MOTS CLÉS :

Les cartes à puce, JavaCard, approche synchrone, approche réactive, les SugarCubes,

le langage junior (Rewrite,Replace,Simple, Storm,gloutan)

République Algérienne Démocratique Et Populaire
Ministère de l'enseignement supérieur
et de la recherche scientifique

UNIVERSITE MOULOUD MAMMERI DE TIZI-OUZOU
FACULTE DE GENIE ELECTRIQUE ET D'INFORMATIQUE
DEPARTEMENT D'INFORMATIQUE

Projet de fin d'étude

En vue de l'obtention du diplôme de Master en informatique

Spécialité : Conduite de projets informatiques

Thème :

Implémentation de l'approche réactive sur une plateforme JAVACARD

Proposé par :

Mr. M.DEMRI

Réalisé par :

Mr Bouadma Farid

2010/2011

remerciement

TOUS MES VIFS REMERCIEMENTS ET MA PROFONDE RECONNAISSANCE S'ADRESSENT À MON ENCADREUR MR M.DEMRI. POUR M'AVOIR CONFIÉ CE SUJET, ET POUR SON SOUCI, SA DISPONIBILITÉ, SES CONSEILS, SON AIDE ...
QU'IL N'A PAS CESSÉ DE ME PRODIGUER DURANT MON TRAVAIL.

MES SINCÈRES REMERCIEMENTS ENCORE SONT ADRESSÉS À TOUS CEUX QUI, SANS AVOIR ÉTÉ IMPLIQUÉS DIRECTEMENT DANS CE TRAVAIL, ONT TOUJOURS ÉTÉ D'UN GRAND SUPPORT : MA FAMILLE, MES ENSEIGNANTS, MES AMIS, MES COLLÈGUES CHACUN SON NOM.

Dédicaces

Je dédie ce modeste travail à :

- *Mon défunt frère, que dieu l'accueille en son vaste paradis.*
- *A ma chère mère et mon père, qui ont été toujours à coté de moi avec leur attention et leurs conseils.*
- *A mes frères et sœurs et leurs familles.*
- *A toutes ma famille.*
- *A tous les gens que je connaisse, de prés ou de loin.*
- *A tous mes ami : Nacer, Brahim, Nourdine,Said,Samir,Athman,Sidia,Amirouche, Ahlem,Smail,Hassan….*
- *A tous les étudiants de la section MastrII LMD.*
- *Je dédie également ce modeste travail à La mémoire de notre chère camarade G.Assia et à ça famille.*

SOMMAIRE

TABLE DE FIGURES :

TABLE DE TABLEAUX :

Introduction Générale

Introduction Générale :

Aujourd'hui les cartes à puce font partie de notre vie courante. Les plus courantes sont les cartes bancaires. Toutefois le développement d'applications pour cartes à puces à toujours été difficile ,en effet au début la puce ne contient que des données relatives à l'utilisateur (comme son code) et des protocoles de communications avec les lecteurs dont la majeure composante est la sécurité (vérification du code).

C'est en 1996 qu'un groupe d'ingénieurs Texan de Schlumberger ont voulu simplifier la programmation des cartes à puces tout en préservant la sécurité des données. C'est finalement le langage Java qui a été retenu. C'est cependant un sous-ensemble de Java qui a été utilisé à cause de la faible quantité de mémoire disponible.

Avec *Java Card*, il devient possible d'installer sur la carte des petits programmes sous forme d'*applets* Java écrites dans une version allégée du langage et qu'on peut supprimer à tout moment .

Ainsi, la carte peut dorénavant contenir, en plus des données personnelles de l'utilisateur, des programmes appartenant à plusieurs entités différentes. Ces entités peuvent être des organismes commerciaux ou publics, des sociétés diverses en partenariat ou en concurrence. Nous pouvons donc nous attendre à une réduction sensible du nombre de cartes dans notre quotidien, une seule carte pouvant faire office à la fois de carte bancaire, de carte de parking et de badge d'accès.

Malgré tout ces avantages qu'offre la plateforme JAVACARD, elle reste limitée par rapport à une plateforme Java standard, notamment en ce qui concerne l'implémentation des threads et la programmation concurrente et biens d'autres fonctionnalités.

Plusieurs difficultés ont été relevées dans les threads java, non seulement au niveau de la programmation qui nécessite beaucoup de compréhension du mécanisme pour se lancer dans un développement multithreads, mais aussi dans la sémantique des primitives qui n'est pas claire. Ce qui a conduit plusieurs chercheurs à développer d'autres solutions, ainsi l'approche réactive est née.

Cette approche dérive de l'*Approche Synchrone* et regroupe un ensemble de formalismes permettant d'implémenter des systèmes réactifs. Elle a pour caractéristique de pouvoir évoluer dynamiquement, en cours d'exécution, en modifiant le nombre de composants parallèles.

Ses principales caractéristiques sont:

- Fournir un ensemble de primitives permettant l'expression simple et puissante du parallélisme au niveau d'un langage de programmation.
- Permettre l'expression de systèmes dynamiquement reconfigurables en cours d'évolution. Ceci est en effet plus adapté dans le cadre de langages de programmation généralistes, en particulier lorsqu'on s'intéresse à des *systèmes distribués*;
- Fournir un modèle *déterministe* de programmation du parallélisme, formellement spécifié, dont les résultats soient reproductibles indépendamment des implémentations et des plates-formes d'utilisation;
- Fournir des mécanismes simples et puissants de communication et de synchronisation entre les différents composants parallèles d'un système;
- Exhiber le caractère modulaire de la programmation parallèle en insistant sur la réutilisabilité des composants parallèles développés, en particulier, par combinaison parallèle de ces composants;
- Fournir un modèle de programmation du parallélisme cohérent avec un modèle de programmation à base d'objets.

Mon travail consiste en l'implémentation de l'approche réactive au dessous de la plateforme JavaCard. Pour cela, je l'ai structuré comme suit :

Premièrement on va donner un petit aperçu sur les cartes à puces, leur évolution, leurs composants, leurs types,…etc. Ensuite on va présenter la technologie JavaCard, où on décrira particulièrement le langage JavaCard. Et puis on parlera de l'approche réactive, ses différents formalismes, précisément les objets réactifs et leurs implémentations.

Enfin, on choisira la meilleure implémentation qu'on va simplifier et l'exécuter sous la plateforme JavaCard.

Les cartes à puce et la technologie JavaCard

I. Les cartes à puce :

I.1. Introduction :

La carte à puce est couramment utilisée pour désigner des supports de sécurité qui ont les mêmes dimensions qu'une carte de crédit, elles sont en matière plastique et, contiennent un circuit électronique intégré capable de mémoriser ou de traiter les informations.

Cette carte a fait ses preuves qu'elle est appelée à couvrir pratiquement tous les secteurs de l'activité humaine en tant que moyen de paiement, clé d'accès ou dossier portable. Aujourd'hui, la télécarte est entrée dans nos habitudes pour le prépaiement des taxes téléphoniques et la carte bancaire à microcalculateur s'est généralisée en France tandis que des développements similaires se réalisent en Europe, au Japon et aux États-Unis, elle est même introduite en Algérie : carte bancaire, carte **Chifa**.

La carte à puce peut donc être vue comme un support électronique permettant d'intégrer des applications à haut niveau d'intégrité et de confidentialité dans des systèmes d'information. Pour ce faire, elle offre un espace sécurisé pour des applications "sensibles".

La carte à puce à été introduite en 1969 et depuis, elle a connu une énorme évolution. Le tableau suivant résume l'évolution des cartes à puce.

Année	Evénement
1969	Jürgen Dethloff et Helmut Grotrupp ingénieurs allemands déposent un brevet
1970	Kunitaka Arimura ingénieur japonais dépose un brevet en mars 1970 au japon
1971	Paul castrucci d'IBM dépose aux USA un brevet intitulé ***information card***
1974	Création de la première carte à puce à circuit intégré par Roland Moreno, dirigeant de la société ***Innovation.***
1979	Création de la première carte à microprocesseur (Bull CP8) avec 1KO de mémoire programmable et un cœur à base de microprocesseur 6805.
1980	Le Groupement Carte à Mémoire est crée par un regroupement de banques françaises afin d'utiliser la carte à puce comme un nouveau moyen de paiement : ce qui deviendra la future

	carte bancaire. Bull, Schlumberger et Philips se lancent alors dans la conception de ces cartes, et c'est réellement en 1984 que la commercialisation des premières cartes à mémoire bancaire a eu lieu.
1983	la carte à puce aida le secteur sanitaire et social. Il s'agit de doter les patients d'une **carte santé** qui renseignerait de manière plus précise le docteur à propos du malade.
1984	On note l'apparition de la **télécarte** avec la "carte pyjama" (appelée ainsi en raison de ses rayures blanches et bleues) crée par Schlumberger pour France Télécom (cartes équipées de micromodules).
1984-1987	Les normes internationales de l'ISO sur la carte à puce à contact voient le jour sous la référence 7816.
1997	Les premières **JavaCard**s sont apparues.
Aujourd'hui	Près de 7 milliards de cartes en circulation .La technologie de la carte à puce fait partie de notre vie quotidienne grâce à **des applications très courantes** comme les cartes SIM des téléphones portables, le e-commerce, la carte d'identité électronique, secteur médical, le passeport biométrique….etc.

Tableau I.1 Evénements importants dans l'histoire de la carte à puce.

I.2. La description physique d'une carte à puce :

- **La puce électronique :**

C'est un micro-circuit (circuit intégré) construit à partir d'une galette de silicium.

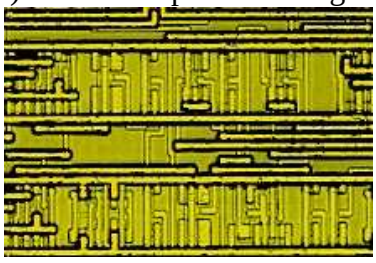


Figure I.1.Détail d'un circuit integré

- **Le micromodule :**

Très mince circuit imprimé logé dans l'épaisseur de la carte qui accueille les contacts (visibles) du connecteur sur une face et la puce (cachée sous les contacts du micromodule) sur l'autre.

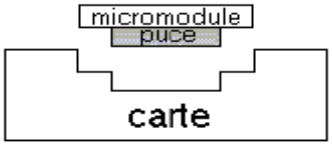


Figure I.2. Le micromodule

- **Les points de contacts :**

On décrit ici les fonctions des huit points de contacts pour la communication entre la carte à puce et le lecteur de cartes.

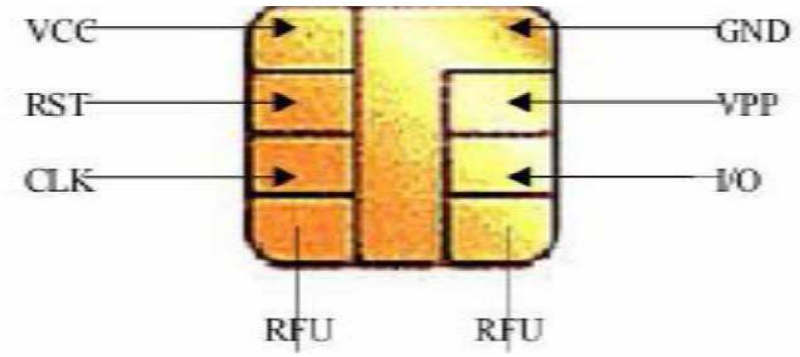


Figure I.3. Les points de contacts.

- **VCC:** Ce point alimente la puce en tension électrique (3 à 5 volts)**.**
- **RST :** c'est le "RESET", il permet l'initialisation du microprocesseur, on parle aussi de "warm reset", il existe aussi le "cold reset" lors de la coupure et de rétablissement de l'alimentation pour la carte.
- **CLK :** il fournit un signal d'horloge pour la carte qui ne dispose pas d'une horloge interne.
- **GND :** il correspond à la masse, c'est le référentiel de la tension.
- **VPP :** c'est la tension de programmation de la carte fournie par le lecteur, utilisé dans les anciens modèles pour avoir une autre source d'alimentation.

- **I/O :** il est utilisé pour le transfert des données et des commandes entre le lecteur et la carte, la communication est half duplex.
- **RFU :** les deux points RFU sont réservés pour des utilisations futures.

➢ **La carte plastique :**

Deux principaux types de plastique sont utilisés

- *Le PVC non recyclable mais embossable*
- *L'ABS non embossable mais recyclable*

Même si on connaît en général deux formats de la carte à puce :

Celui de la carte bancaire et celui de la carte SIM, ils en existent trois formats normalisés : ID1, ID00 et ID000 :

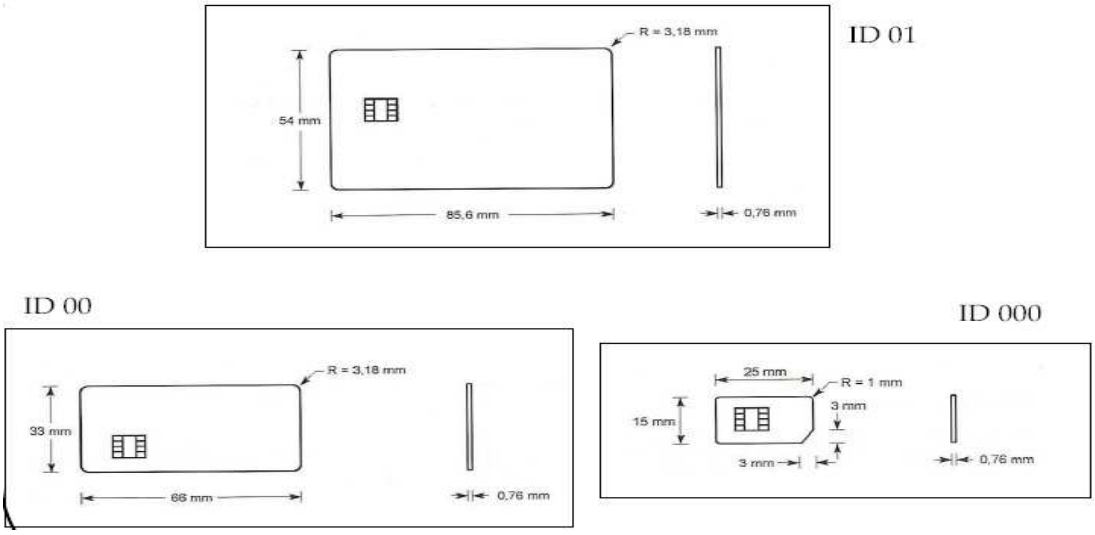


Figure I.4. Les trois formats de cartes à puce.

I.3. Les types de cartes à puce:

Ils existent plusieurs types de cartes à puces, que l'on peut classer selon deux critères ; en effet, on peut réaliser un classement selon l'architecture matérielle ou bien suivant l'interface de communication entre le lecteur et la carte à puce, comme illustré dans le schéma suivant :

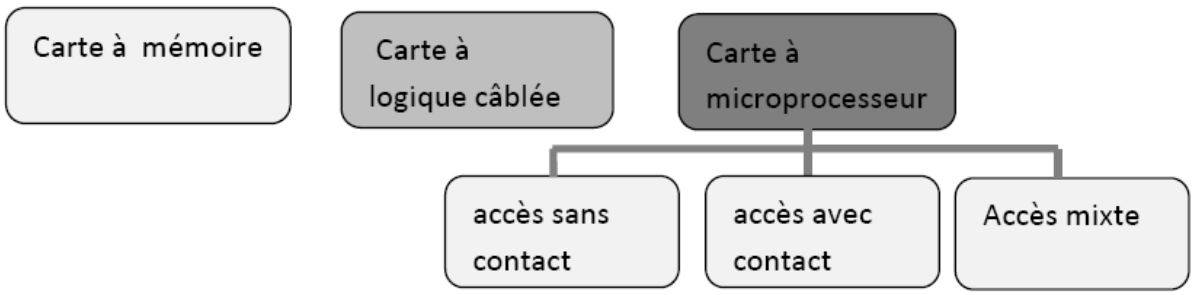


Figure I.5. Les différents types de cartes à puce. [tsa7]

I.3.1. Classement suivant l'architecture matérielle :

On trouve trois types :

I.3.1.1. Les cartes à mémoire ou cartes synchrones :

Les cartes à mémoire sont les premières, appelées aussi cartes synchrones en raison de leur protocole de dialogue. Possédant une mémoire simple (sans processeur) accessible en lecture sans protection, mais l'écriture peut être rendue impossible ; cette famille de carte est destinée à des applications relativement simples. Ce type de carte ne permet pas la mise en œuvre d'applications plus complexes, ce qui explique le recours indispensable aux cartes à microcontrôleur. Par exemple, une carte «porte-jetons» pour applications de prépaiement.

I.3.1.2. La carte à logique câblée :

Sa mémoire est accessible via des circuits préprogrammés et figés pour une application particulière. Par exemple, pour une carte "sécuritaire" pouvant effectuer des calculs figés (accès à un local, à un garage).

I.3.1.3. Les cartes à microcontrôleur ou cartes asynchrones :

Ces cartes «programmables» pouvant effectuer tout type de traitements,les cartes à microprocesseur (ou microcontrôleur) incorpore l'intelligence qui faisait défaut aux cartes à mémoire, et méritent ainsi le nom de ***smart cards****(en anglais),* par opposition aux ***memory cards*** précédentes. Les cartes à puce "intelligentes" renferment un microcontrôleur complet ; c'est à dire l'association en un seul circuit d'une unité centrale à base de microprocesseur, de mémoire morte, de mémoire vive, de mémoire EEPROM et d'une interface d'entrée/sortie.

Les cartes à puce sont de plus en plus répandues :

- cartes bancaires ;
- cartes SIM pour téléphones mobiles ;
- cartes pour les décodeurs satellite et câble ;
- cartes de santé ;
- cartes d'identité et informations biométriques ;
- cartes d'identification (signature et déchifrement) électronique ;
- cartes d'accès (avec ou sans contact).

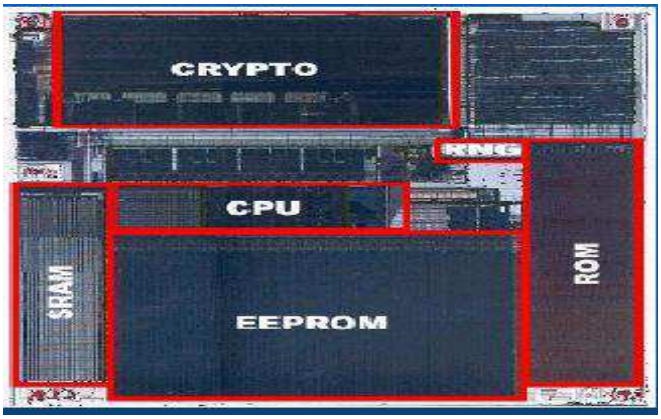


Figure I.6. Carte à puce à microcontrôleur [Sau 02J].

A. La RAM (Random Access Memory):

La RAM est une mémoire volatile et rapide, sa capacité est entre 256 octets et 2 kilos octets. Elle est utilisée comme espace de stockage temporaire pour des données fréquemment mises à jour (comme la pile d'appels), ou pour des données temporaires nécessitant un fort degré de confidentialité (comme des clés cryptographiques de chiffrement de session).

B. La ROM (Read Only Memory):

La ROM est une mémoire persistante, non modifiable, qui sert à stocker le COS (Card Operating System) et les données permanentes, elle est de capacité de 8 à 64 kilos octets.

C. L'EEPROM (Electrical Erasable Programmable Read Only Memory):

Aujourd'hui, sa capacité est entre 8 et 32 kilos octets, L'EEPROM est également une mémoire persistante qui, contrairement à la ROM, peut être modifiée par une application.

D. Le processeur :

La plupart des cartes à puce sont dotées d'un processeur 8 bits. C'est habituellement un simple 8051. La puissance d'un tel processeur est suffisante pour les systèmes d'exploitation natifs. Cependant, si un système d'exploitation doit inclure un interpréteur (ex : **JavaCard**), un processeur 16 bits sera mieux adapté. Il existe des microcontrôleurs qui sont basés sur des processeurs 32 bits tels que le ARM7 et MIPS [**Zhi 02**].

I.3.2. Classement suivant la technologie utilisée pour communiquer avec le monde extérieur :

I.3.2.1. Carte avec contact :

Les cartes à contact pour pouvoir fonctionner doivent être insérées dans un lecteur de cartes. Elles utilisent une communication série via huit contacts.

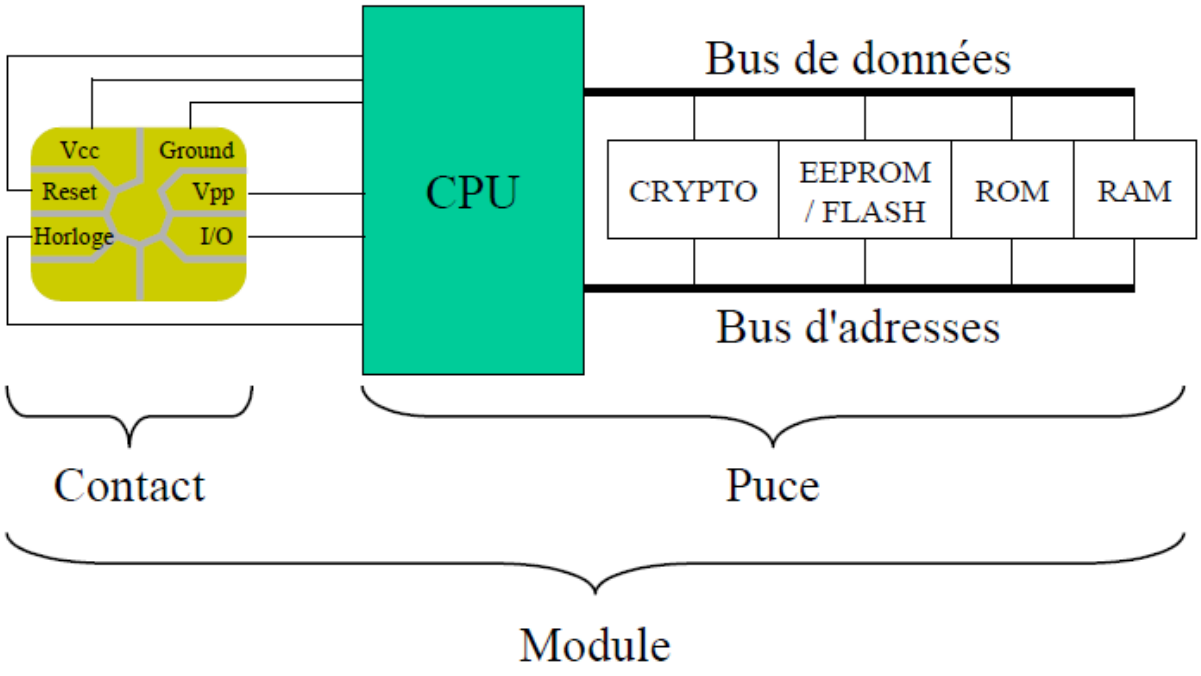


Figure I.7. Architecture d'une carte à puce avec contact [1]

I.3.2.2. Carte sans contact :

Après l'introduction des technologies de communication radiofréquence, une nouvelle catégorie de cartes à puce a été créée, Les cartes à puce sans contact ont fait partie des produits RFID (Radio Frequency Identification Devices). La carte et le lecteur sont reliés par un champ électromagnétique haute fréquence, qui véhicule les données et l'alimentation de la carte à puce. Une antenne bobinée dans la carte reçoit l'énergie radiofréquence transmise par le lecteur via la porteuse. La carte possède après l'antenne un circuit d'accord et de redressement qui permet une alimentation continue à partir de l'alternatif, pour piloter les quelques échanges de données.

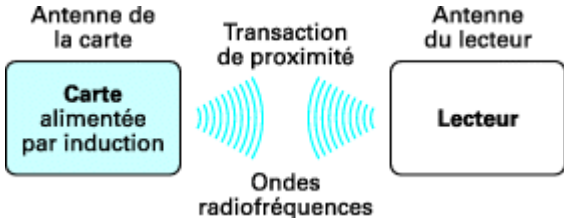


Figure I.8. Interface de communication entre une carte sans contact et un lecteur[2]

I.4. Standards et spécifications des cartes à puces:

L'industrie de la carte à puce a longtemps été ralentie par l'incompatibilité entre les applications, des cartes et des lecteurs en dehors de l'Europe. Pour que le domaine d'application de la carte à puce évolue, les différents constructeurs doivent absolument garantir l'interopérabilité de leurs produits. En effet, un émetteur peut être réticent à investir dans une technologie potentiellement incompatible avec les générations futures. C'est pour cela que de nombreuses normes existent dans les différents domaines d'applications.

I.4.1. Le standard ISO 7816 :

Dans le but de promouvoir l'interopérabilité entre les cartes à puce et les lecteurs, l'International Standards Organization (ISO) a élaboré les normes ISO 7816 pour les cartes à circuit intégré à contacts (Identification Card-Integrated circuit cards with contact).

C'est la norme de référence pour les cartes à puce. Elle spécifie entre autres les éléments physiques et dimensionnels des supports plastiques, la signification et la localisation des

contacts, les protocoles et contenus des messages de bas et haut niveaux échangés avec les cartes ;

Le standard ISO 7816 définie 15 normes.

D'autres standards sont le fruit de la norme ISO dédiés aux nouvelles technologies. En 1996, Europay, MasterCard et VISA (EMV) ont défini une spécification de carte à puce, qui a repris les normes ISO 7816 et défini d'autres types de données et règles de codage pour le domaine de la finance(ex : carte bancaire). Elle définit aussi un ensemble de mécanismes internes de protection des données assurant l'intégrité des secrets stockés dans la carte.

L'industrie des télécommunications européennes a également reconnu les normes ISO 7816 dans sa spécification de carte à puce de communications GSM (Global System for Mobile) pour activer l'identification et l'authentification des utilisateurs de téléphones mobiles.

I.4.2. GlobalPlatform :

GlobalPlatform est une organisation internationale qui rassemble des industriels dans les domaines de la banque, du transport, des télécommunications, de la carte à puce, de l'informatique ou de la défense. La mission de GlobalPlatform est d'établir, de maintenir et de promouvoir l'adoption d'une infrastructure ouverte et interopérable pour les cartes à puce, les terminaux et les systèmes associés ; elle consiste en des spécifications des cartes ouvertes (multi-application) et des lecteurs de sorte à être compatible avec l'ISO et EMV. Les spécifications des cartes définissent la communication avec le lecteur et la gestion d'application de la carte. GlobalPlatform définit un environnement intégré qui concerne principalement le chargement et/ou désinstallation des applications sur la carte.

I.4.3. PC\SC (Personal Computer Smart Card):

Les spécifications du PC/SC (Interoperability Specification for ICCs and Personal Computer System) sont définies en 1997 par un groupe industriel contenant entre autre Bull, Sun, Microsoft, Siemens, etc. Le PC/SC définie l'objectif général d'une architecture utilisant une carte à puce dans un système à ordinateur personnel. Il offre une API (C/C++) standard de haut niveau pour le développement d'application à base de carte à puce sous Microsoft Windows. Cette API permet aux développeurs de se concentrer sur la programmation de la carte en elle-même sans se soucier des différents pilotes de chaque lecteur. Le PC/SC est disponible sous Windows sous forme d'une bibliothèque ; wincard.dll. Grâce à l'open-source PCSC Lite, le PC/SC est aussi implémenté sous UNIX. Pour utiliser une carte à puce, il suffit

d'appeler les fonctions du PC/SC (ex : SCardConnect qui permet de se connecter à la carte à puce).

I.4.4. Open Card Framework:

Open Card Framework(OCF) a été introduit en 1998 par **IBM**, mais actuellement développé par Open Card consortium. C'est un Framework qui fournit une interface standard pour interagir avec les lecteurs de cartes et les applications embarquées dans une carte à puce. L'OCF est implémenté dans un langage de programmation orienté objet (Java).C'est un complémentaire de **JavaCard** pour développer des applications clientes.

Les objectifs d'OCF sont :

- Rendre transparentes les parties spécifiques aux lecteurs de cartes, aux systèmes d'exploitation et les constructeurs de cartes à puces.
- Permettre un développement facile d'application en offrant une API de haut niveau.
- Permettre la portabilité des applications sur différentes plateformes.

I.5. Cycle de vie d'une carte à puce [Ran 03] :

Le cycle de vie d'une carte à puce est relatif à la technologie de production et de la plateforme de développement des applications. Cependant, l'ISO/IEC a défini un cycle de vie standard dont les principales étapes sont les suivantes :

- **Fabrication :** Charger le système d'exploitation (masque) dans la ROM.
- **initialisation :** Charger dans l'EEPROM les données propres à l'application.
- **personnalisation :** Charger dans l'EEPROM les données personnelles relatives à chaque porteur (nom,prenom,..) .
- **utilisation :** Envoie de commandes vers la carte et traitement par le masque. Des modifications peuvent être apportées à la mémoire de la carte.
- **Mort :** Ceci peut être du à un disfonctionnement du système d'exploitation, saturation de la mémoire, perte, vol,…, etc.

I.6. Les communications Carte/Terminal :

Le protocole ISO 7816-4 définit le format des données (**APDU** : Application Protocol Data Units) que doivent suivre les programmeurs d'applications pour que les opérations soient possibles sur n'importe quel lecteur de carte.

La communication **carte/terminal** est une séquence d'échange de commandes APDUs du terminal vers la carte, ou de réponses APDUs de la carte vers le terminal (voir Figure I.9). Les cartes n'initient jamais des communications mais se contentent de répondre aux commandes du monde extérieur.

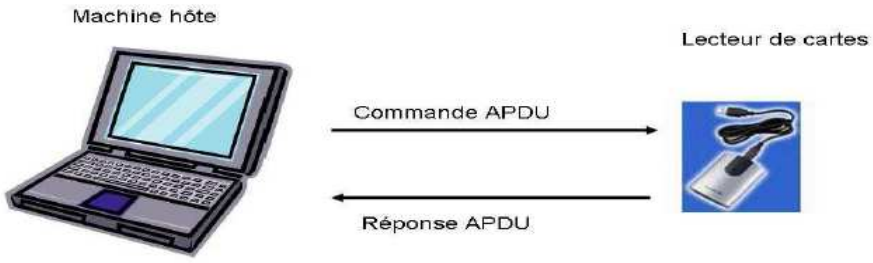


Figure I.9. Modèle de communication carte/ terminal [1]

La figure suivante montre le format des APDUs échangées :

I.6.1. Format d'une APDU de commande :

Commande APDU						
Entête obligatoire				Corps optionnel		
CLA	INS	P1	P2	Lc	Data field	Le

Figure I.10. Format d'une APDU de commande [1]

CLA: un octet qui indique la classe de l'instruction.

INS : un octet qui contient le code instruction.

P1 etP2 : un octet chacun, ils correspondent aux paramètres de l'instruction.

Lc : sa taille varie de 1 à 3 octets. Il indique la longueur des données envoyées vers la carte.

Data : sa taille est spécifiée par **Lc**. C'est une séquence de données additionnelles

Le : sa taille varie de 0 à 3 octets. Il indique la longueur des données attendues en réponse.

I.6.2. Format d'une APDU de réponse :

Réponse APDU		
Corps optionnel	Partie obligatoire	
Data field	SW1	SW2

Figure I.11.Format d'une APDU de réponse [1]

Data field (longueur variable) : une séquence d'octets reçue dans le champ données de la réponse.

Sw1 et **Sw2** : Statuts Words (mot d'état) indiquent l'état du traitement de la commande par la carte (un octet chacun).

I.7. Interface de communication Carte/Lecteur :

Les cartes à puce n'ont pas une source d'alimentation interne pour leur fonctionnement. Donc elles ne sont actives que lorsqu'elles sont connectées à un lecteur qui leur fournit l'énergie nécessaire. Dés qu'une carte est insérée dans un lecteur, elle est mise sous tension puis envoie un message de réponse d'initialisation appelé **ATR** (**A**nswer **T**o **R**eset). Ce dernier contient les paramètres nécessaires pour établir une communication (ex : le protocole de transmission). Le lecteur négocie avec la carte sur les paramètres électriques tel que : le voltage, la fréquence d'horloge et la vitesse de communication. Une fois la connexion établie, la carte revient à l'état inactif jusqu'à réception d'une commande **APDU**. A la réception d'une commande, la carte la traite puis envoie une réponse. La figure ci-dessous résume les messages envoyés entre le lecteur et la carte à puce.

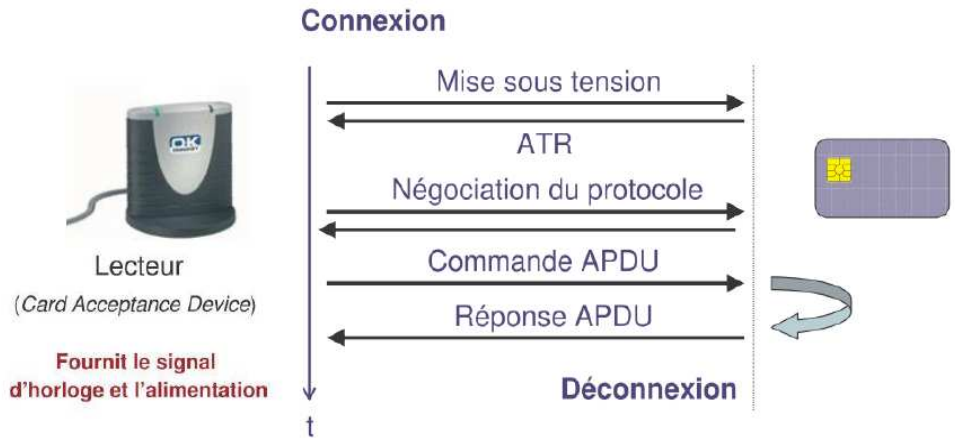


Figure I.12. Communication entre un lecteur et une carte à puce [1].

II. La technologie JavaCard :

II.1. Introduction :

Le développement des technologies de l'information et de communication a augmenté le besoin en applications et leur sécurité. Aujourd'hui, le moyen le plus sûr pour assurer un niveau de sécurité satisfaisant reste la carte à puce. Cependant, le développement d'applications pour carte à puce a toujours été difficile et réservé aux experts. Il a donc fallu développer un langage qui soit à la fois fiable, robuste, peu gourmand en ressources et simple. C'est en 1996 que Sun Microsystems a proposé une solution qui est la **JavaCard**.

La **JavaCard** est une carte à puce possédant une plateforme d'exécution basée sur la technologie Java, en effet elle exécute des programmes java appelés applets, mais à cause de sa la limitation mémoire et de son contexte applicatif (des petites applications de types serveur), seulement un sous ensemble de java à été utilisé pour le développement d'applets.

II.2. Historique :

C'est en 1996 qu'un groupe d'ingénieurs **Texan** de **Schlumberger** ont voulu simplifier la programmation des cartes à puces tout en préservant la sécurité des données. C'est finalement le langage Java qui a été retenu. C'est cependant un sous-ensemble de Java qui a été utilisé à cause de la faible quantité de mémoire disponible.

Ainsi fut créé Java Card, le premier langage orienté objet pour carte à puce.

Schlumberger (devenu Axalto) et Gemplus (tous deux réunis plus tard dans Gemalto) sont les cofondateurs du **JavaCard** Forum (http://www.javacardforum.org) qui regroupe les fabricants de silicium, les encarteurs et les clients, et qui recommande des spécifications à Sun Microsystems en vue d'obtenir une standardisation du langage.

II.3.Présentation de langage JavaCard :

C'est un sous-ensemble de Java adapté aux cartes à puce. Sa syntaxe est légèrement moins riche à cause de la faible capacité mémoire d'une puce.

Le langage Java utilisé pour la programmation des applets **JavaCard** n'est pas le langage Java standard, mais constitue un sous ensemble strict et cohérent de celui ci. Ceci veut dire que l'ensemble des mots-clés et fonctionnalités du langage Java pour l'environnement

JavaCard existent dans **J2SE**, mais pas l'inverse. Le bytecode Java d'une application **JavaCard** pourrait donc en théorie être exécuté dans un environnement J2SE, pourvu que les classes de base de l'environnement **JavaCard** soient portées sur cet environnement.

Les applications **JavaCard** sont appelées des applets, identifiées uniquement par leur AID (*Application IDentifier*) isolées entre elles par des applets firewall.

Elles possèdent 5 méthodes : la méthode *install()* qui permet de créer l'applet, la méthode *register()* qui permet d'enregistrer l'applet au JCRE (Java Card Runtime Environment), les méthodes *select()* et *deselect()* qui permettent la sélection ou la desélection d'une applet et la méthode *process()* qui traite les APDUs de commande. Plusieurs applets pourront cohabiter sur une carte même si elles appartiennent à des fabricants différents. Les applets d'une carte sont gérées par le JCRE. Les minima requis pour faire tourner un programme **JavaCard** sont 16kø de ROM, 8kø d'EEPROM et 256ø de RAM.

JavaCard, bien qu'il soit un sous-ensemble de Java, possède des mécanismes spéciaux liés aux spécificités des cartes à puce. Ainsi il a 2 types d'objets : les objets **transients** (i.e. stockés en RAM qui perdront leurs valeurs lors du retrait de la carte, d'une coupure de courant, de la désélection de l'applet ou après chaque transaction) et les objets **persistants** stockés en EEPROM. **JavaCard** permet de garantir l'atomicité des transactions. Il permet aussi de partager des objets entre applets sous certaines conditions. Les lignes de sécurité sont implémentées par la machine virtuelle.

Le gros avantage de **JavaCard** est la possibilité de charger dynamiquement, n'importe quand, une nouvelle applet sur la carte. Cela signifie que le code des applets pourra être remis à jour.

II.3.1.Les caractéristiques Java supportés :

- Type simple de donnée de petite taille : booléen, byte, short.
- Tableaux à une dimension.
- Paquetages Java, classes, interfaces et Exceptions.
- Caractéristiques orientées objets : héritage, méthodes virtuelles, surcharge et création dynamique d'objets, contrôle d'accès.
- La création dynamique d'objets avec l'opérateur new.
- Le RMC (*Remote Methode Controler*) est supporté depuis la version 2.2.

II.3.2. Les caractéristiques Java non supportés :

- Type simple de donnée de grosse taille : long, double, float .
- Tableaux à plusieurs dimensions.
- Chaine de caractères.
- Chargement dynamique des classes.
- Security Manager.
- Ramasse-miettes et finalisation.
- Les Threads.
- La Sérialisation d'objet.

II.4. Architecture globale d'une JavaCard :

Les **JavaCard** sont des cartes à puce du type *Smart Card*. Ce sont des cartes à microprocesseur qui peuvent gérer des données sur 8, 16 ou 32 bits. Ce type de gestion est dû au fait que la carte fonctionne sur un principe de décodage d'octet. Voici les spécifications qui sont aujourd'hui les plus utilisées pour ce type de carte :

CPU :
Microcontrôleur 8 à 32 bits :

Mémoire :
ROM : 256 – 512 ko (48ko au minimum)
RAM : 512 ko (2ko au minimum)
EEPROM ou FLASH : 128 – 256 ko (32ko au minimum)

L'architecture globale de la **JavaCard** se présente sous cette forme :

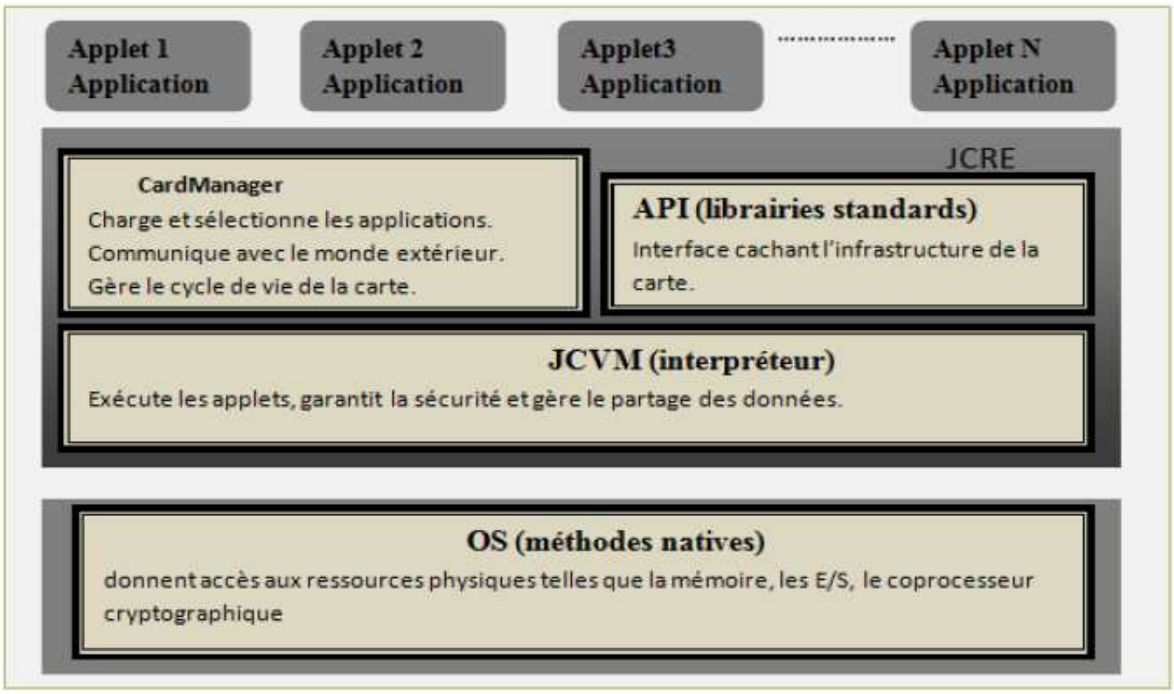


Figure II.1 Architecture globale d'une JavaCard [Tsa 07].

II.4.1.Le système d'exploitation (OS : Operating system) dans la JavaCard :

L'OS d'une **JavaCard** est un système mono-tâche très simple contrôlant les ressources disponibles de la carte (RAM, EEPROM, I/O, et coprocesseur). Il dispose en particulier d'un sous-système d'E/S compatible avec le standard ISO 7816-3 et ISO 7816-4(voir annexe) définissant les méthodes de communication de la carte avec le monde extérieur, il est généralement écrit en C ou en langage d'assemblage et il est stocké en ROM, d'où il est impossible de le modifier durant toute la vie de la carte .

II.4.2. L'environnement d'exécution JavaCard (JCRE : Java Card Runtime Environnement) :

Le JCRE gère les ressources de la carte, les communications, l'exécution des applets et leur sécurité. Il comprend les éléments suivants [**Tsa 07**] :

- La machine virtuelle **JavaCard** (JCVM).
- L'interface de programmations **d'applications JavaCard** (JCAPI).
- Des services incluant le firewall et le partage.

Le JCRE inclut également des services de supports exclusifs aux cartes à puce. En particulier, il inclut un moyen de partager des objets entre les applets et un moyen de gérer des

Transactions (séquences de mise à jour d'objets effectuées dynamiquement). Un autre service très important concernant la répartition des APDUs entre les applets et la sélection de ces dernières.

II.4.2.1. La Machine Virtuelle JavaCard (JCVM) :

A cause de la capacité mémoire limitée de la carte, notamment en mémoire volatile, un découpage spécifique a été défini et qui permet de gagner de l'espace en mémoire et de la consacrer uniquement à l'exécution des applications.

La machine virtuelle **JavaCard** fournit pratiquement les mêmes fonctionnalités que la machine virtuelle Java (**JVM**). La **JCVM** (**JavaCard** Virtual Machine) est composée de deux parties : le convertisseur, situé sur l'ordinateur (**off-card**), et l'interpréteur de byte code, situé dans la **JavaCard** (**on-card**).

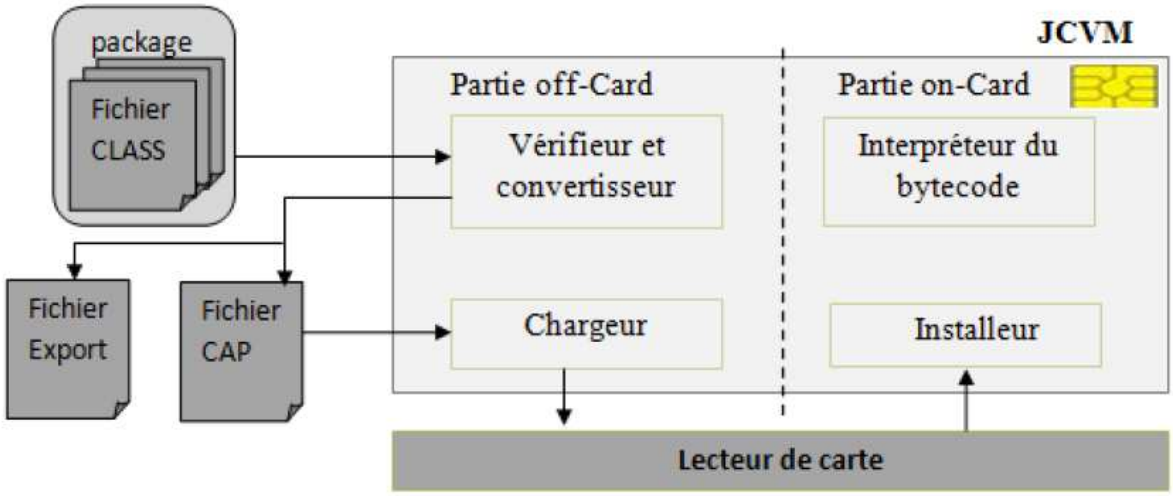


Figure II.2.La machine virtuelle JavaCard [**Tsa 07**]

- **Le vérifieur et le convertisseur de bytecode** :

A partir du code source java, des fichiers *.class* sont produits par le compilateur java. Vu qu'ils ne sont pas exploitables en tant que tels par la machine virtuelle **JavaCard**, le convertisseur prétraite tous les fichiers *.class* contenu dans un package et les convertit en un fichier CAP (**Converted Applet**) contenant une représentation binaire exécutable de classes. Le fichier CAP est ensuite chargé sur une **JavaCard** et exécuté par l'interpréteur. Durant le

processus de conversion, le convertisseur exécute des tâches qui sont normalement exécutées par la JVM au moment du chargement des classes :

- Il vérifie que les images des classes java sont bien formées.
- Il vérifie qu'il n'y a pas de violations du sous-langage java.
- Il initialise les variables statiques.
- Il convertit des références symboliques de classes, des méthodes et des champs en une forme plus compacte supportée par les **JavaCard** (les noms et les descriptions sont remplacés par des numéros).
- Il alloue, stocke et crée des structures de données de la machine virtuelle pour représenter les classes.

Le convertisseur ne prend pas seulement en entrée les fichiers *.class* à convertir en fichiers .CAP mais aussi un ou plusieurs fichiers d'exportation. Si le package importe des classes d'autres packages le convertisseur charge aussi les fichiers d'exportation de ces packages.

Le convertisseur produit donc un fichier CAP et un fichier d'exportation du package converti.

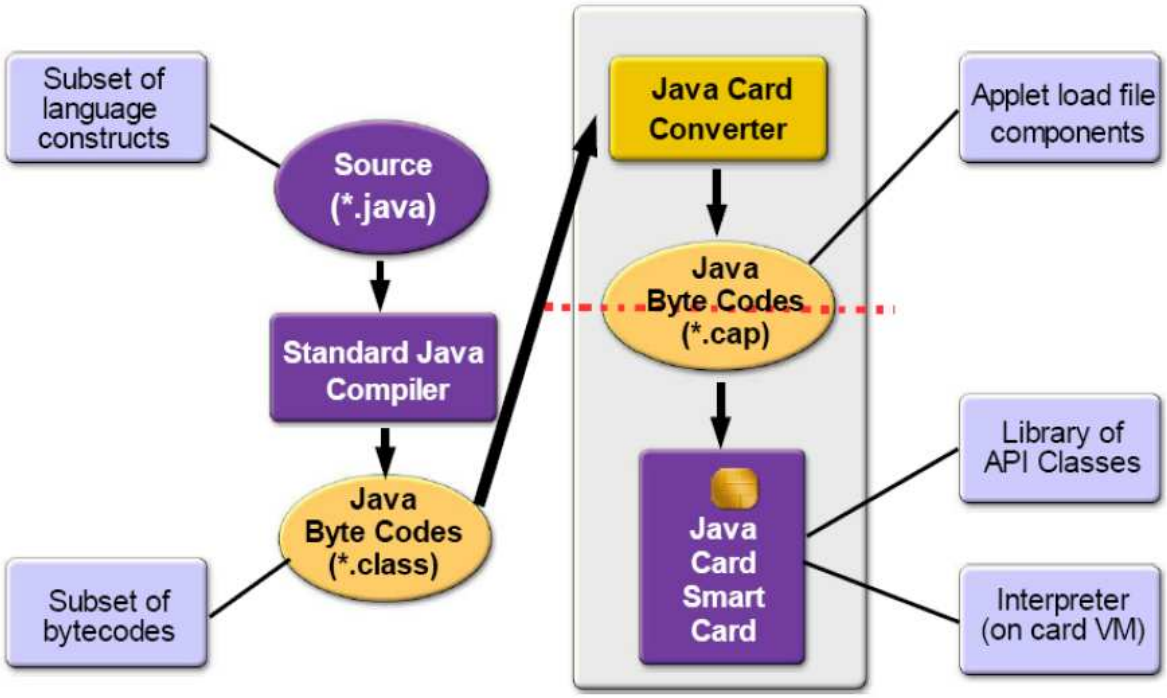


Figure II.3.Le convertisseur [LAN 07-08]

- **L'interpréteur :**

L'interpréteur fournit un support d'exécution pour les applications indépendantes du *hardware* de la carte. Il exécute des instructions *bytecode* obtenu après compilation et édition de lien et contrôle l'allocation et la création des d'objets dans la mémoire.

En effet certaines données peuvent être communes à toutes les applications hébergées mais une donnée utilisée par une application particulière ne pourra être allouée à une autre application.

- **L'installeur :**

La procédure d'installation consiste à décomposer un fichier.*cap* en une suite de commandes APDUs envoyées à la carte et traitées par l'installeur. Ce dernier alloue l'espace mémoire nécessaire et crée une représentation mémoire du package. Le fichier.*cap* est ainsi écrit dans la mémoire EEPROM de la carte et reste inactive jusqu'à une sélection explicite.

II.4.2.2. L'interface de programmation d'applications JavaCard (JCAPI) :

La JC API définie les fonctionnalités offertes par **JavaCard** aux développeurs d'applications. Elle inclut un très petit sous ensemble de classes Java standard et des classes dédiées au développement d'applications pour cartes à puce. L'API définie les conventions par lesquelles une applet **JavaCard** accède aux JCRE et aux méthodes natives incluant les fonctionnalités de l'OS, l'accès à la mémoire et les opérations d'E/S.

Elles incluent des classes adaptées à la plateforme java fournissant un support sur le langage et des services de cryptographie. Elles contiennent également des classes spécialement conçues pour supporter le standard ISO 7816 des smart cards.

Dans le développement d'application à base de **JavaCard**, cette dernière doit avoir une configuration matérielle minimale afin de profiter pleinement de cette API. Ainsi, la configuration requise selon SUN [**Zhi 02**] est :

- RAM : 1 Ko.
- EEPROM : 16 Ko.
- ROM : 24 Ko.

La JC API contient trois packages principaux (**Javacard.lang, Javacard.framwork et javacard.security**) et un package d'extension **javacardx.crypto :**

Le package javacard.lang :

Ce package est un sous-ensemble du package java équivalent java.lang. Les classes supportées sont Object, Throwable, et quelques classes d'exceptions reliées à la machine virtuelle. Beaucoup de méthodes java ne sont toutefois pas disponibles dans ces classes. La classe **JavaCard** Object par exemple ne définit qu'un constructeur par défaut et la méthode *equals*.

Le package javacard.framework :

Le package javacard.framework est essentiel. Il fournit l'architecture des classes et des interfaces pour le fonctionnement basique des applets **JavaCard**. Plus important, il définit la classe de base Applet qui donne le cadre d'exécution et d'interaction des applets durant leur fonctionnement. Deux autres classes importantes de ce package sont la classe APDU qui sert aux protocoles de transmission et la classe PIN qui gère les mots de passe.

Le package javacard.security :

Ce package est utilisé pour les fonctions de cryptographie de la plate-forme Java Card.
Son implémentation est basée sur le package Java du même nom java.security. Il définit notamment la classe keyBuilder et des interfaces variées représentant le système cryptographique de clés (**DES**) et (**DSA** et **RSA**).

Le package javacardx.crypto :

Comme dit précédemment ce package est un package d'extension. Il contient des classes de cryptographie et des interfaces dont les spécifications sont régulées par les Etats-Unis. Il définit la classe abstraite **Cipher** pour l'encryption et la décryption des données.

II.4.2.3. Des services incluant le firewall et le partage :

Chaque applet s'exécute dans un espace qui lui est propre. La JCRE doit assurer cette propriété. Une applet ne peut avoir d'effet sur une autre applet, à moins que cette dernière fournisse une interface d'accès (interface shareable). Il existe des mécanismes de partage, qui

permettent à une applet d'accéder à des services offerts par une autre applet ou par le JCRE. Cette caractéristique est le « firewall » des applets.

Le firewall est une sorte de protection spécifique à **JavaCard**, il est indépendant de la technologie Java et offre une sécurité supplémentaire et automatique. Il fournit une sécurité contre les erreurs fréquemment commises par les développeurs et qui pourraient autoriser la fuite de données sensibles vers d'autres applets. Il fournit aussi une protection contre le code malveillant en empêchant l'accès aux objets.

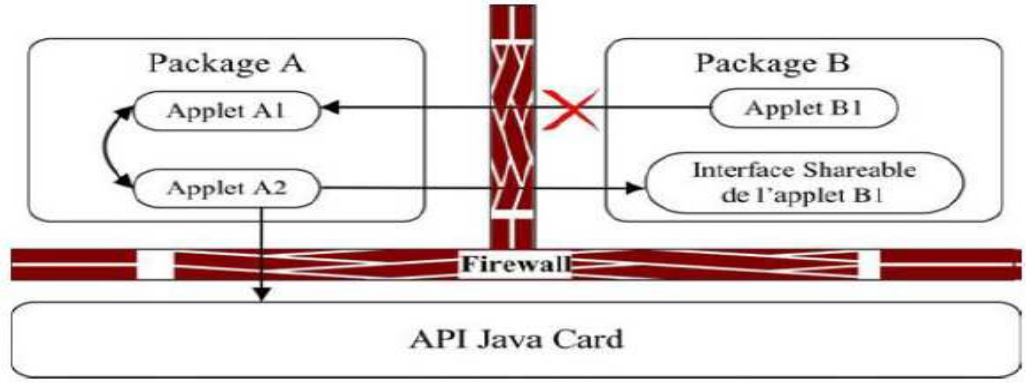


Figure.II.4.Le firewall [Kar 08].

II.4.3.Les Applets JavaCard :

Une applet **JavaCard** est une application qui étend toujours **javacard.framework.Applet** , une fois correctement chargée sur la plateforme, permettra à cette dernière d'accéder à une ou plusieurs fonctionnalités supplémentaires. Ces fonctionnalités sont définies dans le code de l'applet par le développeur.

II.4.3.1.Les étapes de développement d'une applet [Bou] :

- Spécifier les fonctions de l'applet.
- Assigner des AIDs à l'applet et au package contenant la classe de l'applet.
- Concevoir les programmes de l'applet.
- Définir l'interface applet/terminal.

L'assignation des AIDs :

Dans la technologie **JavaCard**, chaque applet est identifiée et sélectionnée par un identificateur

Unique (**AID** : **A**pplet **ID**entifier). De même, à chaque package est assigné un **AID**. Cette convention de nommage est conforme à l'ISO 7816.

Un **AID** est représenté par un tableau dont la taille varie entre 5 et 15 Octets.il est constitué par la concaténation du **RID** (**R**essource **ID**entifier) de taille fixe de 5 octets, et du **PIX** (**P**roprietary **I**dentifier e**X**tension) de taille variant entre 0 et11 Octets.

C'est l'ISO procède à l'attribution des **RID**s aux entreprises afin que chacune ait son propre **RID** ; ensuite c'est à l'entreprise elle-même de contrôler la gestion des **PIXs**.

Définition des méthodes de l'applet [Sun 3]:

Une applet **JavaCard** doit hériter la classe **javacard.framework.Applet**. Celle-ci est une superclasse des applets résidentes sur la carte. Elle définit les méthodes courantes que doit implémenter une applet pour interagir avec le JCRE.

Les méthodes les plus courantes sont présentées dans le tableau qui suit :

Méthode	Définition
Select ( )	Méthode appelée par le JCRE pour informer l'applet de sa sélection
Deselect ( )	Méthode appelée par le JCRE pour informer l'applet couramment sélectionnée qu'une autre applet sera sélectionnée.
Install ( )	Le JCRE appel cette méthode pour créer une instance de l'applet
Register ( )	Cette méthode est appelée par l'applet pour enregistrer l'instance de cette applet au sein du JCRE.
Process ( )	Cette méthode est l'équivalent de la fonction *main* dans un programme Java. Elle est appelée par le JCRE pour exécuter une commande APDU
selectingApplet ( )	Elle est appelée par la méthode process( ) pour distinguer entre la commande SELECT qui permet de sélectionner l'applet et les autres commande SELECT relatives à la sélection de fichiers ou à l'état interne de l'applet.

Tableau II.1.Les méthodes principales définies dans la classe javacard.framework.Applet [Sun 3]

II.4.3.2.Cycle de vie d'une applet :

Une applet est une application serveur. Une fois le code de l'applet proprement chargé sur la carte, la vie de l'applet commence lorsqu'une instance est créée et enregistrée au sein du JCRE.

Une applet doit définir la méthode *install()* pour créer une instance d'applet en utilisant son constructeur et doit enregistrer l'instance au sein du JCRE en invoquant la méthode *register()*,si aucune exception ne sera signalée lors de l'enregistrement, l'installation de l'applet sur la carte serait jugée réussie .

Une applet **JavaCard** reste inactive jusqu'à ce qu'elle soit explicitement sélectionnée.

Lorsque le JCRE reçoit une commande APDU SELECT, il consulte sa table interne pour chercher l'applet dont l'AID correspond à celui spécifié dans la commande. S'il la trouve, le JCRE prépare sa sélection. Cette préparation se fait en deux étapes: d'abord, si une applet couramment sélectionnée est présente en mémoire, alors le JCRE la désélectionne en invoquant la méthode *deselect()*. L'applet exécute la méthode *deselect()* avant de devenir inactive. Le JCRE invoque alors la méthode *select()* pour informer la nouvelle applet de sa sélection. L'applet retourne vrai à la méthode *select()* si elle est prête à devenir active et à traiter n'importe quelle commande APDU. Sinon, l'applet retourne faux pour décliner sa participation.

Une fois l'applet est sélectionnée, le JCRE fait suivre toutes les commandes APDU (y compris la commande APDU SELECT) à la méthode process(). Dans la méthode process(), l'applet interprète chaque commande APDU et exécute la tâche spécifiée. Pour chaque commande APDU, l'applet répond au lecteur en envoyant une réponse APDU qui l'informe du résultat de traitement de la commande APDU.

Ce dialogue commande-réponse continue jusqu'à ce qu'une nouvelle applet soit sélectionnée ou bien que la carte soit retirée du lecteur. Lorsqu'elle est désélectionnée, l'applet devient inactive jusqu'à sa prochaine sélection.

Le cycle de vie d'une applet est illustré dans le schéma ci-dessous :

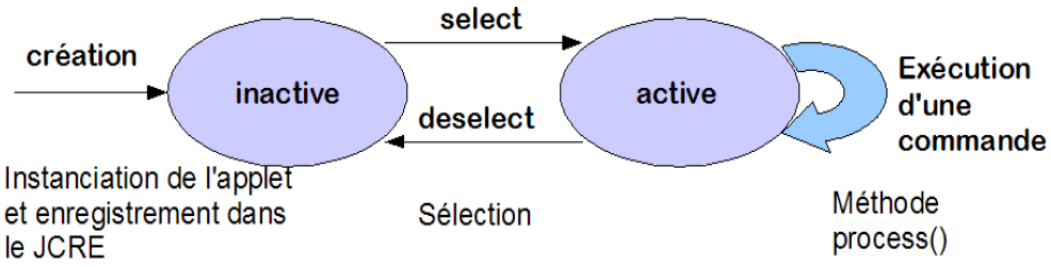


Figure II.5.Le cycle de vie d'une applet JavaCard [1] .

II.4.3.3.La définition de l'interface applet/terminal [Bouzef07]:

Une applet qui fonctionne sur une carte à puce communique avec l'application externe en utilisant le protocole **APDU** (**A**pplication **P**rotocol **D**ata **U**nits défini par l'ISO 7816).

Donc, l'interface entre l'applet et l'application est un ensemble de commandes **APDU** supportées aussi bien par l'applet que l'application.

II.5.Avantages de la technologie JavaCard :

- **Développement facile :**

Avant le développement d'application pour carte à puce était réalisé en assembleur ou en C. La mise au point de programmes avec **JavaCard** sera plus rapide et les temps de développement seront sensiblement réduits (langage haut niveau).

- **Sécurité :**

JavaCard possède un modèle de sécurité qui permet à plusieurs applications de coexister en sécurité sur la même carte et cela grâce à plusieurs niveaux de control d'accès aux méthodes et aux variables (public, protected, private).
Le firewall : Il isole les applets dans des zones appelées contextes

- **Une plateforme multi application :** plusieurs applets peuvent coexister sur une même carte

- **Ouverture, compatibilité et interopérabilité des applets (Write Once, Run Anywhere).**

II.6.Conclusion :

Dans ce chapitre on a présenté les cartes à puce et les technologies **JavaCard**, largement répandues et utilisées à travers le monde. Leur nombre en circulation est estimé à plus d'un milliard. Les applications courantes qui les utilisent sont les cartes SIM 3G, les cartes d'identification d'entreprises ou gouvernementales, les cartes de paiement pour décodeur TV, la finance, etc. Ses cartes peuvent être utilisées avec ou sans contact (RFID).

L'approche réactive au dessous de JAVACARD

III.L'approche réactive au dessous de JavaCard

III.1.Introduction :

Selon D. Harel et A. Pnueli [HP].Tout système informatique est soit transformationnel, interactif ou réactif.

Un système transformationnel effectue des calculs à partir des données fournies en entrée par un utilisateur, pour produire des résultats en sortie puis il se termine, donc son interaction avec son environnement se limite à acquérir des informations et produire des résultats comme le fait un compilateur.

Contrairement à un système transformationnel, un système interactif ou un système réactif se caractérise par une interaction constante avec son environnement en produisant des résultats à chaque invocation. Ces résultats dépendent des données fournies par l'environnement lors de l'invocation, ainsi que de l'état interne du système. Par exemple Les interfaces homme machine et les programmes de contrôle de processus industriels.

Vu les quantités importantes d'information qu'un système informatique doit traiter, et les réponses que doit fournir le plus vite possible en résultat de plusieurs requêtes utilisateur ; ce système doit être en mesure de gérer la concurrence et le parallélisme qui en découlent.

Les systèmes réactifs introduits par **Harel** et **Pnueli** ont été développés dans plusieurs cadres qui ont donné naissance à l'approche synchrone et à un ensemble de langages de programmation, dits synchrones, qui l'implémentent.

Parmi ces langages, on trouve Java qui propose les threads comme outil qui gère la concurrence et le parallélisme, Si le modèle des threads semble bien adapté à la prise en compte du parallélisme, plusieurs difficultés ont été relevées, non seulement au niveau de la programmation qui nécessite beaucoup de compréhension du mécanisme pour se lancer dans un développement multithreads, mais aussi dans la sémantique des primitives qui n'est pas claire(le comportement d'un programme utilisant ces primitives est différent d'une plateforme à une autre, en plus les possibilités des deadloks (verous mortels) et d'inversions de priorité qui rendent la programmation par thread très délicate). Différentes propositions ont vu le jour afin d'étendre ce langage avec différents modèles d'expression de la concurrence et le rendre ainsi plus compatible avec la programmation modulaire à objets parmi celles-ci on trouve l'approche réactive synchrone.

Plusieurs équipes de recherche, à **l'INRIA**, au **CNRS** et à l'école des Mines de Paris ont développés et utilisés un ensemble de langages construits à base de cette approche, dont les plus connus sont :**Esterel, Lustre, Signal, SugarCube, Junior, REJO,…,** etc**.**

Dans les plateformes à faible capacité mémoire et traitement telles que les **JavaCard**s, l'approche réactive n'est pas supportée, d'où la nécessité d'adopter une nouvelle approche pour l'expression de la concurrence et du parallélisme qui sera alors l'approche réactive dans une plateforme **JavaCard**.

III.2.L'approche synchrone:

L'approche synchrone a été créée en 1982 par plusieurs équipes de recherche françaises qui étudiaient les caractéristiques des systèmes de contrôle et les systèmes temps réel (systèmes avec des contraintes temporelles qui sont utilisées de plus en plus dans des contextes critiques).

L'approche synchrone repose sur une hypothèse fondamentale selon laquelle la durée logique des réactions est nulle, c'est-à-dire un système réagit en temps zéro.

À chaque activation, un système réagit, aussi rapidement que possible, à toute modification de son environnement (les *entrées* ou *inputs*) et engendre à son tour une modification de l'environnement (les *sorties* ou *outputs*), de ce fait les sorties sont simultanées avec les entrées qui les ont provoquées ceci peut engendrer des problèmes de causalité.

II.2.1.Langages Synchrones :

On présente ici un récapitulatif de la plupart des langages qui ont été créés en utilisant l'approche synchrone. L'idée de cette liste est de donner une vision générale du développement de ce domaine.

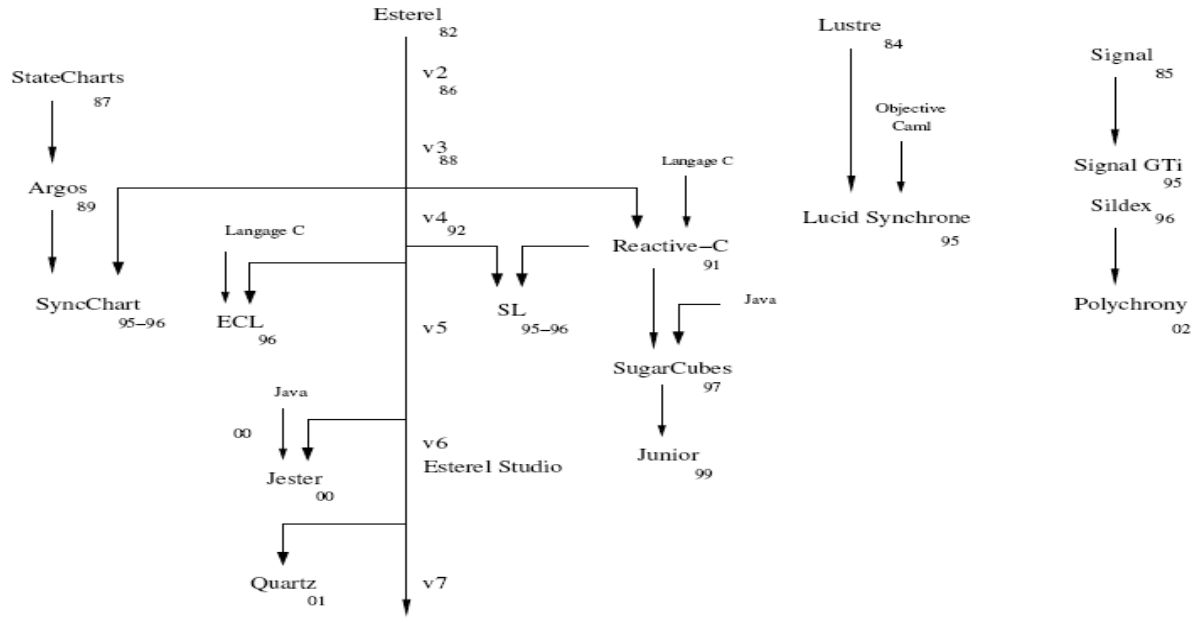


Figure III.1. L'évolution de l'approche synchrone[Raul A]

III.3.L'approche réactive:

L'approche réactive a été conçue on se basant sur l'approche synchrone. La principale caractéristique d'un système réactif est de réagir, rapidement et en continu, aux activations et aux modifications de l'environnement. En reposant sur une découpe logique de l'exécution d'un programme en une succession d'instants. 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=instant+concurrence+diffusion d'événements+dynamisme

III.4. Définitions :

III.4.1.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. 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 : tous les composants parallèles constituant le programme évoluant au même rythme.

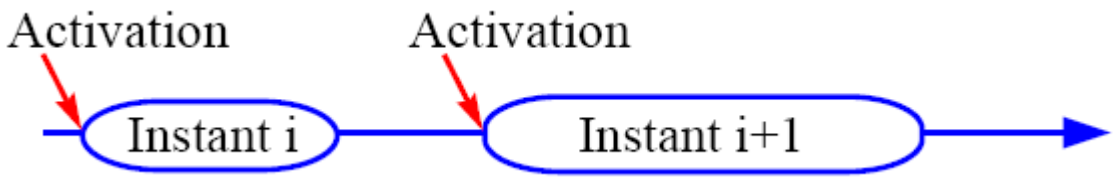


Figure III.2.La notion d'instant [Ch B]

III.4.2.La Concurrence :

Comme dans l'approche synchrone, l'approche réactive 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 i.e. tous les composants parallèles du système ont terminés leur exécution pour l'instant courant, et se synchronisent à chaque fin d'instant.

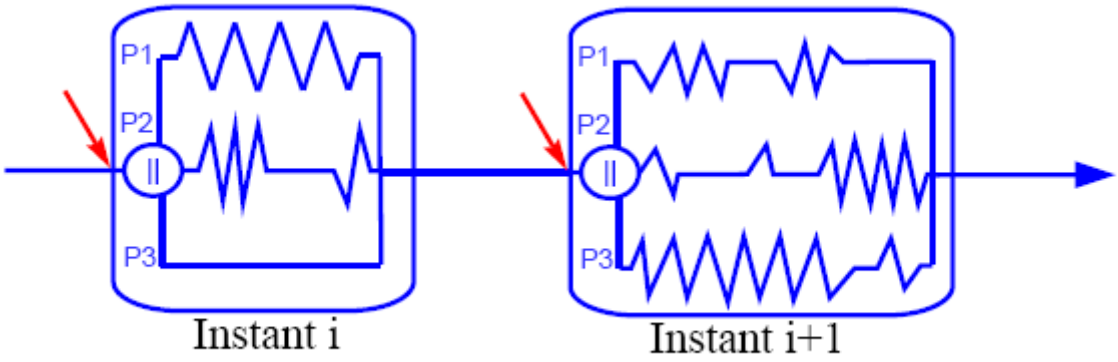


Figure III.3.La concurrence [Ch B]

III.4.3.La diffusion d'événements :

Pour permettre aux composants parallèles de communiquer entre eux, l'approche réactive fait appel aux événements, ces derniers permettent aussi aux composants en attente sur un même événement de se synchroniser. 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. Il faut savoir qu'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 du système, l'approche réactive ne définit aucune hypothèse quant à l'absence d'un événement pendant un instant et elle ne réalise qu'un événement est absent qu'à la fin de l'instant. Donc, aucune réaction n'y est 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.

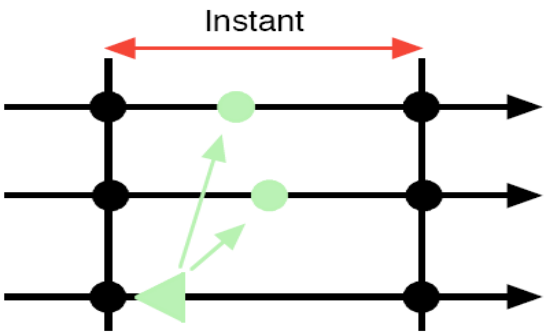


Figure III.4.La diffusion d'événements [jfs]

III.4.4.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.
Afin de garantir la stabilité du système à la fin de celui-ci, ces opérations sont uniquement effectuées entre deux instants.

L'ajout :

Ajout dynamique de nouveaux composants parallèles entre deux instants. Cet ajout dynamique ne pose aucun problème puisque les programmes, de par les restrictions sémantiques des formalismes de l'*Approche Réactive*, ne peuvent générer des problèmes de causalité. La composition parallèle est donc toujours possible.

Le retrait :

Retirer dynamiquement des composants en train de s'exécuter dans un système à la fin d'un instant. Cela permet d'implémenter des systèmes de codes mobiles: le composant retiré est suspendu et son exécution peut être reprise ultérieurement sur un autre site (simple péremption). Il est aussi possible d'implémenter des systèmes persistants par exemple pour la tolérance aux pannes: un programme peut être copié sur un support de stockage à la fin d'un instant pour être repris ultérieurement en cas de panne(une préemption dans laquelle on garde l'état du composant préempté pour pouvoir le réutiliser dans un autre environnement d'exécution).

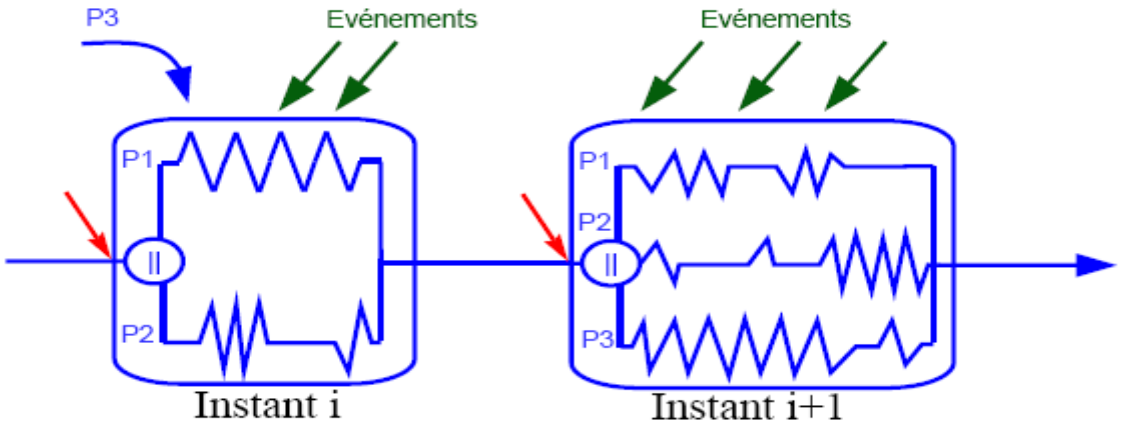


Figure III.5.Le dynamisme [Ch B]

III.5.Les formalismes réactifs :

Issue de l'approche synchrone, l'approche Réactive, a engendré plusieurs formalismes initialement construits aux dessus de langages procéduraux tels que **C** et **Tcl-Tk**. En particulier, le langage **Reactive-C** proposé comme une extension réactive du langage **C**, qui a servi à implémenter d'autres formalismes réactifs de plus haut niveau.
Les principaux formalismes qui ont été développés dans ce cadre sont les suivants :

- **Reactive-C** : Ce langage ajoute un ensemble de primitives réactive, au langage **C** pour coder les réactions.
- Le langage **SL** : Il s'agit d'un langage synchrone très proche d'**Esterel** dans lequel la restriction fondamentale, pour éviter les problèmes de causalité, consiste à interdire la réaction instantanée à l'absence de signaux. Comme en **Esterel**, la création dynamique est interdite en **SL** et il n'y a pas de possibilité de programmation de micro-étapes de réaction.
- Les **Objets Réactifs** : il s'agit en effet d'un modèle de programmation, qui vise à intégrer les concepts de la programmation objet avec les principes de l'*Approche Réactive*.Par exemple : Le modèle **ROM** (**R**eactive **O**bject **M**odel) propose de considérer un type particulier d'objets, et que chaque méthode d'un objet est un

programme réactif qui s'exécute pour la durée d'un instant à chaque fois qu'elle est invoquée.

- Les **Objets Réactifs Distribués :** Il s'agit d'une variante du modèle **ROM**, dans laquelle on considère des systèmes distribués.
- Les **Scripts Réactifs** : Ils apportent toute la souplesse et le dynamisme d'un langage interprété à l'*Approche Réactive*. Les programmes exécutés en parallèle par l'interprète sont ajoutés dynamiquement en cours d'exécution.
- Les **Icobjs** : La **programmation graphique par Icobjs** constituent un formalisme particulier, permettant d'exprimer graphiquement la programmation par combinaison de comportements élémentaires.

III.6.L'approche réactive au dessous de java :

Parmi les langages qui nous offrent l'opportunité d'étudier la pertinence de l'*Approche réactive*, on trouve le langage Java.

Dans un cadre orienté objet et distribué. L'*Approche Réactive* peut être vue comme une solution adéquate aux problèmes rencontrés dans le modèle des threads, pour l'expression de la concurrence au niveau d'un langage de programmation.

L'*Approche Réactive* se focalise sur l'expression de l'ordonnancement et du contrôle des opérations dans un système réactif.

III.6.1.Atouts du langage java :

Java possède un ensemble de caractéristiques attrayantes afin d'exprimer dans un programme la concurrence et la communication entre activités parallèles sans recourir aux threads.

- **Java** possède un modèle objet simple et facile à appréhender (héritage simple, tout est classe et objet…). Le nombre de notions à intégrer est restreint ce qui en fait un langage bien adapté au prototypage.
- **Java** possède une *API* très riche permettant un développement rapide en se concentrant sur l'essentiel. Ses nombreuses facilités d'accès au réseau (Socket, RMI, …) qui permettent l'étude de l'*Approche Réactive* dans un cadre distribué.
- **Java** est un langage à bytecode qui en fait un langage particulièrement portable et ce quelques soient les plates-formes considérées.

III.7.Les formalismes réactifs au dessous de java :

Les langages de programmation à objets se sont imposés dans le développement des systèmes informatiques, en proposant une découpe en modules d'un programme guidée par la structure des données que ce dernier doit manipuler.

Avec l'accroissement des systèmes distribués et plus particulièrement de l'Internet, le besoin d'intégrer dans les modèles objets les notions de parallélisme et de concurrence est devenu inévitable, Le modèle couramment retenu dans les systèmes distribués est un modèle basé sur des *threads*. Tel est le modèle de concurrence proposé initialement en langage java.

Puisque le modèle de threads proposé en java s'est avéré impuissant pour l'expression de très nombreuses activités concurrentes modulaires et fortement communicantes, Différentes propositions ont vu le jour afin d'étendre ce langage avec différents modèles d'expression de la concurrence et le rendre ainsi plus compatible avec la programmation modulaire à objets.

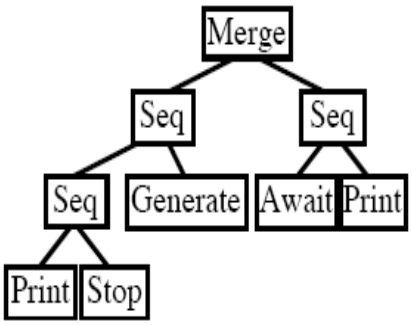
Exemple : **JavaTriveni, Reactive Java, Jester, sugarcubes,junior,…,etc**

On va aborder quelques formalismes visant à étendre Java pour la programmation de systèmes réactifs, leurs avantages et leurs inconvénients, ensuite on va choisir un formalisme qu'on pourra ensuite implémenter après des élagages sous la plateforme **JavaCard**.

III.7.1.Les SugarCubes v1 :

L'implémentation de l'*Approche Réactive* au-dessus de **Java** a pour objectif de fournir des moyens d'expression simples, avec une sémantique précise, de la concurrence et du parallélisme. À ce titre, les **SugarCubes** proposent des outils permettant d'implémenter des systèmes réactifs au-dessus de **Java.** L'*Approche Réactive* propose un ensemble de primitives exprimant l'ordonnancement et le contrôle des tâches d'un programme. Dans les **SugarCubes** ces primitives réactives sont implémentées par des objets, il ne s'agit pas d'un langage à part, mais chaque primitive est décrite par une classe **Java** particulière.

Un programme réactif est une collection d'objets instanciés à partir de ces classes. Ainsi, un *objet-instruction* encapsule l'état de l'instruction réactive qu'il représente dans un programme. À chaque réaction du système, une méthode particulière de cet objet-instruction exécute le comportement de la primitive réactive si celle-ci doit être exécutée. L'état de l'instruction est modifié en conséquence afin de mémoriser l'effet d'une activation et préparer la suivante, jusqu'à ce que la primitive ait été totalement exécutée.




```
Program p = new Merge(
      new Seq(
        new Seq(new Print("HelloWorld"),new Stop())
        ,new Generate("e")
      )
      ,new Seq(new Await("e"),new Print("e!"))
);
```

Figure III.6.Un programme SugarCubes [jfs]. Chaque nœud est un objet Java**.**

Les **SugarCubes** sont un ensemble de classes **Java** qui implémentent les primitives réactives, les machines virtuelles chargées de leur exécution et les éléments implémentant l'environnement d'exécution.

On peut donc découper les **SugarCubes** en 3 sous-ensembles logiques:

- **Les instructions réactives** : Ce sont des objets qui implémentent les primitives réactives; par exemple, l'opérateur de séquence, l'opérateur de parallélisme, la primitive de génération d'un événement, la primitive de préemption, etc. …
- **Les machines réactives :** Ce sont des objets utilisés pour exécuter les programmes construits à partir de primitives réactives. Typiquement, une machine réactive(i) possède un programme qu'elle active cycliquement, (ii) définit les instants d'exécution et (iii) gère l'environnement d'exécution (environnement événementiel, objet **Java** par défaut, etc.…).
- les classes implémentant **l'environnement d'exécution**, comprenant entre autres, les événements.

III.7.1.1.Les instructions réactives :

Une primitive réactive est implémentée par une classe particulière et son utilisation dans un programme réactif est obtenue par instanciation de la classe désirée à la construction du programme. Toutes les primitives réactives répondent à un certain nombre de méthodes déclarées dans la classe abstraite **Instruction** dont toutes les primitives **SugarCubes** héritent.

Un objet-instruction peut être activé par un appel de sa méthode ***activ***. L'instruction réagit alors à cette activation en exécutant une micro-étape de son comportement. À la fin de l'activation, l'instruction retourne un statut représentant l'état d'avancement de son exécution au cours de l'instant, ce statut correspond à l'une des trois valeurs suivantes: ***TERM***, ***STOP*** ou ***SUSP***, déclarées dans l'interface **ReturnCodes**. Ces différentes valeurs sont retournées dans les conditions suivantes :

TERM (terminée) signifie que l'exécution de l'instruction est complètement terminée (pour cet instant et les instants futurs). Ainsi, si on l'active aux instants suivants, rien ne se passera et l'instruction retournera toujours ***TERM***.

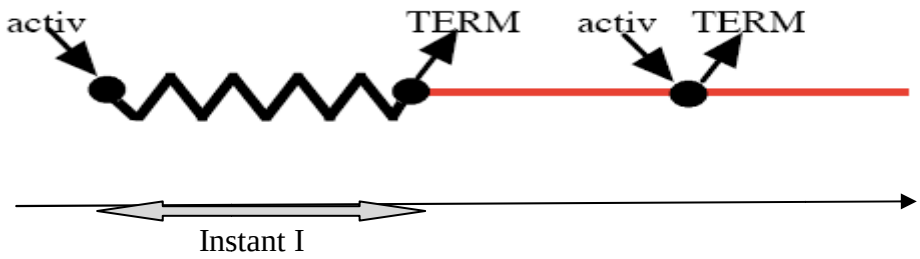


Figure III.7. TERM indique la terminaison d'une instruction réactive [jfs]

STOP (stoppée) signifie que l'exécution d'une instruction est stoppée dans un état stable pour l'instant courant. Son exécution pourra reprendre, à l'instant suivant, à partir de l'endroit précis où celle-ci s'est arrêtée pour l'instant courant. L'exécution n'est donc pas définitivement terminée; simplement il n'y a plus rien à exécuter pour l'instant courant. Par analogie avec la signalisation routière qui demande au véhicule arrivant à un panneau ***STOP*** de s'arrêter avant de redémarrer, le statut ***STOP*** marque un point d'arrêt de l'exécution d'un programme avant de reprendre à l'instant suivant.

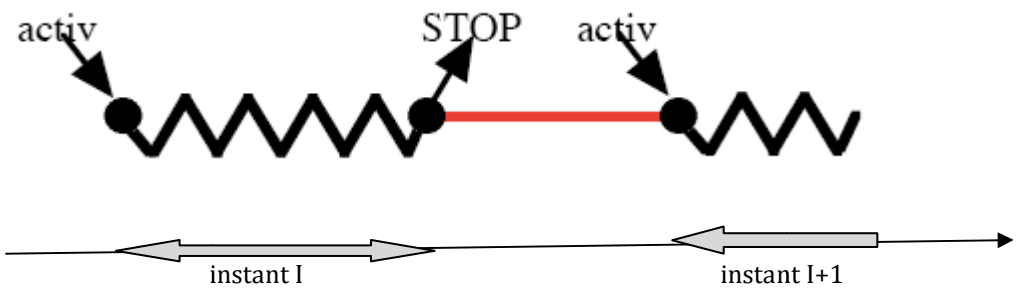


Figure III.8. STOP indique que l'instruction à terminé son exécution pour l'instant courant [jfs].

SUSP (suspendue) signifie que l'exécution de l'instruction n'a pas encore atteint un état stable pour l'instant en cours. D'autres micro-étapes vont être nécessaires pour terminer la réaction et l'instruction doit être activée de nouveau au cours de cet instant. Cela se produit en particulier lorsque l'instruction se trouve en attente d'un événement qui n'a pas encore été généré dans l'environnement d'exécution.

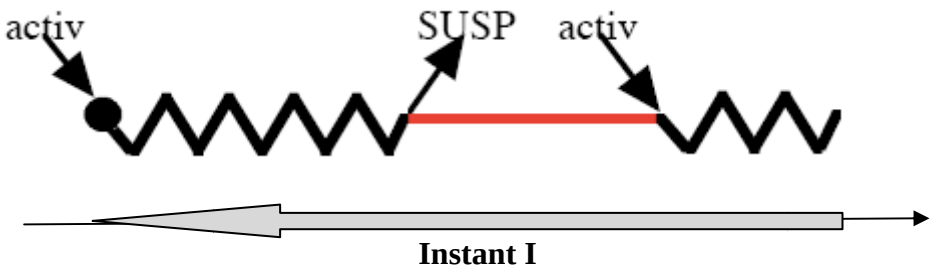


Figure III.9. SUSP indique qu'une instruction se suspend [jfs]

Instructions unaires et binaires [jfs]:

Certaines instructions, comme par exemple l'instruction de séquence, peuvent contrôler l'exécution d'autres instructions. Ces instructions possèdent des références sur les sous-instructions qu'elles contrôlent. En **SugarCubes**, ils existent deux catégories de ces opérateurs particuliers : les opérateurs unaires et les opérateurs binaires; auxquelles correspondent deux classes abstraites: **UnaryInstruction** et **BinaryInstruction**.

La classe **UnaryInstruction** étend la classe **Instruction** et possède un corps (body) qui est lui-même une instruction dont elle contrôle l'exécution. Cette classe est utilisée par exemple pour implémenter une instruction de boucle.

La classe **BinaryInstruction** étend elle aussi la classe **Instruction** et possède deux branches left (gauche) et right (droite) qui sont des références sur les instructions contrôlées par l'opérateur binaire.

Cette classe est utilisée par exemple pour définir l'opérateur de séquence (qui est un opérateur binaire en **SugarCubes**) ou l'opérateur de parallélisme (qui est aussi un opérateur binaire).

III.7.1.2.Les principales primitives :

Chaque primitive réactive des **SugarCubes** est définie par dérivation de la classe **Instruction** ou de sous-classes comme les classes **BinaryInstruction** ou **UnaryInstruction.**Ci-dessous un diagramme qui résume les principales primitives définies dans **SugarCubes**.

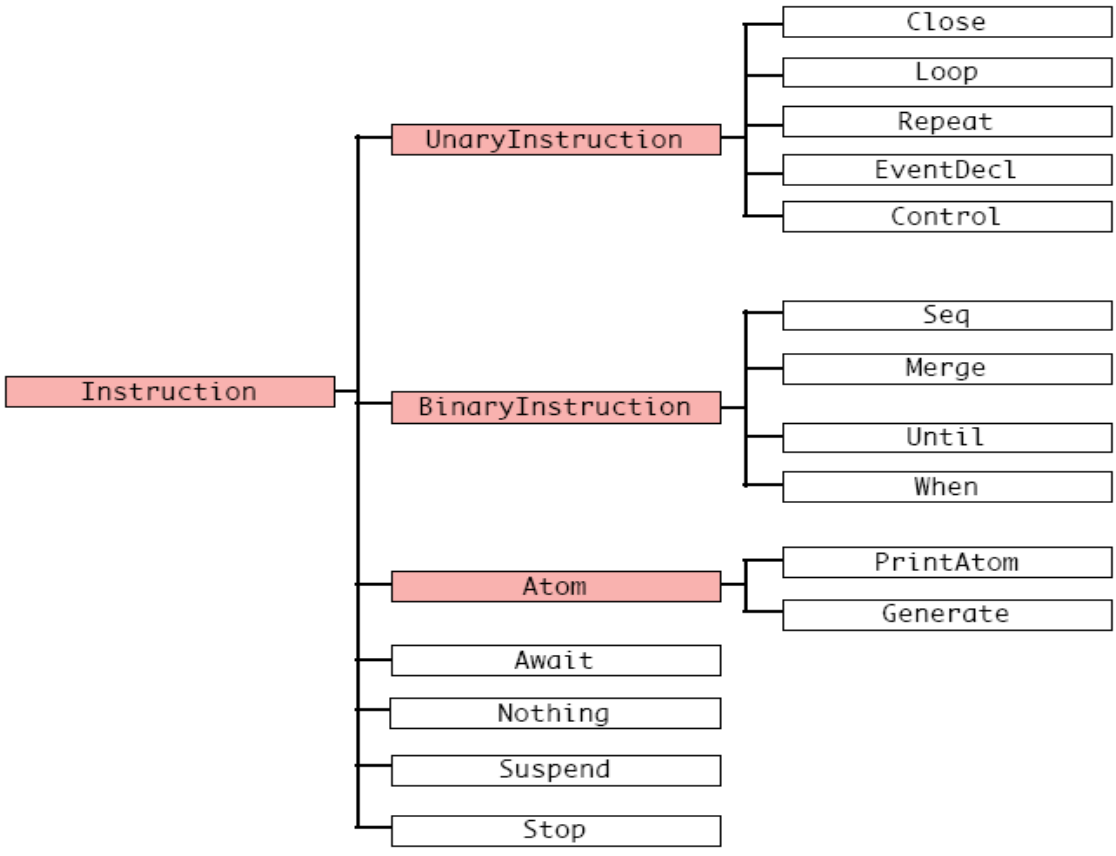


Figure III.10. Les principales primitives définies dans SugarCubes [jfs].

III.7.1.3.La Machine d'exécution :

Un programme réactif constitué d'objet-instructions doit être exécuté par une machine d'exécution réactive. En **SugarCubes**, une machine d'exécution est un objet qui possède un programme à exécuter.

Une machine réactive est une instance de la classe **Machine** qui possède un champ **program** qui est une instruction réactive. Initialement, le programme d'une machine d'exécution réactive est simplement l'instruction **Nothing** qui ne fait rien. Il est possible de rajouter de nouveaux programmes en parallèle en appelant la méthode **add** qui crée un nouveau programme constitué de la mise en parallèle du programme déjà présent dans la machine et du nouveau programme à ajouter.

Une machine d'exécution fournit au programme un environnement d'exécution et va définir les instants partagés par l'ensemble du système. En particulier la machine d'exécution va se charger de déterminer le nombre de micro-étapes nécessaires pour que le système atteigne un état stable. Une machine d'exécution décide donc des fins d'instants pour un système réactif.

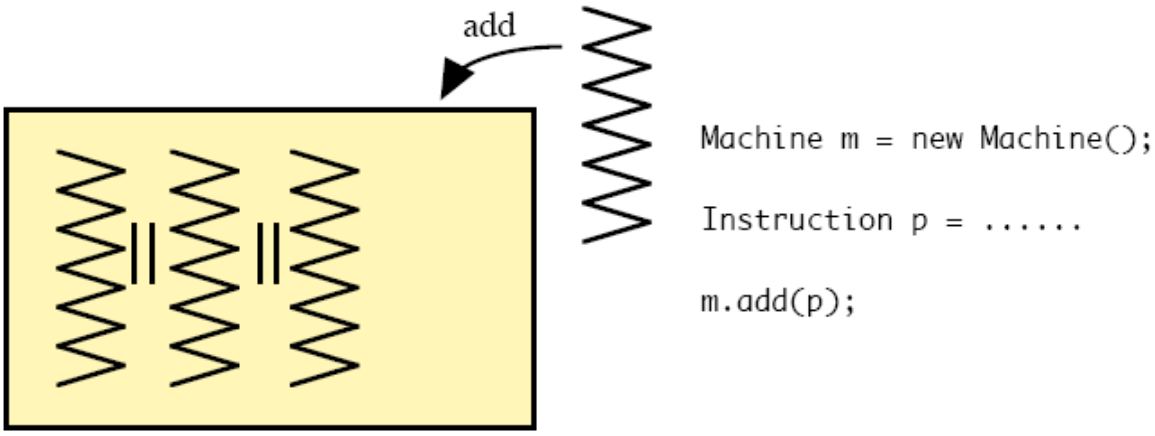


Figure III.11.La machine réactive [jfs].

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 dans 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. A 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**.

III.7.1.4.Les événements :
La notion d'événement de l'*Approche Réactive* est implémentée en **SugarCubes** par la classe Event.
Les événements **SugarCubes** ont les propriétés suivantes:

• Les événements sont automatiquement réinitialisés au début de chaque instant. Ainsi, les événements ne sont pas des valeurs persistantes dans l'environnement, conservées entre les instants.
• Les événements peuvent être générés par l'appel de leur méthode **generate**. Cela détermine la présence de l'événement durant l'instant où cette méthode a été appelée. Générer un événement qui a déjà été généré n'a aucun effet.

• Tous les composants parallèles dans un système réactif ont la même vision de chaque événement à chaque instant.

• Il n'est pas possible de décider si un événement est absent au cours d'un instant, avant que l'instant ne soit terminé. C'est en effet le seul moment où l'on est sûr que personne ne pourra plus le générer et viendrait ainsi contrarier cette hypothèse. De ce fait, la réaction à l'absence d'un événement est toujours reportée à l'instant suivant. Ceci constitue le principe de base de l'*Approche Réactive*.

L'implémentation des événements en **SugarCubes**, repose sur un principe très simple: le champ generated de l'objet Event mémorise le numéro d'instant de la dernière génération de l'événement. La méthode generate de l'événement mémorise donc le numéro de l'instant où intervient la génération. La méthode **isPresent** compare le numéro d'instant courant de la machine d'exécution avec le numéro d'instant de la dernière génération.

Cependant, si une réponse positive de la méthode **isPresent** donne une information immédiatement exploitable sur la présence de l'événement, une réponse négative ne permet pas de conclure à l'absence de cet événement. Pour connaître l'état d'un événement au cours d'un instant, il convient donc d'utiliser la méthode qui renvoie le statut courant de l'événement. À chaque activation d'un programme (c'est-à-dire à chaque micro-étape de l'exécution d'un programme), un événement peut présenter une des trois valeurs suivantes définies dans l'interface **EventConsts**:

```
public interface EventConsts
{

final byte PRESENT = 1;
final byte ABSENT = 2;
final byte UNKNOWN = 3;
}
```

Au début de chaque instant, la présence d'un événement est indéfinie. La méthode **presence** renvoie ***UNKNOWN*** et tout composant d'un système intéressé par cet événement devra se suspendre pour l'instant courant jusqu'à ce que le statut de l'événement soit fixé. Si l'événement est généré dans l'instant, le statut passe alors à ***PRESENT***. Cette information est immédiatement exploitable par un système réactif pour pouvoir poursuivre son exécution. Lorsqu'un événement est déclaré présent au cours d'un instant, rien ne pourra plus contredire cet état jusqu'à la fin de l'instant (un événement généré, reste présent jusqu'à la fin de l'instant). Si l'événement reste dans l'état ***UNKNOWN*** jusqu'à ce que la machine d'exécution décrète la fin d'instant, cela signifie qu'aucun composant du système réactif n'a généré cet événement. La fin de l'instant étant déclarée, l'événement est donc absent et le statut retourné par la méthode **presence** est donc ***ABSENT***. Les instructions en attente sur l'état de cet événement sont donc informées de l'absence de cet événement; comme l'instant est terminé, ces instructions se préparent à réagir pour l'instant suivant. Ainsi, l'évolution du statut d'un événement au cours d'un instant ne peut emprunter que deux chemins:

• ***UNKNOWN*** devient ***PRESENT*** (si l'événement est généré);

• ***UNKNOWN*** devient ***ABSENT*** (si la fin d'instant survient sans qu'il n'y ait eu génération).

Les **SugarCubes** proposent de manipuler les événements dans un programme réactif par l'intermédiaire d'expressions booléennes basées sur la présence d'un ou plusieurs événements. Ces expressions appelées configurations événementielles.

III.7.1.5.Limitations des SugarCubes :[jfs]

Les **SugarCubes v1** constituent une première implémentation des principes de l'*Approche Réactive*, au dessus de **Java**. Ils présentent un certain nombre de limitations ainsi que des problèmes au niveau du choix de l'implémentation et de la sémantique de certains opérateurs.
Les limitations des **SugarCubes v1** ont été envisagées suivant quatre axes principaux :

1. l'intégration à l'environnement objet du langage **Java**.
2. l'interface utilisateur à travers laquelle ce dernier construit et exécute ses programmes réactifs.
3. l'efficacité de l'implémentation, en particulier, en termes de traitement d'un grand nombre d'événements et de composants parallèles par instant.
4. la sémantique précise des opérateurs de l'*Approche Réactive*.

Un certain nombre de travaux ont déjà été menés, afin de proposer de nouveaux modèles de programmation, regroupant les objets et les systèmes réactifs, favorisant l'intégration des comportements réactifs **SugarCubes** avec le modèle objet **Java**. Ils offrent également un certain nombre de nouveaux mécanismes améliorant la modularité et la réutilisation des développements. Les différentes versions des **SugarCubes** qui ont été développées ont permis d'élaborer progressivement plusieurs modèles, ainsi que les primitives facilitant leurs mises en œuvre.

III.7.2.Le formalisme Junior(JR) :

JR est un langage de programmation créé spécialement pour résoudre des problèmes de programmation concurrente. Ce langage est en fait une surcouche de Java qui ajoute à ce dernier les principaux paradigmes de la programmation concurrente.

Junior est né des expériences faites sur la première implémentation des **SugarCubes**. Son objectif principal est de donner une sémantique formelle au modèle réactif synchrone; sous forme de règles de sémantique formelle assez simple exprimé en utilisant des règles de ***s****émantique* ***o****pérationnelle* ***s****tructurelle* (**SOS**).Ce qui permet de donner une meilleure compréhension du modèle et de valider les différentes implémentations.

Le modèle d'exécution de **Junior** est le même que celui de **SugarCubes**, les classes **junior** définissent des primitives réactives permettant d'écrire un programme réactif dans un style impératif à la SugarCube.Ces instructions réactives sont exécutées par des machines réactives.

III.7.2.1.Modèle d'exécution :

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

- Les systèmes considérés sont cycliques, c'est-à-dire qu'ils ne finissent pas.
- Ils doivent être suffisamment rapides pour suivre les changements de leur environnement.

Son approche réactive est caractérisée par 4 éléments :

1. La notion d'instant.
2. Les comportements réactifs.
3. La notion d'événement.
4. La propriété de dynamisme.

III.7.2.1.1.Les instructions réactives :

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 qui 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**.

Le statut nommé 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.

III.7.2.1.2.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 **III.4**.

Le comportement de la machine réactive est le même que celle des **SugarCubes**.

La diffusion d'événements 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.

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.

- **Types de machines et leur API :**

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

Jr.Machine(Program P) : Crée une unsafe-machine et y charge le programme **P**.

Jr.SafeMachine(Program P) : Crée une safe-machine et y charge le programme **P**.

La safe-machine est une machine qui a été conçue pour être exécutée dans un environnement multitâche, c'est-à-dire qu'elle supporte l'accès concurrent de 2 ou plus de threads Java, sans perte d'information. En particulier, elle garantit la consistance de l'information lorsque 2

threads utilisent l'API de la machine réactive en même temps. L'API des deux machines est la suivante:

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

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

generate(Identifier E) :Génère l'événement **E** dans la machine.

getFrozen(Identifier E) : Rend une copie des programmes gelés, grâce à l'instruction **Freezable**, associés à l'événement E.

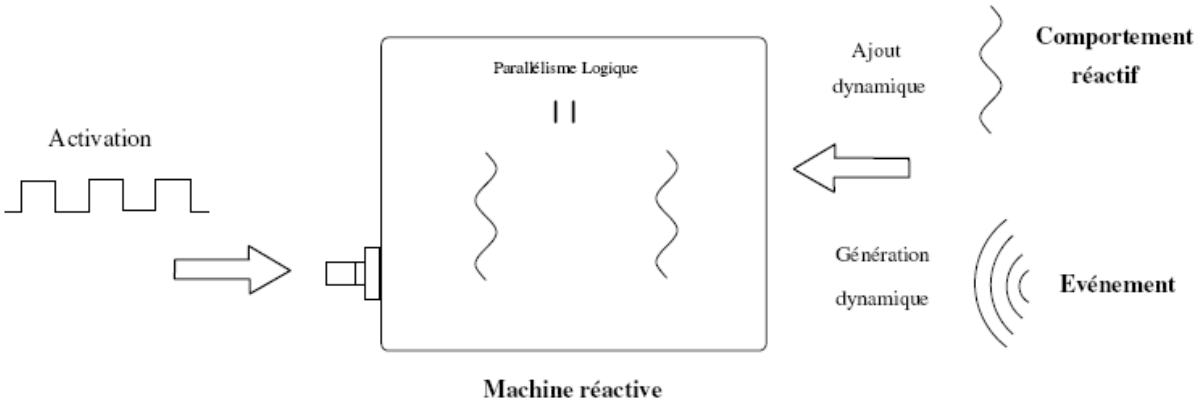


Figure III.12. L'approche réactive Synchrone[Raul A].

III.7.2.1.3. Les API Junior :

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 de base :**

Junior définit une interface de programmation pour l'utilisateur à l'image de ce qui a été présenté en permettant de masquer les différentes implémentations qui ont été expérimentées. **Junior** découpe la liste des primitives mises à la disposition de l'utilisateur en deux classes:

- La classe **JrBase** : propose l'ensemble des primitives réactives implémentées par **Junior**.
- L'interface **Jr** qui étend la classe **JrBase** fournit un ensemble de facilités syntaxiques simplifiant l'écriture de certains programmes réactifs.

On présente maintenant les principales primitives qui ont été retenues en **Junior** :

• **Jr.Nothing**(): ne fait rien et termine immédiatement, ce qui correspond à retourner TERM à chaque activation.

• **Jr.Stop():** marque la fin de l'instant dans une séquence, Cela correspond à retourner **STOP** à la première activation et **TERM** à toutes les suivantes.

• **Jr.Atom(Action a):** exécute une action atomique externe **a** (qui implémente l'interface **JavaAction**) implémentée en **Java** et termine immédiatement.

• **Jr.Seq(program first, program second):** met en séquence deux programmes **first** et **second**. Lorsque **first** termine on passe immédiatement en séquence au programme **second**.

• **Jr.Loop(program body):** exprime une boucle infinie. Le programme **body** est immédiatement redémarré dès qu'il termine.

• **Jr.Repeat(IntWrap Iw, program body):** exprime une boucle finie exécutant un certain nombre d'itérations du programme **body**. Le nombre d'itérations à réaliser est déterminé lors de la première activation de l'instruction **Repeat** par l'évaluation de l'expression externe

IntWrap (qui implémente l'interface **IntegerWrapper**) exprimée en **Java** qui retourne une valeur entière.

• **Jr.Par(program left, program right):** met en parallèle deux programmes **left** et **right**. Contrairement à l'opérateur **Merge**, l'opérateur **Par** ne fixe pas d'ordre particulier dans l'exécution des deux programmes. La seule garantie est que les deux branches seront exécutées à chaque instant. **Par** stoppe pour l'instant courant lorsque ses deux branches ont marqué la fin d'instant. La construction parallèle termine quand les deux branches terminent.

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*

Figure III.13.Statut retourné par l'instruction Par.

• **Jr.Generate(IdWrap,ObjWrap)**: génère un événement dans l'environnement d'exécution. L'événement est identifié par un identificateur particulier déterminé à l'exécution (**runtime)** par l'évaluation de l'expression **IdWrap** (qui implémente l'interface **IdentifierWrapper**) qui retourne un identificateur. La valeur associée à la génération d'événement est déterminée à l'exécution par l'évaluation de l'expression **ObjWrap** (qui implémente l'interface **ObjectWrapper**) qui retourne un objet en guise de valeur.

• **Jr.Await(C):** attend la satisfaction de la configuration **C** au cours d'un instant pour terminer et permettre le passage en séquence.

• **Jr.When(C, then, else)**: teste au cours d'un instant la configuration événementielle **C**. Si elle est satisfaite, la branche **then** est exécutée, sinon la branche **else** est exécutée. L'exécution de l'une ou l'autre des deux branches est instantanée si la configuration peut être évaluée avant la fin d'instant, sinon l'exécution est repoussée à l'instant suivant.

• **Jr.Until(C,body,handler):** réalise la préemption du corps **body** lorsque la configuration de préemption **C** est satisfaite. C'est une préemption faible qui laisse le corps s'exécuter à

l'instant où **C** est satisfaite. Le **handler** est exécuté si la préemption est effective. Son exécution est immédiate si la configuration est satisfaite avant la fin de l'instant et que le corps est stoppé, sinon le **handler** est exécuté à l'instant suivant.

• **Jr.Control(IdWrap,body):** contrôle l'exécution de l'instruction body en fonction de la présence d'un événement de contrôle dans l'environnement d'exécution. L'identificateur de l'événement de contrôle est déterminé à l'exécution lors de la première activation de l'instruction par l'évaluation atomique de l'expression externe **IdWrap** (qui implémente l'interface **IdentifierWrapper**) qui retourne un identificateur d'événement dans l'environnement d'exécution. Le corps n'est exécuté pour un instant que lorsque l'événement de contrôle est présent.

• **Jr.Local(id,p):** déclare un événement local identifié par l'identificateur **id**. La portée de l'événement local est limitée au programme **p**.

• **Jr.If(CondWrap,then,else):** choisit d'exécuter la branche **then** ou la branche **else** en fonction du résultat de l'évaluation de la condition booléenne. L'évaluation de l'expression booléenne externe **CondWrap** (qui implémente l'interface **BooleanWrapper**) exprimée en **Java** s'effectue de façon atomique à la première activation du **If**. Si la condition est vraie la branche **then** est exécutée, sinon c'est la branche **else**.

• **Jr.Link(ObjWrap,body):** associe un objet **Java** implicite au comportement **body**. L'objet est renvoyé à l'exécution par l'évaluation de l'expression atomique externe **ObjWrap** (qui implémente l'interface **ObjectWrapper**) qui est exprimée en **Java** et renvoie l'objet qui sera désormais associé au corps du **Link**.

• **Jr.Freezable(id,body):** permet de geler une instruction **body** si l'événement de gel identifié par **id** est présent. À l'instant où l'événement de gel est présent, le corps est quand même exécuté, à la manière de la préemption faible. Le corps gelé est ajouté dans la table des instructions gelées dans l'environnement, en parallèle avec les autres instructions gelées par le même événement s'il y en a.

III.7.2.1.4. Evénements et configurations événementielles :

❖ **Les événements :**

L'approche réactive dispose d'un moyen de communication puissant qui est 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 :**

Junior définit une notion d'événement avec laquelle on peut décider si un événement est présent ou absent dans un instant. Cette caractéristique permet de définir les opérateurs de la logique booléenne classiques qui utilisent la présence ou l'absence d'un événement. Cette logique booléenne basée sur la notion d'événement a été construite en **Junior** avec la définition de 4 opérateurs de type **Configuration**.

Jr.Presence(IdentifierWrapper Iw) : Teste si un événement est présent. L'´evénement est identifié par l'´evaluation du **wrapper Iw** qui retourne un identificateur,cette évaluation est effectuée lors du premier test de présence d'événement.
Jr.And(Configuration C1, Configuration C2) : Retourne la conjonction des configurations C1 et C2.
Jr.Or(Configuration C1, Configuration C2) : Retourne la disjonction des configurations **C1** et **C2**.
Jr.Not(Configuration C) : Retourne la négation de la configuration **C**, ce qui permet de donner la possibilité aux instructions événementielles de réagir à l'absence d'un ou plusieurs événements.

❖ **Construire un programme en Junior :**

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

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

```
Import   junior . *;
import  j r e .*;

public class Exemple{
public  static void main ( String [ ] args ){

Machine M = Jre .Machine ( );//La création d'une machine réactive.


Program exemple = Jre . Loop ( Jre . Seq ( Jre .Atom(new Action ( ) {

void execute (Environment  env ){
     System . out . print ( "bonjour" ) ;
                                    }
                                                          }
                                                ),  Jre . Stop ( )
                                    )
                              ) ;//  La création d'un programme exemple

M. add (exemple);//Ajout du programme exemple à la machine M

for ( int i =1; i <=5; i++)

M. react ( );//Faire réagir la machine M
                                                }
                              }
```

Figure III.14.Exemple d'un programme Junior [Raul A]

III.7.2.2.Sémantique de Junior [jfs]:

Dans la section **III.7.2.1.3** on a donné la description du comportement des instructions réactives de base dans **Junior**. Cette description donnée en langage naturel s'avère utile pour donner une idée générale du comportement de chaque instruction réactive. Cependant, lorsqu'on commence à construire des programmes réactifs plus complexes, cette description n'est pas suffisante car elle ne donne pas la description exhaustive de toutes les interactions possibles entre les instructions réactives. Pour avoir une description précise du fonctionnement de **Junior**, sa sémantique formelle a été définie.

Junior formalise la sémantique des opérateurs de l'*Approche Réactive* à travers un ensemble de règles *SOS (Sémantique Opérationnelle Structurelle)* très simple. . Pour ce faire, nous opérons un certain nombre de simplifications par rapport à l'API complète de **Junior**:

• On ne considère que la sémantique des opérateurs réactifs. Ainsi tous les mécanismes d'interface avec le monde extérieur aux seules instructions réactives ne sont pas considérés. Par exemple, les méthodes **add** ou **generate** d'une machine d'exécution qui ne sont pas utilisées en tant qu'instructions réactives ne sont pas formalisées.

• L'interfaçage avec le langage **Java** n'est pas considéré ici. En particulier, les instructions **Link**,**If**, etc. et les mécanismes d'évaluation à l'exécution ne sont pas formalisés. En effet, l'environnement **Java** d'un programme réactif est défini indépendamment de la notion d'instant. Il s'agit d'objets **Java** définis par ailleurs dans un programme applicatif standard. De plus, le non déterminisme (en termes d'effet de bord), apporté par l'opérateur **Par** de **Junior**, implique que la prise en compte des effets de bords **Java** dans ces conditions n'apporte aucune information pertinente et contribue à alourdir les notions.

• Les facilités syntaxiques offertes par l'API **Junior** (classe Jr), exprimant des simplifications dans l'utilisation de certains opérateurs ne sont pas considérées (par exemple, l'opérateur **Until** sans **handler**). En effet, ce type d'opérateurs s'exprime directement par des macro-instructions où les branches absentes sont remplacées par une instruction **Nothing**. Leur sémantique peut donc être directement déduite. Il en va de même pour les simplifications permettant l'utilisation de chaînes de caractères directement comme identificateurs d'événements etc.

• Les notions d'événements valués et leur génération ne sont pas considérées. On ne traite ici que le cas des événements purs.

On va maintenant présenter un système de règles de sémantique formelle de référence, appelé **Rewrite**, exprimé sous la forme de règles de réécritures.

A. Les règles SOS :

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

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

$$\frac{\textbf{premisse1 premisse2 … … … … premisse n}}{\textbf{Conclusion}}$$

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

Plusieurs variantes de la SOS ont été proposées, par exemple la sémantique **big-step** ou **macro-step** qui, par opposition à la sémantique **small-step** ou **micro-step**, cherche à capturer le comportement décrit entre deux états considérés comme l'état initial et l'état final. Un exemple de sémantique **macro-step** est celle qui fut développée par **Gilles Kahn**, nommée sémantique naturelle.

On donne un système de règles décrivant le comportement de chaque primitive en termes de micro-étapes (micro-step). On parlera donc de *sémantique micro-step*.

Le format général d'une règle de réécriture est le suivant:

$$t, E \xrightarrow{\alpha} t', E' \quad \textbf{Où} \quad \alpha \in \textbf{\{SUSP, STOP, TERM\}}$$

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

La sémantique de **Junior** est composée de 45 règles de réécriture de type micro-step qui définissent les instructions réactives du langage **Junior** et de 3 règles micro-step, plus une règle de réécriture de type macro-step, qui définissent la notion d'instant. La notion d'instant

du modèle réactif est divisée en micro-instants ou micro réactions; les micro-réactions sont implémentées à l'aide de trois drapeaux de terminaison:

• **TERM** : qui signifie que l'exécution est terminée pour l'instant courant et qu'il ne reste rien à faire au prochain instant.

• **STOP :** signifie que l'exécution est terminée pour l'instant courant mais qu'il reste quelque chose à faire dans le prochain instant.

• **SUSP** signifie que l'exécution n'est pas terminée pour l'instant courant et qu'elle doit être reprise dans le même instant.

B.Syntaxe :

La sémantique repose sur un nombre réduit de règles qu'on va représenter. Celles-ci fournissent une définition extrêmement compacte de **Junior** à la base de ses implémentations. L'interprétation de la syntaxe utilisée est la suivante :

• Les instructions réactives sont représentées par leur syntaxe abstraite, et non leur syntaxe concrète; par exemple l'instruction **Nothing** qui a la syntaxe concrète **Jr**.**Nothing**() est représentée par **Nothing**.

• Lorsque les instructions utilisent des paramètres qui sont des programmes réactifs, on utilise des lettres minuscules (en général **t** et **u**). Par exemple, l'instruction réactive **Loop** qui a la syntaxe concrète **Jr**.**Loop**(**Program t**) est dénoté par **Loop(t).** Si une instruction réactive, par exemple **t**, est réécrite en un nouveau terme, on le dénote par **t0**.

• Les événements sont dénotés par des lettres majuscules: **S** pour les événements simples et **C** pour les configurations (des conditions événementielles formées avec des opérateurs **And** et **Or**). Par exemple, l'instruction réactive **Await** qui a la syntaxe concrète **Jr**.**Await**(**Config c**) est dénotée par **Await(C).**

• Les instructions événementielles utilisent 3 prédicats : **sat(C,E)** pour tester si la configuration **C** est satisfaite dans l'environnement E, **unsat(C,E)** lorsque la configuration **C** n'est pas satisfaite, et **unknown(C,E)** lorsque on peut pas encore savoir son état.

C. Définition de l'environnement d'exécution :

L'environnement d'exécution des instructions réactives, symbolisé par la lettre ***E*** dans les règles de réécriture, est constitué d'un certain nombre de composants:

• Un ensemble d'événements présents au cours d'un instant.

• Une valeur booléenne ***eoi(E)*** qui indique si la fin d'instant a été décidée par la machine d'exécution (***true***) ou non (***false***).

• Une valeur booléenne ***move(E)*** qui est positionnée à vrai lorsqu'un changement dans l'environnement d'exécution intervient au cours d'une activation du programme.

• Une table utilisée pour stocker les instructions gelées au cours d'un instant.

Ainsi, un événement *S* qui est présent dans ***E*** s'écrit: $S \in E$. L'ajout de l'événement ***S*** dans l'environnement ***E*** est noté $E + S$ tandis que $E - S$ représente l'environnement ***E*** auquel on a retiré l'événement ***S***. La notation $E[move = b]$ représente l'environnement ***E*** dans lequel le drapeau ***move*** a été affecté avec la valeur ***b***. De la même manière, la notation $E[eoi = b]$ indique que le drapeau ***eoi*** est affecté avec la valeur ***b***.

La notation $E / F[S]$ représente un environnement égal à $E + S$ si $S \in F$ et $E - S$ sinon. Cette notation signifie que l'environnement $E / F[S]$ est identique à l'environnement ***E*** sauf dans le cas du signal ***S*** dont le statut est déterminé par son statut dans l'environnement ***F***. Ces notations sont utilisées dans la définition des événements locaux.

Enfin, la notation $E\{S = t\}$ associe l'instruction gelée ***t*** à l'événement ***S*** et la notation $E\{S\}$ représente l'instruction gelée associée à ***S***.

D.Primitives de base :

Nothing :

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

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

Stop :

L'instruction *Stop* arrête l'exécution d'une séquence pour l'instant courant. À l'instant suivant il ne reste rien à faire.

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

Les actions atomiques :

Les actions atomiques terminent immédiatement en une seule micro-étape après avoir exécuté une action externe *a*. Celle-ci est exécutée de façon atomique (aucune interférence concurrente ne peut venir perturber l'exécution de *a*).

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

On ne rend pas compte ici de l'effet de bord produit par l'action a comme nous l'avons précisé plus haut.

L'opérateur de séquence binaire :

La séquence est un opérateur défini par deux règles. Le choix de l'une ou l'autre de ces règles est dicté par le statut d'exécution de la première branche:

Si la première branche ne termine pas alors on applique la règle:

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

Si la première branche termine (retourne ***TERM***), alors la seconde branche est immédiatement activée:

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

L'opérateur de parallélisme :

Si les deux branches de l'opérateur **Par** sont suspendues (ce qui est la situation initiale), alors on exécute au choix (non déterministe) l'une des 4 règles suivantes.

Pour les deux premières, une seule branche est activée et on positionne le drapeau move à **true** pour indiquer que la décision de fin d'instant doit être repoussée après la micro-réaction suivante afin de donner une chance à la branche qui n'a pas été exécutée de l'être ultérieurement au cours du même instant:

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

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

Cependant il est également possible que les deux branches soient exécutées au cours de la même micro étape et ce dans un ordre quelconque. Ceci est traduit par les deux règles suivantes:

$$\frac{t, E \xrightarrow{\alpha} t', E' \quad u, E' \xrightarrow{\beta} u', E''}{Par_{SUSP,SUSP}\ (t,u), E \xrightarrow{\gamma(\alpha,\beta)} Par_{\gamma_1(\alpha,\beta),\gamma_2(\alpha,\beta)}\ (t',u'), E''}$$

$$\frac{u, E \xrightarrow{\beta} u', E' \quad t, E' \xrightarrow{\alpha} t', E''}{Par_{SUSP,SUSP}\ (t,u), E \xrightarrow{\gamma(\alpha,\beta)} Par_{\gamma_1(\alpha,\beta),\gamma_2(\alpha,\beta)}\ (t',u'), E''}$$

Notations:

• Le statut correspondant à la notation $\gamma_{(\alpha,\beta)}$ correspond aux valeurs issues du tableau à double entrées suivant:

α/β	**TERM**	**STOP**	**SUSP**
TERM	**TERM**	**STOP**	**SUSP**
STOP	**STOP**	**STOP**	**SUSP**
SUSP	**SUSP**	**SUSP**	**SUSP**

• $\gamma_{1(\alpha,\beta)}$ est égal à ***SUSP*** si α est égal à ***STOP*** et β est égal à ***STOP*** ou ***TERM***, sinon $\gamma_{1(\alpha,\beta)}$ est égal à α.
• $\gamma_{2(\alpha,\beta)}$ est égal à ***SUSP*** si β est égal à ***STOP*** et α est égal à ***STOP*** ou ***TERM***, sinon $\gamma_{1(\alpha,\beta)}$ est égal à β.

Remarque: La première de ces deux règles faisant réagir les deux branches au cours de la même micro-étape correspond à la sémantique de l'opérateur **Merge** des **SugarCubes** qui, à chaque activation, fait réagir ses deux branches toujours dans le même ordre (branche 1 puis branche 2 -gauche puis droite-).

On retrouve bien ici, la volonté de **Junior** d'unifier les différentes formes de parallélisme primitif au sein du même opérateur.

Si une seule des deux branches est suspendue, alors l'opérateur ***Par*** se contente de l'activer

$$\frac{t, E \xrightarrow{\alpha} t', E' \quad \beta \neq SUSP}{Par_{SUSP,\beta}\ (t,u), E \xrightarrow{\gamma(\alpha,\beta)} Par_{\gamma_{1(\alpha,\beta)},\gamma_{2(\alpha,\beta)}}\ (t',u), E'}$$

$$\frac{u, E \xrightarrow{\beta} u', E' \quad \alpha \neq SUSP}{Par_{\alpha,SUSP}\ (t,u), E \xrightarrow{\gamma(\alpha,\beta)} Par_{\gamma_{1(\alpha,\beta)},\gamma_{2(\alpha,\beta)}}\ (t,u'), E'}$$

L'instruction Loop :

Une boucle exécute son corps et se réécrit en une séquence si son corps ne termine pas immédiatement:

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

Lorsque le corps de la boucle termine immédiatement alors la boucle est immédiatement réactivée pour la même micro-étape:

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

L'instruction Repeat :

L'instruction **Repeat** termine immédiatement si son compteur est inférieur ou égal à 0:

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

Sinon, la règle suivante est appliquée:

$$\frac{n > 0\ Seq(t, Repeat(n-1,t)), E \xrightarrow{\alpha} u, E}{Repeat\ (n,t), E \xrightarrow{\alpha} u, E}$$

E. Les primitives événementielles :

Les configurations événementielles :

Une configuration événementielle est une expression booléenne basée sur la présence d'événements dans l'environnement. Ces expressions sont constituées des expressions élémentaires suivantes:

• **Configuration positive**: ***S***. Cette configuration retourne ***true*** si l'événement ***S*** est présent, ***false*** sinon.

• **Configuration négative**: ***not C***. Cette configuration retourne la négation d'une configuration.

• **Conjonction de deux configurations**: *C*1 *and C*2. Cette configuration retourne le "*et*" logique de deux configurations.

• **Disjonction de deux configurations**: *C*1 *or C*2. Cette configuration retourne le "*ou*" logique de deux configurations.

Deux fonctions ***fixed*** et ***eval*** sont définies pour les configurations. L'évaluation d'une configuration (fonction ***eval***) n'est valide que lorsque celle-ci est fixée (fonction ***fixed***).

La fonction ***fixed*** retourne vrai lorsque *C* peut être évaluée dans l'environnement ***E*** (les informations nécessaires à cette évaluation sont disponibles):

- *fixed*(*S, E*) ≡*S* ∈ *E* ou *eoi*(*E*)
- *fixed*(*not C, E*) ≡ *fixed*(*C, E*)
- *fixed*(*C*1 *and C*2 *, E*) ≡ *fixed*(*C*1 *, E*) *and fixed*(*C*2 *,E*)

  ou *fixed*(*C*1 *,E*) *and eval*(*C*1, *E*) = *false*

  ou *fixed*(*C*2 *, E*) *and eval*(*C*2 *, E*) = *false*
- *fixed(C1 or C2 , E)* ≡ *fixed(C1 , E) and fixed(C2 , E)*

  ou fixed(C1 , E) and eval(C1, E)

  ou fixed(C2 , E) and eval(C2 , E)

La fonction **eval** retourne la valeur de la configuration **C** dans l'environnement **E**:

- *eval(S, E)* ≡ *S E*
- *eval(not C, E)* ≡ *not eval(C, E)*
- *eval(C1 and C2 , E)* ≡ *eval(C1 ,E) and eval(C2 ,E)*
- *eval(C1 or C2 , E)* ≡ *eval(C1, E) or eval(C2 ,E)*

Grâce à ces fonctions on construit trois fonctions supplémentaires:

- la fonction de satisfaction:

  sat(C, E) ≡ *fixed(C, E) and eval(C, E)*

- la fonction de non-satisfaction:

  unsat(C, E) ≡ *fixed(C, E) and eval(C, E) = false*

- la fonction de configuration non encore déterminée:

  unknown(C, E) ≡ *fixed(C,E) = false*

La primitive Generate :

Une instruction Generate ajoute l'événement à générer dans l'environnement d'exécution et termine immédiatement:

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

L'instruction When :

La branche *then* est exécutée si la configuration *C* est satisfaite. L'exécution est immédiate si la satisfaction a lieu avant la fin d'instant, sinon l'exécution est reportée à l'instant suivant:

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

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

La branche ***else*** est exécutée si la configuration ***C*** n'est pas satisfaite. Comme pour la branche ***then***, l'exécution est immédiate si la non-satisfaction est connue avant la fin d'instant:

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

Sinon, l'exécution est reportée à l'instant suivant:

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

L'instruction est suspendue si la configuration n'est pas encore déterminée:

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

L'instruction Await :

Await attend qu'une configuration soit satisfaite. Cette instruction termine immédiatement si la configuration est satisfaite avant la fin de l'instant, sinon sa terminaison est repoussée à l'instant suivant:

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

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

Si la configuration n'est pas satisfaite ***Await*** retourne ***STOP*** et attend de pouvoir à nouveau évaluer la configuration à l'instant suivant:

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

Await se suspend tant que la configuration n'est pas déterminée:

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

III.8.Les implémentations de l'approche réactive au dessous de JAVA :

Le modèle d'exécution de **Junior** a été implémenté, principalement, de 6 manières différentes. Chaque implémentation a un nom qui reflète l'essence de l'algorithme utilisé. Les noms des versions créées sont **Rewrite**, **Replace**,**Simple**, **Storm**, **Rvm**, et **Glouton**. En effet **Rewrite**, **Replace** et **Storm** sont de la même famille :**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.

III.8.1.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 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**.

L'implémentation repose sur un invariant important: au cours d'une activation, l'état d'un objet-instruction n'est jamais modifié, ce sont les objet-instructions produits par les différentes réécritures qui tiennent compte des modifications de l'état d'un programme. En **Java**, cet invariant est représenté par le fait que les champs d'un objet-instruction sont déclarés constants par l'utilisation du mot clé **final**.

- **L'environnement d'exécution :**

L'environnement d'exécution d'un programme réactif, symbolisé par la lettre ***E*** dans les règles de sémantiques, est implémenté par un objet de la classe **Environment**:

```
public class Environment
{
public Vector eventSet = new Vector();

public boolean eoi,move;

public Hashtable frozen,tempFrozen = new Hashtable();
...
}
```

L'environnement possède deux champs ***eoi*** et ***move*** qui jouent exactement le même rôle que dans les règles de sémantique pour la clôture des instants.

Un environnement événementiel implémenté par un tableau dynamique ***eventSet*** contient l'ensemble des événements générés au cours d'un instant.

Enfin, deux tables de hachage permettent de gérer les instructions gelées au cours d'un instant: une table permet de collecter au cours d'un instant les instructions gelées, tandis que l'autre table donne accès aux instructions gelées au cours de l'instant précédent.

- **Les Instructions :**

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.

```
public class Instruction implements Flags,Cloneable,Program,java.io.Serializable
{
abstrat public MicroState rewrite(Environment env);

public Instruction copy(){

try{

return (Instruction)clone();
   }

catch(CloneNotSupportedException e){
         throw new InternalError(""+e);
                                  }
                        }
}
```

Une instruction se réécrit au cours d'une micro-étape au moment où elle est activée. Cela se traduit en **Java** par l'appel de la méthode **rewrite** d'une instruction: **t.rewrite(env)**. Cette méthode retourne un objet de la classe **MicroState** constitué du terme réécrit et du statut d'exécution obtenu:

```
public class MicroState
{

final public byte flag;

final public Instruction term;

public MicroState(byte flag,Instruction term)
    {
this.flag = flag;
this.term = term.copy();
    }
}
```

Remarque: un **MicroState** mémorise toujours une copie du terme avec lequel on le construit de sorte que l'on assure une réécriture complète du programme à chaque activation.

Stop :

L'implémentation de la règle de réécriture d'une instruction **Stop** se traduit directement comme suit:

```
public class Stop extends Instruction
{

public MicroState rewrite(Environment env){

return new MicroState(STOP, new Nothing());
                                                }
}
```

Séquence :

Les deux règles conditionnelles de la séquence sont traduites dans la méthode **rewrite** de la classe **Seq** comme suit:

```
public class Seq extends BinaryInstruction
{
...

public MicroState rewrite(Environment env){

MicroState s = left.rewrite(env);

if(TERM == s.flag){ return right.rewrite(env); }

return new MicroState(s.flag,new Seq(s.term,right));
                                                }
}
```

On commence par exécuter le premier terme de la séquence (le terme de gauche). Si celui-ci termine (retourne le statut ***TERM***) alors la séquence se réécrit comme la réécriture de son second terme (les séquences se résorbent ainsi progressivement). Sinon la séquence se réécrit comme une séquence du nouveau terme réécrit de gauche et de l'ancien second terme. Le statut associé à cette dernière réécriture, est le statut retourné par la réécriture du premier terme de la séquence.

L'opérateur Par :

Le non déterminisme (en termes d'ordonnancement des effets de bords sur l'environnement **Java**) est situé dans le choix de la règle à appliquer lorsque le statut associé aux deux branches est ***SUSP***.

L'implémentation **Rewrite** de **Junior** se contente de faire le choix d'un opérateur de type **Merge** comme pour les **SugarCubes**.

Instant :

La classe Instant réalise la clôture d'un instant en exécutant autant de micro-étapes que nécessaire:

```
public class Instant extends UnaryInstruction
{
...

public MicroState rewrite(Environment env){

MicroState s = body.rewrite(env);

 if (SUSP != s.flag){ return new MicroState(s.flag,new Instant(s.term)); }

if (env.move){ env.move = false; }else{ env.eoi = true; }

return new Instant(s.term).rewrite(env);
                                                }
...
}
```

ExecContext :

Finalement, la classe **ExecContext** implémente la ressource d'exécution d'un programme réactif. En particulier la méthode **react** qui permet d'exécuter un instant du système.

```
public class ExecContext extends UnaryInstruction
{
...
public boolean react(){
...

MicroState s = instant.rewrite(env);

instant = (Instant)s.term;

env.newInstant();

return (TERM == s.flag);
}
...
}
```

L'implémentation directe des règles de sémantique **Rewrite** fait de cette implémentation une référence très simple à maîtriser. Cependant, celle-ci s'avère assez peu efficace. En effet, le déroulement des instants d'un programme par cette implémentation ne permet pas une exécution rapide, ni même un passage à l'échelle de programmes de grandes tailles (plusieurs centaines de composants parallèles), du fait des très nombreuses créations dynamiques d'objets éphémères produits par les réécritures d'un programme. La gestion de l'allocation mémoire extrêmement lourde dans cette implémentation est très pénalisante, particulièrement en Java et pratiquement en **JavaCard**, où le langage n'utilise pas un *ramasse-miettes*.

Afin de contourner cette difficulté une autre implémentation de **Junior** a été proposée. C'est cette implémentation qu'on décrit succinctement maintenant.

III.8.2.Replace :

L'objectif de la version **Replace** était, d'éliminer les créations inutiles d'objets à chaque exécution. L'état d'une instruction est désormais contenu dans l'instruction elle-même.

Le principe de base est de considérer que l'arbre du programme réactif doit rester figé tout au long de l'exécution. Seuls les états des nœuds (les objet-instructions) du programme sont modifiés par les différentes activations. L'absence de modification de l'arbre du programme est assurée par l'utilisation du mot clé **final** dans les instructions unaires et binaires pour les champs référençant les branches d'un opérateur.

Quelques instructions :

Une instruction réactive définit un comportement réactif codé par un algorithme qui utilise trois valeurs de retours après exécution.

On peut découvrir dans ce présent chapitre l'utilisation faite de ces trois valeurs déclarées dans l'interface **Flags** :

```
public interface Flags {
byte STOP = 0, TERM = 1, SUSP = 2;}.
```

La classe Instruction décrit le comportement générique d'un objet-instruction en **Replace**, La classe **Instruction** est alors la suivante :

```
abstract public class Instruction implements Flags, Program, Cloneable, java.io.Serializable
{
    public transient InternEnvironment env;
    public void bind(InternEnvironment env){ this.env = env; }

    abstract public byte rewrite();
    abstract public void reset();

    public boolean equals(Instruction inst){ return getClass() == inst.getClass(); }

    public Object clone(){
       try{ return (Instruction)super.clone(); }
       catch(CloneNotSupportedException e){ throw new InternalError(""+e); }
    }

    public Program copy(){ return (Instruction)clone(); }

    public Instruction residual(){ return (Instruction)clone(); }
}
```

Comme l'arbre syntaxique du programme n'est pas changé en cours d'exécution, l'environnement d'exécution peut être fixé une fois pour toutes les activations grâce à la méthode **bind**. Cet environnement d'exécution est ensuite utilisé par la méthode **rewrite** (qui n'a donc plus d'argument d'appel) chargée de dérouler le comportement d'une instruction particulière en fonction des règles de sémantique. Cette méthode ne renvoie que le statut de progression de l'exécution de l'instruction puisque le terme n'est pas réécrit par l'activation, seul son état est modifié (on dit que l'état de l'instruction est réécrit).

Enfin, la méthode **reset** permet de remettre l'instruction dans son état initial. Cette méthode est utilisée pour implémenter les boucles.

Les opérateurs unaires et binaires appliquent de manière récursive cette méthode:

```
abstract public class BinaryInstruction extends Instruction{

public Instruction left, right;

public BinaryInstruction(Program left,Program right){

this.left = (Instruction)left; this.right = (Instruction)right;
                                                        }
public void bind(EnvironmentImpl e){ super.bind(e); left.bind(e); right.bind(e); }

public void reset(){ left.reset(); right.reset(); }
...
}
```

Nothing :

L'instruction **Nothing** ne fait rien et termine instantanément. Cette instruction ne nécessite pas d'état à mémoriser, on reste dans le même environnement E et elle retourne TERM.

L'implémentation de cette classe est la suivante :

```
package junior.kernel;

public class Nothing extends Instruction
{
  public void reset(){}
  public String toString(){ return PrintInstruction.Nothing(); }
  public byte rewrite(){ return TERM; }
}
```

Stop :

L'instruction **Stop** est implémentée par le code suivant:

```
package junior.kernel;

public class Stop extends Instruction
{
  public boolean terminated = false;

  public void reset(){ terminated = false; }

  public String toString(){ return (terminated) ? PrintInstruction.Nothing() :
PrintInstruction.Stop(); }

  public boolean equals(Instruction inst){
     return super.equals(inst) && terminated == ((Stop)inst).terminated;
  }

  public byte rewrite(){
     if (terminated) return TERM;
     terminated = true;
     return STOP;
  }

  public Instruction residual(){ if (terminated) return new Nothing(); else return new Stop();
}
}
```

Le booléen **terminated** code l'état de l'opérateur **Stop**. Si ce booléen vaut **false** l'activation retourne **STOP** et met le booléen de terminaison à **true**. Si le booléen **terminated** vaut **true** alors l'activation se contente de renvoyer **TERM**.

Séquence :

La Séquence binaire utilise un booléen pour mémoriser si son premier terme est fini:

```
package junior.kernel;
import junior.Program;

public class Seq extends BinaryInstruction
{
  public boolean leftTerminated = false;

  public Seq(Program left,Program right){ super(left,right); }

  public void reset(){ super.reset(); leftTerminated = false; }

  public String toString(){
     return (leftTerminated) ? (""+right) : PrintInstruction.Seq(""+left,""+right);
  }

  public boolean equals(Instruction inst){
     return (getClass() == inst.getClass())
         && (leftTerminated == ((Seq)inst).leftTerminated)
         && (right.equals(((Seq)inst).right))
         && (leftTerminated || left.equals(((Seq)inst).left));
  }

  public byte rewrite(){
      if (leftTerminated) return right.rewrite();
      byte s = left.rewrite();
      if (s == TERM){ leftTerminated = true; return right.rewrite(); }
      return s;
  }

  public Instruction residual(){
     if(leftTerminated) return right.residual();
     else return new Seq(left.residual(),(Instruction)right.clone());
  }
}
```

La méthode **reset** réinitialise les deux branches de l'opérateur binaire et remet le booléen **leftTerminated** à false. La méthode **rewrite** suit simplement les règles des deux instructions **if** présentes dans son corps traduisant l'existence des trois règles de l'opérateur **Seq**.

Boucles :

Les boucles en **Replace** utilisent la méthode **reset** d'une instruction pour préparer une nouvelle itération de leur corps:

```
package junior.kernel;
import junior.Program;

public class Loop extends UnaryInstruction
{
  public Loop(Program body){ super(body); }

  public String toString(){ return PrintInstruction.Loop(""+body); }
```

```
  public byte rewrite(){
     byte s = body.rewrite();
     if (s != TERM) return s;
     body.reset();
     return rewrite();
  }

  public Instruction residual(){
      Instruction newBody = (Instruction)body.clone();
      newBody.reset();
      return new Seq(body.residual(),new Loop(newBody));
  }
}
```

Replace est plus efficace dans sa gestion de la mémoire puisque le programme n'est pas systématiquement réécrit. Ainsi, l'exécution des instants est plus rapide que dans l'implémentation **Rewrite**. Le prix de cette optimisation réside dans le fait que la taille du programme ne change pas (par exemple, les séquences ne se résorbent pas). Ainsi, Les *branches mortes* (déjà exécutées) d'un programme restent présentes en mémoire. Il faut donc systématiquement tester l'état de certaines instructions (le booléen **leftTerminated** de l'instruction séquence par exemple) pour savoir si oui ou non il faut exécuter une de leurs branches.

Par contre, dans les deux cas, l'exécution parcourt l'arbre du programme de la racine vers les feuilles.

Ce type d'exécution descendante ne semble pas une solution particulièrement adaptée à l'exécution de systèmes mettant en jeu un grand nombre de composants parallèles et d'événements. En effet, les mécanismes d'exécutions descendantes activent des branches suspendues qui ne sont pas forcément capables de progresser (activations inutiles). La complexité de ce genre de situation nécessite la mise en œuvre d'algorithmes plus efficaces, basés sur des techniques d'exécutions remontant l'arbre du programme. Simple propose un algorithme de ce type que nous allons très succinctement évoquer maintenant.

Analyse de Replace :

L'idée de **Replace** consiste à reprendre l'implémentation de **Rewrite** et à éliminer la création inutile d'objets.

Pour cela, **Replace** optimise, principalement, 2 choses :

1. Les instructions réactives sont réutilisées le plus possible avant d'être détruites, en particulier elles sont réutilisées : 1) tant qu'elles n'ont pas fini, et 2) lorsqu'elles sont sous le contrôle d'une instruction **Loop** ou **Repeat**. Pour réutiliser les instructions réactives, elles sont implémentées avec un état interne qui est réinitialisé, par exemple, lorsque l'instruction est finie et est sous le contrôle d'une instruction **Loop**.

2. Le code de retour décrit dans les règles de réécriture est implémenté à l'aide d'une variable globale et non une structure (**MicroState**) contenant le code de retour et la nouvelle

instruction; cette optimisation élimine la recréation, souvent inutile, de la structure ainsi que l'utilisation des paramètres dans la méthode de réécriture **rewrite().**

3. On ne crée des instructions **Nothing** que lorsque cela est nécessaire. Les instructions qui se réécrivent en **Nothing** (**Atom**, **Stop**, **Repeat**, **Await** et **Freezable**) se comportent comme **Nothing** (elles rendent **TERM**) lorsqu'elles ont fini et qu'elles n'ont pas été réinitialisées.

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

• L'utilisation d'une structure d'arbre pour représenter les programmes réactifs. La structure d'arbre pose des problèmes à la construction et à l'exécution du programme car dans les deux cas on utilise des fonctions récursives pour parcourir l'arbre. Si on compile un programme réactif qui génère un arbre de hauteur considérable, c'est la pile du compilateur qui explose. Si on essaie d'exécuter un programme qui a été chargé dans la machine par petits bouts mais qu'`a la fin on obtient aussi un arbre de profondeur considérable, c'est la pile de l'application qui explose lorsqu'on fait l'appel récursif de la méthode **rewrite().**

• L'activation inutile d'instructions. Ce problème se présente sous plusieurs formes qui peuvent se combiner pour donner des performances très pauvres :

– On exécute un arbre en forme de peigne formé par des instructions **Par** ou **Seq** dans lequel (dans le pire cas) il y a juste un composant à exécuter qui est dans la feuille la plus profonde. On active ainsi des instructions qui n'ont plus rien à faire mais qui restent dans l'arbre, près de la racine, pour garder la cohésion du programme.

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

Ce problème ne se présente pas en **Rewrite** car l'arbre est reconstruit à chaque activation, sans les instructions inutiles; c'est le principe même de réécriture d'un programme en un autre.

– **Replace** (et aussi **Rewrite**) implémente l'attente active d'un événement. Autrement dit, **Replace** parcourt toutes les branches de l'arbre qui contiennent des instructions événementielles même si l'événement attendu n'est jamais généré. Ce parcours est fait au moins une fois même si aucun événement n'est généré. Le pire cas de cette situation est un programme qui contient seulement des instructions événementielles et dans lequel les événements attendus ne sont pas générés dans l'instant.

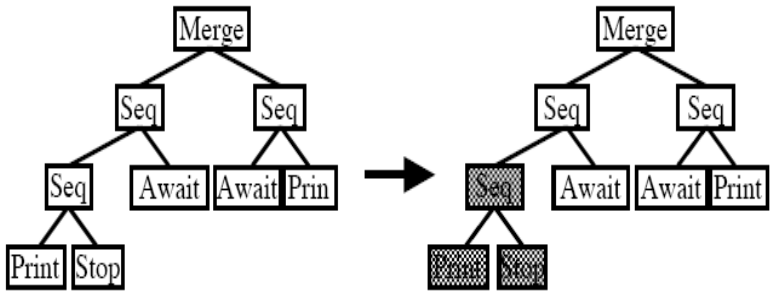


Figure III.15.En Replace,le programme est conservé[jfs].

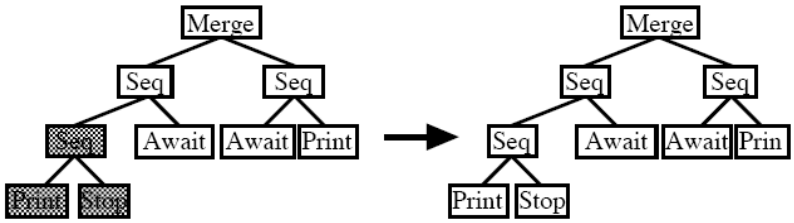


Figure III.16. La fonction *reset* permet de remettre le programme dans son état initial [jfs].

III.8.3.Simple :

Simple est une implémentation particulière de **Junior** qui permet de traiter de manière extrêmement efficace des systèmes constitués d'un très grand nombre de composants parallèles et d'événements. Cet algorithme est développé par *L. Hazard.* (Le lecteur intéressé pourra obtenir plus d'information en contactant **laurent.hazard@francetelecom.com**).

Simple propose de réduire au maximum l'exécution de code inutile afin d'améliorer les performances sur des systèmes de grandes taille.

Les principes de base de cette implémentation sont:

•La machine réactive est chargée avec un programme initial structuré en arbre d'instructions comme avec **Replace** et **Rewrite**.

• La première activation fonctionne sur les mêmes principes que **Replace**.

• Les composants parallèles au cours de leur exécution sont amenés à se suspendre en attente d'événements qui ne sont pas encore connus dans le système. Si tel est le cas, **Simple** crée des files d'attente associées à ces événements dans lesquelles les instructions, bloquées en attente, s'enregistrent.

• Les composants parallèles peuvent aussi stopper leur exécution pour l'instant courant afin de poursuivre leur exécution à l'instant suivant. **Simple** gère là aussi une file d'attente particulière contenant l'ensemble des instructions devant être réactivées à l'instant suivant.

• Ainsi, à l'issue de cette première phase, il ne reste plus que des instructions en attente, enregistrées dans des files. **Simple** débloque alors les différentes files d'attentes en fonction des nouvelles informations à sa disposition:

(i) générations de nouveaux événements,

(ii) fin d'instant permettant de décider l'absence d'événements;

(iii) début d'un nouvel instant.

• L'exécution d'un composant est donc reprise uniquement si l'information qui l'intéresse (présence ou absence d'un ou plusieurs événements, début d'un nouvel instant) est valide. Il n'y a donc plus d'activation inutile de branches comme dans les mécanismes d'exécution descendante.

• Lorsque l'exécution reprend à partir d'une feuille, l'activation des instructions remonte l'arbre syntaxique du programme. Ainsi, une instruction qui termine remonte à son instruction parente présente dans l'arbre de programme. Ce mécanisme d'exécution remontante ne peut plus être décrit formellement par de règles structurelles, la correction de ce type d'implémentation est donc difficile à déterminer.

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.

III.8.4.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.

• 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**, 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'´evénement 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'´etat des instructions **Par** bloquant l'exécution de l'instruction en passant le statut de **WAIT** à **SUSP**.

La première modification par rapport à **Replace** 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 fait 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 connait 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**.

Structure d'une instruction **Storm** :

```
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);
...
}
```

III.8.5.Glouton [Raul.A]:

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

Les différences principales par rapport à **Simple** sont les suivantes:

1. une notion de groupe est utilisée pour :
   - implémenter la séquence.
   - implémenter les boucles.
   - gérer la dépendance d'un programme réactif sur un événement.
2. une sémantique différente de l'instruction de préemption. La sémantique de l'instruction de préemption est celle de l'instruction **Kill** du langage **SL**, c'est-à-dire une instruction qui ne finit pas instantanément en cas de préemption effective.

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

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

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

Ce programme **Junior**, indépendamment de l'implémentation, s'exécute en deux instants :

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

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

Rewrite/Replace :

instant 1 : 1, 2, 4, 1, 2

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

Simple/Glouton :

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

instant 2 : 5, 2, 3, 1

Storm :

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

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

Remarquez que lorsque l'implémentation repasse plusieurs fois sur la même instruction cela signifie qu'elle fait de l'attente active. C'est le cas en **Rewrite/Replace** au cours du même instant, par exemple pour l'instruction1. C'est le cas de **Storm** entre les deux instants. Par contre en **Simple/Glouton** il n'y a jamais d'attente active.

III.8.6.Conclusion :

Le système de règles **Rewrite** constitue une référence pour toutes les implémentations de l'*Approche Réactive* au-dessus de **Java** conformes à l'API **Junior**. Plusieurs algorithmes d'exécution différents ont été expérimentés. Ces différentes implémentations peuvent être comparées à l'implémentation directement déduite des règles de **Rewrite**. Les travaux de formalisation ont été poussés plus loin de manière à rendre compte par des modifications progressives de la sémantique de référence des optimisations mises en œuvre dans des implémentations plus efficaces. Ainsi, un système de règles a été défini dans le cadre de l'implémentation **Replace**, afin de décrire une optimisation consistant à remplacer le mécanisme de réécriture de programme par un mécanisme de modification de l'état des primitives sans jamais altérer la structure de l'arbre syntaxique.

Dans le même esprit, l'algorithme **Storm** propose une optimisation de l'exécution des programmes réactifs dérivée de l'algorithme **Replace**. Cet algorithme utilise un mécanisme de remontée d'information le long de l'arbre de programme, appelé précurseur, afin de limiter le nombre d'activations inutiles à chaque micro-étape. Une formalisation de cet algorithme en règles structurelles a été donnée avec deux implémentations très voisines (la première étant une implémentation directe des règles, l'autre utilisant un mécanisme de files d'attente). Les implémentations de **Storm** conformes à **Junior** sont directement dérivées de **Replace**, ce dont on peut se rendre compte en comparant leur nombre de lignes de code.

algorithme	nb de lignes de code
REPLACE	1400
Formal Storm	1450
Storm	1600

Tableau III.1 : Comparaison entre Replace et les implémentations de Storm [jfs].

Critiques :

- d'après le tableau, il s'avère que **Storm** est plus couteuse en termes d'espace mémoire que **Replace**.

- **Storm** est caractérisée aussi par l'apport d'un nouveau mécanisme qui est le précurseur qui requiert quelques lignes du code pour l'implémenter.

Jusqu'ici, on a vu les différentes implémentations de l'approche réactive au dessous du langage JAVA, dans ce qui suit, on va essayer de choisir le meilleur algorithme qu'on va implémenter après des élagages dans le langage **JavaCard**. On appellera cette implémentation

JCR(**J**ava**C**ard **R**eactive).

III.9.Implémentation de l'approche réactive au dessous de JavaCard :

Puisque, ils existent plusieurs algorithmes d'implémentation de l'approche réactive. Une question se pose : qui est le meilleur ou dans quelles conditions un algorithme est meilleur que les autres.

On a décrit auparavant ces différentes implémentations, leurs fonctionnements ainsi que leurs avantages et leurs inconvénients, on cherche à adapter l'un de ces algorithmes, de sorte qu'il soit implémentable sous une plateforme **JavaCard**.

L'allocation mémoire est une opération extrêmement chère dans les systèmes à ressources limités. En effet dans le cas d'une **JavaCard**, la création d'objets est l'une des opérations les plus couteuses. Parmi les implémentations qu'on a décrit, on a vu que **Replace** permet une gestion efficace de la mémoire contrairement à **Rewrite**. Quant à **Simple** et **Storm** sont des implémentations dérivées de **Replace** avec apport de nouveaux mécanismes tels que le précurseur et les files d'attente et ce pour la gestion des systèmes réactifs de grande envergure. Et comme la plateforme **JavaCard** est limitée en matière de ressources, on a opté pour l'implémentation de **Replace**.

On va présenter dans ce qui suit un nouveau noyau réactif basé sur **Replace** et adapté à la plateforme **JavaCard**(**JCR)** en éliminant toutes les instructions moins utiles ou très difficile à implémenter, et ce en procédant à des élagages.

1. **Le paquetage IO :**

Ce paquetage peut être éliminé car il ne contient que l'instruction **Print** qui n'a aucun sens dans **JavaCard**, puisque il concerne l'affichage d'objets à l'écran.

```
package junior.kernel;
import junior.*;

public class Print implements Action
{
  public String msg;
  public Print(String msg){ this.msg = msg; }
  public String toString(){ return PrintInstruction.Print(msg); }
  public void execute(Environment env){ System.out.print(msg); }
```

}

L'instruction Print.

2. **L'opérateur de parallélisme Merge au lieu de Par :**

Junior introduit l'opérateur non déterministe **Par** afin de prendre en compte une variété plus grande de parallélisme. L'opérateur **Merge** se présente alors comme descendant de l'opérateur **Par**, dans lequel le choix non déterministe de l'ordre d'exécution des branches est remplacé par un ordre déterministe figé ainsi nous obtenons un code optimisé de l'opérateur **Merge** contrairement à l'opérateur **Par.**

3. **L'instruction Freezable :**

Vu que le mécanisme du gel d'instruction n'est pas considéré, car son implémentation est très couteuse en espace mémoire, L'instruction Freezable sera supprimé.

4. **Il n'y aura pas de sérialisation d'objets :**

La sérialisation d'objets est le processus de stockage complet d'objet sur disque ou sur tout autre système de stockage, d'où il pourra être restauré à tout moment .Le processus inverse de la restauration est connu sous le nom de la désérialisation. Un objet sérialisé est dit **persistant**. Par contraste, la plupart des objets en mémoire sont **transcients**, ce qui signifie qu'ils s'en vont quand leurs références sortent de la portée ou quand l'ordinateur est éteint, les objets quant à eux, existent tant qu'il reste une copie stockée quelque part sur disque.

Les instructions réactives dans **JavaCard** n'implémentent pas l'interface **Serializable** car, d'une part cette interface n'est pas supportée et d'autre part la persistance d'objets est garantie par la mémoire **EEPROM**.

5. **Pas de clonage d'objets :**

Les instructions réactives n'implémenteront pas l'interface **Cloneable**, car le clonage d'objets n'est pas supporté par **JavaCard**.

6. **La méthode toString () :**

Elle est implémentée presque par tout les instructions réactives de **Replace**. Elle est utilisée comme un mécanisme de débogage, elle sera supprimée car les systèmes embarqués sont toujours dotés des simulateurs pour éviter des éventuelles erreurs qui peuvent survenir.

7. **Suppression de l'ajout dynamique des composants parallèles :**

Cela pourra s'effectuer en affectant directement l'instruction représentant le programme réactif qu'on veut exécuter au champ Program de la classe **Machine**, car l'ajout dynamique s'effectue en exécutant en parallèle l'instruction que l'on définie comme programme réactif avec l'instruction initialement affectée à ce champ qui est **Nothing**, et cela s'écrit en java comme suit :**Merge**(**Instruction**,**Nothing**).

Cela pourra nous conduire à supprimer la méthode **add ()** de la classe **Machine** en affectant le référant de la racine du programme au champ **Program** de la classe **Machine.**

Les travaux d'optimisation qui ont été effectués sur les **SugarCubes** dans la plateforme **J2ME** ont permis de réduire la taille du fichier **.jar** ou sont stockées les classes **SugarCubes** de 450KB à 95 KB, dans notre cas on propose plus d'optimisations.

III.9.1.l'environnement de développement :

Afin de pouvoir développer des applications **JavaCard** et des applications cliente, on a besoin d'installer un environnement de développement. Les outils utilisés sont :

Outils matériels :

- Un lecteur de cartes **JavaCard SCM Micosystems Inc. CHIPDRIVE Serial 0**.
- Une carte à puce **GEMXPLORE 3G .Univeral-SIM(GSM-CDMA/CDMA).**

Outils logiciels :

- Télécharger et installer **JDK** version **1.6**(ou plus).
- L'environnement de développement **Eclipse** version **3.2.2**.
- Le plugin **Java Card Development Kit 2.2.2**.for windows pour Eclipse.
- Un pilote d'installation **chipdrive driver 3.1** pour le lecteur de cartes.
- Télécharger l'outil de développement de **JavaCard JSR268TK.zip**.

Le modèle de communication de la JavaCard :

La **JavaCard** est un module de traitement réactif à l'environnement (**serveur**). C'est une application externe qui va appeler les applets présentes dans la carte (**client**). Cette application va envoyer des requêtes à la carte pour exécuter les applets qui lui sont nécessaires et cette dernière lui renvoie à son tour des réponses.

Les applets situés sur une **JavaCard** communiquent avec l'application cliente grâce aux échanges de requêtes et de réponse, les **APDUs**.

L'application envoie une commande APDU via le lecteur (**CAD**). Le JCRE en recevant une commande, sélectionne une nouvelle applet ou bien passe la commande à une applet courante (déjà sélectionnée). L'applet courante traite la commande et retourne une réponse **APDU** à l'application. Les commandes et réponses **APDU** sont échangées alternativement par la carte et le **CAD**.

III.9.2.Implémentation :

Dans ce travail on a choisi de simplifier un ensemble de règles de **Replace** afin de les implémenter sous la plateforme **JavaCard**, du fait que **Replace** est la plus optimisée des autres implémentations, elle se base sur **Rewrite** et élimine la création inutile d'objets.

En se basant sur les concepts de programmation sous la plateforme **JavaCard**, on va programmer des instructions réactives tout en éliminant les chaînes de caractères, les entiers, les réels, la sérialisation d'objets,…, etc.

III.9.2.1.Les instructions de base :

Une instruction réactive définit un comportement réactif codé par un algorithme qui utilise trois valeurs de retour après exécution :

- **TERM :** signifie que l'instruction est totalement terminée. Il n'ya rien à exécuter à l'instant présent ni aux instants suivants.
- **SUSP :** signifie que l'instruction est suspendue et elle n'a pas atteint un état stable, en conséquence son exécution doit être poursuivie dans l'instant présent. c'est le cas d'une attente d'un événement.
- **STOP :** signifie que l'instruction est terminée pour l'instant présent, mais qu'il reste du code à exécuter à l'instant suivant.

L'utilisation de ces trois états est utilisée par des instructions en implémentant l'interface **Flags** définie ainsi :

```
public interface Flags {

byte STOP = 0, TERM = 1, SUSP = 2;}
```

Les instructions de base inspirées de **Replace** sont les suivantes :

- **L'instruction Nothing :**

L'instruction **Nothing** ne fait rien et termine instantanément, elle ne nécessite pas d'état à mémoriser, on reste dans le même environnement et elle retourne **TERM**.

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

Son implémentation est la suivante :

```
 public class Nothing extends Instruction
{
  public void reset(){}
  public byte rewrite(){ return TERM; }
}
```

– **L'instruction Stop :**

Cette instruction arrête l'exécution de la séquence dans laquelle elle se trouve pour l'instant courant.

Si l'instruction ***Stop*** n'a pas été exécutée, le booléen indiquant sa terminaison est à *false*. Son activation retourne ***STOP*** et, comme l'instruction termine, la valeur booléenne de terminaison est mise à ***true***:

$$\boldsymbol{Stop(false), E \xrightarrow{STOP} Stop(true), E}$$

A la prochaine activation, elle retournera TERM et permettra ainsi de passer immédiatement en séquence avec le reste du programme.

$$\boldsymbol{Stop(true), E \xrightarrow{TERM} Stop(true), E}$$

L'implémentation de cette instruction est la suivante :

```
public class Stop extends Instruction
{
  public boolean terminated = false;

  public void reset(){ terminated = false; }

  public byte rewrite(){
     if (terminated) return TERM;
     else terminated = true;
     return STOP;
  }
```

– **L'instruction Seq :**

C'est une instruction qui met en séquence deux instructions ***left*** et ***right***. Lorsque ***left*** termine on passe immédiatement en séquence au terme ***right.***

Un booléen permet de distinguer deux états de l'opérateur:

• ***false*** indique que le premier terme n'est pas terminé et doit donc être exécuté;

• ***true*** indique que le premier terme est terminé et que l'on doit passer à l'exécution du second.

Si le premier terme ne termine pas on a:

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

Si la première branche termine, on passe immédiatement à la seconde branche en réécrivant l'état interne à ***true***:

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

Si l'état de l'instruction a déjà mémorisé la terminaison du premier terme, on exécute la règle :

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

Voici l'implémentation de cette instruction :

```
 public class Seq extends BinaryInstruction
{
  public boolean leftTerminated = false;

  public Seq(Program left,Program right){ super(left,right); }

  public void reset(){ super.reset(); leftTerminated = false; }

   public byte rewrite(){

      if (leftTerminated) return right.rewrite();
      byte s = left.rewrite();
      if (s == TERM){ leftTerminated = true; return right.rewrite(); }
      return s;
  }


}
```

- **L'instruction Loop :**

Les boucles utilisent la méthode **reset ()** d'une instruction pour préparer une nouvelle itération de leur corps:

Si le corps de la boucle ne termine pas immédiatement elle réexécute son corps.

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

Lorsque le corps de la boucle termine immédiatement alors la boucle est immédiatement réactivée pour la même micro-étape:

$$\frac{t, E \xrightarrow{TERM} t', E' \quad Loop(reset(t')), E' \xrightarrow{\alpha} t'', E''}{Loop\ (t), E \xrightarrow{\alpha} t'', E''}$$

L'instruction est implémentée ainsi :

```
public class Loop extends UnaryInstruction
{
  public Loop(Program body){ super(body); }

  public byte rewrite(){
     byte s = body.rewrite();
     if (s != TERM) return s;
     body.reset();
     return rewrite();
  }
}
```

– **L'instruction Merge :**

Il étend la classe **BinaryInstruction** et implémente un parallélisme basique: à chaque instant, l'instruction **left** puis l'instruction **right** sont activées, toujours dans cet ordre. La fin d'un instant ne survient que lorsque les deux branches ont stoppé leur exécution pour l'instant courant. **Merge** termine définitivement lorsque les deux branches **left** et **right** ont terminé définitivement.

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

La classe **Merge** est définie ainsi :

```
public class Merge extends BinaryInstruction
{
  public byte leftFlag = SUSP, rightFlag = SUSP;

  public Merge(Program left,Program right){ super(left,right); }
  public void reset(){ super.reset(); leftFlag = rightFlag = SUSP; }

 public byte result(){
     byte b = SUSP;
     if(leftFlag!=SUSP && rightFlag!=SUSP){
        b = (leftFlag==TERM && rightFlag==TERM) ? TERM : STOP;
        if (leftFlag==STOP) leftFlag = SUSP;
        if (rightFlag==STOP) rightFlag = SUSP;
     }
     return b;
  }

  public byte rewrite(){
      if (leftFlag == SUSP && rightFlag != SUSP) leftFlag = left.rewrite();
      else if (rightFlag == SUSP && leftFlag != SUSP) rightFlag =
right.rewrite();
      else{
          leftFlag = left.rewrite();
              rightFlag = right.rewrite();
      }
      return result();
  }


}
```

III.9.2.2.La machine réactive :

La machine réactive est la partie centrale d'un système réactif, elle contient un programme réactif qui exécute les instructions réactifs, elle contient un environnement d'exécution, où la machine diffuse dans l'environnement les événements générés en dehors du programme.

La machine réactive gère l'exécution des instructions, déclare les fins des instants et réinitialise l'environnement.

Une machine finit l'instant courant lorsque toutes les instructions parallèles du programme qu'elle contient sont soit totalement terminées soit stoppées. Sinon, la machine active cycliquement le programme tant qu'il reste des instructions suspendues.

Son code d'implémentation est :

```
public class MachineImpl implements Flags, Machine
{
   public Instant instant;
   public InternEnvironment env;
   public Program toAdd = new Nothing();
   public boolean somethingToAdd = false;
   public boolean terminated = false;
// constructor
public MachineImpl(Program program){
      buildEnvironment();
      instant = new Instant(program);
      instant.bind(env);
   }
   // building the environment
   public void buildEnvironment(){ env = new EnvironmentImpl(); }
   // dynamic add method for next instant
   public void add(Program inst){
      toAdd = new Par(toAdd,inst);
      somethingToAdd = true;
   }
   protected void performAddings(){
      if (somethingToAdd == false) return;
      instant.body = (Instruction)new Par(toAdd,instant.body);
      instant.bind(env);
      toAdd = new Nothing();
      somethingToAdd = false;
      terminated = false;
   }
   // react method
   public boolean react(){
      performAddings();
      if (terminated == false){
         byte res = instant.rewrite();
         env.newInstant();
         terminated = (TERM == res);
      }
      return terminated;
   }
}
```

III.9.2.3.Les événements :

L'approche réactive définit la notion d'événements diffusés instantanément à tous les comportements réactifs comme moyen de communication.

Une configuration événementielle est le moyen avec lequel on peut décider si un événement est présent ou absent dans un instant. Cette dualité dans les valeurs prises par un événement à chaque instant, permet de définir les opérateurs de la logique booléenne classiques qui utilisent, au lieu de l'égalité des valeurs, la présence ou l'absence d'un événement. Cette logique booléenne basée sur la notion d'événement.

– **L'instruction Atom(Action a) :**

En **Replace**, l'approche réactive considère que toute intervention élémentaire sur l'environnement d'exécution se fasse de façon atomique afin de ne pas être perturbée par d'autres actions qui se déroulent en parallèle sur le système.

Une action atomique étend la classe abstraite **Atom** et définit l'opération à exécuter dans la méthode **action**.

La classe **Atom** étend la classe **Instruction**. L'orsqu'elle est activée, elle exécute la méthode **action** définie par l'action atomique et termine immédiatement en retournant la valeur **TERM** :

Si l'action **a** est terminée on a :

$$\boldsymbol{Atom(a, false), E \xrightarrow{TERM} Atom(true, a), E}$$

Sinon :

$$\boldsymbol{Atom(a, true), E \xrightarrow{TERM} Atom(true, a), E}$$

Son code en **JavaCard** est :

```
public class Atom extends Instruction
{
  final public Action action;
  public boolean terminated = false;

  public Atom(Action action){ this.action = action; }
  public void reset(){ terminated = false; }

public boolean equals(Instruction inst){
     return super.equals(inst) && action == ((Atom)inst).action
                               && terminated == ((Atom)inst).terminated;
  }

  public byte rewrite(){
     if (terminated == false){ action.execute(env); terminated = true; }
     return TERM;
  }
}
```

III.9.2.4.Exemple :

Mon travail consiste en la simplification de **Replace**, afin de construire un noyau de primitives réactives qu'on chargera dans une carte à puce, et de lui faire un appel à partir d'une action java extérieure qui fait appeler et fait réagir la machine réactive.

On va présenter un exemple d'implémentation de **Replace** sous une plateforme **JavaCard** et de l'exécuter.

Les étapes suivies pour le développement sont :

1. La création d'un projet **JavaCard**, dans lequel on va :

- Coder un ensemble de base d'instructions réactives.
- Déclarer une machine réactive (statique) qui contient initialement un ensemble de primitives réactives.
- Mettre le projet dans une librairie **JavaCard**, afin d'obtenir un fichier **.jar**

2. La création d'un projet Java.

- Importer la JSR-268.
- Préciser le nom du lecteur dans ManagerConfig.xml.
- Importer les classes **JavaCard**, ainsi que le.cap.
- Construire le programme client.
- Interroger les instructions réactives à partir de la classe client.

L'outil JSR-268, a été développé afin de faciliter les travaux dans la plateforme **JavaCard** Après importation de la JSR-268, on aura la vue suivante :

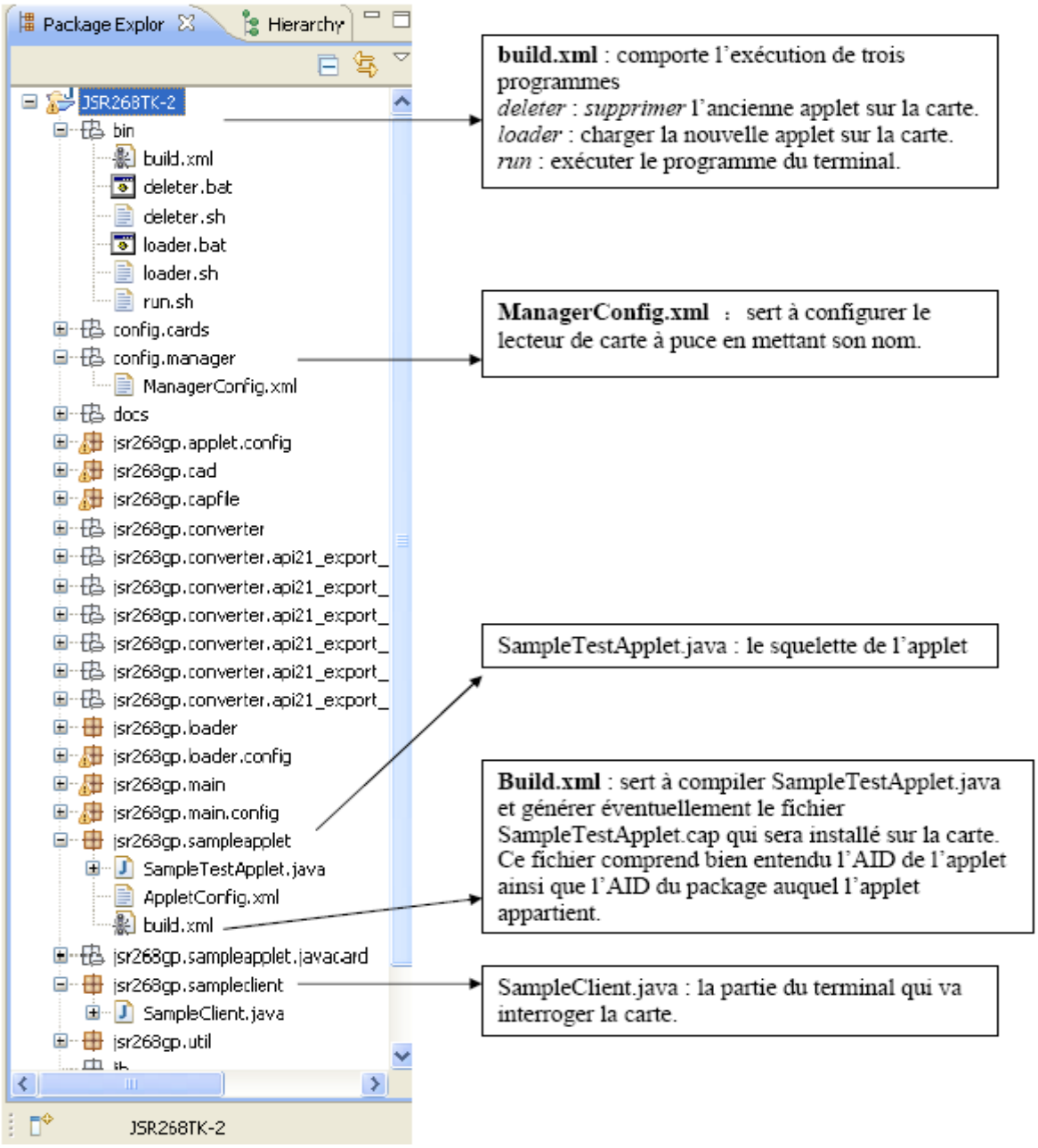


Figure III.17.Vue Eclipse après importation de la JSR-268.

Exemple :

A partir du package **jsr268gp.sample.client**, on va créer une classe (**SampleClient**) qui va faire réagir la machine réactive statique, en invoquant l'Applet (**SampleTestApplet**) qui se trouve sur la **JavaCard.**

Ce programme va exécuter en parallèle (**Merge**) deux instructions réactive binaire **Merge** et **Seq**. Elles mêmes composées respectivement de deux instructions de **Seq** composées de deux actions atomiques (**Atom**) et de deux actions atomiques (**Atom**).

Voici l'arbre du programme :

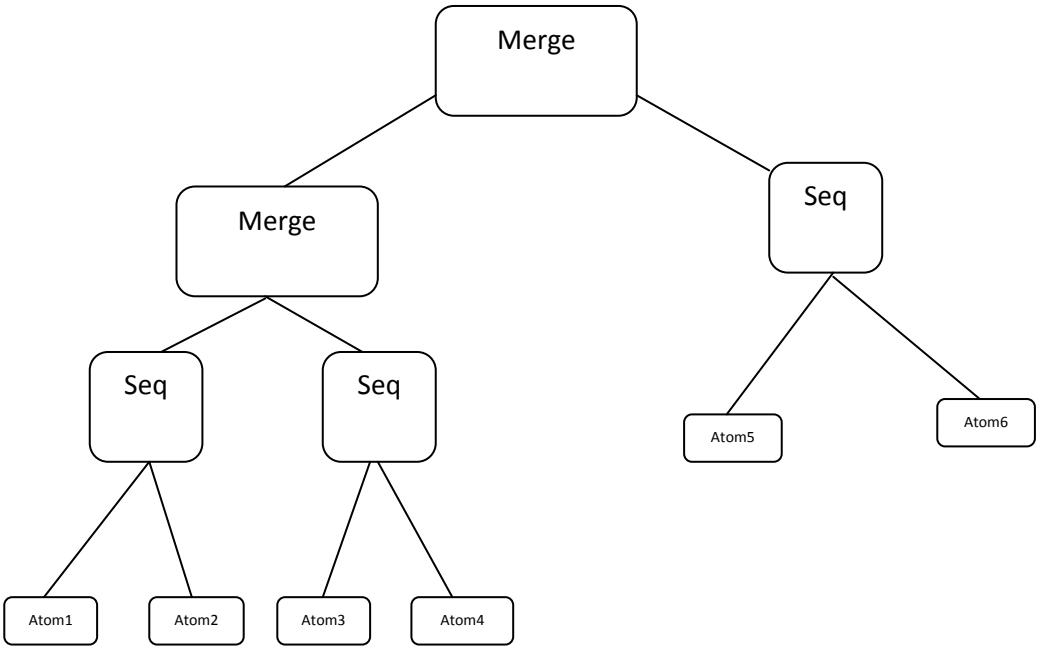


Figure III.18.Exemple d'un programme réactif

Tels que les comportements des actions atomiques de Atom1 à Atom6 sont définis comme suit :

Atom1 : enregistre la valeur initiale d'un compteur.

```
new Atom(new Action ( ) {
                public void execute (Environment  env ){
                 buffer[0]=(byte) compteur;
               }})
```

Atom2 : incrémente et affiche la valeur du compteur.

```
new Atom(new Action ( ) {
                public void execute (Environment  env ){
                compteur=(byte) (compteur+1);
                buffer[1]=(byte) compteur ; }})
```

Atom3 : décrémente le compteur.

```
new Atom(new Action ( ) {
                public void execute (Environment  env ){
               compteur--;
               }})
```

Atom4 : affiche la nouvelle valeur du compteur.

```
new Atom(new Action ( ) {
                public void execute (Environment  env ){
               buffer[2]=(byte) compteur; }}))
```

Atom5 : initialise la valeur du compteur.

```
new Atom(new Action ( ) {
                public void execute (Environment  env ){
               compteur=(byte)0;
               }})
```

Atom6 : affiche la nouvelle valeur du compteur.

```
new Atom(new Action ( ) {
                public void execute (Environment  env ){
               buffer[3]=(byte) compteur; }})
```

Le résultat qui s'affiche est le suivant :

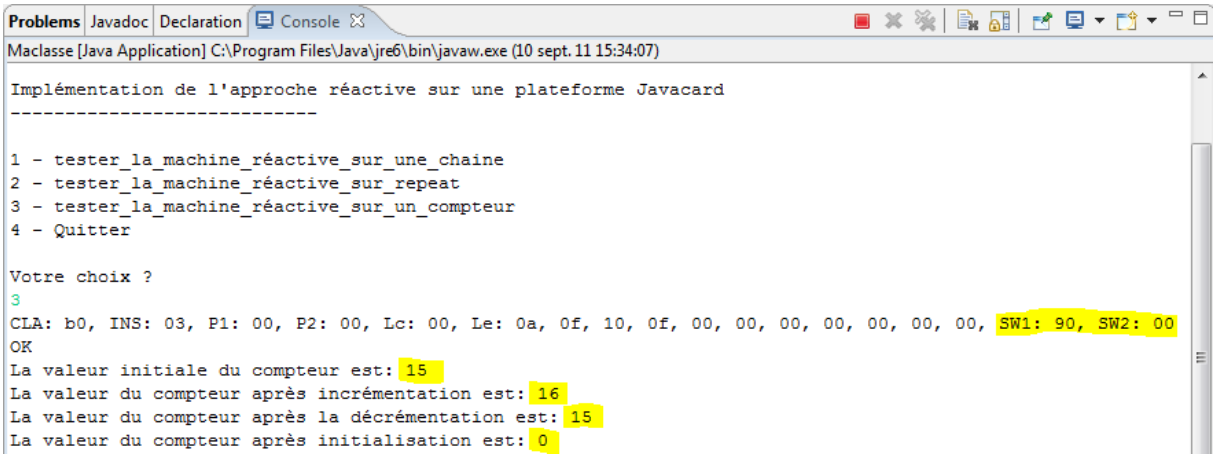


Figure III.19. Résultat du programme réactif.

III.9.2.5.Conclusion :

En se basant sur **junior** et ses différentes implémentations de l'approche réactive au dessous de Java, j'ai pu développer un noyau réactif de taille de 20KO destiné à la plateforme **JavaCard**, et ce en respectant une sémantique bien précise défini à travers des règles de réécriture.

Conclusion générale

Conclusion générale :

En se basant sur les différents travaux de recherche sur l'approche réactive au dessous de JAVA, mon travail consistait à décrire et à implémenter l'approche réactive sur une plateforme JAVACARD.

On a décrit principalement deux formalismes de l'approche réactive, notamment les **SugarCubes** et **Junior** qui permettent de construire sur une base formelle des systèmes réactifs dynamiques adaptés à la programmation d'applications interactives, concurrentes, distribuées et pouvant s'exécuter sur des environnements hétérogènes ,ainsi on a expliciter quelques implémentations qui sont le résultats de ces travaux de recherche.

Certains de ces implémentations cherchaient à améliorer les capacités de traitement des systèmes réactifs créés, en gérant un grand nombre de composants parallèles qui communiquent entre eux en générant un nombre important d'événements.

Afin de bien mener mon travail, on a tenu à présenter en premier lieu les **SugarsCubes** , ensuite on a expliciter le formalisme **junior** ainsi que ses descendant à savoir :**REWRITE**, **REPLACE**, **Simple**, **Storm** et **Glouton**, et après analyse on a soulevé des problèmes liées à une implémentation directe de **REWRITE** sur notre plateforme , on a jugé que **REPLACE** sera la solution de fait de sa gestion efficace de la mémoire. Concernant **Simple**, **Storm**, et **Glouton** elles sont en effet des extensions de Junior.

En tenant compte des contraintes liées à la plateforme JAVACARD (qui est un sous ensemble de Java), et la quantité mémoire disponible, on a dégagé un noyau de primitives caractéristiques de l'approche réactive inspiré de **REPLACE**, et une machine réactive qui fait réagir ce noyau dans le but de maintenir la réactivité entre les instructions de ce dernier se trouvant sur une carte et qu'on pourra interroger à partir d'un client JAVACARD.

L'introduction de la programmation concurrente dans les plateformes à mémoire limitée en se basant sur l'approche réactive au dessous du Java est très intéressente, et cela pour une meilleure performance et aussi pour pouvoir introduire d'autres concepts, de nouvelles options dans des domaines diverses et généraliser l'utilisation des cartes à puces.

API :(**A**pplicataion **P**rogramming **I**interface), interface de programmation d'applications, c'est une couche logicielle au travers de laquelle une application peut accéder d'une façon standardisée aux ressources d'un système d'exploitation, et ce sans avoir besoin de connaitre le fonctionnement.

APDU :(**A**pplication **P**rotocole **D**ata **U**nit),c'est un ensemble de règles qui définit le format des données ,des commandes et des réponses échangées entre une carte à puce et un lecteur pour exécuter des services de bas niveau offert par la carte (celui du système d'exploitation).son format fait l'objet du standard ISO 7816-4.

Applet : c'est une mini application qui ne peut tourner qu'à l'intérieur d'un navigateur.

Bytecode : dans un environnement interprété, le code java n'est conçu pour aucune plateforme spécifique, mais une sorte du code intermédiaire qui doit être traduit en code natif machine par un programme particulier faisant office de compilateur. le code intermédiaire ou bytecode est conçu pour une machine virtuelle ou JVM pour **J**ava **V**irtual **M**achine.

CAD :**C**ard **A**cceptence **D**evice ,Lecteur de carte à puce.

CAP :(Converted **Ap**plet),Le fichier .CAP contient une représentation binaire de toute les classes exécutable de toute les classes d'un package java .Le format, CAP est la format dans lequel les logiciels sont chargés sur La JavaCard.

Classe : Notion de base de la programmation orientée objet, il s'agit du modèle générique d'objets (composante modulaire d'un programme) qui ne diffèrent que par la valeurs de leurs attributs. En java ,qui est un langage objet ,il n'est possible de créer une application qu'à partir des classes.

EEPROM :(**E**lectrical **E**rasable **P**rogrammable **R**ead **O**nly **M**emory), comme la ROM, elle peut conserver les données quant la mémoire n'est plus alimentée.La différence est que l'on peut modifier Les données stockées dans la mémoire.

GSM :(**G**lobal **S**ystem for **M**obile telecommunications),sytème global de communications mobiles.

ISO :(**I**nternational **S**tandards **O**rganization), ensembles de normes décrivant les caractéristiques cartes à puces avec contact.

JCF :(**J**ava**C**ard **F**orum), groupement de professionnels qui est chargé de développer les spécifications de la technologie JavaCard et donner des recommandations à SUN Microsystems, propriétaire des spécifications API de JavaCard. Parmi les attributions de JFC est de promouvoir la technologie JavaCard en tant qu'un standard industriel.

JCRE : **J**ava**C**ard **R**outine **E**nvironnment, l'environnement d'exécution JavaCard (JCRE) gère les ressources de la carte, la communication avec le réseau ,l'exécution des applets et leur sécurité.En fait le système d'exploitation des SmartCartds.

JSR :(**J**ava **S**pecification **R**equest) : Logiciel représentant l'interface entre la carte à puce et le lecteur.

JCVM :(**J**avaCard **V**irtual **M**achine) :c'est la machine virtuelle définit pour les JavaCards.

JVM : **J**ava **V**irtual **M**achine, La JVM est l'équivalent d'un ordinateur virtuel résidant entièrement en mémoire .Cette JVM doit être implémenté sur chaque plateforme. Elle fournit un niveau d'abstraction et d'indépendance entre le programme Java compilé est la plateforme matérielle.

Objet : Instance de classe crée à partir d'une classe lorsque le programme est lancée ,avec ses propres valeurs d'attributs.Tous les objets d'une classe ont en commun leurs méthodes (code exécutable des opérations),leurs attributs(variables),leurs informations de débogage ,leurs propriétés d'héritage et leurs droits d'accès.

ROM :**R**ead **O**nly **M**emory ,mémoire utilisée pour stocker les programmes fixes de la cartes. On ne peut ainsi ni les modifier ni les supprimer .aucune alimentation n'est nécessaire pour conserver les données.

Le standard ISO 7816 a été publié par ISO (organisation internationale de standardisation) en 1987.

C'est le plus important standard définissant les caractéristiques des cartes à puce qui fonctionnent avec un contact électrique, il s'agit de l'ensemble des normes suivantes :

ISO7816-1 : Cette normes définit les caractéristiques physiques des cartes à puce à contact (la géométrie, la résistance, les contacts, …etc.).

ISO7816-2 : Cette norme spécifie le dimensionnement, les fonctions, et placements des contacts sur la puce. Ces paramètres concernant : l'alimentation en énergie, le contact de masse, l'horloge, la mise à zéro, et le port d'entrée/sortie.

ISO7816-3 : Elle définit les protocoles de transmission (TPDU : Transmission Protocol Data Unit) :

- T=0 : Protocole orienté octet.
- T=1 : Protocole orienté paquet.
- T=14 : réservé pour les protocoles propriétaires.

- La sélection d'un type de protocole.
- La réponse à un reset (ATR).
- Les signaux électriques, tels que le voltage, la fréquence de l'horloge et la vitesse de communication.

ISO7816-4 : Cette norme vise à assurer l'interopérabilité des échanges. Elle définit les messages APDU,par lesquels les cartes communiquent avec le lecteur. Les échanges s'effectuent en mode client-serveur, le terminal ayant toujours l'initiative de début de la communication.

Elle spécifie également la structure des fichiers et des données, l'architecture de la sécurité et la sécurisation de la communication.

ISO7816-5 : Elle définit le système de numérotation et les procédures d'enregistrement et d'attribution des identifiants des applications (AID : Application IDentifer).

ISO7816-6 : Elle spécifie des éléments de données inter-industrie pour les échanges, tel que le numéro de porteur de carte, sa photo, sa langue, la date d'expiration…etc.

ISO7816-7 :Définie en 1998 suite aux travaux du RD2P de l'université sciences et des technologies de Lille .Ce standard précise les commandes de langage de requêtes SCQL.

ISO7816-8 : Cette norme définit la sécurité dans les applications inter-industries.

ISO7816-9 : Définie en 2000, elle concerne les commandes et les attributs supplémentaires pour des opérations de sécurité .Elle ajoute les commandes telles que Create File et Delete File, ainsi que les commandes de recherche.

ISO7816-10 : Elle définit les protocoles de communication pour les cartes synchrones(T=14).

ISO7816-11 : Cette norme est destinée à la représentation, l'organisation des données et les vérifications personnelles pour les méthodes biométriques.

ISO7816-12 : Définie en 2005, cette norme présente les spécifications d'interface électrique USB et les procédures de fonctionnement pour les cartes à puce à contact.

ISO7816-13 : Elle spécifie les commandes à utiliser dans un environnement multi-applicatif. Elle reprend les spécifications de Global Platform.

ISO7816-15 : Définit une façon standardisée de stocker les données cryptographiques sur une carte à puce et d'accéder aux clés publique et certificats.

Bibliographie

BIBLIOGRAPHIE :

[Bou] ***Samia Bouzefrane,*** Maître de Conférences **"la technologie java card"** CEDRIC –CNAM, France ; 2009 .

[Bo1] ***F. Boussinot,*** **"La programmation réactive - Application aux systèmes communicants"**,

Collection technique et scientifique des télécommunications, MASSON ; 1996.

[Christiane B]*Christian BRUNETTE*, thèse de doctorat***:*** ***"*****Construction et simulation graphiques de comportements : le modèle des Icobjs*****"*** ; 2004.

[HP] ***D. Harel, A. Pnueli, "*****On the Development of Reactive Systems*****",*** Proc. NATO ASI Series F,

Vol.13 477-498, Springer-Verlag ;1985.

[jfs]*J-F. Susini,* thèse de doctorat: "**L'Approche Réactive au dessus de Java: sémantique et implémentation des SugarCubes et de Junior**" ;2001.

[Kar 08] ***Achraf KARRAY*** , Thèse de doctorat :"**Conception, mise en œuvre et validation d'un environnement logiciel pour le calcul sécurisé sur une grille de cartes à puce de type Java**", université Bordeaux I, Ecole doctorale de mathématiques et d'informatique ; 2008.

[LAN 07-08]*LANNETTE Romain* ,3A ISA SERTR, Projet de Fin d'Etude **:** "**Les JAVA CARD**" ; 2007-2008.

[PlO 81] ***G. Plotkin.*** **"A Structural Approach to Structural Semantics"**. Rapport Technique DAIMI FN-19, Université d'Aarhus, 1981.

[Ran 03] ***Wolfgang Rankl, Wolfgang Effing*** **"Smart Card Handbook"**; third edition, ISBN 0-470-85668-8; 2003.

[Raul A] ***Raul Acosta-Bermejo***, thèse de doctorat:"**Rejo:Langage d'Objets Réactifs et d'Agents**" ;2003.

[Sau 02J] ***Damien SAUVERON*** "**la technologie javacard et sa sécurité** " ; Laboratoire Bordelais de Recherche en Informatique, équipe : Systèmes et Objets Distribuées; juillet 2002.

[Sun 3] ***Sun Microsystems*** **,"Application Programming Interface specification java card platform 2.2.2"**; Inc.www.sun.com.

[Tsa 07] ***Ernest TSASSONG*** **,"Globalisation des résultats dans les mesure de performance sur les plateformes Java Card"** ; rapport de stage, Laboratoire Cédric ; 2007.

[Zhi 02] ***Zhiqun Chen*** **,"Java Card Technology for Smart Cards: Architecture and Programmer's Guide (The Java Series)"**; 1ère édition , Addison-Wesley PubCo; ISBN: 0201703297; 2002.

WEBIOGRAPHIE :

[1]http://cedric.cnam.fr/~bouzefra/cours

[2]http://www.techniques-ingenieur.fr/.