Moteur r´eactif

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.

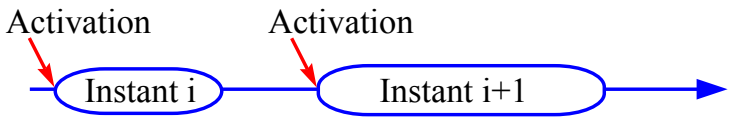

FIG. 2.1 – Notion d'instant

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).

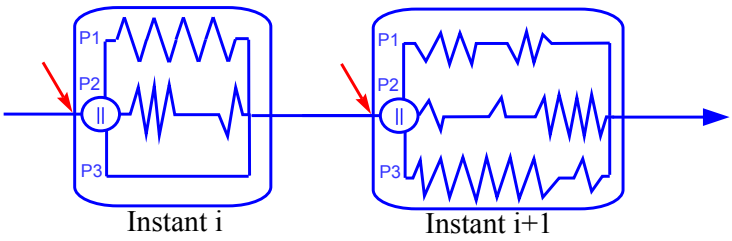

FIG. 2.2 – Concurrence

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

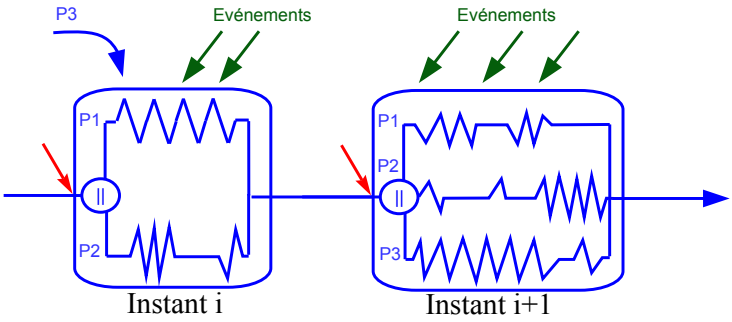

FIG. 2.3 – Dynamisme

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.

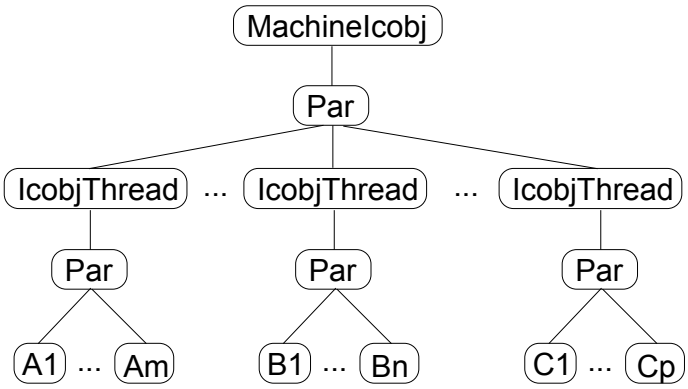

FIG. 2.4 – Structure interne de MachineIcobj

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*.