle l'exécution des branches est ordonnancée de gauche à droite. L'ordonnancement en cascade directe fait que les mécanismes de destruction de la structure de l'arbre et de gestion de file d'attente de *Simple* entraînent un coût supplémentaire qui est dans cet exemple particulier inutile.

La cascade inverse instantanée

La cascade inverse instantanée permet de tester la sensibilité du moteur réactif à la façon dont les branches parallèles sont écrites et ordonnancées dans le programme. Le programme exécuté est présenté sur la table 4.15.

```
Program[] par = new Program[num];
for(int i = 0; i < (num-1); i++)
   par[num-i-1] = Jre.Seq(Jre.Await("s" + i), Jre.Generate("s" + (i + 1)));
par[0] = Jre.Await("s" + (num-1));

Program x = Jre.Loop(Jre.Seq(Jre.Generate("s0"), Jre.Seq(Jre.Par(par), Jre.Stop())));
Machine machine = Jre.SafeMachine(x);
```

TAB. 4.15 – Programme de la cascade inverse instantanée

La figure 4.5 présente les temps pour la première résolution et la figure 4.6 les temps moyens des instants suivants de la cascade inverse instantanée. Nous pouvons remarquer, qu'excepté pour *Storm*, les temps et donc les écarts restent identiques à ceux observés dans l'exemple de la cascade directe instantanée.

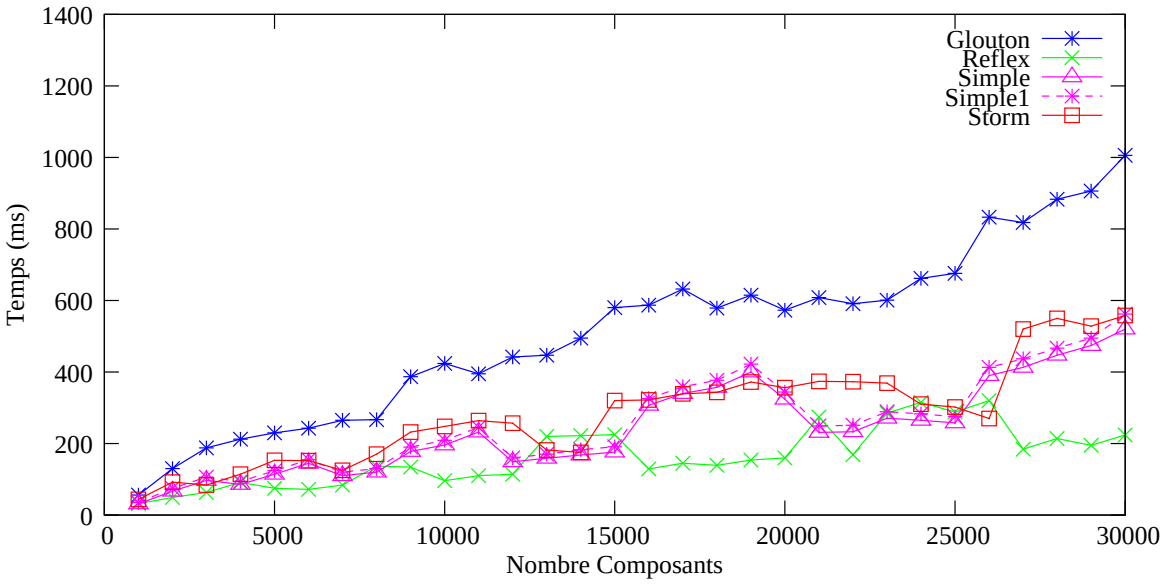


FIG. 4.5 – Cascade inverse instantanée (première exécution)

Storm est plus lent sur la cascade inverse instantanée. En effet, *Storm* parcourt d'abord l'ensemble de l'arbre dont toutes les branches retournent le statut *WAIT*, excepté la dernière qui génère l'événement attendu par l'avant-dernière branche. La résolution de cette cascade

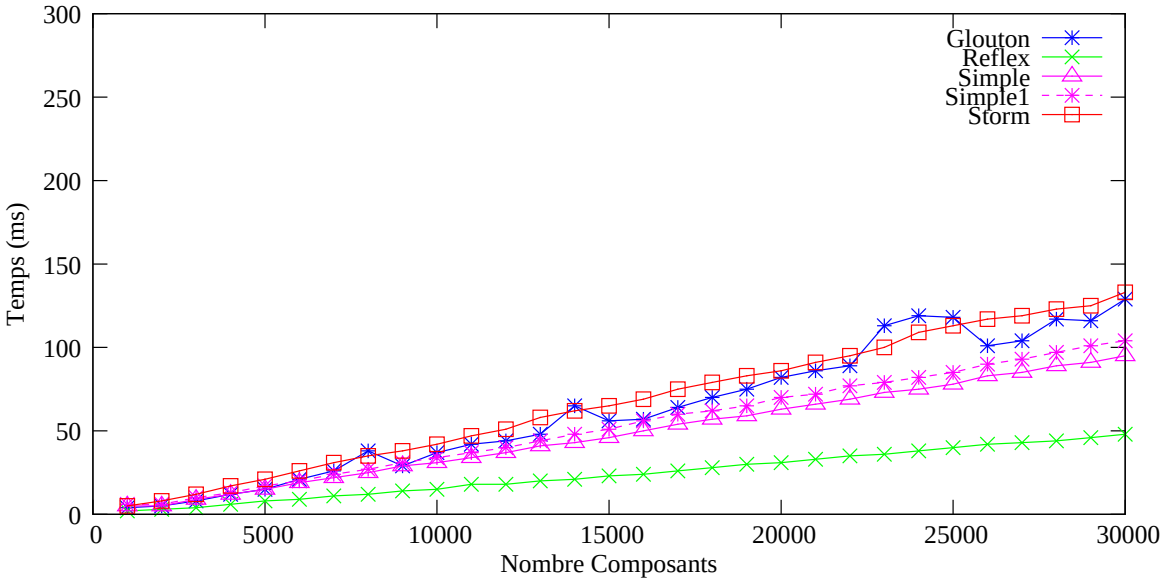


FIG. 4.6 – Cascade inverse instantanée

inverse prendra n micro-étapes, n étant le nombre de branches parallèles. De plus, *Storm* doit descendre en profondeur dans l'arbre pour accéder aux instructions puisqu'il ne dispose pas d'une instruction `Par` n-aire, alors que les autres implémentations accèdent directement à la prochaine instruction à exécuter. Il faut aussi noter que l'utilisation d'une instruction `Par` n-aire implémentée à l'aide d'un tableau d'instructions n'améliorerait pas nécessairement les performances de *Storm*, puisqu'il faudrait tester à chaque instant quelle instruction doit être exécutée. L'utilisation d'un arbre binaire offre une dichotomie dans la décision des instructions à exécuter, ce qui évite de tester toutes les instructions et permet de descendre uniquement dans les branches de l'arbre qui ont besoin d'être exécutées. Le fait d'avoir un `Par` déterministe fait que *Storm* est sensible à la façon dont un programme est écrit, et à l'ordonnancement initial des instructions à exécuter en parallèle.

Nous pouvons également constater que *Reflex* est, au moins, deux fois plus rapide que toutes les autres implémentations. Ceci est principalement lié au fait qu'entre deux réactions de la boucle en *Reflex*, les instructions de `Par` sont réordonnancées dans l'ordre dans lequel elles ont terminé, donc en cascade directe. Nous pouvons d'ailleurs remarquer que les temps de *Reflex* sont identiques à ceux de la cascade directe instantanée de la figure 4.4.

La cascade directe inter-instant

Nous passons aux tests inter-instants. Les mêmes programmes sont exécutés excepté qu'une instruction `Stop` est ajoutée entre les attentes d'événements et les générations. Ces tests permettent d'analyser le coût de l'attente inter-instant des événements.

Le programme de la cascade directe inter-instant est présenté sur la table 4.16.

La figure 4.8 donne le temps moyen de résolution de la cascade directe inter-instants. Cette résolution n'est plus faite en un instant, mais elle prend autant d'instants que de branches parallèles. La figure 4.7 donne le temps de la première résolution de ce programme. Pour améliorer la lisibilité de ces graphes, nous ne présenterons qu'une partie des résultats de *Storm* dont les temps d'exécution deviennent trop importants. Le tableau 4.17 permet de

```
Program[] par = new Program[num];
for(int i = 0; i < (num-1); i++)
   par[i]= Jre.Seq(Jre.Await("s" + i), Jre.Seq(Jre.Stop(), Jre.Generate("s" + (i + 1))));
par[num-1] = Jre.Await("s" + (num-1));

Program x = Jre.Loop(Jre.Seq(Jre.Generate("s0"), Jre.Seq(Jre.Par(par), Jre.Stop())));
Machine machine = Jre.SafeMachine(x);
```

TAB. 4.16 – Programme de la cascade directe inter-instant

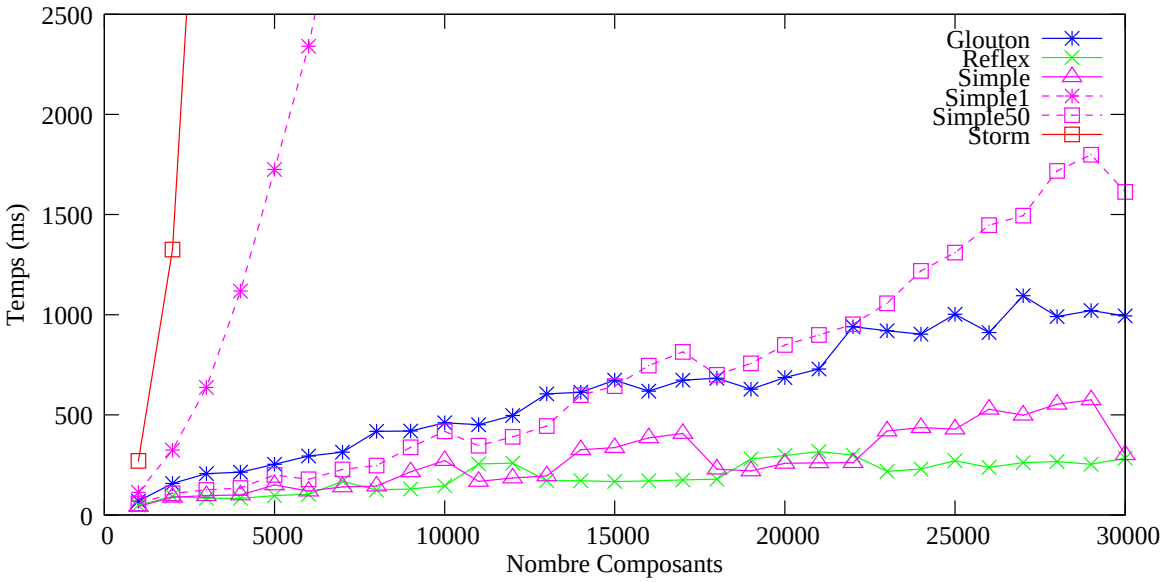


FIG. 4.7 – Cascade directe inter-instant(première exécution)

compléter les résultats de *Storm* et de les comparer à ceux de *Reflex*.

	3000	5000	7000	9000	11000	13000
Storm	3,6s	10s	32,4s	58,3s	1min10s	1min54s
Simple1	558ms	1,6s	3,3s	9,2s	12,7s	27,3s
Reflex	13ms	23ms	29ms	37ms	46ms	54ms

TAB. 4.17 – Temps de résolution de la cascade directe inter-instant pour *Storm*, *Simple1* et *Reflex*

Storm n'est pas efficace pour ce genre de programmes. À chaque instant, *Storm* enregistre des précurseurs dans la liste des événements qui n'ont pas encore été générés et, à la fin de l'instant, il ré-exécute ces précurseurs pour qu'ils se terminent dans l'instant. Par contre, les trois autres implémentations font de l'attente passive inter-instant, et ont toutes les trois des temps similaires. Il faut aussi noter que les temps de résolution sont finalement assez proches de ceux de la cascade directe instantanée. L'écart de temps entre ces deux programmes vient principalement de la gestion de l'environnement entre les instants.

Dans ce test, *Reflex* a perdu son gain de rapidité sur *Simple* et *Glouton*. Ceci est principalement lié à l'instruction interne intermédiaire `IcobjThread`. En effet, cette instruction

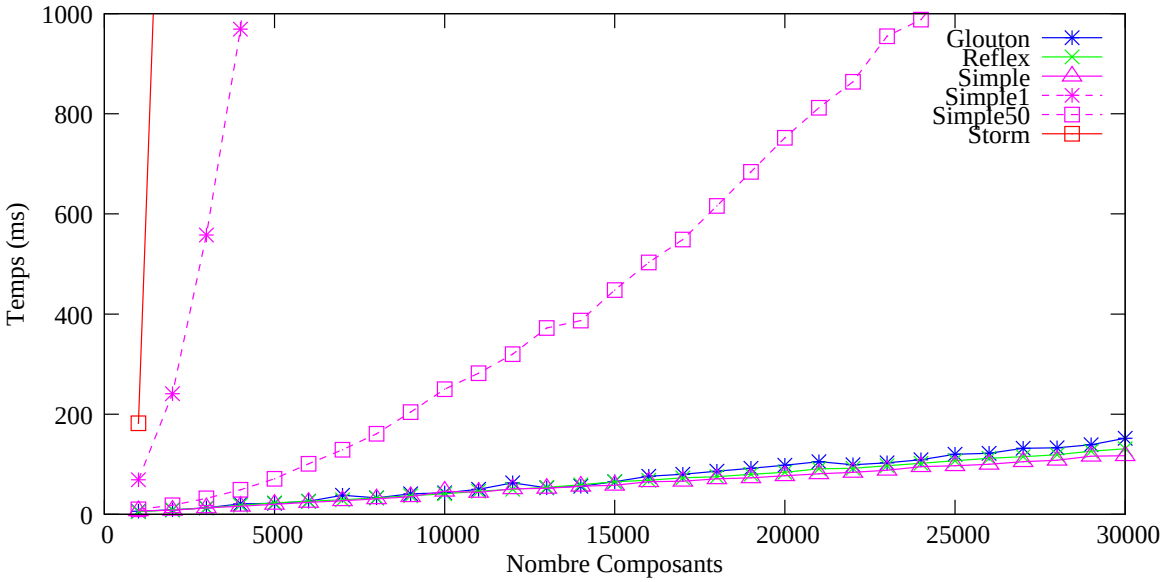


FIG. 4.8 – Cascade directe inter-instant

est traversée à chaque instant. Comme elle réalise la préemption et le gel de l'objet, elle doit attendre la fin de l'instant pour terminer son exécution. Cela signifie que `IcobjThread` est activé deux fois par instant, ce qui dans le cas d'une exécution inter-instant devient relativement coûteux. Il faut ajouter à cela le mécanisme de nettoyage de la table des événements qui efface à chaque résolution tous les événements de la table pour les remettre à la prochaine exécution de la boucle du programme. Dans l'exemple de la cascade instantanée, ce mécanisme n'effaçait pas les événements puisque la résolution de la cascade est instantanée et que les événements sont réutilisés immédiatement à l'instant suivant. Dans le cas où le mécanisme de nettoyage de *Simple* est activé, *Simple1* ou *Simple50*, les performances s'effondrent. En effet, si l'on considère le test avec 5000 branches, dans le cas de *Simple1*, la table d'événements est parcourue 5000 fois pour éliminer à chaque fois un événement alors que, dans le cas de *Simple50*, elle est parcourue 100 fois pour éliminer à chaque fois 50 événements.

La cascade inverse inter-instant

Pour la cascade inverse inter-instant, le programme exécuté est présenté sur la table 4.18.

```
Program[] par = new Program[num];
for(int i = 0; i < (num-1); i++)
   par[num-i-1]= Jre.Seq(Jre.Await("s" + i),
                    Jre.Seq(Jre.Stop(), Jre.Generate("s" + (i + 1))));
par[0] = Jre.Await("s" + (num-1));

Program x = Jre.Loop(Jre.Seq(Jre.Generate("s0"),
                        Jre.Seq(Jre.Par(par), Jre.Stop())));
Machine machine = Jre.SafeMachine(x);
```

TAB. 4.18 – Programme de la cascade inverse inter-instant

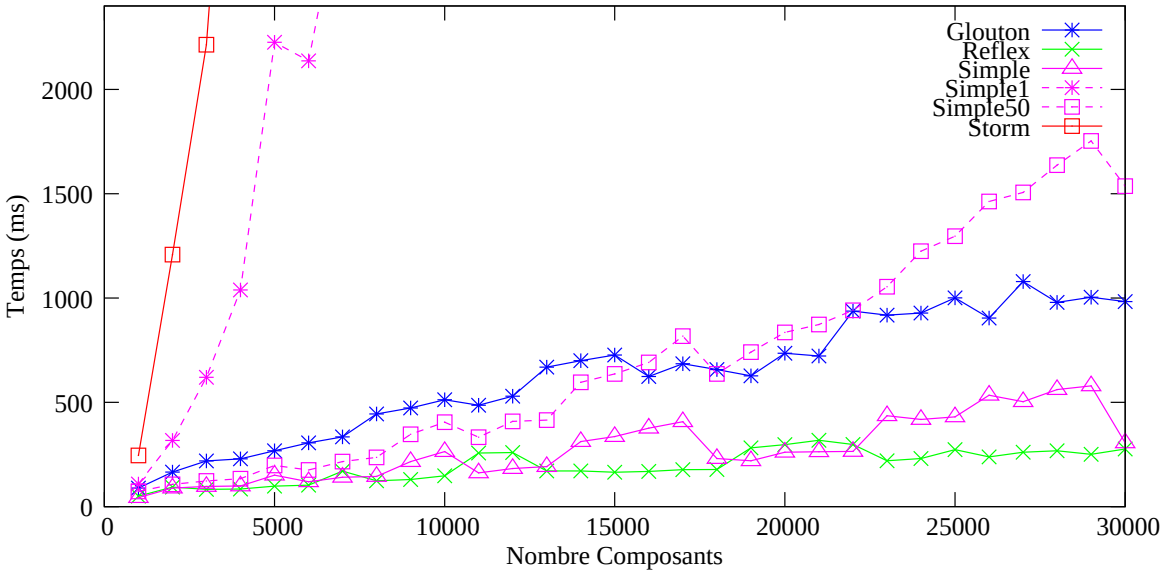


FIG. 4.9 – Cascade inverse inter-instant (première exécution)

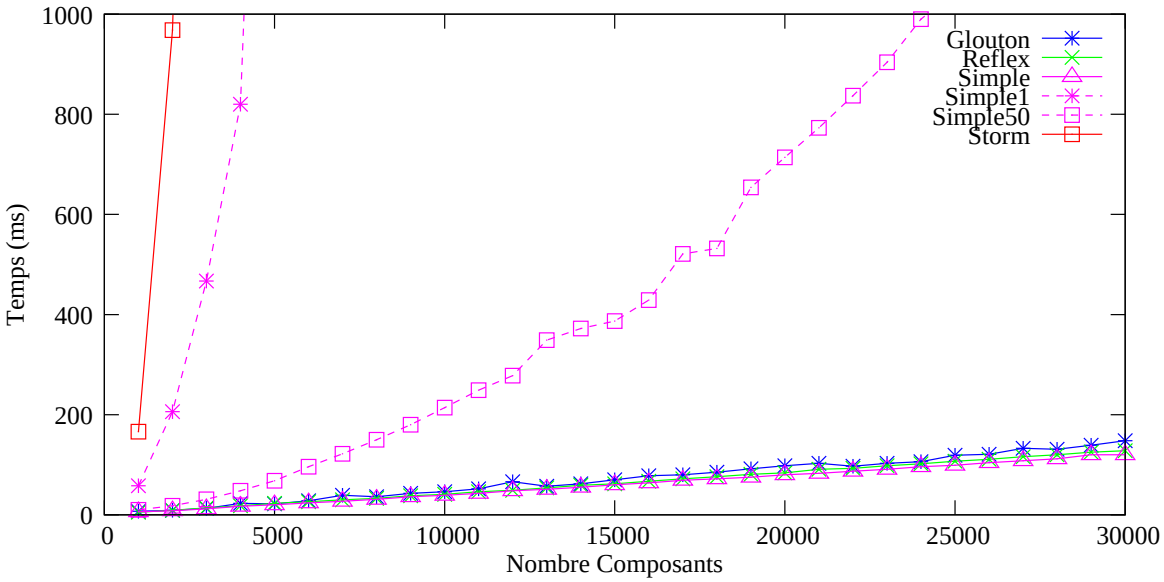


FIG. 4.10 – Cascade inverse inter-instant

	3000	5000	7000	9000	11000	13000
Storm	3,1s	9,6s	28,6s	56,7s	1min11s	1min44s
*Simple*1	467ms	2,6s	5,2s	7,1s	10,6s	16,8s
Reflex	14ms	23ms	30ms	38ms	46ms	54ms

TAB. 4.19 – Temps de résolution de la cascade inverse inter-instants pour *Storm*, *Simple1* et *Reflex*

Les figures 4.9 et 4.10 nous donnent les mêmes résultats que ceux de la cascade directe inter-instants. Par contre, comme le montre le tableau 4.19, *Storm* améliore ses performances par rapport à la cascade directe inter-instant, ce qui nous semble assez étrange.

Test mémoire

Ce test permet de représenter la quantité mémoire utilisée par chaque implémentation. Nous voulons mettre en évidence l'importance d'un mécanisme de nettoyage de la table d'événements. Le programme que l'on exécute pour mettre cela en évidence est présenté sur la table 4.20. Ce programme est constitué de la mise en parallèle du comportement de 100 icobjs. Le comportement des icobjs est de réagir aux événements clavier et souris (`keyBehavior()` et `mouseBehavior()`) et de réagir à un événement de destruction. Chaque comportement génère son propre événement de destruction au bout d'un nombre d'instant compris entre 1 et 200. Dès que cet événement est généré, un nouvel icobj avec le même comportement est ajouté à la machine réactive. Cela signifie qu'il y a toujours 100 comportements d'icobjs qui sont exécutés à chaque instant. Le comportement de chaque icobj se met en attente de 7 événements spécifiques à chacun. Entre chaque icobj, il n'y a aucune réutilisation d'événements.

```
for (int i = 0; i < 100; i++)
{
   Program behav =
      Jr.Par(
         Jr.Par(mouseBehavior(), keyBehavior()),
         Jr.Par(
            Jr.Seq(Jre.Await(new IcobjIdentifier(EventNames.KILL_EVENT)),
                   Jr.Atom(new KillMe())),
            Jr.Seq(Jre.Repeat((int)(random.nextFloat()*200), Jr.Stop()),
                   Jr.Seq(Jre.Generate(new IcobjIdentifier(EventNames.KILL_EVENT)),
                          Jr.Atom(new AddNewOne())))));
   machine.add(Jre.Link(icobj,
                  Jr.Seq(Jre.Freezable(new IcobjIdentifier(EventNames.KILL_EVENT), behav),
                         Jr.Atom(new CopyToIcobj()))));

}
```

TAB. 4.20 – Quantité mémoire utilisée au cours des instants

Dans le cas de Reflex, le comportement n'utilise ni les instructions *Link* et *Freezable* pour encapsuler le comportement de chaque icobj, ni l'atome `CopyToIcobj` qui récupère le résidu du programme pour le placer dans un champ spécifique de l'icobj, puisque cela est réalisé par l'instruction interne *IcobjThread*. Dans le cas de *Glouton*, il est impossible de récupérer le résidu du programme : l'instruction *Freezable* est remplacée par l'instruction *Until*.

La figure 4.11 présente la mémoire occupée par le programme et par son environnement durant 5000 instants d'exécution de ce programme. *Storm*, *Glouton* et *Simple* consomme de plus en plus de mémoire, ce qui peut conduire à une exception `OutOfMemoryException` et donc à l'arrêt de la machine virtuelle. Dans le cas de *Simple50* ou *Reflex*, on observe un niveau constant d'utilisation de la mémoire, ce qui devrait être le cas dans un environnement où le nombre de composants exécutés est toujours le même.

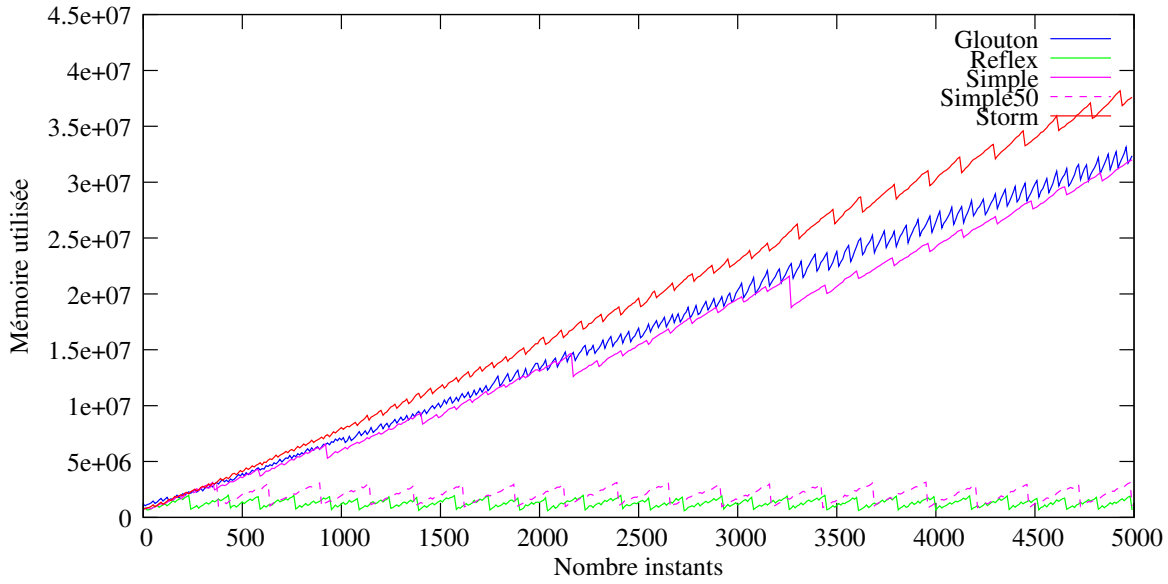


FIG. 4.11 – Consommation mémoire en fonction des instants d'exécution

4.4 Bilan

Notre objectif était d'obtenir un moteur efficace pour l'exécution des objets réactifs graphiques capable de gérer un grand nombre de branches parallèles, un grand nombre d'événements et dont le coût d'exécution est réduit au minimum quand le programme n'a rien à exécuter. Cette implémentation a été réalisée à partir de *Storm* qui était initialement le moteur réactif utilisé dans le cadre du projet IST-PING [82] que nous évoquerons dans la section 7.1. Les principales caractéristiques de *Reflex* par rapport à *Storm* sont :

- il effectue une attente passive inter-instant. Ce point est particulièrement intéressant dans le cas des *Icobjs* où les différentes entités graphiques sont en attente constante d'événements pour réagir et où on ne désire pas payer de coût supplémentaire pour ces attentes. Pour cela, nous avons rajouté un mécanisme efficace de désabonnement des précurseurs.
- tout comme *Rewrite* et *Replace*, l'arbre du programme est conservé. Pour atteindre de bonnes performances, l'instruction `Par` est devenu indéterministe au sens où l'ordre d'exécution des instructions n'est pas conservé, ce qui donne la possibilité d'exécuter uniquement les instructions qui ont besoin de l'être.
- *Reflex* dispose d'une structure interne dédiée aux *Icobjs* qui fournit les fonctionnalités des instructions `Link, Kill et Freezable`. Cette structure regroupe tous les programmes qui sont ajoutés à chacun des icobjs, ce qui permet des retraits d'icobjs en vue de migration à des coûts minimes.
- *Reflex* dispose d'un mécanisme de nettoyage de la table d'événements. Ce mécanisme permet d'éviter une explosion en mémoire dans le cas de simulations où de nombreux objets s'enregistrent et meurent rapidement. De plus, contrairement à *Storm*, les programmes sont immédiatement effacés de la machine réactive au moment où ils se terminent et peuvent donc être récupérés par le *Garbage Collector*.
- *Reflex* dispose enfin d'instructions au comportement régulier qui vont pouvoir se transposer simplement et intuitivement dans le monde graphique.

Comme les performances nous l'ont montré, *Reflex* est au moins aussi efficace que *Simple* et *Glouton* pour le type de programmes qui nous intéresse dans le cadre des *Icobjs* et il est basé sur une sémantique claire.

Dans la partie suivante, nous présenterons le modèle que nous avons créé pour la gestion des objets réactifs graphiques, les *Icobjs*, qui a été construit au-dessus de *Reflex*.

Deuxième partie

Icobjs

Chapitre 5

Modèle des *Icobjs*

Le modèle des *Icobjs* est le résultat de l'application du modèle réactif au monde graphique. Icobj signifie *Iconic Objects*. Un icobj est un objet graphique dont le comportement est défini par l'intermédiaire d'un langage réactif. Le modèle des *Icobjs* possède un mécanisme puissant de construction de comportements réactifs de façon simple et intuitive. Ce mécanisme est basé sur l'utilisation des comportements des icobjs existants pour les combiner graphiquement. Le résultat de la construction est l'ajout d'un nouvel icobj dont le comportement est le résultat de la combinaison. L'objectif principal de ce modèle est de permettre à des non-spécialistes d'*écrire des programmes* de façon graphique, sans connaître une syntaxe particulière et en masquant au maximum la notion d'instant. Cette construction permet également de disposer de toute la puissance des langages réactifs, c'est-à-dire du parallélisme et de la communication par diffusion d'événements, tout en masquant la complexité de leur syntaxe.

Un des principaux problèmes des premières implémentations d'*Icobjs* est qu'il n'y a aucun moyen de visualiser le comportement de chacun des icobjs. Nous proposons dans notre modèle un moyen de visualiser et de modifier ce comportement par l'intermédiaire d'une interface : l'inspecteur des *Icobjs*.

Dans ce chapitre, nous commencerons par présenter, dans la section 5.1, le modèle d'objet réactif avant de donner une définition de ce qu'est un icobj. Dans la section 5.2, nous décrirons le système de construction graphique des *Icobjs* et les modifications que nous y avons apportées. Enfin, dans la section 5.3 nous illustrerons par l'intermédiaire de l'application `Framework`, les fonctionnalités ajoutées au modèle pour visualiser et modifier le comportement des icobjs, et pour intervenir directement sur les paramètres d'exécution.

5.1 Modèle d'objets réactifs

Le modèle des *Icobjs* est une extension du modèle classique d'objet réactif représenté sur la figure 5.1. Ce modèle est composé d'une part de la structure de l'objet contenant toutes les informations relatives à celui-ci et d'autre part d'un programme réactif décrivant le comportement de l'objet. Les actions atomiques constituant le programme manipulent (accèdent et/ou modifient) la structure de données.

Pour implémenter cette approche, *Junior* dispose de l'instruction `Link` qui permet d'associer l'exécution d'un programme à un objet accessible par les actions atomiques à travers l'environnement.

Un icobj dispose à la fois d'un aspect graphique et d'un comportement réactif. Le résultat

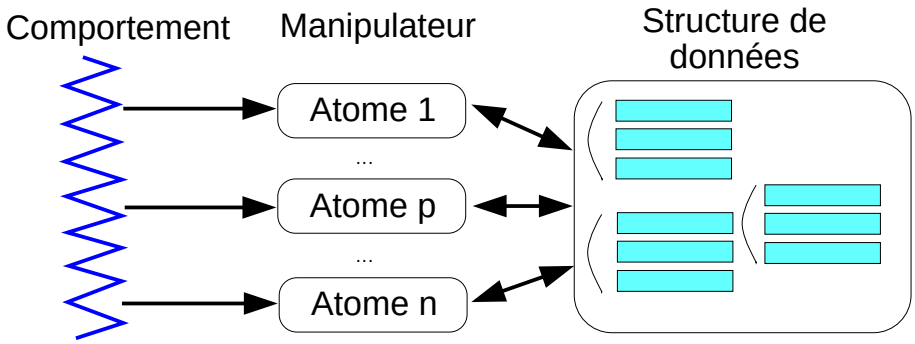


FIG. 5.1 – Objet réactif classique

de la composition de ces deux aspects est la possibilité de créer facilement des animations. Dans le modèle des *Icobjs*, les entités du système sont construites sur le même principe que les objets du modèle représenté sur la figure 5.1. Chaque icobj dispose d'un comportement réactif et d'une structure de données. Les *Icobjs* étendent le modèle d'objet réactif au niveau graphique, c'est-à-dire que le comportement des icobjs doit réagir aux événements graphiques et produire des modifications graphiques et que la structure de données contient des données liées à l'environnement graphique. L'apparence des icobjs est une composante de leur structure de données. Cette apparence est rendue entre deux exécutions de leur comportement réactif.

Par exemple, *Squeak* [42] se base également sur un modèle d'objet graphique. Les objets graphiques appelés *morphs* dispose initialement d'une structure de données de grande taille. Au contraire, nous souhaitons définir, pour le modèle des *Icobjs* [22], une structure minimale aisément extensible. Un icobj devra dans son comportement introduire explicitement des actions atomiques permettant l'ajout de nouvelles données pour étendre sa structure de données. La structure minimale doit contenir les informations suivantes :

- des identifiants. Un icobj doit pouvoir être identifié de façon unique dans l'environnement dans lequel il est exécuté. Réciproquement, il doit aussi pouvoir accéder à l'environnement d'exécution.
- des informations graphiques. Les informations minimales nécessaires à l'utilisation d'entités graphiques sont l'apparence, la position et les dimensions. Dans la suite du document, nous parlerons de zone d'influence lorsque nous ferons référence à l'*encombrement positionné* de l'icobj, c'est-à-dire à l'espace (aire ou volume) occupé par l'icobj situé à une position particulière de son environnement graphique.
- d'un comportement réactif. Le modèle des *Icobjs* possède son propre système de construction graphique qui est basé sur la réutilisation du comportement des icobjs déjà existants. C'est pourquoi nous allons différencier deux parties dans le comportement d'un icobj : le comportement *clonable* qui est utilisé dans le cadre du système de construction et le comportement non-clonable qui décrit les comportements spécifiques de l'icobj.

Nous distinguons deux types d'entités dans le modèle des *Icobjs* : l'icobj qui est une entité réactive graphique et son environnement d'exécution nommé *Workspace*. Si l'on considère que l'icobj est un programme réactif, le workspace représente alors une machine réactive en charge de l'exécution des icobjs. En réalité, les tâches du workspace sont bien plus importantes. Il est en charge, entre autres :

- de l'affichage des icobjs qu'il exécute.
- de la gestion de l'interfaçage avec le "monde extérieur", en particulier des événements clavier et souris provenant de l'utilisateur.

Dans notre modèle, nous considérons que le workspace est un icobj comme les autres avec un comportement particulier. En effet, celui-ci contient, tout comme l'icobj, une partie graphique qui consiste à s'afficher lui-même avant de dessiner les icobjs qu'il contient, et une partie comportementale minimale qui consiste à exécuter les icobjs qu'il contient. Cette caractéristique permet d'encapsuler des workspaces dans d'autres workspaces et d'avoir ainsi un modèle où tout élément graphique est un icobj. De plus, puisqu'un workspace est un icobj, il est possible d'étendre sa structure de données et son comportement. Il faut noter que chaque workspace est un environnement d'exécution indépendant, c'est-à-dire que les événements générés dans un workspace ne sont pas présents dans les autres.

5.2 Construction par *Icobjs*

Le mécanisme de construction des *Icobjs* permet de combiner graphiquement des comportements élémentaires en leur ajoutant une structure de contrôle. Pour réaliser ces constructions, nous disposons de plusieurs icobjs particuliers appelés *constructeurs graphiques* qui permettent de réaliser certaines constructions du langage réactif. Nous allons détailler dans cette partie la manière d'utiliser chacun des ces constructeurs.

Tout d'abord, nous allons faire une distinction dans le mécanisme de construction graphique. En plus des *constructeurs graphiques* que nous décrirons par la suite, le mécanisme de construction graphique utilise également d'autres icobjs qui sont décrits comme des briques élémentaires. Ces icobjs ont la particularité de n'exécuter que la partie non-clonable de leur comportement, alors que tous les autres icobjs vont exécuter la composition parallèle des comportements clonable et non-clonable.

Chaque icobj doit aussi exécuter, en plus des comportements clonable et non-clonable, un comportement de contrôle. Ce comportement de contrôle ajouté par le workspace à chaque icobj réagit aux événements souris et clavier transmis par le workspace. Il consiste à :

- sélectionner un icobj en cliquant avec le bouton gauche de la souris dans la zone d'influence de l'icobj,
- déplacer l'icobj sélectionné en maintenant le bouton gauche de la souris enfoncé et en la déplaçant,
- changer la taille de l'icobj sélectionné en maintenant le bouton droit de la souris enfoncé et en la déplaçant,
- activer un icobj en double-cliquant sur l'icobj.

5.2.1 Comportements élémentaires

Les comportements élémentaires constituent les briques de base pour construire des comportements plus complexes. Les constructeurs présentés par la suite permettent de combiner ces comportements en les ordonnant et en leur ajoutant des instructions de contrôle du langage réactif. Les comportements élémentaires contiennent l'ensemble des manipulations à apporter à l'icobj construit par l'intermédiaire d'actions atomiques. La création de ces comportements élémentaires ne peut se faire qu'en *Java* en créant ses propres actions atomiques. La structure de données de l'icobj et de son comportement, c'est-à-dire les actions atomiques, les wrap-

pers, les diverses classes d'objet nécessaires et l'apparence sont construits à l'aide de l'API des *Icobjs*. Les principales classes de cette API seront détaillées en 6.1.

Sur la figure 5.3, les icobjs représentés par des apparences de triangles orientés vers la droite ou vers le bas sont des exemples d'icobjs disposant de comportements élémentaires. Ils sont identifiés comme étant des constructeurs, c'est-à-dire que leur comportement non-clonable est exécuté, alors que leur comportement clonable sert uniquement de comportement élémentaire dans les constructions. Pour ces constructeurs, il est important que l'apparence reflète au mieux le comportement.

Par exemple, l'icobj dont l'apparence est un triangle qui pointe vers le bas dispose :

- du comportement clonable suivant :

```
Ic.Repeat(20,
   Ic.Seq(Ic.Atom(new Move(0, 5)), Ic.Stop()))
```

- ainsi que du comportement non-clonable suivant :

```
Ic.Seq(Ic.Atom(new NeedAWTImage("images/down.gif")),
   Ic.Seq(Ic.Await(new IcobjIdentifier()),
      Ic.Repeat(20,
         Ic.Seq(Ic.Atom(new Move(0, 5)), Ic.Stop()))))
```

Nous pouvons constater qu'à l'inverse de son comportement clonable, son comportement non-clonable spécifie son apparence et attend une activation représentée par l'attente d'un événement composé de son identifiant. Ces comportements supplémentaires sont spécifiques à l'icobj et ne seront pas réutilisés dans une construction.

5.2.2 Séquence et composition parallèle

Le constructeur "de base", représenté sur la figure 5.2, permet de construire des séquences ou des compositions parallèles de comportements. Ce constructeur a trois apparences possibles qui traduisent l'état dans lequel il se trouve : soit il est dans son état initial 5.2(a), c'est-à-dire qu'aucune construction n'est en cours ; soit il est *chargé* 5.2(b), c'est-à-dire qu'une construction a commencé. La troisième apparence sera détaillée dans la partie 5.2.7 décrivant la construction de comportements cycliques.

(a) Initial (b) Chargé

FIG. 5.2 – Icobj de construction basique

La sémantique de ce constructeur est la suivante :

- si des icobjs partagent la zone d'influence du constructeur lors de son activation, alors il copie les comportements de chacun des icobjs dont il partage la zone, les met tous en parallèle et stocke cette composition parallèle en séquence avec les constructions précédemment effectuées (s'il n'y a qu'un seul icobj, la composition parallèle est inutile).

- si aucun icobj ne partage la zone d'influence du constructeur lors de son activation, alors la construction est terminée et le constructeur ajoute un nouvel icobj dans le workspace. Le comportement de ce nouvel icobj est le comportement construit et stocké par le constructeur. Ce dernier se réinitialise en mettant `Nothing` comme étant le comportement stocké. Le constructeur est alors prêt pour démarrer une nouvelle construction.

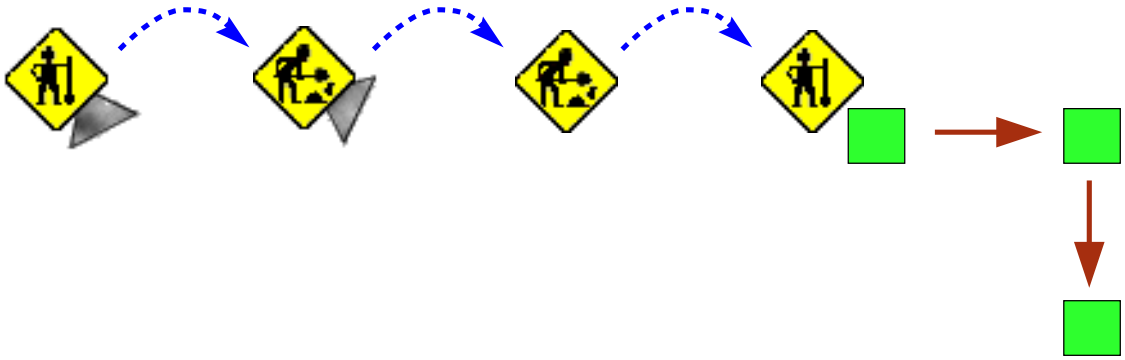


FIG. 5.3 – Construction d'une séquence

La figure 5.3 présente la construction d'un icobj dont le comportement est d'effectuer d'abord un déplacement vers la droite, puis vers le bas. La création de ce comportement consiste à activer d'abord le constructeur sur l'icobj dont le comportement est d'aller vers la droite, puis sur l'icobj dont le comportement est d'aller vers le bas, et enfin dans une zone sans icobj. Cette suite d'activation ajoute un nouvel icobj dans le workspace. L'icobj dont l'apparence est un carré exécute alors immédiatement son comportement.

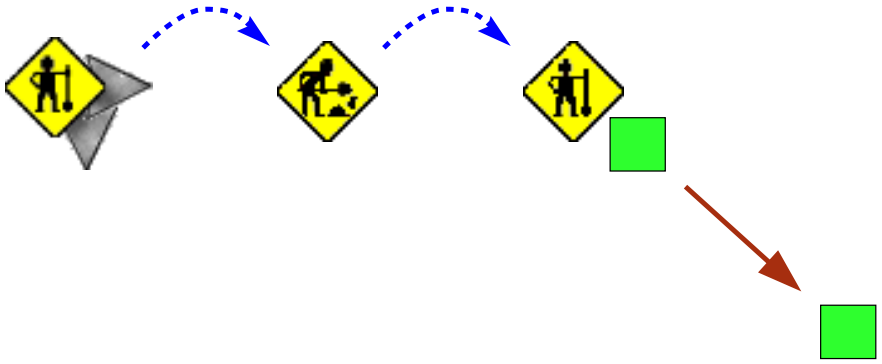


FIG. 5.4 – Construction d'une combinaison parallèle

La figure 5.4 présente la construction d'un icobj dont le comportement est d'effectuer un déplacement en diagonale vers la droite et vers le bas. La création de cet icobj consiste à activer le constructeur sur les deux icobjs à la fois (déplacement vers le bas et déplacement vers la droite), puis dans une zone sans icobj, ce qui crée l'icobj avec le comportement désiré.

Remarque : Si un des icobjs utilisés dans la construction est un workspace, alors l'icobj résultant de cette construction sera aussi un workspace.

Tous ces nouveaux icobjs peuvent servir à leur tour dans une nouvelle construction. Il faut bien noter que le constructeur ne copie ni l'état de l'icobj (l'ensemble de ses champs), ni son comportement au moment de la copie. Il copie uniquement le comportement initialement défini dans les champs de l'icobj contenant son comportement. De plus, le constructeur utilise uniquement sa partie *clonable*. Nous pouvons observer cela sur les exemples des figures 5.3 et 5.4 où l'icobj créé n'a pas du tout la même apparence que les icobjs utilisés dans sa construction. Le code présenté dans la partie 5.2.1 confirme que le comportement clonable ne définissait pas l'apparence de l'icobj qui est définie dans le comportement non-clonable. L'apparence du nouvel icobj n'étant pas définie dans son comportement, celui-ci prend une apparence par défaut qui est définie par le workspace. Dans notre cas, c'est un carré.

Il est à noter que le constructeur de base ne dispose pas de comportement *clonable* (ou plutôt celui-ci est `Ic.Nothing()`).

Pour faciliter l'utilisation de ce constructeur, nous avons ajouté le moyen d'interrompre une construction en réinitialisant le constructeur. Pour cela, il suffit de sélectionner le constructeur et de presser sur la touche *Echap*.

Ce constructeur permet de créer aisément des constructions simples en ordonnant des comportements élémentaires. Cependant, pour profiter au maximum du langage réactif, d'autres constructeurs ont été rajoutés. Ceux-ci se rapprochent plus de la programmation, mais ils s'utilisent néanmoins assez facilement sans avoir à connaître de syntaxe.

5.2.3 Nommage

Un des problèmes rencontrés après plusieurs constructions d'icobjs sans apparence distincte est de pouvoir reconnaître le comportement de chacun des icobjs construits. C'est le cas, par exemple, lorsqu'on veut créer un icobj au comportement complexe et que plusieurs icobjs intermédiaires sont construits pour servir de briques élémentaires pour la construction de l'icobj final.

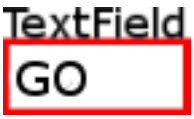


FIG. 5.5 – Icobj de nommage

Pour résoudre ce problème et donc pour identifier le comportement de chaque icobj, nous avons introduit l'icobj *TextField* représenté sur la figure 5.5. Cet icobj est en fait une interface de texte qui permet d'affecter un nom à un icobj.

L'utilisation de cet icobj est assez simple. Pour spécifier le nom, il suffit simplement de sélectionner l'icobj *TextField* et de taper le nom au clavier, comme pour toute interface de texte classique. Ensuite, pour affecter le nom à un icobj, il y a deux moyens :

- soit le constructeur de "base" est activé sur l'icobj *TextField* pendant une construction et l'icobj résultant de celle-ci sera alors directement nommé lors de sa création.
- soit l'icobj *TextField* est directement activé sur l'icobj que l'on souhaite renommer.

Lorsque l'icobj *TextField* est sélectionné, la pression de la touche *BackSpace* permet d'effacer le dernier caractère entré et la pression sur la touche *Echap* efface tous les caractères

entrés. Nous verrons dans la suite que plusieurs constructeurs utilisent l'icobj *TextField* pour paramétrer les comportements qu'ils créent.

5.2.4 Génération et attente d'événements

Les *Icobjs* utilisent deux moyens de communication. Le premier est le partage des zones d'influence qui est utilisé principalement par les constructeurs. Le second est l'utilisation de la diffusion d'événements qui permet, dans le cadre graphique, à plusieurs icobjs de se synchroniser.

L'icobj *Await*, représenté par un poste radio 5.6(b), permet d'attendre un événement donné en paramètre de l'icobj (sur la figure, il s'agit d'*event*). Pour définir le nom de l'événement, il faut utiliser l'icobj *TextField*. Pour cela, il suffit d'entrer le nom de l'événement dans l'icobj *TextField* et d'activer l'icobj *Await* sur ce dernier. Cette opération modifie le comportement clonable de l'icobj *Await* en le remplaçant par `Ic.Await("nouveau␣nom")`. Si le constructeur de base est activé sur cet icobj, il récupérera alors un comportement d'attente sur ce nouvel événement.

(a) Icobj Generate (b) Icobj Await

FIG. 5.6 – Icobjs de génération et attente d'événement

L'icobj *Generate*, représenté par un microphone 5.6(a), permet de générer un événement donné en paramètre de l'icobj. De la même manière que pour l'icobj *Await*, cet événement peut être modifié en utilisant l'icobj *TextField*. Dès que l'icobj *Generate* est activé sur l'icobj *TextField*, son comportement clonable devient `Ic.Generate("nouveau␣nom")`. Si l'icobj *Generate* est activé alors qu'il ne se trouve pas dans la zone d'influence de l'icobj *TextField*, il génère directement l'événement dans le workspace.

Dans le cadre des *Icobjs*, nous avons choisi de ne pas donner de moyens de générer des événements valués et des réactions aux générations valuées. Ces comportements doivent être définis par les comportements élémentaires. Ces constructions nécessiteraient des manipulations peu intuitives, ce qui est contraire à l'esprit des constructions graphiques. La gestion graphique des événements sert donc uniquement à synchroniser différents comportements.

5.2.5 Préemption

Le constructeur de préemption représenté par un réveil 5.7 permet de créer des comportements préemptables. La première étape est de définir l'événement de préemption. Pour cela, nous utilisons l'icobj *TextField* dans lequel le nom de l'événement est saisi. Ensuite, le constructeur de préemption est activé sur l'icobj *TextField* pour récupérer l'événement. La suite de la construction se passe comme avec le constructeur de base 5.2.2 pour créer des séquences, des combinaisons parallèles, puis enfin l'icobj final.

Le constructeur de préemption est utile pour définir un changement de comportement de l'icobj créé. Il est ainsi possible de contrôler le comportement exécuté par l'icobj au moyen

(a) Initial (b) Chargé

FIG. 5.7 – Icobj de préemption

de générations événementielles. Tout comme le constructeur de base, ce constructeur a une apparence différente selon qu'il a commencé ou non une construction. Il ne dispose pas non plus de comportement clonable. De plus, une construction commencée peut être annulée. En pressant sur la touche *Echap*, le constructeur réinitialise le comportement stocké à `Nothing`.

La différence avec le constructeur de base est que le comportement réactif créé est encapsulé dans une instruction `Kill`. Au contraire des implémentations précédentes des *Icobjs*, nous utilisons l'instruction `Kill` et non l'instruction `Until`. Cette instruction a été spécifiquement introduite dans le moteur pour obtenir une plus grande régularité des comportements et ainsi de permettre une construction graphique plus intuitive. Par exemple, deux comportements de préemption placés en séquence et utilisant le même événement de préemption, comme sur le code suivant, ne peuvent pas être préemptés simultanément lors du même instant.

```
Ic.Loop(
   Ic.Seq(
      Ic.Kill(new IcobjIdentifier("change"),
         new PreyBehavior()),
      Ic.Kill(new IcobjIdentifier("change"),
         new PredatorBehavior())))
```

L'instruction `Kill` garantit qu'un comportement placé en séquence avec une instruction `Kill` sera, en cas de préemption, toujours exécuté à l'instant suivant celui de la préemption. Cette régularité du comportement permet une construction graphique plus intuitive, puisqu'il n'est plus nécessaire de tenir compte du comportement construit par le constructeur de préemption. De plus, la construction graphique n'utilise pas le `handler` de l'instruction `Kill` qui est peu intuitif.

Enfin, il faut noter que l'utilisation du constructeur de préemption est différente de celle des implémentations précédentes des *Icobjs*. En effet, dans ces dernières, dès que ce constructeur était activé sur un autre icobj, un nouvel icobj était automatiquement créé avec comme comportement celui du précédent encapsulé dans l'instruction de préemption (`Until`). Nous avons fait le choix de manipuler le constructeur de préemption comme le constructeur de base pour plus de régularité dans la manière d'utiliser les constructeurs graphiques.

5.2.6 Contrôle

Nous avons ajouté le constructeur de contrôle, représenté par un robinet 5.8 qui permet de conditionner l'exécution d'un comportement sur présence d'un événement. L'utilisation de ce constructeur est similaire à celle du constructeur de préemption. Il faut d'abord initialiser le nom de l'événement de contrôle par l'intermédiaire de l'icobj *TextField*, puis créer son

comportement, comme avec le constructeur de base. Ce constructeur n'existait pas dans les versions précédentes des *Icobjs*.

(a) Initial (b) Chargé

FIG. 5.8 – Icobj de contrôle

Le comportement issu de cette construction est encapsulé dans une instruction `Control`. Ce constructeur peut aussi annuler une construction. Comme pour le constructeur de base et le constructeur de préemption, il suffit de sélectionner le constructeur et de presser la touche *Echap*, ce qui réinitialise le comportement stocké par ce constructeur à `Nothing`. Ce constructeur ne comporte pas non plus de comportement clonable.

5.2.7 Construction de boucle

Bien que certains comportements élémentaires puissent être eux-mêmes cycliques, les *Icobjs* proposent un moyen de construire des comportements cycliques sous forme de boucles finies ou infinies. Par défaut, le constructeur de boucle 5.9 crée des boucles infinies (annotation *infinite*).

FIG. 5.9 – LoopIcobj

La construction de boucles finies passe de nouveau par l'utilisation de l'icobj *TextField* qui permet de définir le nombre d'itérations souhaité. Si le constructeur de boucle est activé sur l'icobj *TextField*, son compteur de boucle est initialisé avec la valeur entrée dans le *TextField* si cette valeur est un nombre positif. Pour revenir à des boucles infinies, il suffit soit d'utiliser le *TextField* en entrant le texte *infinite*, soit de presser la touche *Echap* quand le constructeur est sélectionné.

La manière d'utiliser ce constructeur diffère de celle des constructeurs précédents. Ce constructeur est utilisé comme une balise de début et de fin de boucle par les constructeurs de base, de préemption et de contrôle. En particulier, comme le montrent les figures 5.10 et 5.11, l'activation du constructeur de base sur le constructeur de boucle a pour effet de changer l'apparence du constructeur de base. Ce changement traduit aussi une modification du comportement du constructeur. En effet, à partir du moment où le constructeur a l'apparence de la figure 5.10, tous les comportements qu'il va copier sont placés dans une boucle finie ou infinie selon la valeur du compteur du constructeur de boucle.

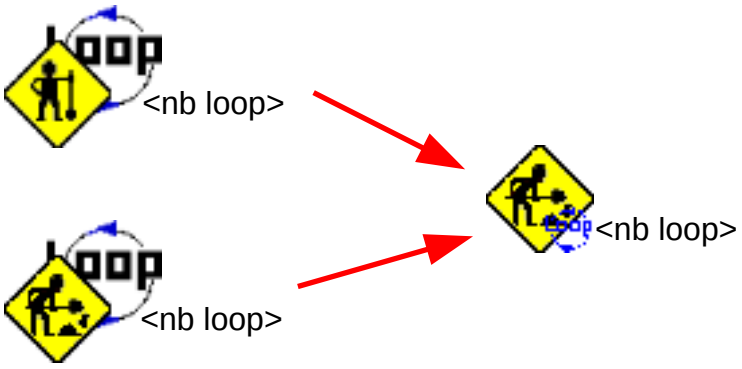


FIG. 5.10 – Débuter une boucle

Si on ré-active une deuxième fois le constructeur de base sur le constructeur de boucle, comme sur la figure 5.11, le comportement cyclique construit par le constructeur de base est alors clos, c'est-à-dire que tous les comportements copiés depuis le début de la construction cyclique sont encapsulés par une instruction `Loop` si la boucle est infinie ou par `Repeat` si cette boucle est finie. L'utilisation du constructeur de base reprend alors normalement. Tous les comportements copiés par la suite seront placés en séquence de cette boucle. Il faut noter que si le comportement cyclique est infini, il est inutile de placer des comportements en séquence, puisqu'ils ne seront jamais exécutés.

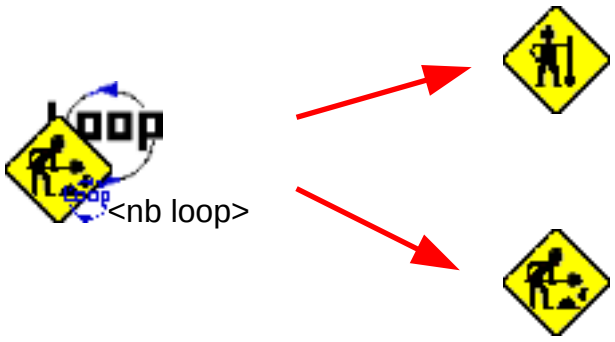


FIG. 5.11 – Finir une boucle

Si le constructeur de base crée un nouvel icobj alors que la boucle n'est pas close, le comportement cyclique comprendra tous les comportements copiés depuis le début de la construction cyclique. Dès que l'icobj est créé, le constructeur de base reprend son apparence initiale. Pour construire de nouvelles constructions cycliques, il faudra à nouveau l'appliquer sur le constructeur de boucle.

Dans le cas des constructeurs de préemption ou de contrôle, la boucle sera encapsulée dans respectivement l'instruction de préemption ou l'instruction de contrôle. La figure 5.12 présente les apparences prises par les constructeurs de préemption et de contrôle lorsqu'ils sont activés une première fois sur le constructeur de boucle.

Dans les implémentations précédentes des *Icobjs*, il suffisait d'activer une seule fois le

(a) Préemption (b) Contrôle

FIG. 5.12 – Autres constructeurs de comportements cycliques

constructeur de base sur le constructeur de boucle pour que tous les comportements copiés soient placés dans la boucle.

Dans la section 2.2.2, nous avons posé le problème des boucles instantanées. Dans l'implémentation de notre moteur, nous avons choisi de ne pas utiliser une heuristique, comme le font les *SugarCubes* et certaines implémentations de *Junior*, pour détecter et bloquer de telles boucles. Nous préférons laisser à l'utilisateur la possibilité de coder son comportement comme il le souhaite. Par contre, dans le cadre d'une programmation réactive graphique, il faut s'en prémunir. En effet, dans la programmation par *Icobjs*, nous nous éloignons au maximum de la notion d'instant et nous ne voulons pas voir surgir ce type de problème qui pourrait déconcerter l'utilisateur.

Pour éviter les problèmes de boucle instantanée, les comportements copiés sont placés automatiquement en parallèle avec l'instruction `Stop` avant d'être encapsulés dans l'instruction `Loop`. Les règles de l'instruction `Par` (cf. section 3.7) garantissent qu'ainsi le corps de la boucle ne peut pas finir instantanément. En effet, pour qu'une instruction `Par` se termine, il faut que toutes ses branches soit terminées. Or, le fait qu'une des branches de l'instruction `Par` soit une instruction `Stop` garantit que même si les comportements copiés se terminent instantanément, l'instruction `Par` ne terminera jamais instantanément.

5.2.8 Interruption d'une séquence

FIG. 5.13 – Icobj Stop

L'icobj *Stop* 5.13 a été introduit pour permettre d'interrompre une séquence d'instructions jusqu'à l'instant suivant. L'utilisation première de cet icobj est d'interrompre une séquence d'attentes ou une séquence de générations. En effet, si un comportement doit attendre deux générations d'un même événement, il est nécessaire d'introduire une instruction `Stop` pour séparer ces deux attentes. De même, si un même événement doit être généré pendant plusieurs instants consécutifs, il faut séparer ces générations par une instruction `Stop`. Dans le cadre de comportements contrôlés, l'icobj *Stop* permet aussi de séparer les comportements correspondants à chaque génération de l'événement. Son utilisation est cependant très proche

de la programmation et reste assez rare.

Nous pouvons noter qu'il n'est pas nécessaire d'utiliser cet icobj dans le cas d'une boucle infinie de générations d'événements car les comportements cycliques sont protégés contre les boucles instantanées (une instruction `Stop` est placée en parallèle de la génération).

5.2.9 Workspace

Une autre nouveauté dans notre implémentation des *Icobjs* est l'introduction du workspace en tant qu'icobj. Un workspace peut être considéré comme une fenêtre graphique qui peut être minimisée. La figure 5.14(b) représente l'apparence d'un workspace quand il est minimisé et la figure 5.14(a) son apparence normale. Nous pouvons observer sur cette dernière figure que le workspace contient le constructeur de base et deux icobjs élémentaires.

Le workspace dispose d'un comportement de contrôle supplémentaire par rapport aux icobjs. On passe d'un affichage minimisé à un affichage normal et inversement en pressant la touche *D* lorsque le workspace est sélectionné.

(a) Normal (b) Minimisé

FIG. 5.14 – Icobj Workspace

Nous avons également ajouté un moyen graphique permettant à l'utilisateur de transférer des icobjs d'un workspace à un autre. Pour cela, le comportement de contrôle ajouté à chacun des icobjs (et au workspace évidemment) permet, en plus des comportements cités dans la section 5.2, de :

- sortir d'un workspace en déplaçant l'icobj avec le bouton gauche de la souris et en maintenant la touche *Ctrl* enfoncée. Cette sortie a pour effet de placer l'icobj dans le workspace de plus haut niveau.
- entrer dans un workspace en déplaçant l'icobj sur le workspace dans lequel il doit être placé et en relâchant le bouton gauche de la souris tout en maintenant la touche *Ctrl* enfoncée.

5.3 Environnement des *Icobjs*

Un des principaux manques dans le mécanisme de construction graphique des versions antérieures des *Icobjs* est qu'une fois une construction effectuée, il n'y a aucun moyen pour vérifier s'il n'y a pas eu d'erreurs dans la construction. Non seulement, il n'est pas possible de

visualiser son comportement, mais le seul moyen de corriger une erreur est de recommencer entièrement la construction graphique depuis son début. Il manque également un moyen de paramétrer les comportements et d'agir directement sur leur paramètres d'exécution.

Dans notre version des *Icobjs*, nous avons mis en place, pour répondre à ce problème, une interface qui permet d'inspecter les icobjs. Elle permet d'inspecter à la fois les champs et les comportements des icobjs. Nous allons maintenant décrire une application, `Framework`, qui permet d'utiliser cet outil d'inspection.

Un autre problème des versions précédentes est que tout icobj construit graphiquement est nécessairement perdu. Les *Icobjs* ne disposait d'aucun moyen pour enregistrer les constructions ou l'état même d'une application. Ce problème a aussi été réglé. Cette fonctionnalité est accessible dans l'application `Framework`.

5.3.1 Présentation du framework

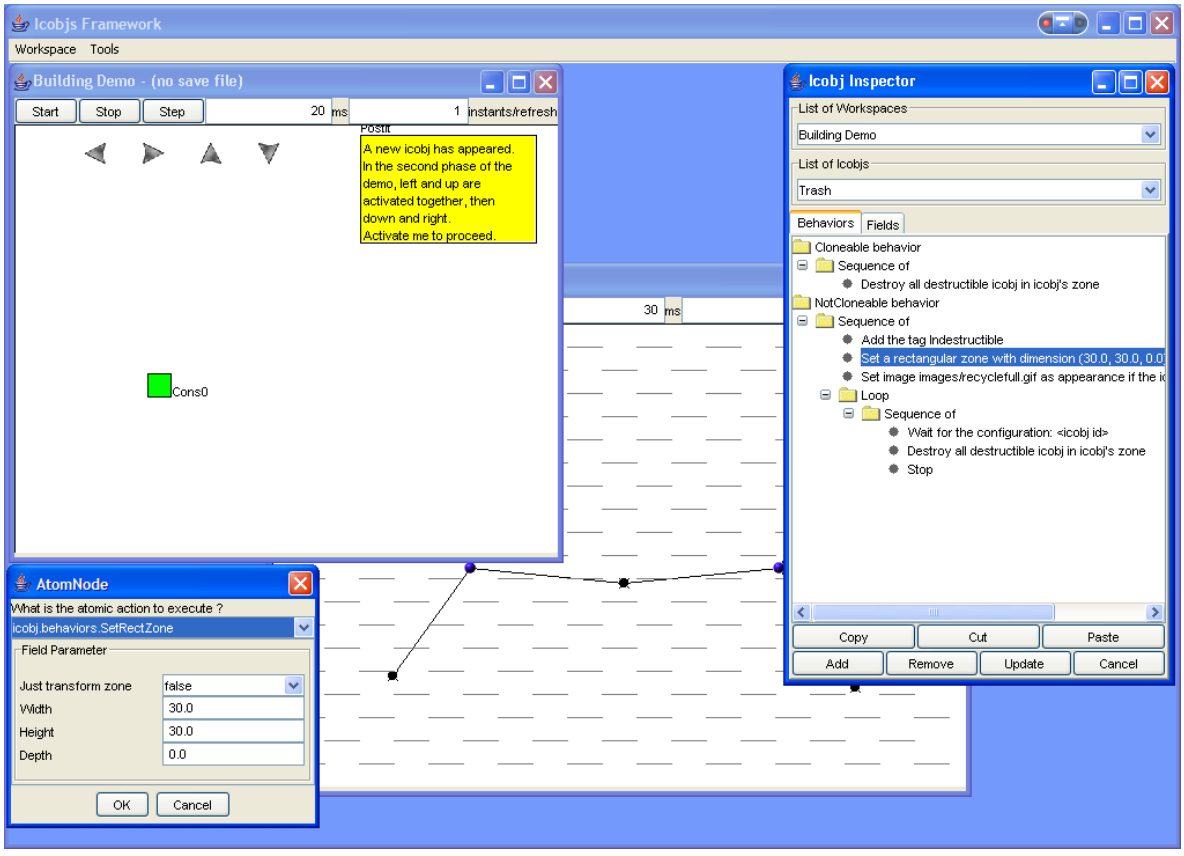


FIG. 5.15 – Capture d'écran du framework

L'application `Framework`, dont on peut voir une capture d'écran sur la figure 5.15, a pour objectif d'être un environnement de développement (IDE) pour les *Icobjs*. `Framework` permet actuellement d'exécuter plusieurs simulations dans un environnement multi-fenêtré. Les fonctionnalités offertes par l'application `Framework` sont :

- d'ouvrir une fenêtre dans laquelle un workspace est exécuté. Ce workspace ne contient

aucun icobj. Cela permet de créer graphiquement des simulations.
- de charger les classes `BootWorkspace` décrivant des simulations que nous détaillerons dans la partie 6.2.
- d'enregistrer dans un fichier ou de charger à partir d'un fichier une simulation.
- d'ajouter un icobj dans une simulation ouverte dans le `Framework`.
- d'inspecter les comportements et la structure de données de chacun des icobjs présent dans une des simulations ouvertes dans le `Framework`. L'inspecteur des *Icobjs* permet à la fois d'interagir directement avec la structure de données en modifiant les valeurs des paramètres d'exécution, et de visualiser et modifier le comportement d'un icobj. L'utilisation de l'inspecteur des *Icobjs* sera détaillée dans la partie 5.3.2 pour ce qui concerne l'interaction avec la structure de données d'un icobj et dans la partie 5.3.3 pour ce qui concerne la visualisation et la modification des comportements.

De plus, l'exécution de chaque simulation peut être contrôlée. En effet, sur chaque fenêtre de simulation, des boutons permettent soit d'exécuter la simulation en continu, soit de l'exécuter pas à pas, soit enfin de la suspendre. Il est également possible de définir le pas d'exécution, c'est-à-dire le nombre de réaction du workspace entre chaque rafraîchissement. De plus, la durée minimale (en millisecondes) d'un pas d'exécution peut être modifiée, ce qui permet, en partie, de contrôler la rapidité et la fluidité de la simulation.

5.3.2 Inspection des champs

La figure 5.16 présente une capture d'écran de l'inspecteur des *Icobjs* et plus particulièrement de la partie permettant d'agir directement sur les données des icobjs.

Dans la partie haute de cet inspecteur, deux listes de choix permettent de sélectionner :
- un workspace parmi tous ceux qui sont chargés dans les différentes simulations. Cette liste contient à la fois les workspaces de plus haut niveau et les workspaces imbriqués.
- un icobj parmi ceux qui se trouvent dans le workspace sélectionné dans la première liste.

L'icobj sélectionné dans cette deuxième liste est celui dont la structure de données est affichée dans la partie basse de l'inspecteur. Cet icobj devient l'icobj sélectionné dans son workspace. Pour chaque icobj, on a au minimum accès à son nom, son statut qui définit si l'icobj est un constructeur, son apparence et sa zone d'influence. Cette liste de paramètres peut s'agrandir en fonction des champs disponibles dans chacun des icobjs. Il y a plusieurs types de champs possibles : booléen, chaîne de caractères, nombre, couleur, fichier...

L'inspecteur représente différemment chacun de ces types : par exemple, il utilise une liste à deux choix pour un booléen, un bouton coloré pour sélectionner une couleur qui, quand on clique dessus, ouvre un panneau de sélection de couleurs. Sur la figure 5.16, on peut remarquer des boutons nommés, comme par exemple le bouton *Zone*. Ils définissent des champs composites, c'est-à-dire des objets *Java* composés eux-mêmes de plusieurs objets paramétrables qu'il est possible d'inspecter et de paramétrer en cliquant sur le bouton du champ. Il y a deux possibilités, soit de nouveaux champs présentant les paramètres de l'objet apparaissent, soit il ne se passe rien. Dans ce dernier cas, il y a deux possibilités : soit l'objet représenté par le bouton n'est pas paramétrable, soit ce dernier ne permet pas d'accéder à ses propres champs. Chaque classe d'objet utilisé comme champ d'un icobj doit définir ses propres champs paramétrables et leur type pour pouvoir les paramétrer dynamiquement. Nous détaillerons cela dans la partie 6.5.

Tous les champs de l'icobj sélectionné sont mis à jour entre deux pas d'exécution de la simulation qui contient cet icobj. La mise à jour inter-instant évite les interférences pendant

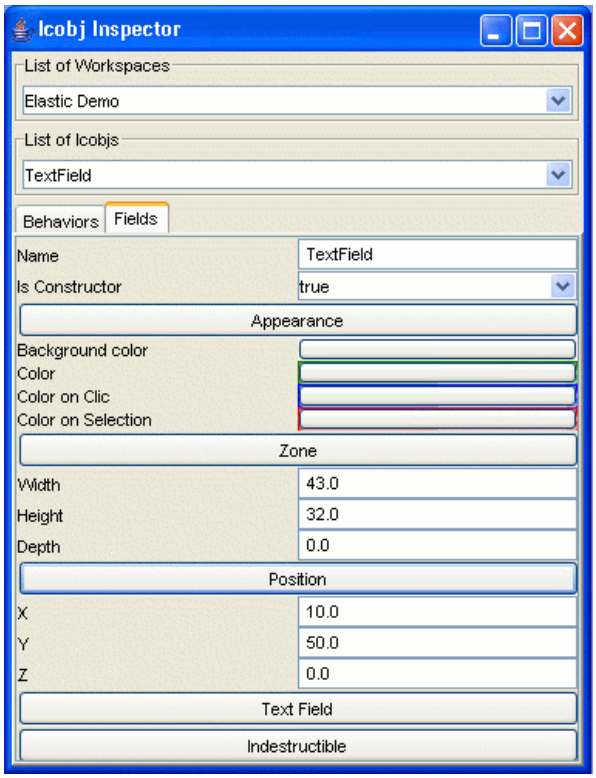


FIG. 5.16 – Paramétrage des champs

l'exécution du comportement des icobjs. Il faut noter que l'inspecteur de champs ne peut pas ajouter des champs à l'icobj directement, il ne permet que d'agir sur les champs existants. Si l'on désire ajouter des champs à un icobj, il faut passer par l'intermédiaire du comportement de l'icobj. Pour cela, il faut utiliser la partie comportementale de l'inspecteur que nous allons détailler maintenant.

5.3.3 Inspection des comportements

La figure 5.17 présente une capture d'écran de la partie de l'inspecteur des *Icobjs* permettant de visualiser les comportements clonable et non-clonable de l'icobj sélectionné.

Les comportements sont représentés par des arbres dont chaque nœud représente une instruction réactive. Il y a différents types de nœuds :

- les nœuds feuilles qui regroupent toutes les instructions terminales : `Atom`, `Stop`...
- les nœuds à un fils qui regroupent toutes les instructions unaires : `Control`, `Loop`...
- les nœuds à deux fils qui regroupent toutes les instructions binaires : `Kill`, `When`...
- les nœuds à n fils qui regroupent les instructions n-aires : `Par` et `Seq`. On y adjoint aussi l'instruction `Nothing` considérée comme une instruction `Par` (ou `Seq`) sans aucune sous-instruction.

L'objectif de cette représentation arborescente est de fournir une spécification textuelle du comportement de l'icobj lorsque la représentation graphique n'est pas suffisante pour comprendre son comportement. De plus, l'inspecteur permet de modifier graphiquement le

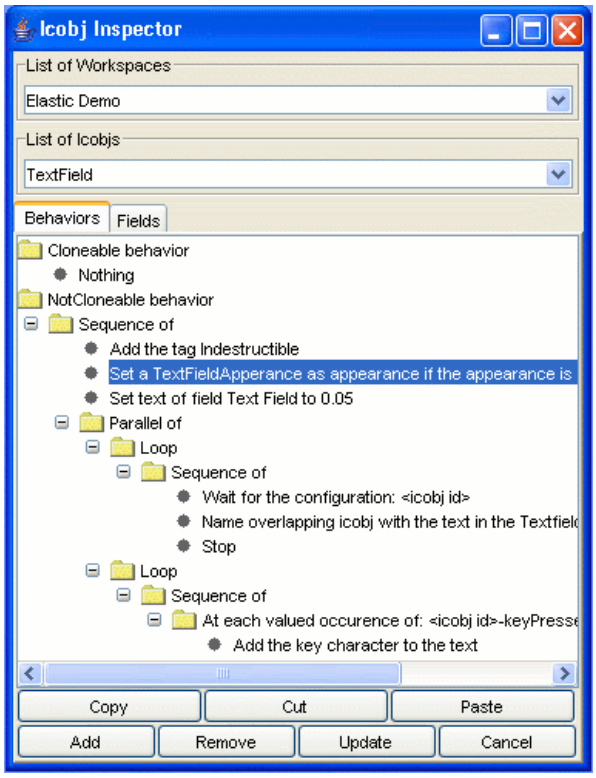


FIG. 5.17 – Inspecteur de comportement

comportement des icobjs si nécessaire. Ces modifications ne sont pas directement appliquées sur l'icobj, mais sur une copie de ses comportements. Il faudra demander explicitement la mise à jour de ces comportements pour que la simulation en tienne compte. Sous la zone d'affichage des arbres, nous pouvons remarquer plusieurs boutons :

- les boutons *Copy, Cut, Paste* permettent comme leur nom l'indique d'utiliser les fonctions classiques de copier/couper/coller sur les nœud de l'arbre. Les opérations couper-coller peuvent également être effectuées par un glisser-déplacer avec la souris, en déplaçant les branches d'un nœud à l'autre.
- le bouton *Add* permet d'ajouter une nouvelle instruction ou un nouveau programme au nœud de l'arbre sélectionné. Lorsqu'on clique sur ce bouton, une boîte de dialogue contenant une liste de programmes apparaît. Cette liste regroupe l'ensemble des classes qui étendent la classe `Instruction` de *Junior*. Il ne reste alors qu'à choisir l'instruction (ou le groupe d'instructions) à ajouter au nœud sélectionné. Ces ajouts sont contraints. Si, par exemple on essaie d'ajouter une instruction à un nœud feuille, alors cette instruction sera ajoutée au nœud parent du nœud sélectionné. Il est à noter que, par défaut, les nœuds fils d'un nœud représentant une instruction unaire ou binaire sont des nœuds n-aires (parallèle ou séquence).
- le bouton *Remove* retire le nœud sélectionné. Tous les nœuds situés sous le nœud sélectionné sont également retirés.
- le bouton *Cancel* réinitialise les copies des comportements clonable et non-clonable par une copie des comportements initiaux de l'icobj.

- le bouton *Update* remplace les comportements clonable et non-clonable de l'icobj sélectionné par les comportements clonable et non-clonable décrits dans l'arbre. Cette mise à jour entraîne une réinitialisation de l'icobj. Le comportement exécuté par le workspace est stoppé et remplacé par le nouveau.

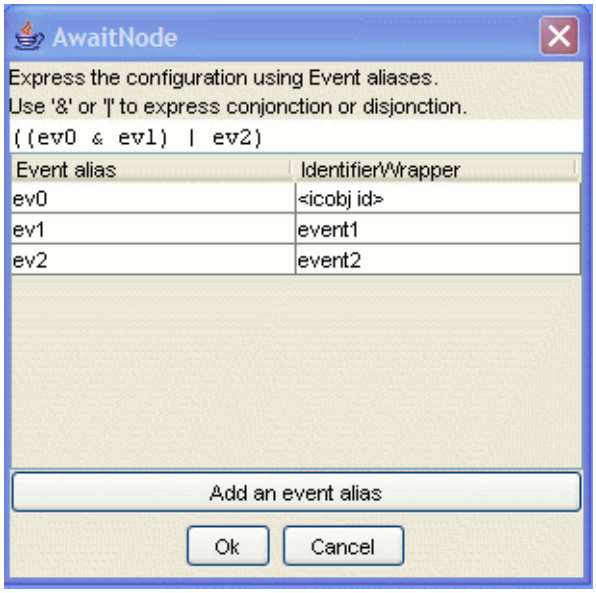


FIG. 5.18 – Paramétrer une configuration événementielle

Il est enfin possible de paramétrer chacun des nœuds en cliquant dessus avec le bouton droit de la souris.

- un clique sur un nœud `Par` ou `Seq` affiche une boîte de dialogue permettant de choisir le mode d'ordonnancement : séquence ou composition parallèle des sous-instructions de ce nœud.
- un clique sur un nœud `Atom` ou `ScanAction` permet de modifier l'action atomique exécutée ou de modifier la valeur des paramètres de cette action.
- un clique sur un nœud `Stop` ou `Loop` n'aura aucun effet.
- un clique sur un nœud `Repeat` (ou `If`) permet de choisir le wrapper d'entier (ou wrapper de booléen) à utiliser ou de modifier les valeurs des champs paramétrables du wrapper sélectionné.
- un clique sur un nœud représentant une instruction événementielle (comme `Await`, `Kill`...) affiche une boîte de dialogue 5.18 qui permet de modifier la configuration événementielle. Elle permet de définir une expression booléenne en utilisant les alias d'événements comme référence et les caractères `&` et `|` pour représenter respectivement une conjonction ou une disjonction d'événements. Chaque alias d'événement correspond à un wrapper d'événement paramétrable qui peut être modifié en cliquant dessus.

Il y a des limitations à ce paramétrage. L'instruction `Generate` peut utiliser comme valeur associer à une génération n'importe quel objet présent dans le système. Nous n'avons pas de moyens pour lister tous les objets potentiellement utilisables. C'est pourquoi, il est de la responsabilité de l'utilisateur de créer les wrappers d'objets nécessaires à la récupération de ces objets s'il désire configurer l'instruction `Generate`.

L'ajout de comportement par l'intermédiaire de l'inspecteur est bien moins intuitive qu'en utilisant les constructeurs graphiques et nécessite une connaissance des primitives réactives. Elle est donc complémentaire à la construction graphique et réservée à des utilisateurs plus expérimentés.

5.4 Bilan

Le système de construction graphique n'a pas été l'objet de grandes modifications depuis les versions précédentes. L'utilisation reste quasiment la même, à l'exception de l'icobj *Loop* dont l'utilisation a été légèrement complexifiée pour permettre une granularité plus fine dans la création de comportements et pour éviter de créer trop de constructions intermédiaires.

Les principales nouveautés consistent en l'ajout du constructeur correspondant à l'instruction `Control` et surtout en la possibilité d'utiliser et de construire des workspaces. Ce dernier ajout permet de considérer les workspaces comme des icobjs à part entière et de définir des sous-ensembles de construction. Il est également possible d'effectuer des constructions et de les tester dans un environnement local indépendant de l'environnement de construction graphique global.

Il est également possible d'utiliser l'inspecteur des *Icobjs* pour visualiser et paramétrer à la fois les champs d'un icobj et son comportement. Le système de construction graphique peut ainsi être étendu en utilisant l'inspecteur des *Icobjs* pour construire le comportement d'un icobj. Ce type de construction nécessite cependant des connaissances supplémentaires quant à l'utilisation des instructions.

L'application `Framework` permet de tester chaque simulation, de construire de nouvelles simulations graphiques en y ajoutant des icobjs, et d'enregistrer ou de charger des simulations sauvegardées dans des fichiers. Toutes ces fonctionnalités font du `Framework` un éditeur de scénarios dans lequel chaque comportement peut être modifié dynamiquement en ajoutant de nouveaux comportements aux entités ou en agissant sur les paramètres d'exécution de ces entités ou de leur simulations.

Le site des *Icobjs* [81] comporte un tutorial expliquant comment utiliser le système de construction graphique par l'intermédiaire d'applets et comment construire ses propres icobjs en utilisant l'API disponible sur le site.

Chapitre 6

Implémentation des *Icobjs*

Nous venons de décrire le modèle et les principes du mécanisme de construction graphique des *Icobjs* en détaillant le moyen mis en place pour visualiser et modifier le comportement de chaque icobj et pour intervenir sur les paramètres d'exécution. Nous allons maintenant passer à l'implémentation des *Icobjs*.

Il y a eu deux implémentations des *Icobjs*. La première [12] était une version en *Tcl/tk* [57] au-dessus des *Reactive Scripts* [16], ces scripts étant implémentés au-dessus du langage *Reactive-C* [11]. La seconde implémentation était réalisée en *Java* au-dessus des *SugarCubes* [13]. L'utilisation de ces deux implémentations ne nécessitait de connaître ni les *Reactive Scripts* ou *Reactive-C*, ni les *SugarCubes*. L'inconvénient de ces deux versions est que leur notion d'*Icobjs* n'était pas basée sur un modèle clair. Il s'agissait surtout d'expérimentations. Le modèle n'était en fait pas aisément utilisable pour quelqu'un qui souhaiterait créer ses propres simulations. De plus, il n'y avait pas d'API claire pour permettre de créer ses propres icobjs ou ses propres comportements.

Nous allons maintenant présenter les points essentiels de notre implémentation des *Icobjs* en détaillant, dans la section 6.1, la structure commune à tous les icobjs et les problèmes liés à l'extension de cette structure. Ensuite, nous étudierons les workspaces qui sont les entités qui exécutent le comportement des icobjs. Nous distinguons deux types de workspace : la classe `Workspace` qui est un icobj et la classe `BootWorkspace` qui exécute le workspace de plus haut niveau par l'intermédiaire d'un thread *Java* autonome. Nous étudierons ces deux types de workspace sous plusieurs aspects. La section 6.2 présentera la gestion des icobjs et de leur exécution. La section 6.3 détaillera la manière dont les événements souris et clavier sont gérés par les workspaces. La section 6.4 présentera les détails liés à l'affichage des simulations graphiques. Ensuite, nous décrirons les mécanismes utilisés par l'inspecteur des *Icobjs* dans la section 6.5. Enfin, avant de conclure en présentant, dans la section 6.7, un tutorial pour construire ses propres simulations et ses propres comportements, la section 6.6 évoquera rapidement l'enregistrement et le chargement de simulation à partir d'un fichier.

6.1 Structure d'un icobj

Dans la section 5.1, nous avons présenté les informations minimales nécessaires dont doit disposer chaque icobj pour être utilisable par notre système de construction graphique. De plus, nous avons également signalé que cette structure minimale pouvait par la suite être étendue selon les besoins. Si un icobj étend sa structure de données et si cet icobj est utilisé

comme brique élémentaire dans le mécanisme de construction graphique, il faudra que l'icobj résultant de la construction dispose des champs nécessaires à son exécution. De plus, tous les icobjs utilisés dans la construction peuvent ou non manipuler les mêmes champs en utilisant le même type d'objet. Notre structure de données doit donc être une structure générique à laquelle de nouveaux champs peuvent être aisément ajoutés, tout en donnant un moyen à différents comportements d'accéder à un même objet si celui-ci a le même rôle dans chacun des comportements.

Il est difficile de créer une telle structure de données en utilisant l'héritage de classe de *Java*. En effet, si l'on compose plusieurs comportements nécessitant chacun une structure de données différentes, il ne sera pas possible de créer une structure de données qui convienne. *Java* ne permet pas non plus le multi-héritage de classes, donc il n'est pas possible de générer des classes héritant de chacun des types d'icobjs entrant dans la construction graphique. De plus, comme le montre la figure 6.1, il n'est pas envisageable de créer une nouvelle classe par composition de comportements car cela risquerait de faire exploser le nombre de classes résultant d'une composition graphique.

Le modèle d'objets réactifs décrit sur la figure 5.1 ne permet pas non plus de répondre à ce problème. L'utilisation de l'instruction `Link` pour associer une structure de données à chacun des sous-comportements réactifs permettrait une composition assez aisée. Le problème est qu'il ne serait pas possible de faire interagir différents comportements qui devraient agir sur le même champ d'un icobj. En effet, les objets liés à chaque comportement seraient indépendants les uns des autres. Par exemple, le comportement inertiel et le comportement de collision nécessitent chacun l'accès à un champ vitesse. Si un champ vitesse était lié à chacun de ces comportements, il n'y aurait aucun partage de données puisque chacun accèderait uniquement à son propre champ.

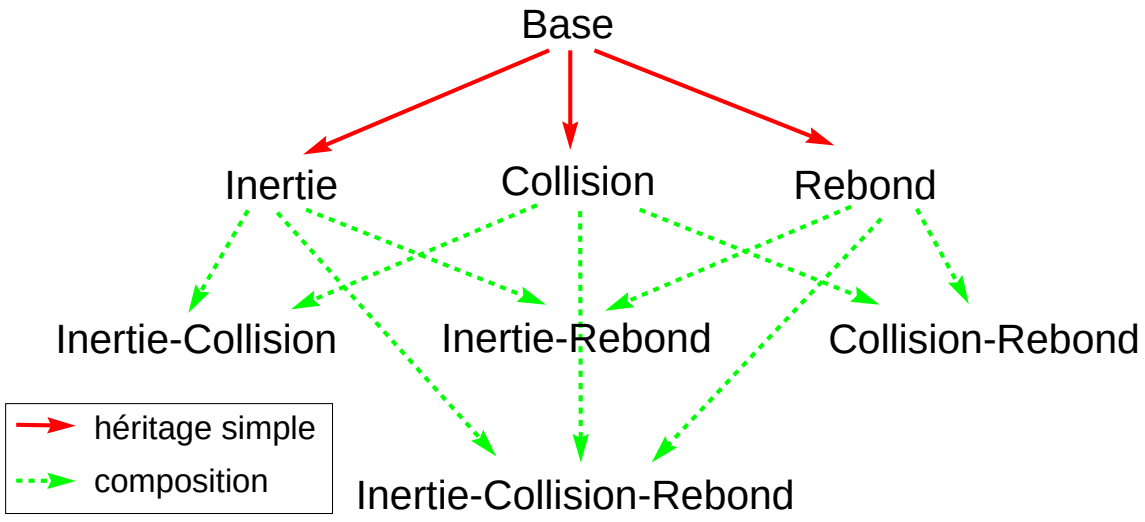


FIG. 6.1 – Problème d'héritage

Les premières versions *Java* des *Icobjs* utilisent l'héritage de classe. En pratique, tous les champs communs à plusieurs comportements finissaient par apparaître dans la classe de base `Icobj` pour éviter les problèmes de structure de données. Or, notre but est d'obtenir une structure de données minimale extensible pour permettre à chacun des comportements

d'avoir accès aux champs nécessaires à son exécution.

```
public class Icobj implements Parametrable
{
   ...
   private transient Workspace workspace;
   private transient String identifier;
   private String name;
   private Zone zone;
   private Appearance appearance;
   private Program cloneable;
   private Program notCloneable;
   private Program frozenBehavior;
   private Hashtable fields;
   private boolean constructor;
   ...
}
```

TAB. 6.1 – Structure d'un icobj

La solution adoptée est l'utilisation d'une table de hachage pour stocker dynamiquement tous les champs nécessaires à chacun des comportements. L'accès à ces champs est fait par l'intermédiaire de variables globales qui identifient chaque type de champ. Cette solution permet de partir d'une structure de données initiale générique implémentée dans la classe `Icobj` (cf. table 6.1). Chaque comportement peut ajouter les champs nécessaires à son exécution et réutiliser les champs déjà existants. Chaque identifiant de champ étant une variable globale, tout comportement qui utilise la même variable globale accèdera au même champ. De plus, la classe `Icobj` permet d'utiliser des champs temporairement et de les retirer dès qu'ils ne sont plus utiles.

Tous les champs contenus dans la structure minimale de l'icobj sont déclarés **`private`**. Il faut donc obligatoirement passer par des méthodes de l'icobj pour atteindre et modifier ces champs. Cela permet entre autre de garantir leur bonne initialisation. Notons que certains champs doivent absolument être initialisés (valeur différente de **`null`**) pour que le modèle fonctionne, c'est par exemple le cas de la zone d'influence de l'icobj (le champ `zone`) ou de l'identifiant de l'icobj (le champ `identifier`). Nous allons maintenant détailler chacune des méthodes disponibles pour manipuler les icobjs.

- `String getIdentifier()`
  Cette méthode retourne une chaîne de caractères représentant l'identifiant de l'icobj. Cet identifiant est unique. Pour garantir l'unicité, l'identifiant est construit en utilisant le moment (le temps en millisecondes) auquel il est construit et un compteur incrémenté à chaque nouvel identifiant. Dans notre cas, comme les icobjs sont potentiellement des objets migrants, un nouvel identifiant est généré à chaque changement de workspace. Cette unicité permet de créer des identificateurs d'événements propres à chaque icobj, comme par exemple les événements clavier ou souris. Ces identificateurs d'événements sont créés par le wrapper d'événement `IcobjIdentifier`. Cette classe construit l'identificateur de l'événement en prenant comme base l'identifiant de l'icobj exécuté en le suffixant par une chaîne de caractère indiquant le type de l'événement traité. Par exemple, l'activation d'un icobj (cf. section 5.2) est faite en générant un événement basé sur l'identifiant de l'icobj suffixé par une chaîne vide. L'utilisation de la

classe `IcobjIdentifier` permet de reconstruire des noms d'événements communs sans avoir à connaître l'identifiant de l'icobj.

- `String getName()`
  `void setName(String name)`
  La première méthode retourne le nom de l'icobj et la seconde le modifie. À l'inverse de l'identifiant de l'icobj, il n'est pas nécessaire que ce nom soit unique.

- `String getWorkspace()`
  `void setWorkspace(Workspace workspace)`
  La première méthode retourne une référence vers le workspace dans lequel l'icobj est enregistré. Cette méthode permet de récupérer les données relatives au workspace, comme par exemple les icobjs qu'il contient pour pouvoir interagir avec eux. Dès qu'un icobj s'enregistre dans un workspace, ce dernier appelle la méthode `setWorkspace` pour changer la référence contenue dans l'icobj. Si l'icobj était déjà enregistré dans un workspace, cette méthode demande son désenregistrement et re-génère l'identifiant de l'icobj. Il est cependant préférable de laisser le workspace se charger d'appeler la méthode `setWorkspace`. En effet, si la référence du workspace est changée avant que l'icobj ait fini de s'exécuter, le comportement risque d'utiliser des données erronées. De plus, puisque l'identifiant de l'icobj est re-généré par l'appel à la méthode `setWorkspace`, les identificateurs d'événements générés à partir de ce moment seront construits sur le nouvel identifiant, alors que les programmes encore exécutés dans la machine réactive sont en attente d'événements dont l'identificateur est formé à partir de l'ancien identifiant de l'icobj.

- `Zone getZone()`
  `void setZone(Zone zone)`
  La première méthode retourne la zone d'influence de l'icobj et la seconde permet d'en définir une nouvelle. La zone d'influence correspond à l'aire (dans un environnement 2D) ou l'espace (dans un environnement 3D) occupé par l'icobj. L'interface `Zone` dispose de méthodes pour définir et modifier les positions et dimensions de l'icobj (qui peuvent être exprimées en deux ou trois dimensions) et pour tester les intersections entre les zones d'influences. Pour effectuer un calcul d'intersections, il est nécessaire de savoir si les icobjs sont dans un environnement 2D ou 3D. Cette information est fournie par la méthode `getDimensionNumber` du workspace.
  L'API des *Icobjs* définit actuellement deux classes qui implémentent l'interface `Zone` : `SphericalZone` et `RectZone`. Ces classes représentent respectivement des zones circulaires (sphériques) ou rectangulaires (en forme de boîte). Initialement, un icobj dispose d'une zone rectangulaire. Une zone doit toujours être initialisée.

- `Appearance getAppearance()`
  `void setAppearance(Appearance app)`
  La première méthode retourne l'objet qui définit l'apparence de l'icobj. Cette méthode est appelée par exemple au moment de l'affichage de l'icobj. La seconde méthode permet de spécifier l'apparence de l'icobj. L'interface `Appearance` ne contient que deux méthodes pour l'instant : une pour le rendu graphique et une pour signaler l'activation de l'icobj. L'API des *Icobjs* contient différents types d'apparences. Par exemple, la classe

`AWTImage` utilise une image chargée à partir d'un fichier comme apparence de l'icobj. Comme pour la zone d'influence, l'apparence peut être en 2D ou 3D. Actuellement, l'implémentation ne dispose que d'apparences en 2D, mais ceci pourra être étendu par la suite en utilisant *Java3D* ou *VRML-X3D*.

- `boolean isConstructor()`
  `void setConstructor(boolean constructor)`
  Ces deux méthodes permettent de lire ou modifier le statut de constructeur d'un icobj. Rappelons que si un icobj est un constructeur, il n'exécute que son comportement non-clonable. Le statut de constructeur doit être modifié avant d'enregistrer l'icobj dans un workspace. Une fois que le comportement est chargé, la modification de ce statut ne provoque plus aucun changement. Si l'on veut changer le comportement exécuté par l'icobj, il sera nécessaire de réinitialiser l'icobj.

- `Program getBehavior()`
  `void setBehavior(Program behav)`
  La première méthode retourne une copie du comportement clonable et la seconde méthode change le comportement clonable de l'icobj en utilisant une copie du programme passé en argument. Ce champ doit toujours être initialisé. Par défaut, sa valeur est `Ic.Nothing()`.

- `Program getNotCloneableBehavior()`
  `void setNotCloneableBehavior(Program behav)`
  Ces méthodes sont similaires aux deux méthodes précédentes, mais elles sont appliquées au comportement non-clonable.

- `void addToBehavior(Program behavior)`
  `void addToNotCloneableBehavior(Program behavior)`
  Ces deux méthodes permettent d'ajouter respectivement la copie du programme passé en argument en parallèle du comportement clonable ou non-clonable. Si l'icobj n'est pas encore enregistré dans un workspace, ces méthodes modifient uniquement le champ. Par contre, si l'icobj est déjà enregistré, alors le programme est ajouté à la fois au champ de l'icobj et dans le workspace en parallèle des programmes déjà exécutés.
  Ces méthodes permettent d'étendre le mécanisme de construction graphique. Il devient maintenant possible d'ajouter des comportements à un icobj déjà existant. Il est possible de concevoir des constructeurs qui, lorsqu'ils sont activés sur d'autres icobjs, leur ajoutent directement un comportement en parallèle.

- `Program getFrozenBehavior()`
  `void setFrozenBehavior(Program behavior)`
  Ces deux méthodes sont utilisées par le workspace. La première est appelée au moment de l'enregistrement de l'icobj dans un nouveau workspace pour tester si le comportement à exécuter doit être construit ou s'il faut utiliser le résidu du comportement exécuté par l'icobj dans son ancien workspace. La méthode `setFrozenBehavior` est appelée par les instructions `IcobjThread` au moment du gel du comportement d'un icobj pour stocker le résidu du comportement. Il est préférable de ne pas appeler directement ces deux méthodes et de laisser le workspace les utiliser.

- `void reinit()`
  Cette méthode réinitialise l'ensemble des champs d'un icobj. Cela signifie que la table de hachage des champs est vidée et que le comportement `frozenBehavior` et l'apparence de l'icobj sont réinitialisés à la valeur `null`. Cette méthode doit être redéfinie dans chaque classe étendant la classe `Icobj` dont un des champs est ajouté immédiatement à la table de hachage de l'objet sans passer par le comportement de l'icobj. Cependant, il est préférable que tous les champs soient systématiquement ajoutés par le comportement.

- `Serializable getValueOfField(String fieldName)`
  `void setValueOfField(String fieldName, Serializable value)`
  `void removeField(String fieldName)`
  Ces trois méthodes permettent de gérer les champs supplémentaires nécessaires à chaque icobj. La première retourne le champ identifié par une chaîne de caractère. La seconde méthode ajoute un nouveau champ dans la table de hachage ou change la valeur d'un champ existant. Enfin, la troisième méthode retire un champ à la structure de données. Les noms des champs doivent être connus par tous les comportements pour que ces derniers puissent utiliser les mêmes objets.

- `void destroy()`
  Cette méthode appelle la méthode `destroyIcobj` du workspace si l'icobj ne dispose pas d'un champ appelé `Indestructible` dans sa table de hachage. La présence d'un champ `Indestructible` évite, par exemple, la destruction par inadvertance des constructeurs de la simulation.

Pour construire un icobj, il suffit d'étendre la classe `Icobj` et d'initialiser chacun de ses champs. Il ne faut en aucun cas ajouter directement des champs dans les classes étendues sous peine que le système de construction graphique ne fonctionne plus. Nous admettons cependant une exception pour la classe `Workspace`. En effet, pour des raisons de performances, nous avons ajouté quelques champs à la classe `Workspace` pour gérer l'affichage, la migration et l'exécution des icobjs. Le mécanisme de construction graphique tient compte de cette distinction et peut créer un icobj ou un workspace. Si l'un des icobjs utilisés pendant une construction graphique est un workspace, alors l'icobj résultant de cette construction sera aussi un workspace.

6.2 Exécution des icobjs

Nous allons dans cette partie décrire la manière dont les icobjs sont gérés et exécutés par les workspaces. Nous commencerons par présenter l'objet `BootWorkspace`. Le rôle de l'objet `BootWorkspace` est d'exécuter le workspace enregistré par la méthode `setWorkspace`. Dans la suite du document, nous évoquerons ce workspace en l'appelant le workspace de plus haut niveau.

Le comportement d'un objet `BootWorkspace`, dont le code est décrit dans la table 6.2, est exécuté par un thread *Java* autonome. De façon cyclique, ce thread fait réagir le workspace de plus haut niveau en mettant à jour l'affichage du workspace entre chaque réaction (ou plutôt chaque `nbInstantPerRefresh` réactions). Ce comportement effectue en plus une mise à jour de l'inspecteur des *Icobjs* et permet d'effectuer une sauvegarde entre deux réactions d'un workspace. Ces fonctionnalités seront détaillées plus loin dans les parties 6.5 et 6.6.

De plus, nous avons défini une durée minimale (le champ `minDuration`) pour une étape d'exécution. L'utilisation d'une durée minimale pour l'exécution d'un cycle évite de surcharger

```
while (running)
{
   now = System.currentTimeMillis();
   try
   {
      if (nonStop || onestep)
      {
         oneStep = false;
         for (int i = 0; i < nbInstantPerRefresh; i++)
            workspace.react();
         render();
      }
      if (inspector != null)
         inspector.updateField(this);
   }
   catch (Exception ex){}
   if (save)
      savingProcess();
   delta = now + instant_duration - System.currentTimeMillis();
   if (delta > 0)
   {
      try
      {
         Thread.sleep(delta);
      }
      catch (InterruptedException e){}
   }
}
```

TAB. 6.2 – Comportement d'un objet de type `BootWorkspace`

le processeur avec l'exécution du thread *Java* exécutant l'objet `BootWorkspace`. Si la durée d'un cycle est plus courte que la valeur `minDuration`, alors le thread *Java* s'endort pendant la durée restante ce qui permet aux autres threads *Java* de s'exécuter. Il ne faut pas non plus que la durée minimale soit trop importante car la fluidité de la simulation en dépend. La classe `BootWorkspace` dispose de la méthode `void setMinimumDuration(long duration)` pour modifier la durée minimale d'une étape. Nous n'avons pas défini de durée maximale d'une étape d'exécution car il n'est pas possible de borner le temps d'exécution d'une réaction qui dépend du nombre de comportements exécutés.

Nous allons maintenant détailler la structure d'un objet `Workspace`. Comme le montre le code de la table 6.3, le workspace est un icobj comme toutes les entités graphiques du modèle des *Icobjs*. Les champs supplémentaires spécifiques au workspace sont également représentés dans cette table.

Comme pour la classe `Icobj`, il n'est pas possible d'accéder directement aux champs du workspace. Cela garantit par exemple que seul le workspace peut modifier l'état de la machine réactive. Les méthodes disponibles dans `MachineIcobj` pour générer des événements externes purs ou valués et pour faire réagir la machine réactive sont ajoutées à la classe `Workspace`. Pour pouvoir gérer les icobjs contenus dans un workspace et les comportements exécutés par sa machine réactive, la classe `Workspace` dispose des méthodes suivantes :

- `void registerIcobj(Icobj icobj)`
  Cette méthode enregistre un icobj dans le workspace. L'enregistrement consiste d'abord

```
public class Workspace extends Icobj
                       implements KeyListener, MouseListener, MouseMotionListener
{
   protected transient boolean reacting = false;
   protected transient MachineIcobj machine;
   private WorkspaceAppearance workApp = null;
   protected boolean workspaceVisible = false;
   protected transient MigrationCell migrationList = null;
   protected transient SavingCell savingList = null;
   protected transient Stack icobjStack = null;
   protected transient Icobj editedIcobj = null;
   protected transient boolean needToRender = true;
   ...
}
```

TAB. 6.3 – Structure d'un workspace

à déterminer le comportement à exécuter selon que l'icobj est un constructeur ou non et selon qu'il provient ou non d'un autre workspace. Cette méthode modifie d'abord la référence de l'icobj vers le workspace dans lequel il est enregistré. L'algorithme suivant est utilisé pour déterminer le comportement à exécuter :

- si l'icobj dispose d'un comportement gelé (le champ `frozenBehavior` est différent de **`null`**), alors le workspace exécute ce comportement et le champ `frozenBehavior` de l'icobj est réinitialisé à la valeur **`null`**.
  Il n'est pas nécessaire de rajouter le comportement standard de contrôle au comportement gelé puisque celui-ci est issu d'une composition parallèle comprenant déjà celui-ci. Le nouveau comportement construit sera exécuté à partir de l'instant suivant (cf. section 4.2.3).
- si ce n'est pas le cas, le comportement de l'icobj est une composition parallèle du comportement de contrôle de l'icobj (pour gérer les événements souris et clavier) et
  – du comportement non-clonable si l'icobj est un constructeur.
  – du comportement clonable et non-clonable si ce n'est pas le cas.

- `Icobj[] getIcobjList()`
  Cette méthode retourne un tableau contenant tous les icobjs enregistrés dans le workspace courant.

- **`boolean`** `add(Icobj icobj, Program program)`
  Cette méthode ajoute un programme à un icobj enregistré dans le workspace. Si l'icobj n'a pas encore été enregistré dans le workspace, alors il ne se passe rien. Il faut noter que le workspace peut ajouter directement des programmes à exécuter dans sa propre machine réactive par l'intermédiaire de cette méthode. Tout programme ajouté à la machine sera exécuté à l'instant suivant.

- **`void`** `resetIcobj(Icobj icobj)`
  Cette méthode réinitialise le comportement en appelant la méthode `reset` de la machine réactive en utilisant l'identifiant de l'icobj. Cette méthode demande ensuite la réinitialisation des champs de l'icobj en appelant la méthode `reinit` de la classe `Icobj`.

Le comportement est alors immédiatement reconstruit et ajouté à la machine réactive de la même manière que lorsque la méthode `registerIcobj` est appelée.

- **`boolean`** `destroyIcobj(Icobj icobj)`
  Cette méthode retire l'icobj du workspace. Ce retrait se traduit par un retrait de l'icobj de la liste d'icobjs et d'une demande à la machine de geler le comportement de l'icobj. Si l'instant n'est pas encore terminé au moment de l'appel, ce retrait aura lieu à la fin de l'instant, sinon, il aura lieu à la fin de l'instant suivant. À l'instant du retrait, un événement constitué par l'identifiant de l'icobj et suffixé par la chaîne "-kill" est généré dans la machine.

- **`void`** `migrateIcobj(Icobj icobj, Workspace destination, Position pos)`
  Cette méthode enregistre l'icobj comme devant changer de workspace à la fin de l'instant. Cela consiste à demander à la machine réactive le retrait de l'icobj à la fin de l'instant, puis à enregistrer la requête de migration dans une liste (le champ `migrationList` de la classe `Workspace`) en indiquant la position à laquelle il devra être placé dans le workspace de destination.
  La migration est finalement effectuée par la méthode `react()` du `Workspace`. Lorsque la réaction de la machine est terminée, tous les icobjs devant migrer sont enregistrés dans les workspaces destinataires et sont placés à la position désirée.
  Le fait de passer par une liste intermédiaire retarde la migration à la fin de l'instant où l'icobj est dans un état stable. Si cette migration était faite manuellement (appel à `destroyIcobj` puis à `registerIcobj`) sans s'assurer que la fin d'instant a été atteinte dans le workspace dans lequel l'icobj était, cela pourrait donner lieu à des erreurs. D'une part, il n'est pas possible de récupérer le résidu du comportement de l'icobj avant la fin de la réaction du workspace et d'autre part, l'icobj pourrait être exécuté en même temps dans deux workspaces différents qui sont eux-mêmes exécutés par deux threads différents. Dans de telles circonstances, le modèle de l'approche réactive n'offrirait plus aucune garantie. De plus, si l'on considère des migrations à travers le réseau, il faut attendre que le comportement de l'icobj soit terminé pour pouvoir le sérialiser avant de l'envoyer (pour l'instant, la migration ne permet que de changer de workspace dans la même machine virtuelle *Java*).

Le fait que le workspace est un icobj signifie qu'il dispose aussi des comportements réactifs clonables et non-clonables qui sont exécutés par le workspace parent. Lorsqu'un workspace est contenu dans un autre, la réaction du workspace n'est pas réalisée par un thread *Java* dédié, mais par l'intermédiaire du comportement clonable du workspace. Par défaut, ce comportement correspond à appeler, par l'intermédiaire d'une action atomique, la méthode `react()` du workspace à chaque fois que le workspace parent est exécuté. Le fait qu'un instant du workspace fils corresponde à une action atomique du workspace père a pour conséquence que la durée des instants est différente. Un instant du workspace fils correspond à une partie du workspace père. Il est par exemple possible qu'un workspace "fils" puisse réagir plusieurs fois de suite alors que son parent n'a réagi qu'une seule fois et inversement. De plus, les événements générés dans un workspace ne sont vus ni par son workspace parent ni par ses workspaces "fils". Chaque workspace dispose d'un environnement d'exécution propre.

Enfin, le changement d'apparence présenté sur la figure 5.14 est aussi géré par le comportement clonable du workspace. En fait, son comportement ajoute une action sur pression de

la touche *D* dans le comportement de contrôle qui réagit aux événements clavier. Nous allons maintenant détailler la gestion de ces événements clavier et souris.

6.3 Gestion des événements clavier et souris

Dans la section 5.2, nous avons signalé que chaque icobj réagit aux événements de souris pour sélectionner, activer ou déplacer un icobj et aux événements du clavier pour changer l'apparence du workspace. Comme un icobj n'est pas un composant graphique *Java*, il ne lui est pas possible de réagir à ces événements en utilisant les méthodes de rappel définies dans les interfaces `KeyListener`, `MouseListener` et `MouseMotionListener`.

Par contre, l'objet `BootWorkspace` est un composant graphique *Java* (un `JPanel`) qui peut donc récupérer ces différents événements. Ces derniers sont transmis au workspace de plus haut niveau qui les récupère par l'intermédiaire des méthodes de rappel et dont la tâche est de les rediriger vers les icobjs appropriés. Pour cela, les événements asynchrones *Java* sont transformés par le workspace de plus haut niveau en événements réactifs générés dans sa machine. Ces méthodes de rappel ne peuvent cependant fonctionner que pour le workspace de plus haut niveau qui est directement connecté au conteneur *Java*. Pour les workspaces imbriqués, les événements de souris et clavier sont relayés par les comportements de contrôle des workspaces. Nous allons maintenant détailler le fonctionnement de ces méthodes de rappel.

6.3.1 Les événements de souris

Les méthodes de rappel récupèrent un objet de type `MouseEvent`. Cet objet contient les coordonnées de la souris dans le composant graphique *Java* au moment de la génération de l'événement souris. À partir de ces coordonnées, il est ainsi possible de déterminer l'icobj qui doit être sélectionné dans le workspace de plus haut niveau. Le workspace conserve l'ensemble des icobjs qu'il contient sous forme d'une pile. La sélection d'un icobj retourne le premier icobj en descendant dans la pile qui contient les coordonnées mentionnées dans l'événement souris. Si cette sélection retourne un icobj, alors le workspace génère un événement valué en utilisant l'objet `MouseEvent` comme valeur. L'identificateur de cet événement est obtenu en concaténant une constante du système à l'identifiant de l'icobj sélectionné. Cette constante est déterminée selon le type d'action souris. Nous distinguons trois types d'actions : soit une pression d'un bouton, soit un maintien d'un bouton enfoncé, soit un relâchement d'un bouton.

Pour des raisons d'affichage dans un environnement 2D, l'icobj sélectionné est placé au sommet de la pile des icobjs, ce qui permet d'afficher l'icobj sélectionné au-dessus de tous les autres icobjs. Pour pouvoir manipuler l'icobj sélectionné, la classe `Workspace` dispose des méthodes `Icobj getEditedIcobj()` et `void setEditedIcobj(Icobj icobj)`.

6.3.2 Les événements du clavier

Les méthodes de rappel récupèrent un objet de type `KeyEvent` qui sert de valeur à la génération de l'événement réactif. L'identificateur de cet événement est basé sur une constante du système préfixée par l'identifiant de l'icobj sélectionné, s'il y en a un.

Si l'icobj sélectionné est un workspace et que celui-ci contient également un icobj sélectionné alors cet événement clavier sera relayé à ce dernier par l'intermédiaire du comportement de contrôle du workspace, sinon le workspace exécutera le comportement associé à cet événement clavier.

6.3.3 Les comportements

Pour pouvoir réagir aux événements souris et clavier, chaque icobj exécute, en plus de son comportement de départ, un comportement de contrôle. Par exemple, le programme décrit dans la table 6.4 permet de réagir aux événements générés quand un bouton de souris est pressé. Le comportement de contrôle gère aussi les événements *mouseDragged*, *mouseReleased* et *keyPressed*.

Ces comportements doivent réagir à toutes les générations de ces événements à chaque instant. En effet, la durée d'un instant n'étant pas bornée, plusieurs événements de souris peuvent être générés au cours du même instant. C'est pourquoi nous utilisons une `ScanAction` pour réagir à toutes les générations et pour effectuer les opérations d'activation, de sélection, de redimensionnement ou de déplacement. Le problème de l'instruction `Scanner` est qu'elle fait de l'attente active sur les événements. Pour palier à ce problème, elle est précédée d'une instruction `Await` pour obtenir des comportements dont l'exécution ne coûte rien en l'absence d'événement.

```
Ic.Loop(
   Ic.Seq(Ic.Await(new IcobjIdentifier(EventNames.MOUSE_PRESSED_EVENT)),
      Ic.Scanner(new IcobjIdentifier(EventNames.MOUSE_PRESSED_EVENT),
                 new MousePressedCallback())));
```

TAB. 6.4 – Comportement de contrôle des événements *mousePressed*

Le comportement de gestion du clavier consiste en une redirection. En effet, l'action atomique réagissant aux événements claviers récupère d'abord un champ particulier qui contient les liens entre les touches et les actions à exécuter si ces touches sont pressées. Si ce champ est défini et si une action est associée à la touche pressée, alors elle est exécutée. Le champ définissant les relations entre la touche pressée et les actions à exécuter est paramétrable. Il est possible d'ajouter dynamiquement de nouveaux liens ou d'en retirer certains.

Les comportements de contrôle des workspaces utilisent les mêmes actions atomiques que celles de tous les icobjs. Une action atomique commence par tester si l'icobj qui l'exécute est un workspace. Si ce n'est pas le cas, le comportement correspondant au type d'événement est exécuté (redimensionnement, déplacement, activation,...). Si, par contre l'icobj est un workspace, alors le comportement consiste à tester si l'événement est dédié à l'un des icobjs que le workspace contient. Si c'est le cas, l'événement est redirigé vers cet icobj. Par contre, si l'événement est dédié au workspace et non à un icobj qu'il contient, alors il exécute le comportement correspondant au type d'événement. Il faut noter que dans le cas des événements souris, si l'événement est dédié à un icobj contenu dans un workspace imbriquée, alors les coordonnées définies dans l'objet `MouseEvent` sont modifiées pour correspondre au référentiel du workspace qui contient cet icobj.

En utilisant ce modèle de transmission d'événements par le comportement des workspaces, il n'est pas possible de générer un événement qui n'est pas associé à un icobj en particulier dans un workspace imbriqué. Dans le workspace de plus haut niveau, il est possible de générer un événement qui n'est destiné à aucun icobj en particulier si son champ `editedIcobj` est à **`null`**. Un workspace imbriqué ne peut recevoir un tel événement car ce workspace le prendra en compte comme si l'événement lui était directement destiné dans son workspace parent.

6.4 Affichage des icobjs

La règle importante ici est que de l'affichage d'un icobj ne doit pas avoir lieu en même temps que l'exécution de son comportement. Comme pour le programme réactif, les données ne sont pas nécessairement dans un état stable avant la fin d'une réaction. En effet, durant l'exécution des comportements des icobjs, les informations de position, de taille, d'apparence ou toutes autres informations utilisées par l'affichage peuvent être modifiées plusieurs fois par instant. L'affichage d'un icobj est donc nécessairement une opération inter-instants.

Un autre aspect de l'affichage dans le modèle des *Icobjs* est de pouvoir passer dynamiquement d'un environnement 2D à un environnement 3D et de faire en sorte que les comportements des icobjs s'adaptent automatiquement à ces modifications. Il est évident que les comportements ne peuvent s'adapter que si les actions atomiques tiennent compte du type d'environnement.

Un workspace étant un icobj, l'affichage d'une simulation doit être réalisé par l'apparence du workspace. La figure 5.14 présente les deux apparences de base d'un workspace. Dans la classe `Workspace`, nous différencions ces deux apparences. L'apparence minimisée correspond à l'apparence d'un icobj et donc au champ `appearance` de l'icobj. Cette apparence peut être une image, un dessin ou n'importe quelle autre apparence d'icobj. L'autre apparence que nous nommons *apparence normale* doit afficher la fenêtre graphique et l'apparence de chaque icobj présent dans cette fenêtre. Pour cette apparence, nous avons ajouté le champ `workApp` à la classe `Workspace`. Nous distinguons aussi cette apparence par un type spécifique : l'interface `WorkspaceAppearance` dont le code est donné dans la table 6.5. Ce type d'apparence doit implémenter 3 méthodes supplémentaires :

- La première retourne le nombre de dimensions gérées par cette apparence (2D ou 3D) et les axes affichés (X, Y ou Z).
- La seconde permet d'initialiser l'apparence. Elle est appelée par le workspace dès que celui utilise l'apparence. Le workspace passe en argument la référence vers la liste des icobjs qu'il contient.
- La dernière permet de spécifier les icobjs qui ont besoin d'être réaffichés lors du prochain rendu de la simulation. En effet, si les icobjs ne changent pas de position ou d'apparence, l'objet de type `WorkspaceAppearance` peut effectuer des optimisations en ne mettant à jour que les icobjs ayant été modifiés de façon apparente.

```
public interface WorkspaceAppearance extends Appearance
{
   Dimension getDimensionNumber();
   void setWorkspaceInfo(Stack icobj, Workspace workspace);
   void render(Icobj icobj);
   ...
}
```

TAB. 6.5 – Interface WorkspaceAppearance

Pour gérer les paramètres graphiques d'une simulation, la classe `Workspace` dispose des méthodes suivantes :

- `Dimension getDimensionNumber()`
  Cette méthode retourne le nombre de dimensions traitées par le workspace. La valeur

retournée indique à la fois le nombre de dimension et les axes gérés par l'environnement graphique. Cette dernière information permet aux comportements d'identifier les dimensions auxquelles ils doivent s'appliquer.

- `WorkspaceAppearance getWorkspaceAppearance()`
  `void setWorkspaceAppearance(WorkspaceAppearance workApp)`
  La première méthode retourne l'apparence normale du workspace et la seconde la modifie. Le champ `workApp` doit toujours être initialisé. Par défaut, ce champ est initialisé avec un objet de type `AWTWorkspaceAppearance` qui effectue un affichage 2D sur les coordonnées XY.
  De plus, il faut noter que la méthode `getAppearance` héritée de la classe `Icobj` est modifiée pour retourner l'apparence utilisée par le workspace au moment de l'appel. Cette méthode retourne le champ `appearance` si le workspace est minimisé. Sinon, elle retourne le champ `workApp`.

- `Zone getWorkspaceZone()`
  `void setWorkspaceZone(Zone zone)`
  De la même manière que pour l'apparence, le workspace dispose de deux zones selon que le workspace est minimisé ou non. Ces deux zones ont des dimensions différentes mais partagent la même position. De la même manière que pour l'icobj, un workspace doit toujours disposer d'une zone, que ce soit en affichage normal ou minimisé.

- `void render(Icobj icobj)`
  `boolean needToRender()`
  La première méthode permet explicitement de demander une mise à jour de l'affichage pour l'icobj passé en argument. La seconde permet au composant graphique *Java* de tester s'il est nécessaire de rafraîchir l'affichage du workspace de plus haut niveau.

- `void hide()`
  `void show()`
  `boolean isWorkspacevisible()`
  Ces méthodes permettent respectivement de minimiser le workspace, de l'afficher normalement et de tester son mode d'affichage. Le passage du mode minimisé au mode normal (ou inversement) entraîne nécessairement un rafraîchissement de l'affichage.

- `Icobj[] listOfOverlappingIcobj(Zone zone)`
  Cette méthode retourne tous les icobjs qui se trouvent dans une zone donnée du workspace. L'ordre des icobjs dans ce tableau est l'ordre dans lequel les icobjs sont placés dans la pile `icobjStack`.

- `void renderScene(Graphics g)`
  Cette méthode permet de faire le rendu du workspace dans le contexte graphique passé en paramètre. Elle est utilisée par l'objet `BootWorkspace` pour afficher le champ `workApp` du workspace de plus haut niveau.

L'API n'offre pour l'instant que la possibilité d'afficher des workspaces en 2D par l'intermédiaire de la classe `AWTWorkspaceAppearance`. L'affichage produit par cette classe n'est pas très évolué. Toute la simulation est en fait redessinée, même si un seul icobj doit être

réaffiché. Quand cette classe fait le rendu, elle commence par effacer le fond, puis demande le rendu des apparences des icobjs qui sont positionnés dans la zone affichée. Ces apparences sont rendues dans l'ordre dans lequel les icobjs sont enregistrés dans le champ `icobjStack` du workspace en commençant par le bas de la pile. Cela permet à l'icobj sélectionné, c'est-à-dire à l'icobj qui se trouve au sommet de la pile, d'être rendu en dernier et donc d'être affiché au-dessus de toutes les autres apparences. Si un icobj ne dispose pas d'une apparence, alors `AWTWorkspaceAppearance` utilise une apparence par défaut de type `AWTDefaultAppearance`.

Si un des icobjs à réafficher est un workspace, alors celui-ci utilisera la même technique pour afficher ses propres icobjs. Pour éviter que l'affichage des icobjs des workspaces imbriqués ne déborde de leur zone d'affichage, la classe `AWTWorkspaceAppearance` utilise la méthode du *Clipping* disponible dans l'API de *Java*. Cette méthode consiste à restreindre la surface disponible pour les rendus. Seule la partie de l'apparence contenue dans cette surface restreinte est affichée. Avant de rendre tous les icobjs d'un workspace, celui-ci limite la zone de rendu à ses propres dimensions et coordonnées dans le conteneur *Java* et change le système de coordonnées de la zone d'affichage pour que l'origine coïncide avec le coin haut-gauche du workspace. Dès que le rendu des icobjs est terminé, la zone d'affichage et les coordonnées sont restaurées dans leur état initial.

L'objet `BootWorkspace` est un composant graphique *Swing* qui hérite de la classe `JPanel` sur lequel est rendu l'apparence du workspace de plus haut niveau. Dans le modèle de *Swing*, plus particulièrement depuis la version 1.4 de *Java*, le rafraîchissement de composant graphique est une opération asynchrone. Ainsi, il faut d'abord demander au thread graphique de rafraîchir un composant et quand ce thread graphique l'a décidé, il permet alors au composant graphique de se dessiner. De plus, le rendu est effectué sur une image appelée `VolatileImage` [52], disponible à partir de la version 1.4 de *Java*, qui permet de bénéficier des accélérations matérielles des cartes vidéo.

Dans les versions précédentes des *Icobjs* qui étaient exécutées avec les versions 1.3 et antérieures de *Java*, il était possible de rafraîchir le composant graphique immédiatement après une réaction de la machine réactive. Cette façon de faire est désormais trop coûteuse car elle nécessite de prendre des verrous sur le thread graphique pour pouvoir dessiner sur le composant de façon synchrone.

Comme le montre le code de la classe `BootWorkspace` représenté dans la table 6.2, le rafraîchissement de l'affichage est demandé tous les `nbInstantPerRefresh` instants. Cette demande n'est faite que si le workspace a besoin d'être réaffiché, c'est-à-dire si les icobjs ont demandé explicitement leur réaffichage. Pour pouvoir gérer de façon plus fine la fluidité de l'animation, nous avons introduit le paramètre `nbInstantPerRefresh`. En effet, si le rendu prend beaucoup de temps, la vitesse à laquelle les instants sont exécutés (la vitesse de l'animation) peut devenir lente. Il est donc préférable dans certains cas d'augmenter le nombre de réactions effectuées par le workspace de plus haut niveau entre deux rafraîchissements de l'affichage sur le conteneur *Java*. Il faut tout de même faire attention que ce nombre d'instants ne soit pas trop grand car cela risque de rendre l'animation saccadée. Pour avoir une animation fluide, il faut ajuster la valeur de ce champ et celle du champ indiquant la durée minimale d'un cycle. L'objet `BootWorkspace` dispose des méthodes `byte getNumberOfInstantsPerRefresh()` et `void setNumberOfInstantsPerRefresh(byte nbInstants)` pour ajuster le nombre d'instants par étape d'exécution.

6.5 Inspecteur des *Icobjs*

L'inspecteur des icobjs permet de visualiser les champs et les comportements d'un icobj et de les modifier. Pour cela, il utilise des mécanismes d'introspection sur les icobjs. Nous allons présenter les mécanismes utilisés dans le cadre de la structure de données de l'icobj et dans celui du comportement

6.5.1 Introspection des champs d'un icobj

Dans le modèle des *JavaBeans* [53], les méthodes d'introspection permettent de découvrir dynamiquement les informations concernant une classe d'objet. Pour cela, chaque bean utilise des mécanismes de réflexion pour pouvoir analyser sa propre structure. Ces méthodes permettent de récupérer les signatures de chacun des constructeurs de l'objet, d'obtenir et fixer les valeurs des champs, d'invoquer les méthodes sur les objets ou sur les classes et de créer de nouvelles instances d'une classe. Ces mécanismes vont découvrir les champs qu'il est possible de modifier en cherchant des méthodes `setXxxx` et `getXxxx` où `xxxx` est un des champs paramétrables.

Pour les *Icobjs*, nous n'avons pas utilisé ces mécanismes sous la forme proposée par *Java*. Les champs d'un icobj étant ajoutés dynamiquement par le comportement, la classe `Icobj` ne dispose pas des méthodes `setXxxx` et `getXxxx` pour chaque champ disponible dans la table de hachage. Ainsi, il n'est pas possible d'utiliser les mécanismes d'introspection proposés par *Java*. De plus, nous ne voulons pas que ces opérations puissent avoir lieu au cours d'un instant d'exécution. C'est pourquoi, nous avons défini un mécanisme qui permet, à chaque classe, de définir ses champs paramétrables et que ceux-ci soient accédés et modifiés uniquement entre deux instants. Toute classe pouvant être inspectée et/ou paramétrée doit étendre l'interface `Parametrable` (cf. table 6.6) et implémenter les trois méthodes de cette interface. La classe `Icobj` et toutes les classes correspondant aux champs prédéfinis de l'icobj étendent cette interface.

```
public interface Parametrable extends Serializable
{
   Parameter[] getParameter(Icobj self);
   Serializable getValue(String fieldName);
   void setValue(String fieldName, Serializable value);
}
```

TAB. 6.6 – Structure d'un objet modifiable

- la méthode `getParameter` permet de spécifier les champs paramétrables. Chaque classe paramétrable déclare ses propres champs en précisant le type d'interface de paramétrage à utiliser pour chacun d'entre eux. Toutes ces interfaces sont de type `Parameter` (cf. table 6.7). Cette classe abstraite offre différents types prédéfinis d'interface de paramétrage dont, entre autres des interfaces pour les nombres entiers (`getLong`), les nombres réels (`getDouble`), les chaînes de caractères (`getString`), les booléens (`getBoolean`), les couleurs (`getColor`), les champs paramétrables (`getEntete`). La liste d'interfaces disponibles n'est pas exhaustive et peut être étendue si nécessaire. Toutes ces méthodes prennent au minimum deux arguments. Le premier argement est une

chaîne de caractères qui identifie le champ à modifier. Cette chaîne de caractères, qui sera affiché dans l'inspecteur, sert de description au champ à paramétrer. Le second argument est une référence vers l'objet paramétrable qui donne accès aux deux autres méthodes que nous allons maintenant détailler.

- la méthode `getValue` retourne la valeur du champ identifié par la chaîne de caractères passée en argument. Si aucun champ n'est identifié par la chaîne de caractères, cette méthode doit retourner **`null`**.
- la méthode `setValue` remplace la valeur du champ identifié par la chaîne de caractères par la valeur du second argument de cette méthode. Il est possible de définir des contraintes dans cette méthode. Par exemple, la classe `BasicAbsorbtion` contient un champ représentant le taux d'une force d'absorption qui doit être compris entre 0 et 1. Sa méthode `setValue` ne mettra à jour ce champ que si la nouvelle valeur est comprise dans cet intervalle.

Le code de la classe `BasicSpeed` donné dans la table 6.9 donne un exemple d'implémentation de ces trois méthodes.

```
public abstract class Parameter extends JPanel
{
   protected String name = null;
   protected Parametrable pointer = null;

   protected Parameter(String name, Parametrable obj)
   {
      super(false);
      this.name = name;
      this.pointer = obj;
   }
   public abstract void updateValue();
   ...
}
```

TAB. 6.7 – Classe abstraite Parameter

Nous allons maintenant détailler comment l'inspecteur utilise ces méthodes. Nous présentons ici un inspecteur particulier, le `IcobjInspector` qui est celui accessible par le `Framework`. Il est cependant possible de créer son propre inspecteur pour des besoins particuliers.

L'objet `IcobjInspector` récupère la liste des paramètres d'un icobj dès que celui-ci est sélectionné. Un icobj étant un objet paramétrable, l'appel à sa méthode `getParameter` permet de récupérer toutes ses interfaces. Pour la classe `Icobj`, cette méthode crée un tableau d'interfaces de paramétrage qui comprend au minimum : une interface pour le nom de l'icobj, une interface pour préciser si l'icobj est un constructeur ou non, une interface pour la zone de l'icobj et une pour son apparence. Pour chacun des champs présents dans la table de hachage de l'icobj, une interface `EnteteParameter` est créée. Cette interface permet d'afficher, pour un objet paramétrable, les interfaces de ses propres champs ou de les faire disparaître en cliquant sur le bouton comme cela est décrit dans la partie 5.3.2. Il est possible de trouver dans l'inspecteur des identifiants pour des champs avec lesquels on ne peut interagir. C'est le cas pour tous les objets qui n'ont pas défini d'interface dans leur méthode `getParameter` ou pour tout objet qui n'implémente pas l'interface `Parametrable`.

La mise à jour des interfaces récupérées par la méthode `getParameter` est effectuée par

l'objet `BootWorkspace` (cf. table 6.2) en appelant la méthode `updateField` de l'inspecteur des *Icobjs*. Le comportement de cette méthode diffère selon que l'icobj sélectionné dans l'inspecteur est ou non dans un workspace exécuté par cet objet `BootWorkspace` :

- Si l'icobj sélectionné n'appartient pas à ce `BootWorkspace`, il ne se passera rien.
- Si, au contraire, l'icobj appartient à cette simulation (au plus haut niveau ou dans un workspace imbriqué), alors toutes les interfaces correspondant à l'icobj sont mises à jour. Dans un premier temps, toutes les interfaces modifiées par l'utilisateur mettent à jour la valeur du champ de l'icobj qu'elle représente en utilisant la méthode `setValue`. Dans un second temps, toutes les interfaces mettent à jour la valeur qu'elles affichent en récupérant la valeur du champ de l'icobj par la méthode `getValue`.

Nous avons choisi que le `BootWorkspace` demande lui-même la mise à jour de l'inspecteur pour garantir que ce dernier n'interfère pas avec les comportements des icobjs pendant leur exécution.

Notons une petite optimisation qui permet de ne mettre à jour que les valeurs des interfaces affichées dans l'inspecteur. Ainsi, les interfaces de type `EnteteParameter` mettent à jour les champs de l'objet qu'elles représentent uniquement s'ils sont visibles dans l'inspecteur.

6.5.2 Introspection du comportement d'un icobj

La visualisation et la modification du comportement d'un icobj ne sont pas des opérations inter-instants comme pour les champs d'un icobj. Dès que l'icobj est sélectionné dans l'inspecteur, celui-ci récupère une copie des comportements clonable et non-clonable de cet icobj. Il n'y a pas de synchronisation automatique entre la copie affichée dans l'inspecteur et les champs des comportements de l'icobj.

De la même manière que chaque objet paramétrable spécifie les interfaces affichables dans l'inspecteur pour communiquer avec ses propres champs, chaque instruction réactive renvoie, par l'intermédiaire de la méthode `Node getTreeNode()`, une interface qui permet de visualiser et/ou de modifier les paramètres d'exécution de ces instructions.

```
public abstract class Node extends DefaultMutableTreeNode
{
   public abstract Program createProgram() throws Exception;
   public abstract void parametrize(Component owner, Icobj self);
}
```

TAB. 6.8 – Classe abstraite Node

Pour chaque instruction, nous avons créé une interface de type `Node` (cf. table 6.8). Par exemple l'instruction `Kill` retourne un arbre dont la racine est une interface `KillNode` permettant de visualiser et de modifier sa configuration événementielle et disposant de deux nœuds fils qui sont respectivement l'arbre retourné par le corps de l'instruction `Kill` et l'arbre retourné par le handler de l'instruction. Les objets de type `Node` implémentent deux méthodes :

– `parametrize`
  Cette méthode affiche, si nécessaire, une boîte de dialogue pour paramétrer le nœud de l'arbre. Par exemple, pour un nœud de type `KillNode`, la boîte de dialogue (cf.

figure 5.18) permet de paramétrer la configuration événementielle. La boîte de dialogue est composée d'une part d'une zone dans laquelle la configuration événementielle est exprimée par une expression booléenne, et d'autre part d'une table qui relie un alias utilisé dans l'expression booléenne à un wrapper d'événement. Dès que l'utilisateur clique sur `OK`, l'expression booléenne est traduite en une configuration événementielle. Pour cela, nous avons généré un traducteur à partir d'une grammaire écrite avec *ANTLR* [59].

– `createProgram`
Cette méthode retourne le programme réactif représenté par le nœud lui-même. Elle est appelée quand l'utilisateur demande la mise à jour des comportements clonable et non-clonable de l'icobj. Lors de cette mise à jour, chaque nœud teste si les paramètres qui lui ont été donnés sont conformes. Par exemple, le nœud de type `LoopNode` retournera `Nothing` si son nœud fils retourne le programme `Nothing` pour éviter de créer une boucle instantanée. Cependant, hormis ce test, aucune autre vérification n'est faite pour déterminer si l'exécution du corps est instantanée.

Pour pouvoir correctement paramétrer les noeuds, il faut pouvoir instancier dynamiquement et paramétrer des objets *Java*. Pour cela, nous utilisons la même technique que pour les champs de l'icobj. Tout objet de type `Action`, `ScanAction` et tous les wrappers sont des objets de type `Parametrable`. Pour pouvoir instancier ces objets, ils doivent disposer d'un constructeur sans argument comme c'est le cas dans les *JavaBeans*. Cette règle doit d'ailleurs être généralisée à tout objet de type `Parametrable` que l'on souhaite instancier dynamiquement.

Pour faciliter la vie de l'utilisateur, l'application `Framework` commence par scanner les fichiers *jar* et les répertoires dans lesquels se trouvent l'API des *Icobjs* et toutes les classes définies par l'utilisateur. Toutes ces classes sont triées et regroupées dans des listes qui représentent les principaux types utilisés par les comportements : les actions atomiques (pour `Atom` et `Scanner`), les différents types de wrappers, les identifiants d'événements, les objets paramétrables, les icobjs et les simulations. Ces différentes listes seront utilisées par l'application `Framework` et par l'inspecteur des *Icobjs* pour pouvoir paramétrer plus facilement les comportements et les champs des icobjs.

6.6 Chargement et enregistrement dans un fichier

L'enregistrement de l'état d'un icobj est une opération qui, comme la migration, doit être effectuée entre deux instants. L'enregistrement d'un icobj ou d'un workspace est fondamentalement une sérialisation de cet objet.

- Dans le cas d'un icobj, la sérialisation consiste à récupérer le résidu du comportement de cet icobj, à le placer dans son champ `frozenBehavior`, à sérialiser l'icobj et enfin à remettre son champ `frozenBehavior` à **`null`**. La méthode `residualBehaviorOf` de la machine réactive permet de récupérer le résidu du comportement de l'icobj sans avoir à le préempter.
- Dans le cas d'un workspace, la sérialisation est un peu plus complexe. Nous ne voulions pas sérialiser l'ensemble de la machine avec tout l'environnement d'événements. Notons d'abord que le workspace étant un icobj, sa sérialisation commence et termine comme celle d'un icobj. En plus de cette partie, le workspace sérialise séparément chacun des icobjs qu'il contient comme cela est décrit précédemment.

Pour charger les icobjs ou les workspaces sérialisés, il suffit simplement de les désérialiser et de les enregistrer dans le workspace cible avec la méthode `registerIcobj`.

Pour effectuer des enregistrements d'icobjs ou de workspaces dans un fichier, la classe `BootWorkspace` dispose de la méthode `void save(File file)`. Cette méthode n'effectue pas l'enregistrement de façon synchrone, mais elle enregistre la requête d'enregistrement qui sera prise en compte à la fin d'une étape d'exécution. L'enregistrement de la requête se traduit par la mise d'un booléen (`save`) à **`true`**, booléen qui est testé dans le comportement du `BootWorkspace` comme cela est montré dans la table 6.2. La méthode `savingProcess` réalise la sérialisation du workspace de plus haut niveau. Cette méthode enregistre, en plus du workspace de plus haut niveau, les paramètres de durée minimale d'une étape d'exécution et de nombre d'instants par étape d'exécution.

Pour enregistrer l'état d'un icobj particulier, la classe `Workspace` dispose de la méthode `boolean saveIcobj(Icobj icobj, File file)`. Cette méthode fonctionne de la même manière que `migrateIcobj`, c'est-à-dire que la requête d'enregistrement d'un icobj est placée dans la file `savingList` (cf. table 6.3). La méthode `react` du workspace exécute d'abord la réaction de la machine réactive, puis se charge de sérialiser les icobjs qui ont demandé à être enregistrés avant de faire migrer tous les icobjs qui ont demandé leur changement de workspace.

Ces deux méthodes d'enregistrement garantissent que le workspace est dans un état stable au moment d'un enregistrement dans un fichier. Pour charger un fichier, la classe `BootWorkspace` dispose d'un constructeur prenant en paramètre un tel fichier. Pour charger les icobjs enregistrés par un workspace, il revient à chaque comportement d'ouvrir le fichier adéquat et d'effectuer la désérialisation lui-même avant d'enregistrer l'icobj dans le workspace.

6.7 Créer sa propre simulation

Dans cette section, nous présentons un tutorial pour programmer ses propres simulations. Pour cela, il faut déterminer les besoins de la simulation en terme de champs à ajouter aux icobjs, de comportements nécessaires à l'initialisation et l'utilisation de ces champs, et d'entités à ajouter à cette simulation. Nous allons donner plusieurs conseils pour implémenter chacun de ces objets pour qu'ils soient utilisables dans notre modèle et pour qu'ils s'interfacent correctement avec l'inspecteur des *Icobjs*. Pour illustrer ce tutorial, nous présenterons le code de classes présentes dans l'API des *Icobjs*.

Les champs

Un champ est un objet *Java* classique dont la seule contrainte est d'être sérialisable pour permettre la sauvegarde dans un fichier ou la migration de l'objet à travers le réseau. Il est cependant préférable que les nouvelles classes créées étendent l'interface `Parametrable` pour permettre à l'inspecteur des *Icobjs* d'accéder et/ou modifier les différentes variables composant cet objet. Cela n'est pas indispensable au fonctionnement des icobjs, mais permet une plus grande interaction à l'exécution de la simulation.

Nous pouvons distinguer trois types de champs paramétrables utilisés dans la structure de données des icobjs :

```
public class BasicSpeed implements Speed
{
   protected float[] value;
   private transient Parameter[] tab;

   public BasicSpeed()
   {
      value = new float[3];
      value[0] = value[1] = value[2] = 0F;
      tab = null;
   }
   ...
   public String toString()
   {
      return "(" + value[0] + ", " + value[1] + ", " + value[2] + ")";
   }
   public Parameter[] getParameter(Icobj self)
   {
      if (tab == null){
         tab = new Parameter[3];
         tab[0] = Parameter.getDouble(SPEEDX_FIELD, this);
         tab[1] = Parameter.getDouble(SPEEDY_FIELD, this);
         tab[2] = Parameter.getDouble(SPEEDZ_FIELD, this);
      }
      return tab;
   }
   public Serializable getValue(String fieldName)
   {
      if (fieldName.equals(SPEEDX_FIELD))
         return new Float(value[0]);
      ...
      return null;
   }
   public void setValue(String fieldName, Serializable value)
   {
      if (fieldName.equals(SPEEDX_FIELD))
      {
         if (value instanceof Number)
            this.value[0] = ((Number)value).floatValue();
      }
      ...
   }
}
```

TAB. 6.9 – Exemple de champ : la classe BasicSpeed

- les champs de type `Zone` qui spécifient la zone d'influence d'un icobj. Les classes qui étendent cette interface doivent pouvoir exprimer cette zone dans des environnements 2D et 3D, sachant qu'il est possible de passer d'un environnement à l'autre dynamiquement. Ces classes doivent aussi pouvoir calculer les intersections entre différents types de zones. Actuellement, le calcul d'intersection est fait à partir des boîtes englobantes des icobjs.
- les champs de type `Appearance`. Les classes qui étendent cette interface doivent permettre de différencier, dans la méthode de rendu, l'affichage normal, l'affichage quand l'icobj est sélectionné et l'affichage lorsque l'icobj est activé. Nous différencions ces différents types d'affichage pour permettre à l'utilisateur de visualiser le fait que l'icobj

a bien reçu les événements de souris (sélection ou activation).
- les champs de type `Parametrable` définis par l'utilisateur pour être utilisés dans la table de hachage de l'icobj. Parmi ce type, nous distinguons deux catégories :
  - ceux qui ne sont que des conteneurs de données et qui disposent uniquement de méthodes pour lire et modifier ces données. C'est par exemple le cas pour la classe `BasicSpeed` (cf. table 6.9).
  - ceux qui sont utilisés comme des actions atomiques. En effet, il est parfois préférable de coder le comportement d'une action atomique, devant conserver son état entre les instants, dans un champ de l'icobj. Par exemple, le comportement du constructeur de base exécute, à chaque activation du constructeur, une action atomique qui recherche un tel champ dans sa structure de données. Si ce champ existe, alors l'action atomique appelle une méthode de ce champ qui exécute le comportement réel du constructeur de base. Cette façon de faire donne la possibilité de modifier dynamiquement le comportement d'un icobj en modifiant simplement le champ stocké dans la table de hachage.

Pour toutes les classes de type `Parametrable`, il faut, comme c'est le cas pour la classe `BasicSpeed` (cf. table 6.9) que l'objet :
- dispose d'un constructeur par défaut, c'est-à-dire d'un constructeur sans argument pour pouvoir charger dynamiquement la classe à partir de l'inspecteur.
- surcharge la méthode `toString` pour permettre une description textuelle de la fonction de l'objet dans l'inspecteur.
- implémente les trois méthodes `getParameter`, `getValue` et `setValue` pour paramétrer les champs composant cet objet à travers l'inspecteur.

Les comportements

Dans le modèle des *Icobjs*, il est préférable que le comportement d'un icobj initialise lui-même les champs de sa propre structure. Ainsi, dans le cas d'une construction graphique, les champs seront automatiquement ajoutés à l'icobj résultant de la construction. Donc, pour chaque nouveau champ, il faut définir à la fois les actions atomiques qui utilisent ces nouveaux champs et celles pour les initialiser et les placer dans la table de hachage de l'icobj. Dans l'API, nous avons distingué deux types d'actions atomiques d'initialisation : celles qui forcent l'initialisation d'un champ de l'icobj et celles qui n'initialisent le champ que si celui-ci n'existe pas encore. Par exemple, pour un champ de type `BasicSpeed`, l'API contient l'action atomique `SetBasicSpeed` qui force l'initialisation du champ vitesse de l'icobj avec un objet de type `BasicSpeed` et l'action atomique `NeedBasicSpeed` qui initialise le champ vitesse lorsque celui-ci n'est pas déjà présent dans la table de hachage.

En ce qui concerne les actions atomiques qui utilisent ces champs, il est préférable de tester le type du champ récupéré dans la table de hachage de l'icobj. En effet, puisque les champs de la table de hachage ne sont pas strictement typés, il vaut mieux tester que l'action manipule le bon type d'objet. Il serait intéressant de pouvoir définir le type du champ dans la table de hachage ; cela devrait être fait dans de futurs travaux.

Pour implémenter une nouvelle action atomique, il faut étendre l'interface `Action` (ou `ScanAction` pour réagir à un événement valué) et implémenter la méthode `execute`. Pour l'interfaçage avec l'inspecteur des *Icobjs*, il suffit de suivre les mêmes indications que pour un champ. De plus, il est préférable de ne garder aucun état dans une action atomique. Si

l'action atomique a besoin de conserver un état au cours de ses différentes exécutions, il faut enregistrer cet état dans la table de hachage de l'icobj.

À partir des actions atomiques, il est possible de créer des comportements plus complexes en utilisant d'autres instructions réactives. L'API contient par exemple la classe `InertialBehavior` décrite dans la table 6.10. Cette classe définit un comportement cyclique qui gère l'inertie d'un icobj sur plusieurs instants consécutifs. Ce comportement ajoute un champ vitesse s'il n'existe pas encore, puis exécute à chaque instant l'action atomique `Inertia`.

```
public class InertialBehavior extends BasicBehavior
{
   public InertialBehavior()
   {
      super("Basic␣Inertial␣behavior");
      set(Ic.Seq(Ic.Atom(new NeedBasicSpeed()),
                 Ic.Loop(Ic.Seq(Ic.Atom(new Inertia()), Ic.Stop()))));
   }
   ...
}
```

TAB. 6.10 – Exemple de comportement : la classe InertialBehavior

Pour implémenter ce type de comportements, il faut étendre la classe `BasicBehavior` qui est une instruction réactive. Cette instruction exécute simplement son corps et retourne le statut renvoyé par l'exécution de son corps. Cette classe permet uniquement de donner un nom à un comportement complexe, ce qui favorise la lisibilité dans l'inspecteur des *Icobjs*. Pour que les comportements de type `BasicBehavior` puissent être chargés par l'intermédiaire de l'inspecteur, ils doivent également disposer d'un constructeur par défaut.

Les entités

Une fois que les comportements et les champs de l'icobj sont implémentés, il ne reste qu'à créer les classes d'icobjs qui les utilisent. Pour cela, il suffit de procéder comme sur le squelette de classe donné dans la table 6.11. Il faut spécifier le comportement clonable, le comportement non-clonable et définir si l'icobj est un constructeur. Il ne faut pas ajouter directement de nouveaux champs dans la classe d'icobj. Tout champ supplémentaire doit être ajouté dans la table de hachage de l'icobj, si possible par l'intermédiaire du comportement de l'icobj. Cependant, si un champ est directement ajouté dans la table de hachage, il faut redéfinir la méthode `reinit` pour qu'elle ajoute ce champ à la table de hachage en cas de réinitialisation de l'icobj. Enfin, pour permettre de charger dynamiquement une classe d'icobj à partir de l'inspecteur, il faut aussi définir un constructeur sans argument.

Les simulations

Enfin, pour créer une classe de simulation qui puisse être chargée dans l'application `Framework`, celle-ci doit suivre le modèle donné par le squelette de la table 6.12. Il faut préciser le nombre d'instants par rafraîchissement de la simulation, et le temps minimal pour chaque étape d'exécution. Ensuite, il ne reste qu'à créer le workspace de plus haut niveau dans lequel les icobjs seront exécutés et de l'ajouter à l'objet `BootWorkspace`.

```
public class <Nouvel Icobj> extends Icobj
{
   public AttractorCreatorIcobj()
   {
      super(<Nom du Nouvel Icobj>);

      // definir si c'est un constructeur ou non
      setConstructor(...);

      // initialiser le comportement clonable et non clonables
      setBehavior(...);
      setNotCloneableBehavior(...);
   }
   // surcharger cette émthode si des champs doivent exister avant
   // l'execution de l'icobj
   public void reinit()

   ...
}
```

TAB. 6.11 – Squelette d'un nouvel icobj

```
public class <Demo name> extends BootWorkspace
{
   final static int width = DEFAULT_WIDTH, height = DEFAULT_HEIGHT;
   ...
   public <Demo name>()
   {
      super(width, height);

      // specifier le nombre d'instant par rafraichissement
      setNumberOfInstantsPerRefresh(...);

      // specifier la édure minimale d'uneé tape
      setMinimumDuration(...);

      // Specifier le workspace de plus haut niveau
      Workspace workspace = WORKSPACE_TO_USE_AS_TOP_LEVEL;
      setWorkspace(workspace);

      // Enregistrer les icobjs necessaires dans le workspace
      workspace.registerIcobj(...);
   }
}
```

TAB. 6.12 – Squelette d'une simulation

Un objet de type `BootWorkspace` étant un conteneur graphique *Java* pouvant être ajouté dans n'importe quel autre conteneur *Java*, il peut être utilisé dans une fenêtre d'une application (cf. table 6.13) ou dans une applet *Java*. Dans l'exemple de la table 6.13, dès que l'objet de type `BootWorkspace` est ajouté à une `Jframe`, il ne reste qu'à lancer un thread *Java* autonome pour exécuter son comportement.

```
public static void main(String[] args)
{
   // Instancier le BootWorkspace
   BootWorkspace demo = new <Demo name>();

   // Creer le conteneur graphique et ajouter le BootWorkspace

   // Demarrer la simulation pour qu'elle soit éexcute en boucle
   demo.start();

   // Creer et demarrer le thread autonome Java qui va se charger
   // d'executer le BootWorkspace
   Thread t = new Thread(demo, "Fight␣Club␣Demo");
   t.start();
}
```

TAB. 6.13 – Utilisation de la simulation en tant qu'application

6.8 Bilan

Le modèle des *Icobjs* dispose maintenant d'une API utilisable définissant une structure minimale extensible pour les icobjs et les workspaces et disposant d'un certain nombre de méthodes pour manipuler ces objets. Cette API contient aussi les constructeurs graphiques décrits dans la partie 5.2 et quelques exemples de comportements et de classes pour des champs ou comportements supplémentaires. Nous avons réglé les problèmes d'affichage inhérents à l'utilisation d'un système synchrone dans un environnement asynchrone. Par contre, la façon de gérer la partie graphique pourrait être améliorée. Par exemple, l'affichage complet du workspace est rendu dès qu'un de ses icobjs doit être réaffiché. De plus, nous n'avons pas encore mis en place de workspace qui prenne en charge un environnement 3D, même si les champs et comportements sont prévus pour les simulations 3D. Nous avons également défini un moyen pour réagir aux événements souris et clavier de *Java* en les transposant en événement réactif.

Au-dessus du modèle des *Icobjs*, nous avons ajouté des mécanismes d'introspection pour accéder aux différents champs composant un icobj et pour pouvoir les modifier. Nous offrons également un moyen de visualiser de façon textuelle le comportement d'un icobj et un moyen complémentaire pour le modifier. Ces mécanismes supplémentaires constituent une extension du modèle de construction graphique des *Icobjs* puisqu'il est possible de construire de nouveaux comportements par l'intermédiaire de l'inspecteur. Ces mécanismes sont actuellement accessibles en utilisant l'application `Framework` présente dans l'API. Cette application est un environnement permettant de tester rapidement les simulations et de les modifier dynamiquement en changeant les comportements des icobjs.

Chapitre 7

Expérimentations

Nous avons effectué plusieurs expérimentations pour tester la simplicité d'utilisation du modèle des *Icobjs*. Nous allons en particulier présenter trois expérimentations. La première concerne l'utilisation des *Icobjs* dans des simulations distribuées (cf. section 7.1). Cette expérimentation a été effectuée dans le cadre du projet européen IST-PING. La seconde expérimentation décrit la construction d'un modèle d'exécution pour les comportements physiques au-dessus des *Icobjs* (cf. section 7.2). Enfin, la troisième expérimentation concerne la mise en place de simulations multi-horloges (cf. section 7.3), c'est-à-dire de simulations dans lesquelles certains comportements doivent être exécutés à des rythmes différents. D'autres expérimentations ont été menées comme par exemple, celle menée par D. Pous sur l'utilisation des *Icobjs* pour simuler le fonctionnement de différents modèles d'*Ambients* [62] [80], mais nous ne les décrirons pas ici.

7.1 Projet PING

IST-PING [82] signifie *Platform of Interactive Networked Games*. Ce projet avait pour but de spécifier, développer une architecture ouverte pour l'Internet qui supporte des applications distribuées de mondes virtuels et qui peut supporter plusieurs milliers de connexions simultanées dispersées géographiquement. L'infrastructure du projet consistait en des composants de communication, de stockage et de simulations temps réel. La plate-forme IST-PING devait aussi proposer des facilités de programmations à partir d'objets actifs de haut niveau. Pour que l'utilisateur conserve une vue cohérente de son environnement tout en maintenant une interactivité et une navigabilité fluide, plusieurs techniques ont été utilisées à différents niveaux. Ces techniques utilisent la structure spatiale et la sémantique des grands mondes virtuels. Parmi les différents types de simulations distribuées, le projet IST-PING s'est concentré autour des jeux en réseau. En effet, ce type d'applications est très exigeant en ce qui concerne l'interactivité et le temps de réponse.

Le but était donc de créer une architecture qui permet de contenir aussi bien des simulations non persistantes, de courte durée avec peu d'utilisateurs et sans contrainte temps réel, que des simulations temps réel, persistantes, de longue durée avec un grand nombre d'utilisateurs et de gros volumes de données à partager à travers l'Internet. L'architecture devait répondre à certains critères. Entre autres, elle devait :

- contenir un espace partagé pour des centaines voire des milliers d'utilisateurs dispersés géographiquement ;

- être capable de supporter des mondes virtuels partagés de grandes tailles (niveau espace mémoire) qui nécessitent de grandes ressources de calcul ;
- permettre de continuer à faire vivre le monde même en l'absence des utilisateurs ;
- permettre de partager l'espace du monde entre les objets contrôlés par les utilisateurs et des objets autonomes et de les faire interagir en temps réel ;
- être supportée par des plates-formes et des environnements hétérogènes.

7.1.1 Architecture de la plate-forme

Pour répondre aux objectifs du projet, la plate-forme a été bâtie sur une architecture réseau distribuée et non centralisée, dans laquelle toutes les machines participent aux calculs de la simulation. Les avantages d'une telle architecture sont de pouvoir répartir la charge de calculs sur un très grand nombre de machines, d'avoir une meilleure tolérance aux pannes et d'être plus interactif. En effet, sur une architecture centralisée, seule la machine serveur (ou un nombre prédéfini de machines) se charge des calcul après avoir récupéré les informations envoyées par chaque client. Dans une architecture centralisée, les performances des simulations et le nombre d'objets pouvant être simulés sont donc fonction des ressources de la (ou des) machine(s) serveur(s) et de la qualité du réseau.

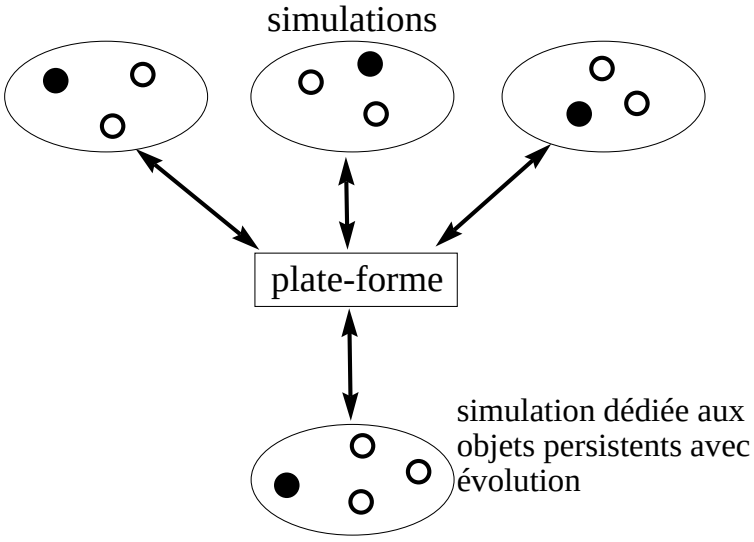


FIG. 7.1 – Architecture de la plate-forme IST-PING

Dans une architecture distribuée, chaque machine connectée au monde virtuel calcule sa propre représentation du monde virtuel en fonction des informations qu'elle a reçues. En ce sens, une simulation est plus réactive sur architecture distribuée que sur une architecture centralisée car elle calcule son propre résultat et n'a pas besoin d'attendre qu'une machine distante le fasse et que le résultat lui soit transmis par le réseau.

Techniquement, les spécifications essentielles de la plate-forme sont les suivantes :

- elle est basée sur un modèle orienté objet.
- elle dispose de protocoles de communication efficaces (pour avoir des temps de latence minime) et adaptables (selon le trafic).

- elle dispose de services de gestion d'objets de haut niveau comme par exemple la persistance des données, un système de réplication des objets et des mécanismes pour garantir la cohérence.
- elle utilise la notion d'*Aura*. Une aura est une zone définie autour de chaque objet partagée sur le réseau qui délimite la zone dans laquelle l'objet interagit directement avec les autres objets de la simulation.
- elle utilise l'approche réactive (*REJO*, *Icobjs*) pour décrire les comportements des objets partagés.

7.1.2 Mondes virtuels

Un monde virtuel, comme un jeu, est un environnement modifiable composé de deux types d'objets (cf. figure 7.2) : les entités passives comme par exemple des murs ou des éléments de décors sans aucun comportement autonome ; les entités actives qui évoluent au cours du temps par l'intermédiaire de leur comportement. Le fait que plusieurs machines partagent le même environnement entraîne qu'il peut y avoir plusieurs points de vue sur le même monde. Il y a en fait autant d'occurrences de la même entité qu'il y a de point de vue sur la simulation. La représentation d'une entité dans une simulation locale est appelée un *répliqua*.

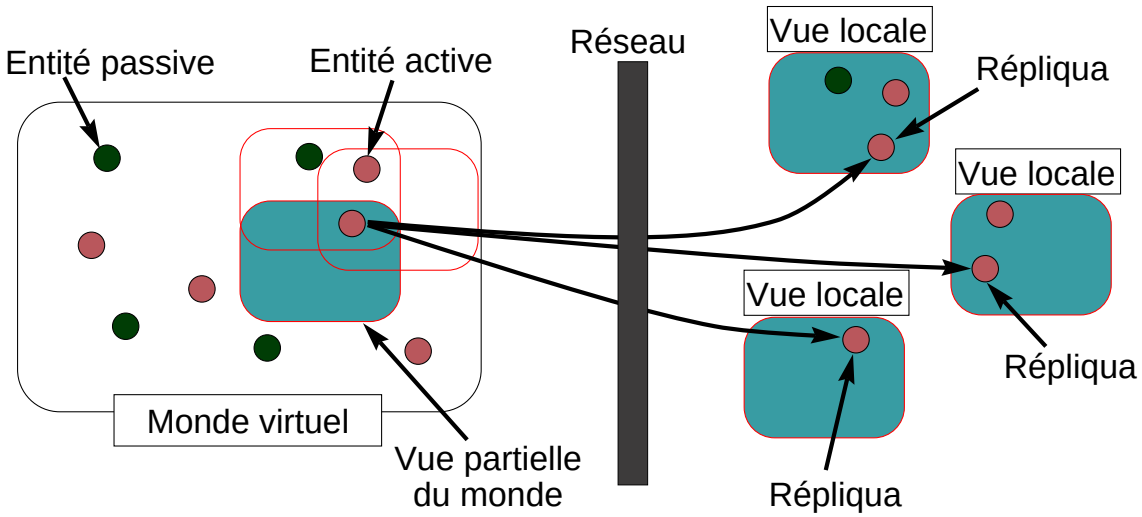


FIG. 7.2 – Vue d'un monde virtuel distribué

Notre rôle dans le projet IST-PING était d'intégrer *Junior* comme moteur pour exécuter les comportements des entités et d'utiliser les *Icobjs* comme environnement de programmation de haut niveau. Le moteur des *Icobjs* devait pouvoir exécuter un très grand nombre de comportements et réussir à garder une bonne interaction avec l'utilisateur. C'est pourquoi nous avons défini une architecture maître-esclave pour définir les comportements des entités le la même manière que pour la plate-forme. Cela signifie que, pour chaque entité, un répliqua (le maître) exécute l'ensemble du comportement de l'entité alors que tous les autres répliquas (les esclaves) ne disposent que d'un comportement dégradé dont le rôle est de reproduire ce que le maître fait. Cette architecture a l'avantage de permettre d'économiser les ressources en limitant la complexité des programmes des esclaves. Elle permet donc d'exécuter un plus grand nombre d'entités.

Le principal problème était de spécifier le comportement esclave en fonction du comportement maître. Pour cela, nous avons défini une nouvelle instruction de type `Behavior`. L'instruction `DistributedBehavior` permet de définir statiquement, pour un comportement donné, le programme que le maître doit exécuter et celui que les esclaves doivent exécuter. La plate-forme pouvant transférer le statut de maître d'une simulation à une autre, toutes les entités devaient connaître l'ensemble du comportement. Cela rendait la simulation d'autant plus réactive que la plate-forme n'avait pas à transférer le comportement de la machine du répliqua maître à celle des répliquas esclaves, mais seulement l'information du statut de maître. Il faut noter que le comportement exécuté par une machine réactive était modifié dynamiquement dès que le statut de maître changeait. De plus, dans le cadre de la construction graphique des *Icobjs*, il fallait pouvoir accéder aux deux comportements sur n'importe quelle simulation pour favoriser la réutilisation des icobjs.

Il a fallu ajouter quelques méthodes[1] dans la classe `Icobj` pour qu'elle puisse fonctionner avec la plate-forme IST-PING et plus particulièrement pour pouvoir contrôler les simulations distantes ou du moins modifier les valeurs des champs créés dynamiquement :

- `PingEntity getPingEntity()`
  Cette méthode retourne l'objet de type `PingEntity` associé à l'icobj. En effet, chaque entité partagée dans la simulation est associée à un objet au niveau de la plate-forme. Cet objet permet de connaître le statut du répliqua présent dans la simulation (maître ou esclave) et de récupérer l'objet `EntityManager` qui est chargé de contrôler, pour la simulation locale, tous les objets partagés par la simulation. L'objet `EntityManager` est celui associé au workspace de plus haut niveau.
- **`void`** `_getField(String name)`
  Cette méthode permet à un répliqua maître de demander la mise à jour du champ identifié par l'argument `name` dans toutes les simulations distantes contenant un répliqua esclave correspondant à la même entité. Tous les répliquas esclaves recevront alors la nouvelle valeur du champ.
- **`void`** `_setField(String name, String type, String value)`
  Cette méthode appelée par la plate-forme IST-PING force la mise à jour d'un champ local par la valeur reçue d'une simulation distante. Le champ `type` différencie simplement les objets *Java* des types primitifs. La valeur est envoyée sous forme de chaîne de caractères par la plate-forme. De même, il existe des méthodes correspondant à chacun des champs prédéfinis de l'icobj (nom, zone d'influence, comportements clonable et non-clonable).

Toutes les données provenant de la plate-forme sont mises à jour après chaque réaction de la machine réactive. De même que la machine réactive, la plate-forme dispose aussi d'une notion d'instant. Toute valeur envoyée vers une autre simulation ne sera pris en compte par celle-ci qu'en faisant réagir la plate-forme. Les mécanismes de cohérence de la plate-forme permettent de retarder certaines mises à jour de valeurs pour qu'elles aient lieu en même temps sur toutes les simulations.

Au niveau de la classe `Workspace`, nous avons ajouté la méthode :

`void` `registerPingIcobj(Icobj icobj)`

Cette méthode permet, tout comme `registerIcobj` le fait de façon locale, d'ajouter un icobj dans le workspace et de partager cet icobj à travers le réseau. Il suffit simplement de demander à la plate-forme de partager un nouvel icobj et les services se chargent de la

[1] ces modifications ont été apportées au modèle initial réalisé au début de la thèse

réplication et de la persistance. Au cours d'une construction graphique avec les *Icobjs*, il est inutile de partager tous les objets intermédiaires qui sont créés durant la construction. C'est pourquoi, nous avons ajouté un constructeur graphique (cf. figure 7.3) qui permet de partager un icobj. En activant ce constructeur un ou plusieurs icobjs, ceux-ci deviennent partagés entre toutes les machines connectées au monde virtuel.

FIG. 7.3 – Constructeur de *pingification*

D'autres modifications ont été apportées aux comportements eux-mêmes. Par exemple, l'algorithme de collision a été modifié pour s'exécuter de manière distribuée. Seul le répliqua maître détecte et réagit à la collision, alors que les répliquas esclaves ne font qu'émettre l'événement indiquant qu'on peut entrer en collision avec eux. Cela permet de diminuer la complexité locale qui est initialement en $O(\frac{N^2}{2})$. Avec ce nouvel algorithme, la complexité locale est en $O(M.(N-1))$, M étant le nombre de répliquas maîtres présents localement. Par contre, la complexité globale est en $O(N^2)$. La complexité globale est 2 fois plus importante car les répliquas maîtres détectent et calculent chacun l'effet de la collision. La mise en place de ces modifications permet d'économiser les ressources de calculs de chaque machine, ce qui permet d'augmenter le nombre d'objets simulables. Des informations complémentaires peuvent être trouvées dans [19].

7.1.3 Communication entre les objets

Hormis les communications pour établir la connexion, les communications entre entité vont toujours du répliqua maître vers tous les répliquas esclaves. La figure 7.4 représente l'objet de liaison créé par la plate-forme qui permet aux répliquas d'une même entité de communiquer. Il n'y a aucune communication réseau entre répliquas représentant différentes entités. Les communications entre les entités se font localement dans chaque simulation en utilisant les événements réactifs.

Au niveau réseau, l'objet de liaison correspond à une adresse multicast. Le principal intérêt d'attribuer un canal de communication par entité est que le traitement des informations est accéléré puisque les entités accèdent directement aux données qui leur sont dédiées. De plus, pour économiser des ressources au niveau du réseau, une simulation locale ne se branche que sur les adresses multicasts qui concernent les objets présents dans son aura.

7.1.4 Cohérence

Dans une architecture distribuée, un des problèmes récurrents est celui de la cohérence de la simulation. En effet, puisque les sources de calculs sont multiples, il n'y a aucune garantie que les simulations arrivent aux mêmes résultats. Les possibilités d'incohérences sont multiples :

- L'incohérence est liée principalement au temps de latence dans le transport des informations entre les différentes simulations. De plus, pour éviter de surcharger le réseau, les variables des objets partagées ne peuvent être mises à jour constamment. Il n'y a

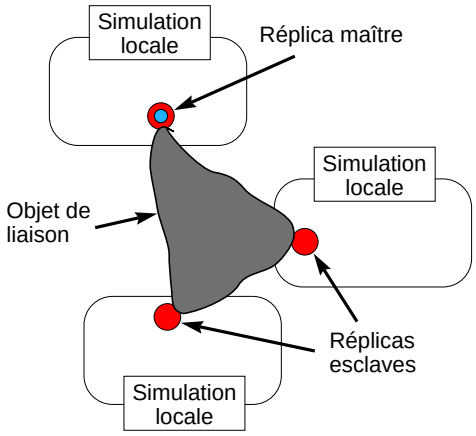


FIG. 7.4 – Schéma de communication de la plate-forme

donc pas de synchronisation forte concernant les valeurs des objets partagés. Cela a pour conséquence que les simulations effectuent des calculs à partir de données qui ne sont pas identiques sur toutes les simulations.

- Dans le cadre des *Icobjs*, le nombre d'instants exécuté par chaque simulation est fonction de la puissance de la machine. Le nombre d'instants qui s'écoule sur chaque machine n'est donc pas nécessairement le même et la vitesse à laquelle évoluent les entités partagées est différente sur chaque simulation.
  Un modèle de machines réactives distribuées a été spécifié dans [15]. Dans ce modèle, toutes les machines réactives présentes sur différents sites sont synchronisées. Une machine spécifique désignée comme le *Synchroniseur* diffuse les événements générés dans chacune des autres machines. Dans ce modèle, toutes les machines partagent le même instant et la fin de l'instant est déclarée par le *Synchroniseur*. L'utilisation d'un tel modèle dans des simulations distribuées gérant un grand nombre d'entités (et donc un grand nombre d'événements) est coûteuse. En effet, pour chaque événement généré dans une machine, il faut transmettre cet événement aux autres sites en utilisant le réseau. De plus, de nombreuses communications réseau sont nécessaires pour garantir qu'aucun événement n'a été généré sur une des machines pendant que le synchroniseur déclare la fin d'instant. Ces opérations prennent beaucoup de temps, ce qui est incompatible avec la bonne réactivité de la simulation où la durée des instants doit être courte.
- Le mécanisme d'*aura* permet d'exécuter uniquement les entités se trouvant dans une certaine partie du monde virtuel. Ce mécanisme permet d'économiser des ressources, mais apporte en même temps son lot d'incohérence. En effet, il est difficile d'obtenir les mêmes résultats sur plusieurs simulations à partir du moment où chacune n'exécute pas nécessairement les mêmes comportements et ne reçoit donc pas nécessairement les mêmes événements.

Toutes ces raisons font qu'il est nécessaire de mettre en place des mécanismes pour permettre de maintenir la cohérence globale sans que toutes les variables soient mises à jour en permanence. La plate-forme IST-PING offre quelques mécanismes pour maintenir la cohérence dans la simulation, comme par exemple le fait de retarder certaines mises à jour de champs.

Au niveau applicatif, nous distinguons deux types d'incohérences : spatiales et temporelles. Les incohérences spatiales se caractérisent par des erreurs de position des répliquas esclaves par rapport à celle de leur maître. Les incohérences spatiales sont liées aux délais de déclenchement de la même action sur différentes simulations. Ces différences de délais peuvent affecter les décisions prises par chacune des simulations locales. Pour répondre à ces incohérences au niveau applicatif, nous avons expérimenté un algorithme de *Dead-Reckoning* que nous allons maintenant considérer.

7.1.5 Mécanisme de dead-reckoning

Les algorithmes de *Dead-Reckoning* sont des algorithmes qui permettent d'extrapoler les positions des différentes entités en l'absence de mise à jour. Le comportement inertiel peut être considéré comme une méthode primitive de dead-reckoning. En effet, ce comportement permet à un répliqua esclave d'être autonome dans son déplacement pendant les instants où il ne récupère pas d'informations de son maître. Cependant ce mécanisme est insuffisant pour pouvoir garantir une bonne cohérence, puisque les différentes simulations locales ne fonctionnent pas nécessairement à la même vitesse.

Nous ne détaillerons pas notre algorithme dont voici uniquement les idées générales. La première idée est de compenser, pour chaque répliqua esclave, la dérive de rapidité d'exécution entre sa simulation locale et la simulation qui exécute son répliqua maître. Par exemple, si la simulation du maître va 1.5 fois plus vite que celle du répliqua esclave, le vecteur vitesse du répliqua esclave est multiplié par 1.5. Cela permet de compenser la différence entre le nombre d'instants exécutés sur chacune des deux simulations et donc la différence de distance parcourue. La seconde idée est de rattraper progressivement l'écart de position [30] qu'il peut y avoir entre le répliqua maître et le répliqua esclave, ce qui permet d'éviter les sauts de position qui pourraient paraître incohérents aux yeux de l'utilisateur.

Le mécanisme de dead-reckoning doit être appliquée individuellement par chaque répliqua esclave car, comme le montre la figure 7.4, chaque entité dispose de son propre canal de communication. Chacun des canaux n'est pas nécessairement soumis aux mêmes temps de latence. De plus, le temps de transmission diffère selon que la distance géographique est grande ou non entre une simulation exécutant un maître et une autre exécutant un des esclaves correspondant à cette entité.

7.1.6 Bilan

Les expérimentations faites autour des algorithmes de *Dead-Reckoning* sont décrites dans [21]. On y trouve une description plus détaillée de l'algorithme et les résultats de son application. En l'absence de plate-forme, le mécanisme de dead-reckoning a été testé par l'intermédiaire d'un petit jeu réalisé en utilisant *Java RMI*. D'autres techniques pour assurer une meilleure cohérence restent à étudier. Il serait par exemple intéressant de faire varier dynamiquement la durée de resynchronisation entre le maître et les esclaves en fonction des erreurs observées par les esclaves. Un autre technique consisterait à utiliser des mécanismes de *"flous artistiques"* qui masquent les erreurs d'approximations (autre que des erreurs de positionnement) et les corrigent à l'aide de communications supplémentaires.

7.2 Simulation physique

Dès les premières expérimentations des *Icobjs*, des simulations ont été créées pour représenter des phénomènes physiques comme la gravité ou des liens élastiques. La modularité de l'approche réactive permettait de représenter ces phénomènes, non par des équations différentielles portant sur toute la simulation comme c'est le cas dans ACSL [54] ou dans *Dymola* [28], mais par la composition de petits comportements élémentaires. Le problème des simulations décrites à l'aide de comportements élémentaire est que, même si elles semblaient correspondre aux modèles physiques à l'écran, ce n'était absolument pas le cas d'un point de vue numérique. Pour répondre à ce problème, A. Samarin a mis en place un modèle de comportements physiques [68] dont nous nous sommes servis pour effectuer des expérimentations sur le modèle actuel des *Icobjs*. Nous allons décrire ce modèle et la manière dont il est mis en oeuvre dans l'API des *Icobjs*.

7.2.1 Modèle des comportements physiques

Pour modéliser les comportements physiques, l'idée de départ est que chaque force soit représentée par la génération d'un événement. Par exemple, une planète génère un événement valué pour simuler le champ de gravité. La valeur de l'événement est l'objet émetteur lui-même. Tout astéroïde devant être soumis à la gravité récupère alors l'événement et réagit à cet événement par l'intermédiaire d'une action atomique. Si plusieurs planètes sont présentes dans la simulation alors chacune d'elles génère son propre événement et les astéroïdes réagissent à tous ces événements, ce qui est une des raisons de l'introduction de l'instruction `Scanner`.

Le problème d'un tel modèle de comportements est que l'action atomique (la `ScanAction`) est exécutée à chaque occurrence d'un événement et que cette action atomique applique directement les modifications sur l'objet sans connaître les modifications apportées par les occurrences précédentes de l'événement. Donc, l'état final de l'objet dépend de l'ordre dans lequel les événements de forces ont été générés. Pendant un instant, un objet réagit et se déplace plusieurs fois en fonction des différentes générations d'événements. Le problème est que dans un modèle physique réel, toutes les forces sont appliquées en même temps sur les objets et non séquentiellement. En appliquant séquentiellement les forces, la réaction à chacune des forces reçues modifie l'état de l'objet et cette modification change l'effet que produisent les événements appliqués ensuite. Pour reprendre l'exemple de la gravité, la loi de *Newton* sur la gravitation universelle indique que les corps s'attirent proportionnellement à leur masse et à l'inverse du carré de leur distance. En réagissant à une occurrence d'un événement de gravité, un objet sensible à celui-ci modifie sa position. Cette modification de position, alors que l'objet peut encore réagir à d'autres occurrences du même événement, entraîne une erreur dans l'application de la loi de *Newton*. En effet, si l'objet réagit à une autre source de gravité au cours du même instant, la distance entre cet objet et la seconde source n'est plus la même qu'au début de l'instant et donc la valeur de la force de gravité qui dépend de la distance entre les deux objets est modifiée.

Pour répondre au problème de non-instantanéité (au sens physique) de l'application des différentes forces, A. Samarin a proposé un modèle où une réaction physique élémentaire est découpée en deux phases qui correspondent chacune à un instant de la machine réactive (cf. figure 7.5). Durant la première phase, chaque objet émet toutes les forces qu'il applique aux autres objets (gravité, lien élastique, absorption...) sous forme d'événements puis il récupère

l'ensemble de ces forces par l'intermédiaire d'actions atomiques. La différence par rapport aux modèles présentés plus haut est que la force n'est pas immédiatement appliquée à l'objet. Les actions atomiques ont seulement pour rôle de calculer la force résultante de l'ensemble des forces déjà reçues au cours de cette phase et de la stocker dans une variable de l'objet. La seconde phase applique la force résultante à l'objet en utilisant une méthode de résolution d'équations, ce qui assure une réaction instantanée (au sens physique) à l'ensemble des forces.

La solution numérique [3] étant une approximation, plusieurs algorithmes sont possibles que l'on va choisir entre autres en fonction de leur niveau de conformité aux lois physiques, de la précision nécessaire et des ressources disponibles. A. Samarin a étudié l'utilisation de quatre algorithmes : *Euler*, *Euler-Richardson*, *Velocity Verlet* et *Runge-Kutta d'ordre 4*. Selon la méthode de résolution utilisée, une réaction physique complète peut nécessiter plusieurs réactions physiques élémentaires. L'algorithme d'*Euler* n'utilise qu'une seule réaction physique élémentaire comme cela est représenté sur la figure 7.5. Les autres algorithmes cités utilisent des estimations d'étapes intermédiaires, ce qui nécessite plusieurs réactions physiques élémentaires : l'algorithme *Velocity Verlet* nécessite deux réactions physiques élémentaires donc quatre instants de la machine réactive et celui de *Runge-Kutta d'ordre 4* nécessite quatre réactions physiques élémentaires donc huit instants de la machine réactive.

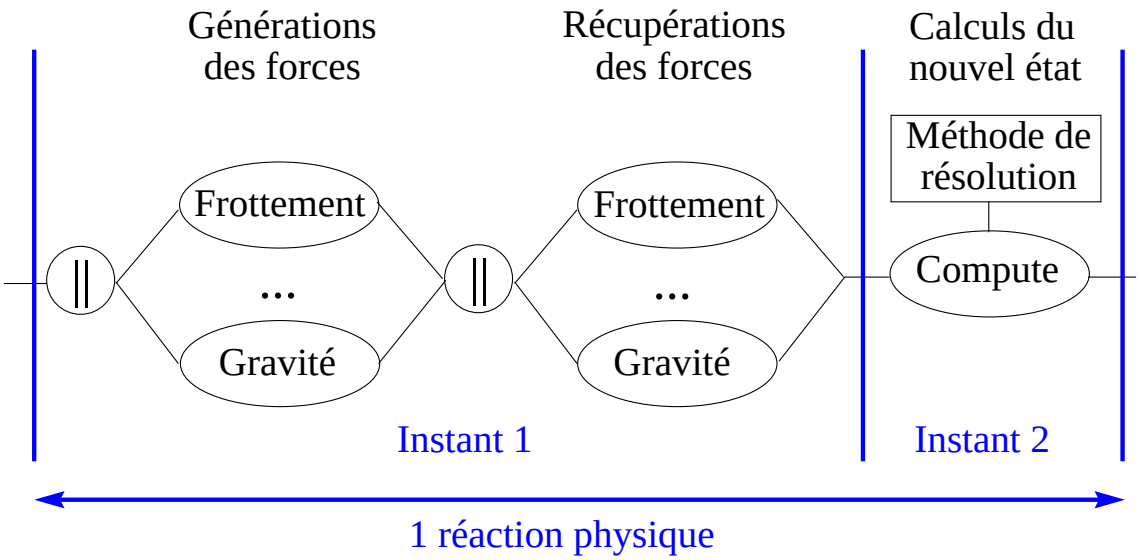


FIG. 7.5 – Les deux phases d'une réaction physique élémentaire

7.2.2 Implémentation

Un des points importants est de synchroniser tous les comportements pour qu'ils soient dans la même phase en même temps, c'est-à-dire soit dans la phase génération et récupération des forces, soit dans la phase de calcul du nouvel état. Pour effectuer cette synchronisation, le workspace génère tous les deux instants, dans sa propre machine réactive, un événement commun (`Constants.STEP`) pour déclencher la phase de génération et de récupération des événements de forces.

```
public class PhysicalBehavior extends Behavior
{
   private Program init = Ic.Nothing();
   private Program eventPhase = Ic.Stop();
   private Program computePhase = Ic.Stop();
   private Program executingBehavior;

   ...
      executingBehavior = Ic.Seq(init,
         Ic.Loop(
            Ic.Seq(Ic.Await(Constants.STEP),
               Ic.Seq(eventPhase, computePhase))));
   ...
}
```

TAB. 7.1 – Classe pour les comportements physiques

Chaque entité physique exécute un comportement de type `PhysicalBehavior` (cf. table 7.1) qui définit trois programmes :

- la partie `init` contient l'ensemble des champs à rajouter à l'icobj et leur initialisation ;
- la partie `eventPhase` contient l'ensemble des générations des événements de forces de l'icobj et la récupération de tous les événements de forces auxquels il est sensible. Ce comportement se traduit par une séquence réalisant d'une part les générations d'événements en parallèle et d'autre part une attente de tous les événements en utilisant l'instruction `Scanner`. Toutes les instructions `Scanner` doivent être mises en parallèle car leur exécution prend un instant. Cette phase est précédée d'une attente sur l'événement `Constants.STEP` qui garantit que toutes les entités sont synchronisées sur la même phase.
- la partie `computePhase` comprend la partie calcul du nouvel état et les autres comportements comme par exemple les rebonds sur les bords de la simulation. Cette partie comprend également la réinitialisation du champ contenant la force à appliquer à l'icobj car ce champ est réutilisé pour récupérer les forces de l'instant suivant.

Il faut faire attention à ce que les comportements utilisés pour les parties `eventPhase` et `computePhase` prennent exactement un instant chacun. Si l'une des deux phases dure plus longtemps, ou si les deux phases sont instantanées, le modèle des comportements physiques ne fonctionne plus. En effet, les événements qui seraient générés durant la deuxième phase ne seraient pas pris en compte ou les instructions `Scanner` exécutées durant la deuxième phase ne récupèreraient aucun événement de force.

7.2.3 Bilan

L'implémentation initiale d'A. Samarin a été réalisée au-dessus des *SugarCubes*. Le problème est que l'API n'était pas clairement définie et que l'implémentation des *PhysIcobj* n'était pas basée sur une notion claire et modulaire des *Icobjs*. Nous avons porté toutes les classes définies par A. Samarin en les conformant à notre modèle des *Icobjs*. Cela se traduit par une séparation des classes en fonction de leur utilisation (champs, comportements, entités) pour plus de modularité et par la création d'actions atomiques pour initialiser chacun des champs. De plus, les comportements et les champs ont été réimplémentés pour pouvoir interagir dynamiquement avec eux par l'inspecteur des *Icobjs*. Nous pouvons toutefois noter que ce modèle

a pu être porté assez facilement dans notre version des *Icobjs* puisque l'implémentation de ce modèle n'avait modifié ni la machine réactive pour prendre en compte des éléments particuliers, ni le workspace en ajoutant des méthodes spécifiques. La figure 7.6 représente une des simulations portées sur notre modèle des *Icobjs*. Dans cette simulation, les particules sont reliées entre elles par des liens rigides et l'ensemble de la structure est relié à un point fixe par l'intermédiaire d'un lien élastique. Toutes ces particules sont soumises à la gravité.

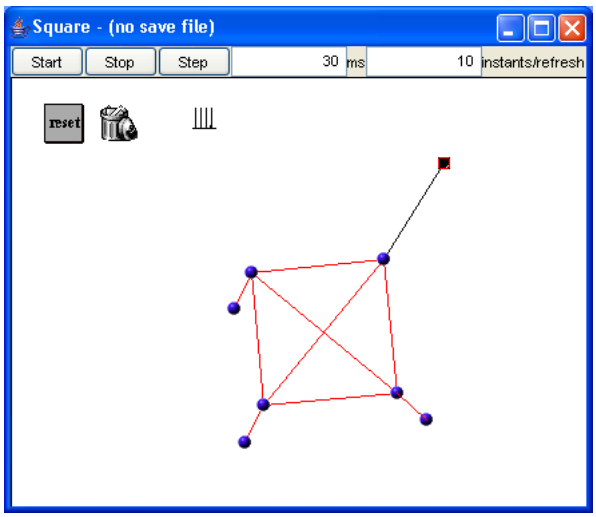


FIG. 7.6 – Capture d'écran d'un exemple de simulation physique

Il reste cependant des problèmes et des limitations dans le modèle d'A. Samarin. En utilisant le mécanisme de construction graphique des *Icobjs*, la mise en séquence des comportements n'est pas possible puisque tous les comportements bouclent à l'infini. De plus, pour assurer le modèle physique, il faut que tous les comportements soient exécutés en parallèle.

Un des aspects contraignants est qu'il faut que toutes les entités utilisent la même méthode de résolution. Si des entités utilisent l'algorithme de *Euler* et d'autres celui de *Runge-Kutta d'ordre 4*, elles ne seront pas synchrones. En effet, le premier algorithme nécessite une réaction physique et le second en nécessite quatre. Par exemple, il serait intéressant qu'une entité puisse passer dynamiquement d'une méthode de résolution à une autre pour des situations particulières où le calcul nécessite d'être plus précis et donc d'utiliser des étapes intermédiaires supplémentaires. Par exemple, dans le cas d'une collision, il serait utile d'utiliser une méthode de résolution précise pour calculer la trajectoire exacte résultant de la collision. De même, en ajoutant un nouvel icobj à la simulation, il faudrait pouvoir le synchroniser avec les autres entités déjà présentes. En plus du mécanisme de synchronisation qui permet à tout comportement d'être dans la même phase de réaction physique élémentaire, il faudrait un mécanisme pour synchroniser toutes les réactions physiques complètes.

7.3 Simulation multi-horloges

Dans le projet IST-PING, nous avons orienté nos travaux vers des comportements dédiés à des jeux. Dans la partie précédente, nous avons décrit un modèle d'exécution pour les comportements physiques qui nécessitent plusieurs instants de la machine réactive. La dernière expérimentation que nous allons présenter concerne les simulations multi-horloges. L'objectif

de cette expérimentation est de réunir dans une même simulation des comportements orientés jeux et des comportements physiques en définissant une notion d'instant commun aux deux types de comportements.

7.3.1 Description

Nous avons décrit dans la partie précédente un modèle de réaction physique élémentaire (cf. figure 7.5) qui prend deux instants de la machine réactive. En mélangeant comportements de jeux et comportements physiques, il faudrait exécuter un seul instant des comportements de jeux pour deux instants des comportements physiques pour que effectue une réaction complète à chaque instant global de la simulation. Nous avons évoqué le fait que certains algorithmes de résolution des comportements physiques nécessitent plusieurs réactions physiques élémentaires pour être dans un état stable physiquement. Il serait intéressant que les entités puissent changer de méthode de résolution quand elles en ont besoin tout en synchronisant chaque entité pour qu'à chaque instant de la simulation globale, chaque entité exécute une réaction physique complète.

Si tous les programmes (physiques ou non) sont placés dans la même machine réactive, les événements qui y sont attachés sont enregistrés dans le même environnement d'exécution. Considérons le cas de comportements physiques dont la réaction complète nécessite quatre instants de la machine alors que les comportements de jeu ne doivent réagir qu'une fois. Cela signifie que les comportements physiques nécessitent la réinitialisation de l'environnement d'exécution (des événements présents) quatre fois alors que les comportements de jeux doivent le réinitialiser qu'une seule fois et donc que des événements peuvent être perdus dans la gestion des comportements de jeux. La solution à ce problème est de multiplier les environnements d'exécution en fonction de l'horloge des comportements, c'est-à-dire de dédier un environnement d'exécution aux comportements de jeux et d'en dédier un autre aux comportements physiques. Pour cela, nous avons réalisé un workspace contenant deux machines réactives que nous présentons dans la partie suivante.

7.3.2 Implémentation

L'objectif de cette expérimentation est d'exécuter dans le même workspace des entités aux comportements de jeux représentées sur la figure 7.7 et des entités aux comportements physiques reprises de la simulation représentée sur la figure 7.6. La figure 7.8 représente une capture d'écran de la simulation multi-horloge obtenue.

La figure 7.7 représente une simulation dont chaque entité reprend des comportements de jeux. Cette simulation contient deux robots aux comportements autonomes et deux robots contrôlables par l'utilisateur. Le comportement des robots autonomes est de fuir (comportement de proie) les deux autres robots lorsqu'ils sont trop proches d'eux et de chasser les deux autres robots (comportement de prédateur) si l'un des deux est détruit. Les deux autres robots sont contrôlés en utilisant les touches du clavier pour changer la direction de leur canon ou pour lancer des projectiles : soit des bombes qui détruisent les robots qui sont, au moment de l'explosion, dans la zone d'influence de la bombe, soit des toiles d'araignées qui attachent par un élastique à un point fixe les robots qui sont, au moment de l'explosion, dans la zone d'influence de la toile d'araignée. Le lancement d'un projectile donne une impulsion au robot qui se déplace dans la direction opposée à celle du canon. Tous les robots ont un comportement inertiel, un comportement de rebond sur les bords du workspace et un comportement

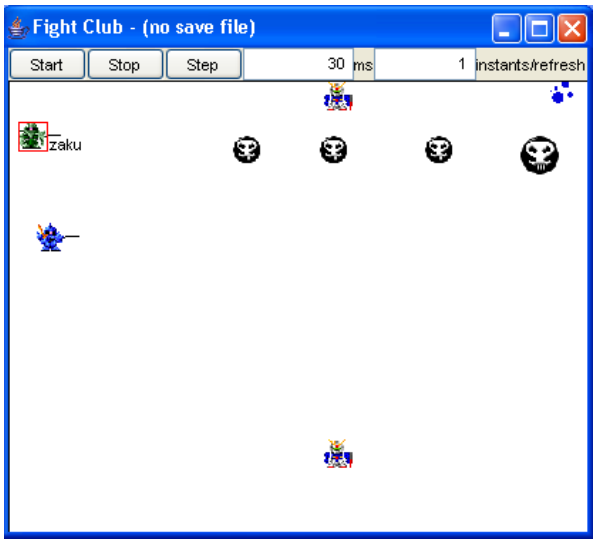


FIG. 7.7 – Capture d'écran d'une simulation orientée jeu

de collision. Aucun de ces comportements n'est basé sur le modèle physique.

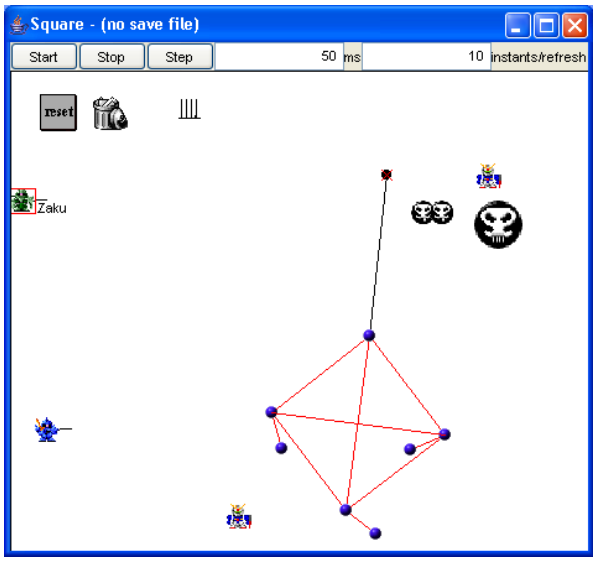


FIG. 7.8 – Capture d'écran d'une simulation multi-horloge

Pour réaliser notre simulation multi-horloge, nous avons défini un workspace, `MultiClockWorkspace` qui exécute deux machines réactives différentes : l'une dédiée à l'exécution des comportements physiques (comme pour le `PhysicalWorkspace`) et l'autre dédiée à l'exécution des comportements de jeux. Dans ce nouveau workspace, nous avons dû redéfinir les méthodes `react`, `addIcobj`, `destroyIcobj`, `resetIcobj` et les méthodes `generate` et nous avons ajouté une méthode `registerPhysicalIcobj`. Voici le rôle de chacune de ces méthodes :

- `void registerPhysicalIcobj(Icobj icobj)`
  Cette méthode enregistre l'icobj dans le workspace en ajoutant le comportement de celui-ci dans la machine chargée d'exécuter les comportements physiques. Le compor-

tement ajouté dans la machine est construit comme dans la méthode `registerIcobj`.

- **boolean** `add(Icobj icobj, Program program)`
  Cette méthode ajoute un programme à un icobj enregistré dans le workspace. Cette méthode ajoute le programme dans la machine dans laquelle l'icobj est déjà enregistré.
- **boolean** `destroyIcobj(Icobj icobj)`
  Cette méthode retire l'icobj du workspace. Ce retrait se traduit par celui de la liste des icobjs et par une demande à la machine dans laquelle l'icobj est exécuté de geler le comportement de l'icobj à la fin de l'instant suivant. De plus, un événement constitué par l'identifiant de l'icobj et suffixé par "-kill" est généré dans les deux machines pour signaler le départ de l'icobj.
- **void** `resetIcobj(Icobj icobj)`
  Cette méthode réinitialise le comportement dans la machine dans laquelle l'icobj est exécuté et les champs de l'icobj en appelant la méthode `reinit` de l'icobj. Le comportement est alors immédiatement reconstruit de la même manière que lorsque la méthode `registerIcobj` est appelée, puis il est ajouté dans la machine.
- **void** `generate(...)`
  Les quatre méthodes de génération (avec ou sans valeur, et en utilisant une chaîne ou un `Identifier`) sont modifiées pour effectuer les générations externes dans les deux machines. Cela donne un moyen de communication par événement entre les entités physiques et les entités de jeux.
- **boolean** `react()`
  Cette méthode exécute une réaction de la machine contenant les comportements de jeux et exécute quatre réactions de la machine contenant les comportements physiques. Le nombre d'instant est ici défini statiquement puisque nous utilisons la méthode de résolution *Verlet-Velocity* qui nécessite quatre instants pour exécuter une réaction physique complète.

Pour créer la simulation représentée sur la figure 7.8, nous avons repris le code des simulations représentées sur les figures 7.7 et 7.6 et nous avons simplement changé, pour les entités de la simulation physique, la méthode appelée pour enregistrer les entités dans le workspace en utilisant `registerPhysicalIcobj`.

7.3.3 Bilan

Les résultats de cette simulation sont que les entités de jeux peuvent agir sur le modèle physique, par exemple par l'intermédiaire des bombes qui détruisent la structure. Le moyen de communication utilisé entre les entités physiques et les entités de jeux est le partage de la zone d'influence des icobjs. De plus, certaines actions atomiques peuvent effectuer des générations externes d'événements, c'est-à-dire appeler la méthode `generate` sur le workspace. Par contre, toute génération effectuée par l'instruction `Generate` est locale à la machine qui l'exécute.

La simulation multi-horloge réalisée n'est qu'une première mise en oeuvre qui nous a permis de tester l'utilisation des *Icobjs* dans le cadre multi-horloge. Nous avons défini le comportement du `MultiClockWorkspace` statiquement sur la base de quatre instants physiques pour un instant de jeu. Il serait intéressant de réaliser d'autres modèles pour rendre plus générique le workspace multi-horloges. En effet, cette simulation exécute soit des entités aux comportements physiques, soit des entités aux comportements de jeux. Il faudrait pouvoir exécuter des icobjs qui ont à la fois des comportements de jeux et des comportements physiques et faire que chaque comportement s'exécute sur la machine correspondante.

Chapitre 8

Conclusions et perspectives

Les travaux présentés dans ce document 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, Jean-Ferdinand Susini, Raùl Acosta-Bermejo et Louis Mandel sur le développement de l'Approche Réactive Synchrone au-dessus de *Java*. Nous nous sommes principalement intéressés à réaliser un outil permettant à la fois de simuler des entités graphiques dans un environnement synchrone et de construire graphiquement et dynamiquement de nouveaux comportements en réutilisant le modèle initial des *Icobjs*.

8.1 Réalisations

À partir du modèle des *Icobjs* décrit par F. Boussinot dans [12] et [13] et des expérimentations menées par J-F Susini, nous avons développé une véritable API des *Icobjs*. Pour assurer une exécution efficace des *Icobjs*, nous avons réalisé un moteur réactif dédié à leur utilisation, appelé *Reflex*. Pour manipuler et interagir dynamiquement avec les simulations et les entités qui les composent, nous avons réalisé un environnement, appelé `Framework`. Nous avons enfin effectué plusieurs expérimentations en utilisant les *Icobjs* dans différents cadres. Voyons chacun de ces points plus en détails.

Les *Icobjs*

L'API que nous avons réalisée reprend le système de construction graphique de comportements de la version initiale des *Icobjs*. Pour rendre cette construction graphique plus générique, nous avons défini un modèle d'objet, `Icobj`, ayant une structure de données minimale. Cette structure contient à la fois des informations graphiques (position, dimensions, apparence) et des informations comportementales. La partie comportementale est divisée en deux parties : l'une étant utilisée par le système de construction graphique et l'autre étant un comportement spécifique à l'icobj. Cette structure de données peut être dynamiquement étendue par l'intermédiaire des comportements. Le `Workspace` prend en charge l'exécution des comportements des icobjs en utilisant la machine réactive de *Reflex* et l'affichage des apparences des icobjs entre les réactions de la machine. Le workspace, étant également un icobj, dispose aussi d'un comportement et peut être ajouté à l'intérieur d'une simulation comme un environnement indépendant dans lequel tout événement généré est local à ce workspace.

Le principe de construction reste le même que dans les versions précédentes des *Icobjs* : des entités particulières, appelées constructeurs graphiques, récupèrent les comportements des autres entités et combinent ces comportements en rajoutant des primitives de contrôle (séquence, parallèle, boucle, préemption, etc.). Nous y avons apporté plusieurs modifications. Nous avons en particulier ajouté de nouveaux constructeurs graphiques et homogénéisé la manière de les utiliser. Pour rendre la construction graphique plus régulière et plus intuitive, nous avons introduit de nouvelles instructions dans *Reflex*. De plus, le système de construction graphique prend en compte la notion de workspace et un icobj résultant d'une construction peut être lui-même un workspace.

Au niveau de l'implémentation, nous avons mis en place des mécanismes pour récupérer les événements souris et clavier de *Java* et les transformer en événements réactifs générés. Au niveau de l'affichage, nous ne disposons actuellement que d'un affichage en 2D en utilisant la bilbiothèque *Java-Swing*. Les opérations comme l'affichage d'un workspace, la migration d'un icobj ou son enregistrement dans un fichier sont effectuées entre les réactions du workspace pour garantir la cohérence de l'affichage et des données enregistrées. La migration ne peut se faire actuellement qu'entre workspaces de la même machine virtuelle *Java*.

Reflex

Pour pouvoir exécuter efficacement les comportements des icobjs, nous avons réalisé notre propre moteur réactif, *Reflex*. Pour son implémentation, nous sommes partis de la version *Storm* de *Junior* en y apportant plusieurs modifications concernant l'efficacité et concernant le changement de sémantique de certaines instructions. Voici les principales modifications :

- *Reflex* est dédié à l'exécution des icobjs. Tout programme ajouté à la machine doit nécessairement être rattaché à un icobj. Nous avons ajouté une instruction interne à la machine `IcobjThread` pour remplacer les instructions `Link` et `Freezable`. L'utilisation de cette instruction interne permet d'effectuer des migrations dès la fin de l'instant.
- Nous avons ajouté le statut d'exécution *LONGWAIT* utilisé par les instructions événementielles dont les attentes peuvent être inter-instants. Ce nouveau statut évite de ré-exécuter des instructions dont la configuration événementielle n'est pas satisfaite. Ce point est particulièrement intéressant pour les icobjs dont les comportements sont en attente de différents événements.
- Nous avons retiré la configuration événementielle `Not` pour plus de régularité dans l'exécution des instructions événementielles. Le retrait de cette configuration assure que toutes instructions événementielles réagiront toujours à l'instant courant si la configuration est satisfaite. Cette régularité peut permettre d'envisager des analyses statiques sur les comportements.
- L'instruction `Par` est, à l'inverse de *Storm*, non-déterministe, ce qui permet un gain d'efficacité en accédant directement aux instructions qui ont besoin d'être exécutées.
- Nous avons mis en place un mécanisme de nettoyage de l'environnement d'exécution pour économiser la mémoire en retirant tous les événements obsolètes.

Reflex est aussi rapide que les implémentations les plus efficaces de *Junior* (*Simple* et *Glouton*). Il est basé sur une sémantique formelle constituée de 58 règles de réécritures utilisant le formalisme SOS. Le nombre de règles est assez important, mais cette sémantique définit les règles strictes du fonctionnement de *Reflex* en se rapprochant au plus près de l'implémentation.

Le Framework

Autour du modèle des *Icobjs*, nous avons développé un environnement pour manipuler les simulations construites à partir de l'API des *Icobjs*. Le `Framework` permet, entre autres :

- de charger des simulations à partir des classes *Java*, de les exécuter dans un environnement multi-fenêtré et de paramétrer leur vitesse d'exécution ;
- de charger des icobjs à partir des classes *Java* et de les ajouter dans un workspace en train d'être exécuté ;
- de sauvegarder l'état d'une simulation dans un fichier et de pouvoir recharger ce fichier ;
- et d'exécuter l'inspecteur des *Icobjs*.

L'inspecteur des *Icobjs* du `Framework` permet de visualiser et de modifier dynamiquement les comportements et les champs des icobjs enregistrés dans les simulations. Pour cela, nous avons introduit deux mécanismes d'introspection. Le premier consiste à récupérer, pour chaque champ d'un icobj, une liste d'interfaces qui permet de visualiser et de modifier dynamiquement les valeurs contenues dans ces champs. Pour cela, il faut que, dans chaque classe représentant un champ de l'icobj, le programmeur spécifie de manière explicite les attributs visualisables et modifiables. Le second permet de récupérer une copie des programmes exécutés par l'icobj et de l'afficher de manière textuelle sous forme d'arbre. L'utilisateur peut modifier cette copie soit en ajoutant de nouvelles instructions réactives, soit en en retirant, soit en changeant leur ordonnancement. La mise à jour du comportement doit être demandée explicitement par l'utilisateur alors que celle des champs est faite automatiquement entre les réactions de la simulation exécutant l'icobj.

La possibilité que l'inspecteur fournit de transformer le comportement des icobjs est un mécanisme complémentaire à celui des constructeurs graphiques. Cependant, ce type de construction nécessite des connaissances supplémentaires sur l'utilisation de l'approche réactive synchrone. Ainsi, le `Framework` peut être considéré comme un éditeur de scénarios dans lequel on peut tester rapidement et dynamiquement les modifications apportées aux comportements et aux paramètres d'exécution de chaque entité.

Les expérimentations

Enfin, nous avons effectué quelques expérimentations. Les premières concernant l'utilisation des *Icobjs* dans une plate-forme de simulations distribuées ont été interrompues suite à l'arrêt du projet européen IST-PING. Cependant, ces expérimentations ont tout de même permis d'étudier les problèmes liés à la synchronisation des icobjs exécutés sur différentes machines ne partageant pas la même notion d'instant. Nous avons réalisé un modèle de comportement distribué où, pour chaque entité, nous avons différencié le comportement d'une des copies de l'entité appelée le maître et celui des autres copies appelées esclaves. Le maître exécute l'intégralité du comportement alors que les esclaves n'exécutent qu'une version dégradée suffisante pour reproduire les déplacements du maître et assurer l'interaction avec les autres entités. Pour compenser les erreurs de calculs entre différents sites exécutant les mêmes entités, nous avons développé des algorithmes de *Dead-Reckoning*.

D'autres expérimentations ont consisté à adapter le modèle développé par A. Samarin sur l'exécution de comportements physiques à notre modèle des *Icobjs*. Cette adaptation a principalement consisté à introduire plus de modularité et de réutilisabilité dans l'implémentation du modèle physique.

Enfin, nous avons réalisé rapidement quelques expérimentations dans le cadre de simula-

tions multi-horloges. Pour cela, nous voulions enregistrer dans une même simulation des icobjs aux comportements physiques et des icobjs aux comportements de jeux. Nous avons réussi à simuler un système multi-horloges en séparant les comportements devant être exécutés à différents rythmes sur différentes machines réactives que le workspace fait réagir à différentes cadences. Nous avons développé un workspace spécifique contenant deux machines réactives : l'une exécute les icobjs construits dans le modèle physique et l'autre exécute les icobjs aux comportements de jeux. Une communication événementielle est maintenue entre les deux machines réactives en générant des événements depuis l'extérieur du workspace.

8.2 Perspectives

Suite à ces travaux, nous envisageons plusieurs axes de recherches concernant les différents développements que nous avons présentés dans ce document.

Moteur réactif

Actuellement, *Junior* utilise les instructions `Freezable` et `Link` pour simuler la notion d'objets réactifs. Mais, comme nous l'avons vu dans ce document, l'utilisation de ces instructions présente plusieurs failles qui peuvent amener à des incohérences dans l'exécution de ces programmes. Un des développements possibles serait d'intégrer la notion d'objet réactif telle que nous la proposons au sein de la machine réactive *Junior*.

Dans *Reflex*, nous avons retiré l'instruction `Freezable` pour les différentes raisons exposées dans la section 2.3.5. Dans le cas de *Junior* ou de *REJO*, cette instruction pourrait être gardée mais en lui faisant subir quelques modifications au niveau de sa structure et de sa sémantique. Il faudrait d'une part, rendre le comportement de `Freezable` plus régulier (de la même manière que pour l'instruction `Kill`) et d'autre part, éviter de placer le résidu des programmes dans l'environnement d'exécution. Une solution serait d'ajouter un handler à l'instruction `Freezable` et de permettre au programme utilisé comme handler d'accéder au résidu du programme gelé. Ainsi, il ne serait plus nécessaire de copier le résidu dans l'environnement en l'attachant à l'événement de gel. Il suffirait de conserver le résidu dans l'instruction `Freezable` et de le rendre accessible pendant l'exécution du handler, en le stockant par exemple dans une variable de l'environnement comme le fait l'instruction `Link`. Cette technique permettrait à la fois de conserver le résidu des programmes gelés dans le comportement d'un objet réactif et d'utiliser une configuration événementielle pour l'instruction `Freezable`.

Enfin, *Junior* ne dispose pas de mécanisme pour récupérer les exceptions générées par les actions atomiques et par les différents types de wrapper. Ainsi, lorsqu'une exception est générée, la machine réactive est stoppée. R. Acosta a proposé l'introduction d'une nouvelle primitive réactive `Try` [2] qui permettrait de traiter ces exceptions. Cependant, cette solution introduit un mécanisme de préemption forte dans *Junior*. Or, la préemption forte introduit de l'indéterminisme dans l'exécution des programmes réactifs, ce que nous ne voulons pas. Pour éviter cet indéterminisme, il serait possible d'introduire un nouveau statut d'exécution qui permettrait à la fois de continuer à exécuter le programme et de gérer l'exception par l'intermédiaire d'un handler. On pourrait aussi envisager des solutions plus radicales comme le retrait de la branche d'exécution ayant générée l'exception (comme dans *oRis*) ou celui de l'objet réactif avec tous les programmes qui lui sont associés.

Les *Icobjs*

Nous nous sommes principalement intéressés à la partie comportementale des *Icobjs* et à l'efficacité de leur exécution. Plusieurs fonctionnalités supplémentaires pourraient être ajoutées à l'API :

- Nous pourrions réutiliser les travaux de R. Acosta concernant *REJO/ROS* pour permettre la migration des *Icobjs* à travers le réseau.
- La partie graphique des *Icobjs* nécessiterait aussi des améliorations. Par exemple, il faudrait mettre en place des interfaces pour permettre l'utilisation d'outil de rendu 3D. La solution la plus simple serait d'utiliser *Java3D* [75] pour rester dans le monde *Java*. Il existe également des API pour manipuler des simulations *VRML-X3D* à partir de *Java*.
- L'inspecteur permet principalement de visualiser le comportement existant et de le modifier légèrement. Pour plus de souplesse et de facilité dans la programmation des *Icobjs*, il serait intéressant d'intégrer un interpréteur comme les *Reactive Scripts*.
- Enfin, il n'y a pas de typage strict sur les champs présents dans la table de hachage d'un icobj. Il serait intéressant de mettre en place un mécanisme de typage de ces champs. L'identifiant d'un champ pourrait par exemple être non plus une chaîne de caractères, mais une structure particulière contenant à la fois une description du champ et le type de ce champ.

Les simulations

Une piste intéressante à suivre serait d'utiliser les *Icobjs* dans le cadre d'enseignements à distance. En effet, la simplicité de construction des interactions physiques et la dynamicité du système concernant l'ajout de nouveaux composants pourraient être un avantage dans la création d'un laboratoire virtuel. Les expérimentations menées autour des simulations distribuées dans le cadre du projet IST-PING constituent une première étape dans l'utilisation de comportements réactifs dans des mondes virtuels distribués. Ainsi, il serait intéressant d'étudier l'intégration du modèle physique proposé par A. Samarin dans une plate-forme comme IST-PING. Il faudrait peut être réaliser des algorithmes spécifiques de *Dead-Reckoning* dans le cadre du modèle physique.

En ce qui concerne les simulations multi-horloges, nous avons fait une distinction entre les icobjs aux comportements physiques et des icobjs aux comportements de jeux. Il serait intéressant de faire cette distinction, non au niveau de l'entité, mais au niveau des comportements. De plus, il serait intéressant de réaliser une seule machine réactive exécutant les différents types de comportements, en associant une horloge aux événements générés dans l'environnement comme cela est fait en SIGNAL.

Bibliographie

[1] R. Acosta-Bermejo. La programmation en rejo. *Les intergiciels, développements récents dans CORBA, JavaRMI et les agents mobiles, Publications Hermes Science, Lavoisier*, 2002.

[2] R. Acosta-Bermejo. *Rejo - Langage d'objets réactifs et d'agents.* PhD thesis, Ecole des Mines de Paris, October 2003.

[3] F.S. Acton. *Numerical Methods That Work (corrected edition).* Mathematical Association of America, 1990.

[4] C. André. Representation and Analysis of Reactive Behaviors : A Synchronous Approach. In *Computational Engineering in Systems Applications (CESA)*, pages 19–29, Lille (F), July 1996. IEEE-SMC.

[5] C. André and M-A Peraldi-Frati. Behavioral Specification of a Circuit Using SyncCharts : a Case Study. In *Euromicro 2000, Digital System Design*, pages 91–98, Maastricht (NL), September 2000. IEEE.

[6] B.J.L. Berry, L. Kiel Douglas, and E. Elliott. Adaptive agents, intelligence, and emergent human organization : Capturing complexity through agent-based modeling. *PNAS*, 99(90003) :7187–7188, 2002.

[7] G. Berry. The Constructive Semantics of Pure Esterel Draft Version 3, 1999.

[8] G. Berry. The Esterel v5 Language Primer. Technical report, INRIA-CMA, 1999.

[9] G. Berry. *The Foundations of* Esterel. MIT Press, 2000. Editors : G. Plotkin, C. Stirling and M. Tofte.

[10] G. Berry and G. Gonthier. The ESTEREL synchronous programming language : design, semantics, implementation. *Sci. Comput. Program.*, 19(2) :87–152, 1992.

[11] F. Boussinot. Reactive C : An Extension of C to Program Reactive Systems. *Software Practice and Experience*, 21(4) :401–428, april 1991.

[12] F. Boussinot. Icobj Programming. Technical Report RR-3028, INRIA-CMA, Octobre 1996.

[13] F. Boussinot. *Objets réactifs en Java.* Collection Technique et Scientifique des Télécommunications, Presses Polytechniques Universitaires Romandes, 2000.

[14] F. Boussinot. Java Fair Threads. *Inria research report, RR-4139*, 2001.

[15] F. Boussinot, G. Doumenc, L. Hazard, Y. Mainguy, J-B. Stefani, and J-F. Susini. Programmation réactive d'applications distribuées. *Proceedings of NOTERE'97, Pau*, September 1997.

[16] F. Boussinot and L. Hazard. Reactive Scripts. *INRIA Research Report 2868*, April 1996.

[17] F. Boussinot and R. De Simone. The SL Synchronous Language. *IEEE Trans. on Software Engineering*, 22(4) :256–266, april 1996.

[18] F. Boussinot and J-F. Susini. The SugarCubes tool box - a reactive Java framework. *Software Practice and Experience*, 28(14) :1531–1550, december 1998.

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

[20] Y. Bres. *Exploration implicite et explicite de l'espace d'états atteignables de circuits logiques* ESTEREL. PhD thesis, Université de Nice-Sophia Antipolis, December 2002.

[21] C. Brunette. Etude de la cohérence dans les jeux en réseau distribués. Master's thesis, Université de Nice-Sophia Antipolis, July 2001.

[22] C. Brunette. A Visual Reactive Framework for Dynamic Behavior Creation. *2nd Workshop on Domain Specific Visual Languages, OOPSLA, Seattle*, 2002.

[23] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE : A DECLARATIVE LANGUAGE FOR REAL-TIME PROGRAMMING. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188. ACM Press, 1987.

[24] P. Caspi and M. Pouzet. A Functional Extension to Lustre. In M. A. Orgun and E. A. Ashcroft, editors, *International Symposium on Languages for Intentional Programming*, Sydney, Australia, May 3-5 1995. World Scientific.

[25] M. Daniels. Integrating Simulation Technologies With Swarm. In *Proceedings of the Workshop on Agent Simulation : Applications, Models, and Tools*, october 1999.

[26] J. Demaria. Programmation réactive fonctionnelle avec Senior. Master's thesis, Université de Nice-Sophia Antipolis, 2001.

[27] Conal Elliott and Paul Hudak. Functional Reactive Animation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, volume 32(8), pages 263–273, 1997.

[28] H. Elmqvist. *Dymola — User's Manual*, 1995. Dynasim AB, Research Park Ideon, Lund, Sweden.

[29] J.-Cl. Fernandez. ALDEBARAN : UN SYSTÈME DE VÉRIFICATION PAR RÉDUCTION DE PROCESSUS COMMUNICANTS. PhD thesis, Univ. Joseph Fourier-Grenoble I, France, July 1984.

[30] R. M. Fujimoto. Parallel and distributed simulation. In *Proceedings of the 31st conference on Winter simulation*, pages 122–131. ACM Press, 1999.

[31] A.J. Goldberg and D. Robson. *Smalltalk-80 : The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, USA, 1984.

[32] Silicon Graphics. The industry's foundation for high performance graphics : http ://www.opengl.org.

[33] P. L. Guernic, T. Gautier, M. L. Borgne, and C. de Marie. Programming real-time applications with signal. *Proceedings of the IEEE*, 79(2) :1321– 1335, september 1991.

[34] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic, 1993.

[35] D. Harel and A. Naamad. The STATEMATE SEMANTICS OF STATECHARTS. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4) :293–333, 1996.

[36] D. Harel and A. Pnueli. On the development of reactive systems. *Logics and Models of Concurrent Systems*, NATO ASI Series(13) :477–498, 1985.

[37] David Harel. Statecharts : A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3) :231–274, 1987.

[38] Fabrice Harrouet. *oRis : s'immerger par le langage dans le prototypage d'univers virtuels à base d'entités autonomes*. PhD thesis, Université de Bretagne Occidentale, December 2000.

[39] L. Hazard, J-F. Susini, and F. Boussinot. The Junior Reactive Kernel. *INRIA Research Report 3732*, July 1999.

[40] L. Hazard, J-F. Susini, and F. Boussinot. Programming with Junior. *Inria Research Report 4027*, 2000.

[41] M. Teresa Higuera-Toledano, V. Issarny, M. Banatre, G. Cabillic, J-P. Lesot, and F. Parai. Java Embedded Real-Time Systems : An Overview of Existing Solutions. In *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 392. IEEE Computer Society, 2000.

[42] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future : the Story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM Press, October 1997.

[43] S. Peyton Jones. *Haskell 98 Languages and Librairies*. Cambridge University Press, 2003.

[44] G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39. Springer-Verlag, 1987.

[45] J. Klein. BREVE : a 3D Environment for the Simulation of Decentralized Systems and Artificial Life. *Proceedings of Artificial Life VIII, the 8th International Conference on the Simulation and Synthesis of Living Systems*, 2002.

[46] S. Luke, G. Catalin Balan, L. Panait, C. Cioffi-Revilla, and S. Paus. MASON : A JAVA MULTI-AGENT SIMULATION LIBRARY. *Proceedings of the Agent 2003 Conference*, 2003.

[47] O. Maler and S. Yovine. Hardware Timing Verification using KRONOS. In *In Proc. 7th 20,300,600 Israeli Conference on Computer Systems and Software Engineering*, 1996.

[48] L. Mandel. Aspects dynamiques dans les langages synchrones : le cas des SugarCubes. Master's thesis, Laboratoire d'Informatique Paris 6, september 2002.

[49] F. Maraninchi. The Argos language : Graphical Representation of Automata and Description of Reactive Systems. In *IEEE Workshop on Visual Languages*, october 1991.

[50] The MathWorks. Simulink : http ://www.mathworks.com/products/simulink/.

[51] Sun Microsystems. Bean builder :
http ://java.sun.com/products/javabeans/beanbuilder/index.jsp.

[52] Sun MicroSystems. The VolatileImage API User Guide : Managing Hardware-accelerated Offscreen Images with VolatileImage.

[53] Sun MicroSystems. JavaBeans.

[54] E. E. L. Mitchell and J. S. Gauthier. *ACSL : Advanced Continuous Simulation Language - User Guide and Reference Manual*, mitchell & gauthier assoc., concord, mass edition, 1986.

[55] B. Nichols, D. Buttlar, and J. Proulx Farrell. *Pthreads Programming.* O'Reilly, 1996.

[56] N. Nikaein. RAMA : Reactive Autonomous Mobile Agents. Master's thesis, Université de Nice-Sophia Antipolis, 1999.

[57] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison Wesley, 1994.

[58] M. Parker. Ascape : an agent based modeling environment in java. In *Proceedings of the Workshop on Agent Simulation : Applications, Models, and Tools*, october 1999.

[59] T. J. Parr and R. W. Quong. ANTLR : a predicated-LL(k) parser generator. *Software Practice And Experience*, 25(7) :789–810, 1995.

[60] O. Parra. Programmation réactive sur systèmes embarqués. Master's thesis, Université de Nice-Sophia Antipolis, 2003.

[61] G.D. Plotkin. A Structural Approach To Operational Semantics. Technical report, FN 19, Department of Computer Science, University of Aarhus, Denmark, 1981.

[62] D. Pous. Les Mobile Ambients en Icobj. Master's thesis, École Normale Supérieure de Lyon, July 2002.

[63] The Real-Time for Java Expert Group. *The Real-Time Specification for Java.* Addison-Wesley, 2000.

[64] P. Reigner, F. Harrouet, S. Morvan, and J. Tisseau. AReVi : A virtual reality multi-agent platform. *Lecture Notes in Computer Science*, 1434 :218– ? ?, 1998.

[65] N. Richard. *Description de comportements d'agents autonomes évoluant dans des mondes virtuels.* PhD thesis, École Nationale Supérieure des Télécommunications, september 2001.

[66] N. Richard. InViWo agents : write once, display everywhere. In *Proceeding of the eighth international conference on 3D Web technology*, pages 123–127. ACM Press, 2003.

[67] E. Sahin, T.H. Labella, V. Trianni, J.-L. Deneubourg, P. Rasse, D. Floreano, L. Gambardella, F. Mondada, S. Nolfi, and M. Dorigo. SWARM-BOTS : Pattern Formation in a Swarm of Self-Assembling Mobile Robots. In A. El Kamel, K. Mellouli, and P. Borne, editors, *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, pages 145–150, Hammamet, Tunisia, october 2002. Piscataway, NJ : IEEE Press.

[68] A. Samarin. Application de la programmation réactive à la modélisation physique. Master's thesis, Université de Nice-Sophia Antipolis, 2002.

[69] J-F. Susini. Implémentation de l'approche réactive en java : les sugarcubes v2. *Proc. MSR'99, Hermes*, 1999.

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

[71] O. Tardieu and R. De Simone. Instantaneous Termination in Pure Esterel. In Radhia Cousot, editor, *Static Analysis, 10th International Symposium, SAS 2003, June 11-13, Proceedings*, volume 2694 of *Lecture Notes in Computer Science*, pages 91–108, San Diego, California, USA, 2003. Springer.

[72] O. Tardieu and R. De Simone. Curing schizophrenia by program rewriting in esterel. In *Second ACM/IEEE International Conference on Formal Methods and Models for Co-Design MEMOCODE 2004*, San Diego, California, USA, June 2004.

[73] A. Tinkham and R. Menezes. Simulating robot collective behavior using StarLogo. In *Proceedings of the 42nd annual Southeast regional conference*, pages 396–401. ACM Press, 2004.

[74] URL. de *Alice* : http ://www.alice.org.

[75] URL. de la communauté Java3D http ://www.j3d.org.

[76] URL. de LOFT : http ://www-sop.inria.fr/mimosa/rp/LOFT/.

[77] URL. de *Mason* : http ://cs.gmu.edu/ eclab/projects/mason/.

[78] URL. de RePast : http ://repast.sourceforge.net/.

[79] URL. de web3d : http ://www.web3d.org.

[80] URL. des ambicobjs http ://www-sop.inria.fr/mimosa/ambicobjs/.

[81] URL. des *Icobjs* : http ://www-sop.inria.fr/mimosa/rp/Icobjs/.

[82] URL. du projet IST-PING : http ://www.pingproject.org.

[83] E. Vecchie and R. De Simone. Syntax-driven behavior partitioning for model-checking of ESTEREL PROGRAMS. In *Synchronous Languages, Applications, and Programming*, Barcelona, Spain, March 2004.

[84] Virtools. Virtools dev :
http ://www.virtools.com/solutions/products/virtools_dev_features.asp.

[85] W. W. Wadge and E. A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., 1985.

Résumé :

Intuitivement, une simulation graphique (monde virtuel, jeu,...) peut être vue comme un espace borné dans lequel plusieurs entités, disposant chacune d'un comportement propre, évoluent en parallèle. Il paraît donc naturel d'utiliser des langages concurrents pour programmer ces comportements. Cependant, c'est rarement le cas pour des raisons de complexité de programmation et de déboguage, de non-déterminisme et d'efficacité.

Nous proposons d'utiliser l'*Approche Réactive* introduite par F. Boussinot qui permet de définir clairement les comportements d'entités graphiques. Nous avons enrichi les *Icobjs* qui est un modèle d'objets réactifs graphiques. Il offre la possibilité à des non-spécialistes de construire graphiquement des comportements complexes à partir de comportements simples et de *constructeurs graphiques*.

Nous avons réalisé une implémentation dont l'objet central est `Icobj`. Il définit une structure minimale dynamiquement extensible pour permettre une construction graphique générique. Un icobj particulier, le `Workspace`, exécute et affiche les icobjs qu'il contient. Les comportements sont exécutés par un moteur réactif dédié aux *Icobjs* appelé *Reflex*. Il reprend les principales primitives du formalisme *Junior* en modifiant la sémantique de certaines instructions et en y ajoutant de nouvelles. Toutes les instructions sont formalisées par des règles de réécritures au format SOS. De plus, nous avons développé un environnement qui permet d'interagir dynamiquement avec les simulations. Enfin, nous présentons quelques expérimentations autour de l'utilisation des *Icobjs* dans le cadre de simulations distribuées, de simulations physiques ou de simulations multi-horloges.

Mot-clefs : approche réactive/synchrone, simulation graphique, programmation graphique, concurrence, *Java*, *Junior*, sémantique formelle.

Abstract :

Intuitively, a graphical simulation (virtual world, game,...) can be seen as a finite space in which several entities, each having its own behavior, evolve in parallel. So it seems natural to use concurrent languages to program those behaviors. However, it is seldom the case due to the complexity of programming and debugging them, because they introduce non-determinism and are inefficient.

We proposed to use the *Reactive Approach* introduced by F. Boussinot to define clearly the behaviors of graphical entities. We enhanced a model of graphical reactive objects called *Icobjs*. This model enables non-specialists to graphically build complex behaviors using elementary behaviors and *graphical constructors*.

We realized an API whose main class is `Icobj`. It defines a minimal structure which can be dynamically extended and which allows a generic graphical construction. A particular icobj called Workspace executes and displays all the icobjs it contains. Behaviors are executed by a reactive engine called *Reflex* which is dedicated to Icobjs. It reuses Junior's formalism's main primitives, while modifying some instructions' semantics and introducing new instructions. We formalized all instructions using rewriting rules in the SOS format. Moreover, we developed a framework for dynamic interaction with simulations. Finally, we present a few experiments with Icobjs in distributed, physical or multi-clock simulations.

Keywords : reactive/synchronous approach, graphical simulation, graphical programmation, concurrency, *Java*, *Junior*, formal semantics.