REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR
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'études

En vue de l'obtention du diplôme de Master en informatique
Spécialité : Systèmes Informatiques

Thème

Implémentation des événements de l'approche réactive sur une plateforme JAVACARD

Proposé par :

Mr. M.DEMRI

Réalisé par :

Mr BERREKAIL Sofiane

2010/2011

Remerciement

Louanges et remerciements à DIEU le tout puissant.

Tous mes plus vifs remerciements et ma profonde reconnaissance s'adressent à mon encadreur **<<Mr DEMRI Mohamed>>** *pour m'avoir confié ce sujet, et de m'avoir dirigé tout au long de sa réalisation. Ses compétences, ses critiques et son bon sens mon largement aidés à réaliser ce travail et je lui exprime mes sincères reconnaissances.*

Je tien à remercier les membres du jury à qui m'ont fait l'honneur de juger mon modeste 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 : mes enseignants, mes amis, mes collègues chacun son nom.

Tout l'honneur de ce travail revient à mes parents pour tant de sacrifices tout au long de mon parcours

Merci à tous.

Dédicaces

Louanges et remerciements à DIEU le tout puissant.

Je dédie ce projet de fin d'études à :

Ma mère, mon père pour leurs sacrifices et leurs encouragements sans qui rien ne serait possible.

Mes sœurs et mon frère qui m'ont toujours soutenu et qui ont toujours répondu présent.

Mes amies, en particuliers Akh Takfarinas, aussi je rends un vibrant hommage à mon défunt amie Bessalem Youcef, que dieu ait pitié de son âme et l'accueille dans son immense paradis, Amène.

Chapitre I - Les cartes à puces

Chapitre II–Java Card

Chapitre III

Implèmentaionde l'approche réactive au-dessus de JavaCard

Tables des Figures

Chapitre I

Chapitre II

Chapitre III

Tableaux

Chapitre I

Chapitre II

Chapitre III

Introduction Générale

Introduction Générale

Inventée en 1974 par le français, Roland Moreno, aujourd'hui les cartes à puce font partie de notre vie courante. C'est un support très répondu pour stocker des informations et leur traitement vue quelle dispose d'un microprocesseur. Les plus courantes sont les cartes bancaires, les cartes SIM. 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 composante majeure 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 de 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 la 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 l'indéterminisme de l'exécution des programmes lié a l'ordonnanceur ‘‘sheduler’’ qui est diffèrent d'une implémentation d'une plateforme à l' autre . 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 et une communication entre eux via des événements.

Ses principales caractéristiques sont:

- Fournir un ensemble de primitives permettant l'expression simple et puissante du parallélisme au niveau d'un langage de programmation.
- 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;
- 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 dessus de la plateforme JavaCard, cela concerne en plus des concepts de base, le parallélisme et les événements.

Pour cela, je l'ai structuré comme suit :

Premier chapitre << Les cartes à puces >> où on étalera les différents aspects des cartes à puce ; leurs évolutions, leurs composants, leurs types,…etc. Suivie du deuxième chapitre << Java Card >> on va présenter la technologie JavaCard. Enfin dans le troisième chapitre << L'implémentation de l'approche réactive au-dessus de JavaCard >> où on parlera de l'approche réactive, ses différents formalismes, précisément les objets réactifs et leurs implémentations, puis, on choisira la meilleure implémentation qu'on va simplifier et l'exécuter sous la plateforme JavaCard.

CHAPITRE I

Cartes à puce

Chapitre I

Les cartes à puce

I.1 Introduction

Dès sa création, la carte magnétique a montré ses énormes capacités. Mais, avec l'utilisation répandue de ce type de carte, elle révèle de graves inconvénients de jour en jour. Des données qui sont stockées dans cette carte peuvent facilement être lues et changées illégalement, la carte à puce considérait comme une carte qui puisse surmonter ces inconvénients. C'est est une carte en plastique voir de papier ou de carton de dimension réduite (format de poche) dotée d'au moins un circuit intégré ou "puce" capable de contenir de l'information.

Ce circuit intégré peut se limiter à des circuits de mémoire non volatile ou bien comporter un microprocesseur capable de traiter cette information. Certaines cartes à puce sont dotées de coprocesseur cryptographique fournissant un accélérateur de calcul pour les fonctions cryptographiques.

Cette technologie est principalement utilisée comme moyen d'identification personnelle (dans le cas des badges d'accès, d'une Carte SIM ou de la carte Vitale dans le domaine médical) si la carte est pourvue de fonctionnalités cryptographiques, aussi comme moyen de paiement, notamment dans le cas de la Carte bancaire et du porte-monnaie électronique ou tout autre service prépayés (cartes de décryptage de télévision par satellite, etc…).

1.2 Historiques des cartes à puce [Ser10]

– Dès **1947**, une mémoire portative est décrite par un ingénieur britannique : un substrat en bakélite sur lequel sont imprimées de très fines pistes de cuivre qui, sous l'effet d'un courant important, se volatilisent irréversiblement, créant un effet mémoire. Il est question, à l'époque, de 64 bits.

–En**1951**, Carte plastique Diners Club (Frank McNamara).

–En **1968**, deux inventeurs allemands Jürgen Dethloff et Helmut Gröttrup introduisent en premier un circuit intégré dans une carte plastique.

– En **1970**, le chercheur japonais Kunikuta Arimura de l'Institut de Technologie Arimura dépose un brevet sur la carte à puce.

– En **1971**, Paul Castrucci d' IBM dépose aux USA un brevet intitulé Information Card

– Entre **1974** et **1978**, le français Rolland Moreno dépose 47 brevets en tout sur la carte à puce parmi lesquels on peut citer [197 ; Mor77 ; Mor76 ; Mor78 ; Mor83] ; il est considéré comme étant l'inventeur de la carte à puce telle qu'on la connait aujourd'hui. En effet, il crée : une mémoire portative dotée de moyens inhibiteurs. Ces inhibiteurs (matériels ou logiciels) ont pour but de protéger l'accès à la mémoire aussi bien en lecture qu'en écriture. Sans ceux-ci, l'utilité de la carte est limitée en termes d'application. Si l'on prend l'exemple de la carte de crédit, il est indispensable de protéger la lecture du code confidentiel ou le changement du numéro de compte inscrit à l'intérieur. La carte décrite dans le premier brevet de Rolland Moreno est d'ailleurs couplée à un lecteur par radiofréquence, comme le seront par la suite les cartes sans contact.

– En **1977**, Derthloff dépose un brevet [197] pour une carte à mémoire portative dont les moyens inhibiteurs sont constitués par un microprocesseur. Ce brevet significatif permet le changement des fonctions de la carte par simple reprogrammation. Aujourd'hui la majorité des cartes à puce sont dotées d'un microprocesseur.

– En **1979**, la première carte est créée et assemblée à Toulouse par Motorola pour Bull CP8 avec 1 ko de mémoire programmable et un microprocesseur à base de 6805.

– En **1983**, apparaissent les premières cartes téléphoniques.

– En **1984** le groupe d'intérêt économique carte bancaire adopte le prototype de carte bleue basé sur la technologie Bull CP8. Cela donne naissance à la carte bleue moderne (la version B0').

– Entre **1984** et **1987**, les normes internationales de l'ISO sur la carte à puce avec contact voient le jour sous la référence 7816 [Sta87].

– En **1989**, premières cartes GSM pour téléphones mobiles.

– En **1990**, le laboratoire français RD2P a développé la CAVIMA (Card Virtual Machine). Ce projet, conçue pour la lecture de données des cartes médicales, c'est le premier pas vers les cartes ouvertes.

– En **1991**, les premières cartes SIM (Module d'indentification de souscription)

– En **1992**, un brevet est déposé en France pour la création du premier interpréteur de code pour une carte à puce.

– En **1994**, Europay, MasterCard et Visa publient les spécifications ENV pour le paiement par la carte à puce, devenue un standard assurant la compatibilité mutuel des cartes de crédit.

– **En 1996**, l'implémentation d'une machine virtuelle java (une partie du langage java) pour la carte à puce sur un microprocesseur standard 8-bit par Schlumberger, offrant à l'utilisateur d'ajouter des applets.

– En **1997**, les premières cartes à puce multi-applicatives voient le jour.

– En **1998**, Microsoft crée la machine virtuelle indépendante du langage de programmation.

– Depuis **2006** en France :
Passeport électronique comporte une puce électronique qui stocke les données personnelles du détenteur : (son nom, sa date de naissance, sa nationalité, son numéro de passeport et la photo numérisée du titulaire).

– Depuis le **15 Juin 2009**, le passeport biométrique : sur une puce RFID (Radio Frequency IDentification Tag), qui permet de lire les informations à courte distance, sont enregistrés - outre les informations personnelles classiques et la photo numérisée - deux empreintes digitalisées des doigts du détenteur (à partir de l'âge de 6 ans).

– Même année, une nouvelle machine virtuelle java (sous langage de java) qui permet l'ajout, en plus des applets, les Servlets pour une programmation client/server. Cette CardVM support la couche IP.

1.3 Fonctions de la carte à puce :

Les fonctions assurées par le système constitué de la carte en interaction avec un lecteur, lui-même pouvant être connecté à un serveur, sont centrées sur la sécurité. Elles comprennent :

- des fonctions de service
  - Etre portable.
  - Stocker, modifier et restituer des données.
  - **Identifier** la carte et éventuellement son porteur.
  - **Authentifier** : le lecteur par la carte, la carte par le lecteur, le porteur par le système.
  - Signer des données pour un contrôle ultérieur.

- des fonctions techniques
  - S'interfacer avec le lecteur (interface physique, électrique, magnétique, logicielle).
  - alimenter la puce de la carte et dialoguer avec le lecteur.
  - Stocker, traiter et échanger des données.
  - Résister à l'environnement et à l'utilisation (frottement, contraintes mécaniques et thermiques, champ électrique, décharge électrostatique, champ magnétique, cycles de lecture et d'écriture…).
  - Résister aux attaques malveillantes.

1.4 Typologies de la carte à puce

On peut distinguer quatre principaux types de cartes à puces selon la présence ou non d'un microprocesseur et le type d'interface, avec ou sans contact :

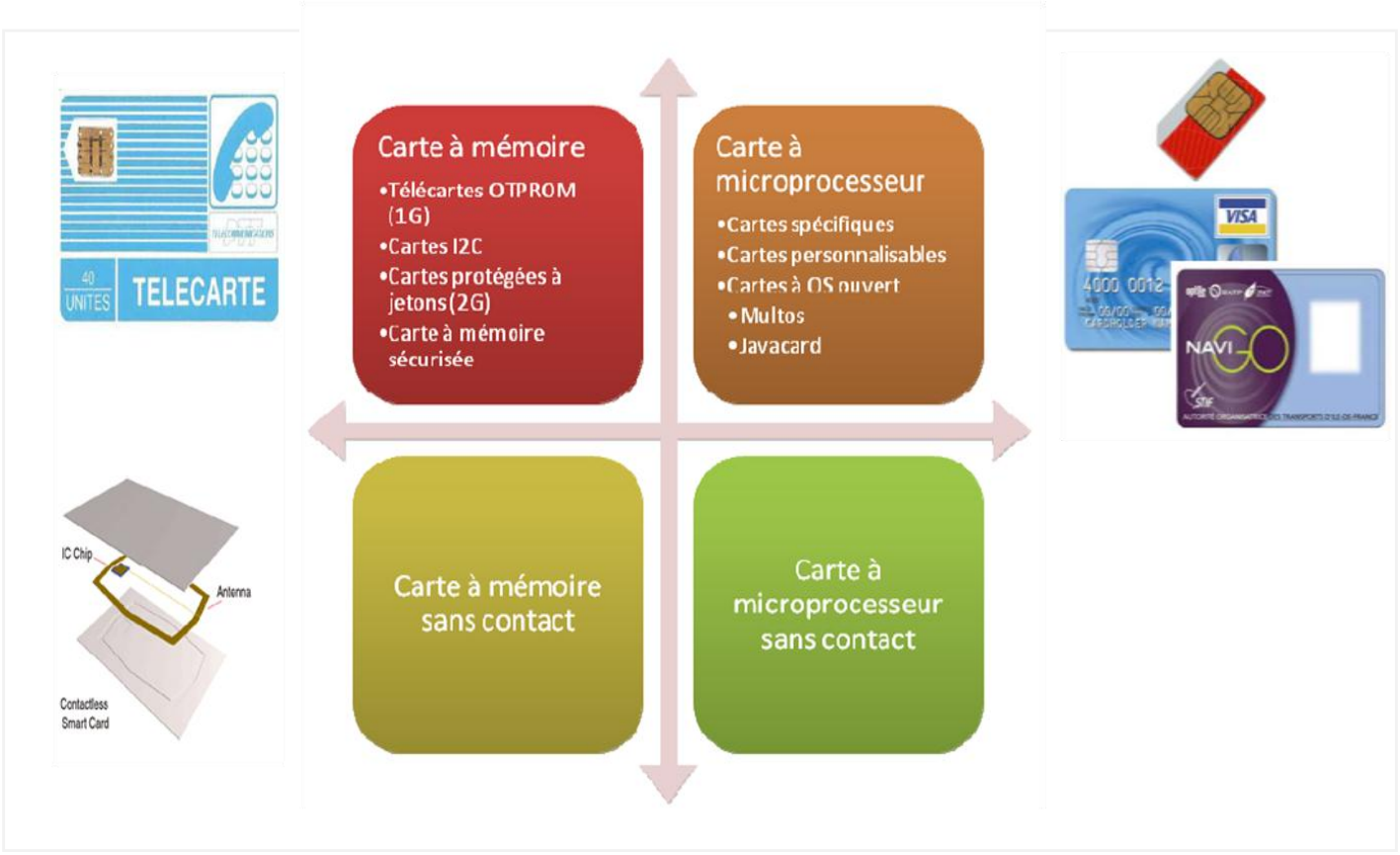

Figure I.1 les différents types de cartes à puce [DEI09].

1.4.2 Cartes à contacts

1.4.2.1 Carte à logique câblée

Cette carte se caractérise par une mémoire accessible via des circuits préprogrammés et figés pour une application particulière, elle peut effectuer des calculs préprogrammés sans CPU, par exemple la gestion de compteurs sous forme de bouliers électroniques, initialisé à une valeur de départ, ils sont décrémentés a chaque usage. Certaines versions supportent une forme limitée d'authentification par clé secrète. Elle est typiquement utilisée pour les cartes téléphoniques.

1.4.2.2 Carte à mémoire

Historiquement les plus anciennes, les cartes à mémoire possèdent un circuit intégré qui ne contient qu'une zone mémoire avec le minimum de logique nécessaire pour la mettre en œuvre intégrant éventuellement un mécanisme de protection.

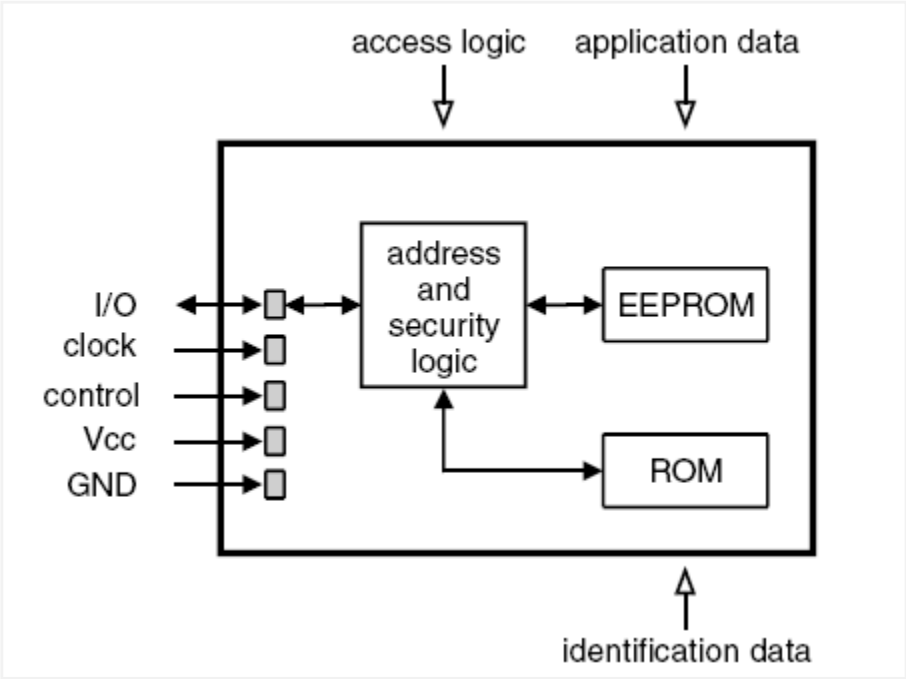

Figure I.2 Schéma typique d'une carte mémoire avec sécurité [Ran 03].

Elles permettent le stockage d'informations en quantité plus importante qu'une piste magnétique, mais aucune capacité de traitement de l'information n'est offerte, ce qui limite les possibilités d'application. Les cartes à mémoire se répartissent en deux sous-catégories :

- *Cartes à mémoire simples* : sans réel mécanisme de protection, le niveau de sécurité de ces cartes est très faible. Les télécartes de première génération (OTPROM) et les cartes I2C (Inter Integrated Circuit) font partie de cette catégorie.

- *Cartes à mémoire protégée*: un premier mécanisme de protection est introduit par la limitation de l'accès de certaines zones mémoire à leur seule lecture ou lors de la phase de personnalisation, et par la mise en œuvre d'automates simples.

1.4.2.2 Carte à microprocesseur

Les cartes à microprocesseur (ou microcontrôleur) incorpore l'intelligence qui faisait défaut aux cartes à mémoire, et méritent ainsi le nom de smart cards, traduction courante de carte à puce, par opposition aux memory cards précédentes. Elles intègrent ainsi une capacité de traitement de l'information apportée par un véritable microprocesseur et permettent d'élever le niveau de sécurité éventuellement par association avec un crypto-processeur.

Selon l'approche utilisée pour leur programmation, on peut distinguer trois sous-catégories des cartes à microprocesseur :

- *Cartes spécifiques*: le programme de la carte est inscrit dans le masque du composant, avec un OS (operating system) propriétaire.

- *Cartes personnalisables*: la personnalisation de la carte en fonction de l'application est permise par un OS supportant un jeu d'instructions spécifique et un certain nombre de répertoires et de fichiers.

- *Cartes à OS ouvert* : la mise en œuvre d'un jeu d'instructions et de mécanismes de gestion de fichiers spécifiques est rendue possible par la programmation de l'interpréteur de commande laissée ouverte à l'utilisateur dans un langage évolué qui est précompilé avant d'être chargé dans la carte. La Java Card fait partie de cette catégorie.

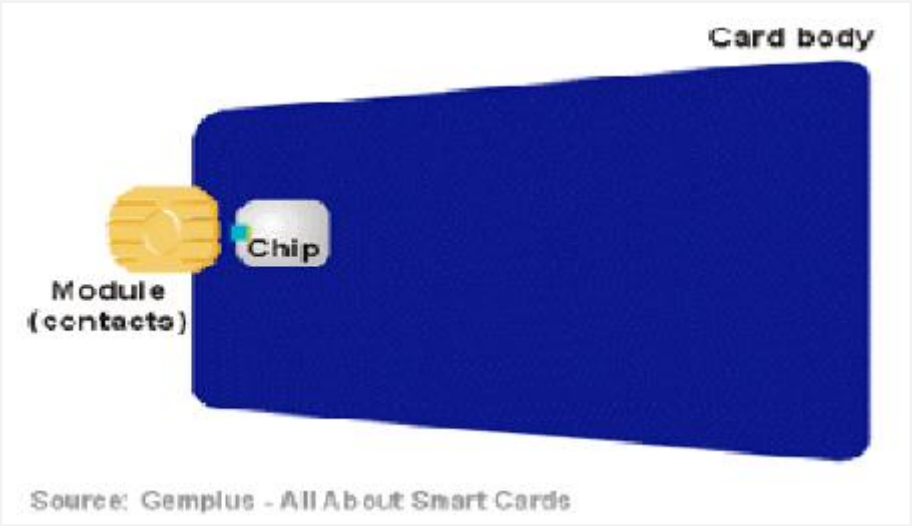

Figure I.3 Schémas d'une carte à puce avec contact

1.4.3 Cartes sans contact

L'introduction des technologies de communication radiofréquence a permis de créer une nouvelle catégorie de cartes à puce : les cartes sans contact. Elles font partie de la famille de produits appelés RFID (Radio Frequency Identification Device) qui est plus vaste. Le principe consiste à assurer l'alimentation de la carte et l'échange de données avec le lecteur par l'intermédiaire d'un champ électromagnétique plutôt que par des connexions électriques

physiques. Il existe trois familles de normes selon la distance de couplage entre carte et lecteur, du millimètre à la dizaine de centimètres.

Parmi les cartes sans contact, on retrouve la même distinction que pour les cartes avec contacts entre les cartes à mémoire et les cartes à microprocesseur. Il faut cependant retenir que même les cartes sans contact à mémoire comportent un microcontrôleur dont la fonction est principalement d'assurer le dialogue avec le lecteur, ses fonctions applicatives étant très limitées. Ainsi l'interface de connexion physique des cartes avec contacts est remplacée par un ensemble antenne + microcontrôleur de gestion haute fréquence quel que soit le type de carte. Dans certains cas, les deux types d'interface sont conservés, on parle alors de cartes « combi ».

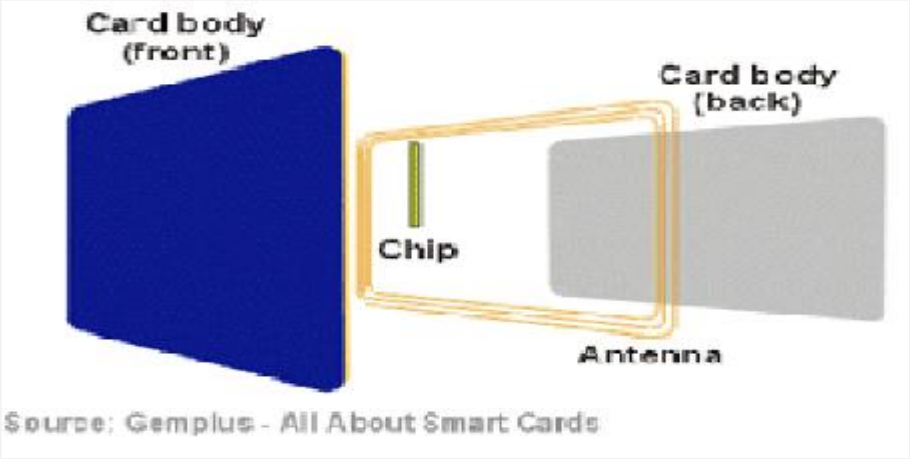

Figure I.4 Schémas d'une carte à puce sans contact

1.5 Caractéristiques physiques et électriques de carte à puce

1.5.1 Format

La carte à puce apparaît le plus souvent sous son format carte de crédit mais d'autres facteurs de forme sont apparus comme le format GSM. Elle se présente, généralement, sous forme d'un rectangle de plastic à bords arrondi de 54 mm x 85.6 mm de taille et 0.76 mm d'épaisseur. Cette aspect de la normalisation est régit par la norme ISO 7816-1.

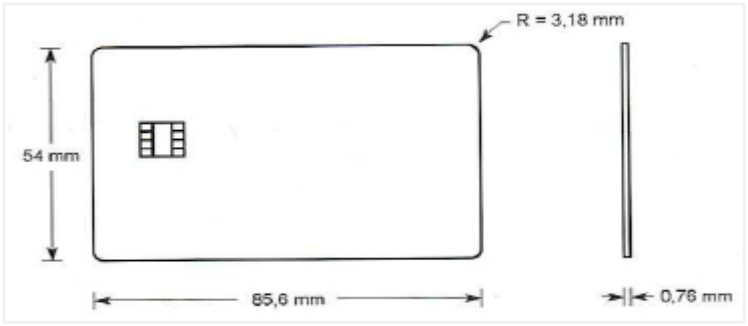

Figure I-5 Aspect et dimensions normalisées d'une carte à puce de type ID 1[BS09].

Bien que ces deux formats sont les plus connues (carte bancaire et carte GSM), il existe théoriquement trois formats normalisées ID1, ID00, ID000. De la on voit que la carte bancaire correspond au format ID1 et la carte mini ou micro SIM correspond au format ID000. Cela dit, le fabricant ne produit qu'un seul format (ID1) et les autres formats de carte sont obtenus en cassant les découpes prévus à cette effet dans la carte en plastique, ceci est rendu possible grâce à la position de la puce et de ses contacts comme le montre la figure I.5.

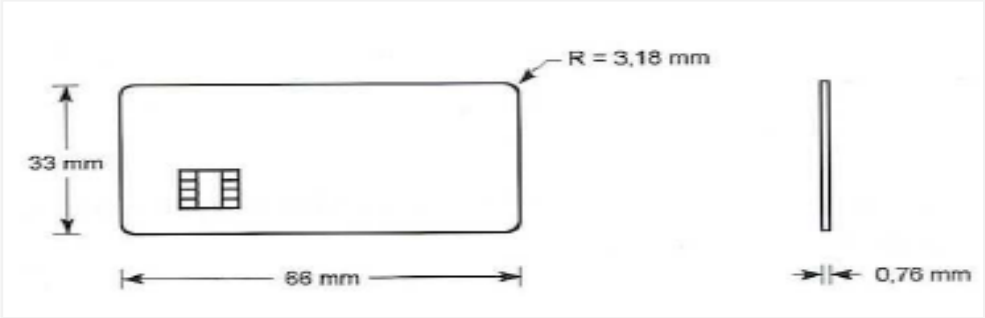

Figure I-6 Aspect et dimensions normalisées d'une carte à puce de type ID 00 [BS09].

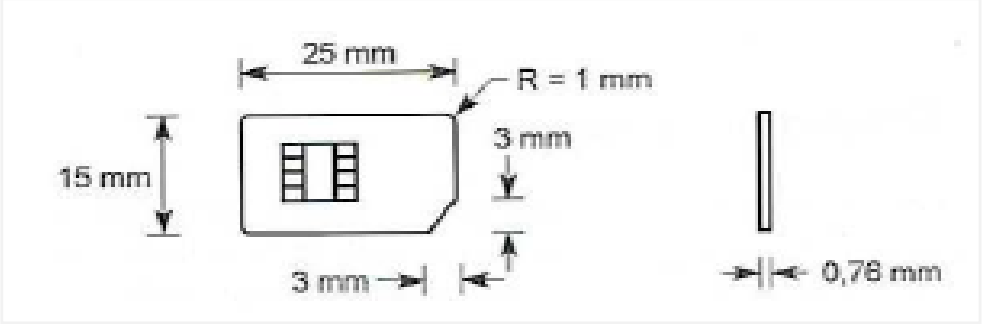

Figure I-7 Aspect et dimensions normalisées d'une carte à puce de type ID 000 [BS09].

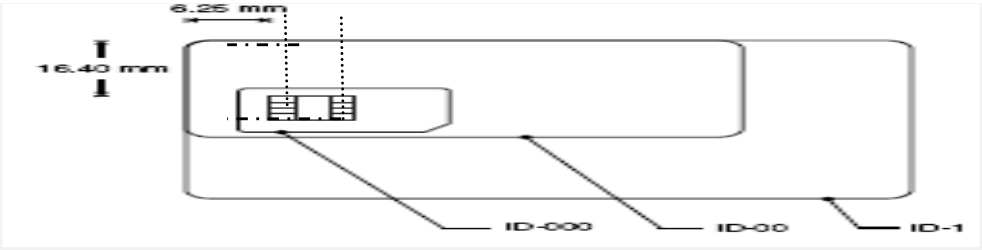

Figure I-8 Mise en évidence de la compatibilité entre les trois formats de cartes [BS09].

1.5.2 Les contact de la puce

Les huit contacts, repérés C1 à C8 dans la *figure I.9*, sont définis dans la norme ISO 7816-2 de la manière suivante :

- C1 (*Vcc*) : la tension d'alimentation positive de la carte, fournie par le lecteur.
- C2 (*RST*) : la commande de reset de la carte, également fourni par le lecteur.
- C3 (*CLK*) : abréviation de *clock* qui correspond à l'horloge fournie à la carte par le lecteur.
- C4 et C8 (*RFU*) : abréviation de *reserved for future use,* ce qui signifie donc que ces contacts sont réservés à une utilisation future ; à ce jour, pas encore définit.
- C5(*GND*) : la masse électrique de la carte.
- C6 (*Vpp*) : tension de programmation de la carte fournie par le lecteur, il est à noter que, de nos jours, ce contact n'est pas utilisé, du fait que la mémoire EEPROM est programmée par la seule tension fournie par le contact Vcc, ceci est rendu possible grâce à la technologie actuelle, ce qui n'était pas le cas des premières cartes, qui nécessitaient 21 volt pour la programmer.
- C7 (I/O) : le signale d'entré et sortie de données en provenance ou à destination de la carte. Cette ligne est donc bidirectionnelle.

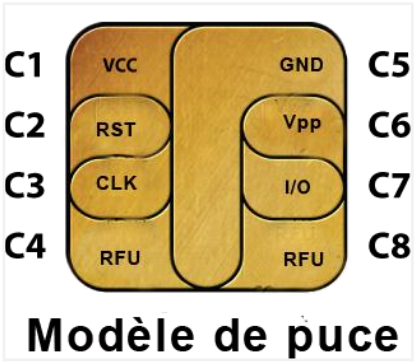

Figure I-9 Les contacts de la puce [NTK07].

Il est à remarquer qu'i existe une autre position des contacts autre que celle de ISO, dite position AFNOR, et ça pendant plusieurs années notamment en France. Cette position est caractérisée par une puce décalée vers l'angle supérieur gauche de la carte par rapport au brochage ISO et par un brochage complètement inversé.

1.5.3 Les composants électroniques de la puce

La carte à puce jouit d'un avantage considérable d'abord par sa taille qui lui permet d'affronter les tests de torsion sans être détériorée, puis par le fait que tout sur une carte à puce est rassemblé au même endroit. En effet, le cœur de la puce, le microprocesseur, est un bloc monolithique. Il contient à la fois les unités de calcul tels le processeur et le co-processeur cryptographique, les mémoires contenant le code de la carte dans la ROM, les mémoires de travail, persistantes comme l'EEPROM, ou temporaires telle la RAM, et les éléments de communication. Tout cela tient en 25 millimètres carrés, la surface maximum allouée au microprocesseur par la norme ISO : cela correspond au petit carré noir intitulé « chip ». Ainsi, il est plus dur pour un éventuel attaquant d'isoler facilement telle ou telle partie de la carte. Pour compliquer le tout, elle dispose de capteurs permettant de détecter une perte de puissance électrique, une surchauffe des composants ou encore la mise à nu des circuits (capteur de lumière) .

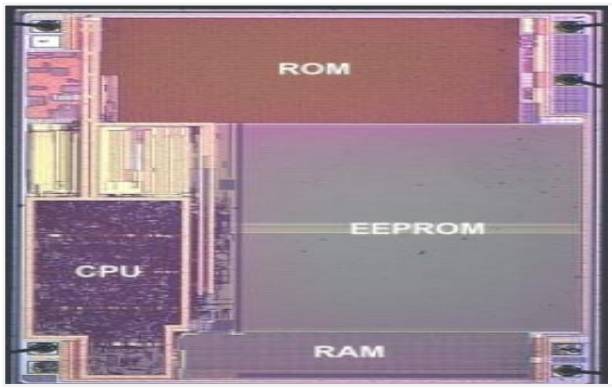

Figure I.10 Un processeur de la carte à puce observé en microscopie optique (source : SERMA Technologies).

1.5.3.1 le microcontrôleur

Précédemment, les cartes à puce étaient basées sur des processeurs de type Motorola 6805 ou l'Intel 8051, 8 bits CISC, cadencés selon une horloge externe à 5 MHz.

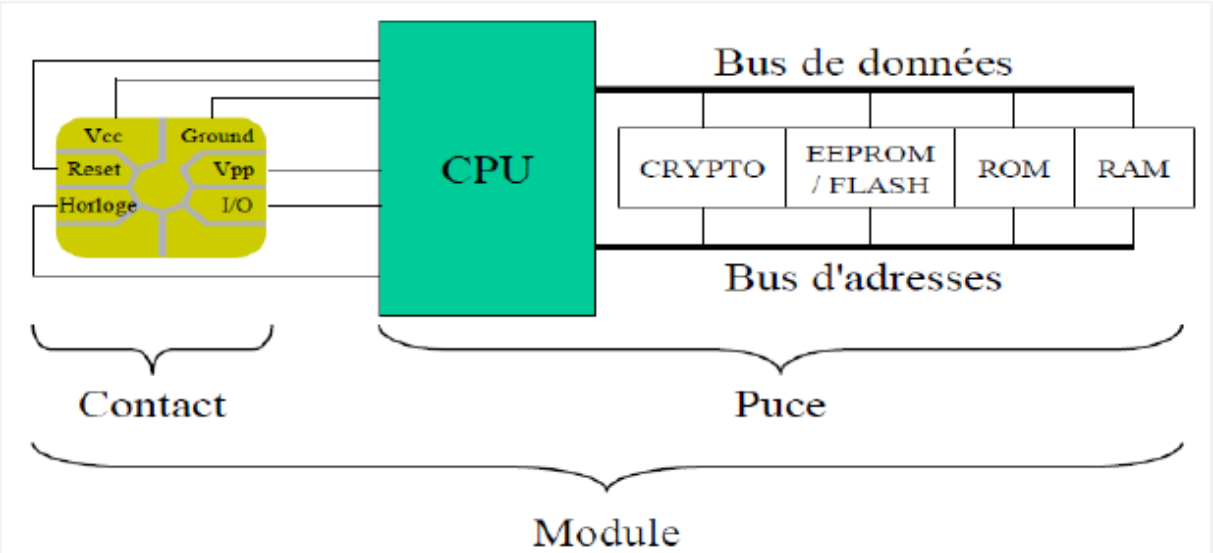

Figure I.11 Architecture simplifiée dune carte à puce [Ser10].

Aujourd'hui, on dispose de processeurs plus puissants 16 ou 32 bits CISC ou RISC travaillant à des fréquences comprises entre 5 et 40 MHz. Cependant, Le standard ISO7816 obligeant à respecter une certaine plage de fréquences d'horloge pour rester compatible avec les matériels déjà en place (lecteurs, etc), les fabricants utilisent alors en interne des multiplicateurs pour atteindre de plus hautes fréquences permettant de faire les calculs dans la carte plus rapidement. Actuellement, il existe ainsi des cartes à microprocesseur de dernière génération RISC fonctionnant avec une horloge interne de 100 à 200 MHz en utilisant un multiplicateur *x20* ou *x40*.

Ces puces possèdent également un coprocesseur cryptographique qui réalise les opérations de chiffrement et déchiffrement de manière câblée améliorant ainsi grandement les performances. Ce coprocesseur cryptographique est associé à un générateur de nombres aléatoires RNG (Random Number Generator).

1.5.3.2 Les différents types de mémoire

On retrouve trois types de mémoires dans la carte que l'on peut classer en fonction de leurs propriétés : une mémoire figée en usine (ROM), une mémoire persistante (EEPROM) et une mémoire volatile (RAM).

- *La ROM (Read Only Memory) ou mémoire morte*

La ROM constitue la mémoire programmée en usine, c'est une mémoire persistante et figée. Elle n'a donc pas besoin d'énergie pour sauvegarder l'information qu'elle contient et son contenu n'est pas modifiable. Elle sert à stocker le système d'exploitation de la carte (COS ou Card Operating System) ainsi que des données permanentes. Sa taille est le plus souvent de 32 ko, mais on peut trouver des cartes contenant 300 ko ou plus (sur les cartes modernes haut de gamme).

- *La mémoire volatile*

La mémoire volatile est utilisée comme espace de stockage temporaire pour le calcul grâce à la vitesse de son temps d'accès. Elle possède un caractère non persistant et perd donc son contenu entre deux utilisations. Elle a la particularité de pouvoir être lue et écrite à l'infini. Il s'agit d'une mémoire RAM (Random Access Memory) comme celle que l'on peut trouver sur un ordinateur classique. Néanmoins dans le cas de la carte, sa capacité est seulement de 1 ko à 24 ko (sur les cartes modernes haut de gamme).

- *La mémoire persistante*

La mémoire persistante a les mêmes caractéristiques que la ROM à ceci près qu'elle peut être modifiée. Elle est en général désignée par le terme de mémoire non volatile (NVM). Elle est chargée du stockage des informations qui peuvent évoluer dans le temps tels que les applications, les clefs cryptographiques et le code PIN de l'utilisateur ; c.-à-d. les informations qui doivent être conservées même lorsque la carte n'est plus sous tension.

Il n'existait encore récemment qu'un seul type de mémoire possédant ces caractéristiques : l'EEPROM (Electrical Erasable Programmable Read Only Memory). Aujourd'hui, on trouve des produits basés sur d'autres types de mémoire aux propriétés similaires comme la mémoire Flash, la FeRAM (Ferroelectric Random Access Memory) ou MRAM (Magnetic Random Access Memory).

L'EEPROM présente deux inconvénients :
– sa durée de vie limitée en nombre de cycles d'écriture (son endurance) qui est de 100000 cycles et en temps de rétention de l'information (10 ans) ce qui nécessite de changer régulièrement la carte pour éviter des dysfonctionnements ;
– sa lenteur d'accès (sa latence) puisqu'elle est 1000 fois plus lente en écriture que la RAM.

1.6 Fabrication et cycle de vie de la carte à puce

Cette section a pour vocation de présenter les étapes de la vie d'une carte et les différents acteurs qui y prennent part. En effet, ces intervenants sont tous essentiels pour la sécurité de la carte car pouvant être tour à tour commanditaire d'une évaluation utilisateur bien ou mal intentionné, etc.

1.6.1 Les acteurs

On peut distinguer dans le monde de la carte à puce, quatre principaux acteurs :
– le fabricant de puces (e.g. Fujitsu, Philips, ST Microelectronics), aussi appelés fondeurs
– le fabricant de cartes (e.g. Axalto, Gemplus, Oberthur)
– l'émetteur de cartes ou fournisseur de cartes (card issuer) (e.g. les banques) qui fournit la carte à l'utilisateur final
– le porteur de carte qui est l'utilisateur final

1.6.2 La fabrication

Le processus de fabrication décrit ci-après est volontairement simplifié mais reste assez proche de la réalité.

Au commencement de la vie d'une carte à puce se trouve le fabricant de puces. C'est lui qui va construire la puce, choisir les technologies pour le microprocesseur, les différentes mémoires embarquées, etc. Il va aussi, lors de l'opération de masquage, figer dans la mémoire ROM un système d'exploitation minimum et certaines données afin de permettre à son produit de fonctionner.

Dans les faits, l'opération de masquage est une suite itérative d'opérations consistant à reproduire un circuit électronique par dépôts successifs de couches de métallisation puis de couches de passivation (i.e. un isolant) sur un substrat de silicium. La réalisation d'un circuit

électronique dans un monde à deux dimensions est impossible puisqu'obligatoirement des pistes électriques se croiseraient. Aussi, les fabricants utilisent la troisième dimension pour câbler leurs circuits et emploient des plots de tungstène pour relier les différents niveaux de métal.

Par la suite, le fabricant de cartes reçoit du fabricant de puces des galettes de silicone (Silicon wafer) contenant les puces qu'il va découper, tester, puis assembler avec le microcontact pour former le micromodule (s'il désire par exemple faire une carte à contact). Parallèlement il produit le corps de carte à partir de PVC (Poly Vinyl Chloride), d'ABS (Acrylonitrile Butadiène Styrol) ou de PC (PolyCarbonate). Le matériau choisi pour le corps de carte dépendra de l'usage auquel est destinée la future carte. En effet, ces matériaux n'ont pas les mêmes caractéristiques (e.g. de longévité), donc pas les mêmes fonctions, ni les mêmes coûts.

L'étape suivante consiste à insérer le micromodule sur le corps de carte dans le cas d'une carte à contact – opération appelée encartage. On a maintenant une carte vierge qui va pouvoir commencer son cycle de vie.

1.6.3 Le cycle de vie de la carte

Souvent le fabricant de cartes développera pour le compte de l'émetteur de cartes les applications à embarquer et les installera sur la carte.
Le cycle de vie de la carte commencera avec une phase d'initialisation de la carte qui consiste à inscrire en mémoire persistante des données communes aux applications et éventuellement à charger d'autres caractéristiques de sécurité. Cette étape, comme la suivante, est souvent réalisée par le fabricant de cartes en fonction de la demande de son client émetteur de cartes. L'étape suivante est la personnalisation et se fait à deux niveaux :

Au niveau électrique :

- Par l'inscription en mémoire persistante des données relatives à chaque porteur de carte ;

Au niveau graphique :

- Par la personnalisation du corps de carte suivant le désir de l'émetteur de carte (e.g. son logo).
- Par l'embossage de données relatives à chaque porteur de carte.

La carte peut alors passer du fabricant de cartes au porteur de carte par l'intermédiaire de l'émetteur de cartes. Le cycle de vie de la carte se poursuit ainsi avec son utilisation par le porteur de carte et se termine soit par invalidation logique, saturation de la mémoire, bris, perte, vol, etc…

1.7 La communication Lecteur/Carte

Pour interagir avec un environnement extérieur la carte à puce dispose de fonctions de communication. Plusieurs technologies de communication furent mises au point pour réaliser des échanges avec le monde extérieur. Cependant, les cartes, en raison de leur conception extrêmement compacte, sont des ordinateurs passifs qui doivent être alimentés depuis l'extérieur pour pouvoir fonctionner. Ainsi, les cartes ne communiquent pas directement avec d'autres ordinateurs, mais plutôt avec un lecteur de cartes à puce qui leur fournit l'énergie requise pour fonctionner. Donc le lecteur sert d'interface entre la carte et la machine hôte avec laquelle elle doit dialoguer. On dit alors de cette communication qu'elle est en mode semi-duplex, c.-à-d. qu'à l'instant ***t*** seule une des parties peut utiliser la ligne de communication : soit le lecteur, soit la carte, mais jamais les deux en même temps.

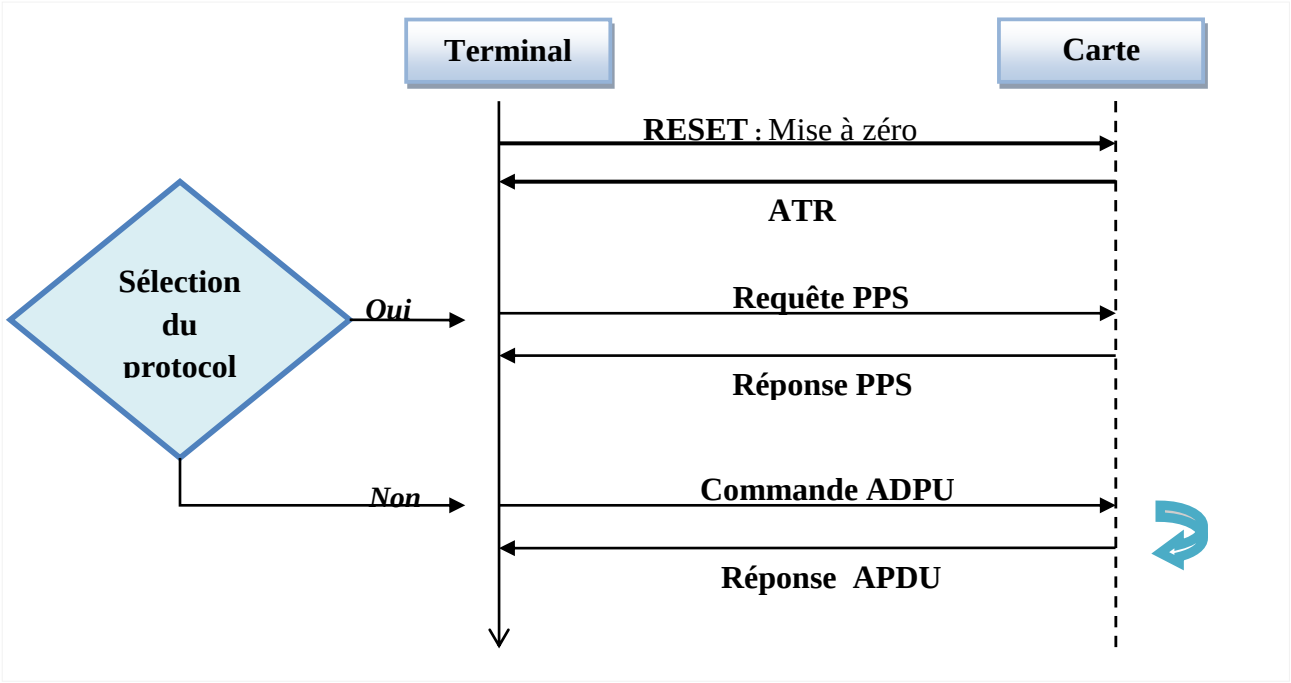

Figure I.12 communication entre le lecteur et la carte à puce

Les échanges entre la carte et le terminal qui est régie par la norme ISO 7816-3, passe par plusieurs commandes, je cite :

La répense au reset (ATR)

Une fois la carte insérée dans le lecteur, celle-ci est mise sous tension, elle envoi une réponse au terminal dite ATR (*Answer to Reset*), c.à.d. une réponse à la mise a zéro, ceci via la ligne IO. Puis le terminal évalue l'ATR, laquelle contienne divers paramètres concernant la carte et la transmission de données. Le terminal peut alors envoyer sa première requête. Cette commande est spécifier en détaille par la norme ISO/IEC 7816-3

La sélection du protocole

Entre l'ATR et la première commande envoyée à la carte, le terminal peut envoyer une commande dite ; sélection des paramètres de protocole PPS (*Protocol Parameter Selection*). Elle permet au terminal de changer le protocole si celui indiqué dans l'ATR n'est pas le bon. La carte réposnd alors par une commande PPS de réponse. Cependant, si la requête contient des éléments que la carte ne peut pas réaliser ou satisfaire, alors elle attend jusqu'a ce que le terminal envoi un reset.

Le principal inconvénient de cette procédure est le temps que le terminal doit attendre pour finalement utiliser l'actuel protocole. Cette commande est décrite par la norme ISO/IEC 7816-3.

Le protocole utilisé pour le transport des données est le protocole TPDU (Transmission Protocol Data Unit). Tous les échanges de données pour ce protocole sont également définis dans l'ISO 7813-3. Ce protocole définit deux modes principaux de transmission qui sont :

- le T=0 qui utilise une transmission par octet, dans la trame généralement utilisée, comprend un en-tête de 5 octets : classe (CLA), instruction (INS), paramètres (P1, P2, P3) (figure I.12).

Généralement P1 et P2 donnent une référence, P3 donne la longueur du champ de données attendu. Les données peuvent être *entrantes* ou *sortantes* ou fonction de l'instruction considérée. Le dernier champ contient deux octets d'état ME1, ME2 qui donnent un compte rendu d'exécution.

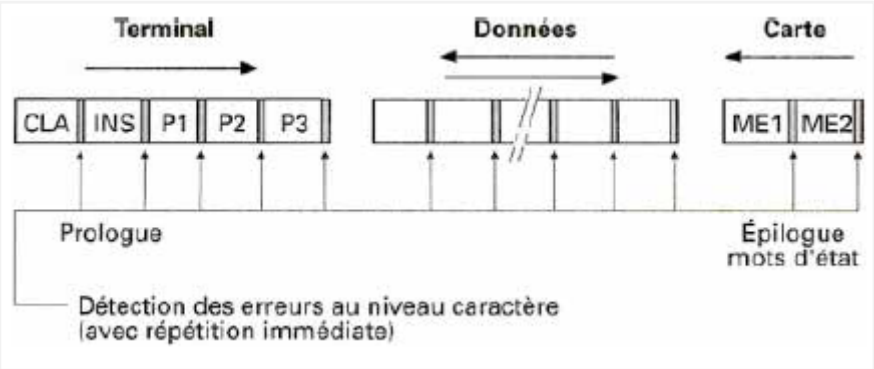

Figure I.13 Trame du protocole octet (T0) [MU].

Le mécanisme de reprise d'erreur permet de répéter un octet erroné immédiatement après sa détection. Lorsque le récepteur détecte une erreur, il force la ligne pour signaler à l'émetteur qu'il faut répéter immédiatement l'octet transmis.

- le T=1 qui utilise une transmission par paquet ou bloc et Il est rarement utilisé en mode contact.

Prologue			Informations	Épilogue
Adresse nœud NAD	Contrôle du protocole PCB	Longueur champ LEN	Champ de données	Détection des erreurs LRC ou CRC
1 octet	1 octet	1 octet	0 à 254 octets	1 ou 2 octets

Longueur des données

Code de détection des erreurs

3 types de blocs (codés par le PCB)
- bloc I (*Information*)
- bloc R (*Receive ready*)
- bloc S (*Supervisory*)

Figure I.14 Trame du protocole orienté paquet (T1) [MU].

Il y a aussi certaines cartes qui utilisent le T=14 qui est réservé aux protocoles propriétaires.
On trouve également des cartes qui utilisent T=CL pour les communications sans contact.

Les Commandes APDU

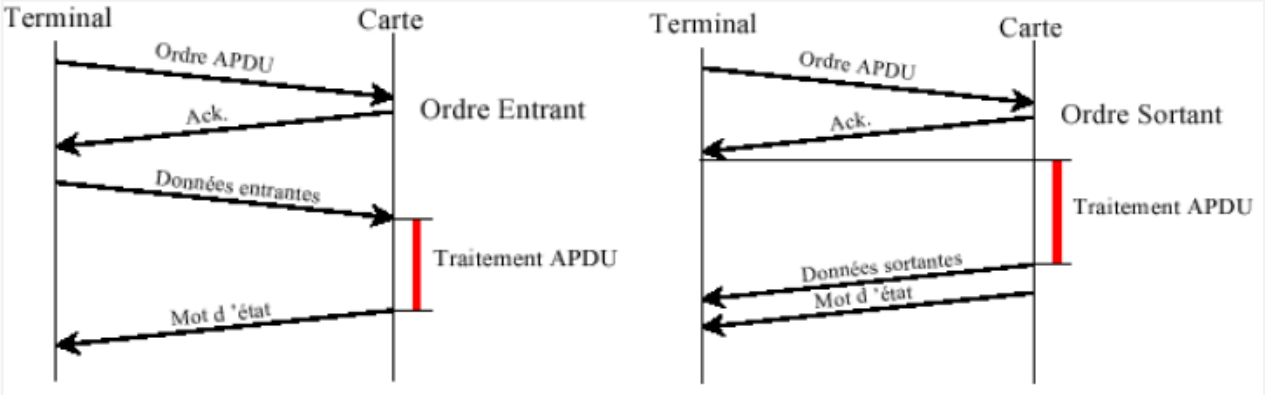

Figure I.15 modèle de communication Terminal/carte [MU].

Le modèle de discussion entre le terminal et la carte est un modèle maître-esclave où le terminal est le maître et la carte est l'esclave. La carte est toujours en attente d'une commande APDU. Une réponse APDU aura toujours lieu en retour d'une commande APDU. Les deux types de structures de communication commande et réponse sont toujours couplés.

La commande et réponse APDU est constituée par :

<table>
<tr><th colspan="7">L'instruction APDU</th></tr>
<tr><td colspan="4">Entête obligatoire</td><td colspan="3">Corps optionnel</td></tr>
<tr><td>CLA</td><td>INS</td><td>P1</td><td>P2</td><td>Lc</td><td>Champ du données</td><td>Le</td></tr>
<tr><td colspan="7">• CLA (1 octet): Classe d'instruction--- indique la structure et formate pour une catégorie d'instruction et réponse APDU
• INS (1 octet): code de l'Instruction: spécifie l'instruction P1 (1 octet) et P2 (1 octet): paramètres de l'Instruction--fournissez des qualifications à l'instruction plus loin
• Lc (1 octet): Nombre des octets qui présentent dans le champ du données de l'instruction
• Champ du données (les octets égalent à la valeur de Lc): Une séquence des octets dans le champ du données de l'instruction
• Le (1 octet): Maximum des octets a attendu dans le champ du données de la réponse à l'instruction</td></tr>
<tr><th colspan="7">La response APDU</th></tr>
<tr><td colspan="3">Corps optionnel</td><td colspan="4">Partie obligatoire</td></tr>
<tr><td colspan="3">Champ du données</td><td>SW1</td><td colspan="3">SW2</td></tr>
<tr><td colspan="7">• Champ du données (longueur variable): Une séquence des octetst a reçu dans le champ du données de la réponse
• SW1 (1 octet) et SW2 (1 octet): mots de la situation--dénotez l'état du traitement dans la carte</td></tr>
</table>

Tableau 1.1 Format de la commande et de la réponse APDU [NTK07].

Il y a quatre cas possibles d'échanges APDU :

- cas 1 : Aucune donnée n'est échangée entre le CAD et la carte autre que l'entête de la commande APDU et la partie obligatoire de la réponse APDU.

- cas 2 : Aucune donnée (dans le champ de données de la commande APDU) n'est envoyée à la carte. Le corps de la commande contient le champ Le qui spécifie le nombre d'octets que la carte lui fournira dans le champ de données de sa réponse APDU.

- cas 3 : Ici des données, de taille Lc octets, sont fournies à la carte (dans le champ de données de la commande APDU) et la carte ne renvoie dans la réponse APDU que la partie obligatoire.

- cas 4 : On a des données échangées dans le champ de données de la commande et de la réponse APDU.

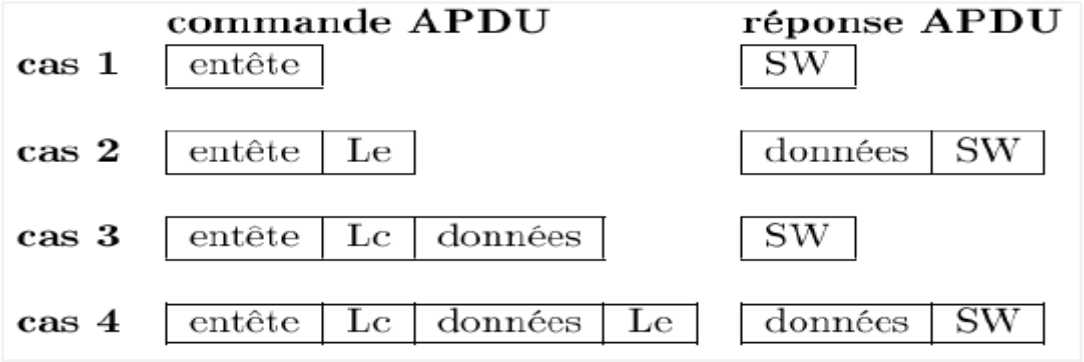

Figure I.16 les quatre cas possibles d'échanges de commande APDU [Ser10].

1.8 Le system de fichiers

Les cartes à puce telles qu'elles sont définies dans l'ISO 7816-4 peuvent avoir un système de fichier à structure hiérarchisée. Un système de fichier conforme à l'ISO supporte 3 types de fichiers : fichier racine (Master File : MF), fichier dédié (Dedicated File : DF) et fichier élémentaire (Elementary File : EF). Chaque fichier possède un identificateur (File IDentifier :FID) sur 2 octets et un nom symbolique jusqu'à 16 octets(le gestionnaire de fichier est détaillé dans l' annexe 2).

1.9 Les Normes d'application

La technologie à base de carte à puce est normalisée par plusieurs comités, mais également par divers forums industriels.

ISO/IEC 7816

La norme ISO7816 est un ensemble de documents destinés à assurer l'inter opérabilité entre cartes à puce et lecteurs. Elle comporte les parties suivantes:

1 Caractéristiques physiques.

2 Dimensions et positions des contacts.

3 Signaux électriques et protocoles de transmission.

4 Commandes intersectorielles pour les échanges.

5 Système de numérotation et procédure d'enregistrement pour les identificateurs d'applications.

6 Eléments de données interindustrielles.

7 Commandes intersectorielles pour langage d'interrogation de carte structurée (SCQL).

8 Commandes intersectorielles de sécurité

9 Commandes intersectorielles additionnelles et attributs de sécurité.

10 Réponse à la RAZ des cartes synchrones.

11 Architecture de sécurité.

OpenCard.

Le consortium OpenCard a défini un ensemble d'API java et une architecture côté terminal, nommé Open Card Framework (OCF). L'interface carte terminal est réalisée par des objets java et non par l'interaction avec des protocoles de communication.

PC/SC.

PC/SC (Interoperability Specifications for Integrated Circuit Cards (ICCs) and Personal Computer Systems) permet d'intégrer des lecteurs de carte à puces aux PCs (Interface Device IFD). La version actuelle est 1.0. Le concept de cette pile consiste à offrir une interface de type API (niveau 6) aux applications qui utilisent les ressources des cartes aux moyens de DLLs.

Microsoft Smartcard for Windows (Windows SC)

En 1999 Microsoft a introduit la spécification d'une carte associée à un kit de développement. Cette carte réalise l'émulation d'un système de fichier (FAT) muni d'un *ACL (Access Control List*). Un ou plusieurs utilisateurs utilisent les ressources de la carte après une phase d'authentification. Le développement d'application utilise le langage *visual basic* dont les *code byte* sont chargés et exécutés par le système d'exploitation de la carte. Il est possible d'utiliser cette dernière à travers des APDUs classique ou des APIs spécifiques.

EMV'96, EMV'2000 et les suivants.

Ce standard publié en 1996 par Europay, Mastercard et Visa, est relatif au domaine bancaire. Il décrit les terminaux, les cartes et leur interaction.

EMV'96 Specification for Payment Systems, comporte quatre parties:

- 1 caractéristiques mécaniques, électriques, interface logique et protocoles de transport des cartes (ICC).
- 2 Jeu de commandes APDUs.
- 3 Mécanismes de sélection des applications.
- 4 Sécurité, méthodes d'authentification et de signature.

Global Platform

Cet ensemble de spécifications est d'origine VISA, il est destiné à la gestion des applications et concerne les cartes et terminaux. Les APIs associées sont basées sur la technologie Java.
Côté carte, trois types de composants sont définis :

- Card Manager. Ce module contrôle la carte et son contenu.
- Global Platform APIs. GP
- Provider Security Domains. Gestion de la sécurité d'une carte multi application.

1.10 Les Avantage de la carte à puce

La carte à puce est résistante aux attaques car elle n'a pas besoin d'être dépendante d'une ressource externe potentiellement vulnérable. La sécurité de la carte à puce se fait à plusieurs niveaux :

- au niveau physique :

- par des techniques d'impression sur le corps de carte très sophistiquées.

➢ au niveau hardware :

- par un numéro de série unique.
- par l'utilisation de mémoire de type PROM. L'information peut être stockée de manière durable hors usine. et aussi être déplacée par moment à l'intérieur de cette mémoire afin d'éviter des attaques localisées (c'est le software qui gère cette possibilité).
- par un blindage physique du composant par collage du micro-chip au micromodule avec des techniques spécialisées et par insertion de couches de métallisation.
- par un contrôle des accès aux mémoires (EEPROM, ROM).
- par des détecteurs de conditions anormales telles que la température, l'humidité, la tension d'alimentation, la fréquence d'horloge, etc.
- par un brouillage des informations dans le composant : dissémination de la mémoire, brouillage du bus, sous-ensembles de leurres.
- grâce à l'utilisation de témoins d'intégrité disséminés dans la mémoire.
- grâce à un générateur hardware de nombres aléatoires.
- grâce à des coprocesseurs cryptographiques qui permettent de calculer le DES en hardware. Les coprocesseurs mathématiques permettent d'implémenter des algorithmes tels que RSA, etc.

➢ au niveau software :

- par des contrôles d'accès aux données grâce par exemple à un code secret ou par authentification cryptographique.
- par un maintient de l'intégrité des données en utilisant la vérification des témoins, les calculs de signatures sur les données internes et l'atomicité des transactions.
- grâce à des entrées/sorties sécurisées par chiffrement/signature et temps d'exécution des commandes constants.

1.11 Fabricant des cartes à puce et des lecteurs de cartes

LOGO DU FABRICANT	PRODUITS
axalto	C'est le nouveau nom de l'activité cartes de Schlumberger. On retrouve donc chez axalto les cartes (Cryptoflex, Cyberflex) qui existaient chez Schlumberger par le passé.
BasicCard	Fabricant de la BasicCard qui est une carte à puce à OS ouvert.
GEMPLUS	Gemplus s'est fusionné avec Axalto pour créer Gemalto en juin 2006
MULTOS	Dédié à des applications sécurisées (paiement, contrôle d'accès, etc.) compte tenu des contraintes de développement qu'il impose, c'est un système réservé aux professionnels.
acs	Fabricant des cartes personnalisables ACOS 1 et ACOS 2.
gemalto security to be free	Le leader mondial de la sécurité numérique, avec un chiffre d'affaires pro forma 2005 de 1,7 Md€, des opérations dans quelque 100 pays. Il a annonce qu'il détient maintenant 100% de Gemplus.
Infineon	Moins connu du grand public. Ce fabricant n'en propose pas moins de diverses cartes.
Oberthur Card Systems The Key to an Open World	Orienté essentiellement vers les professionnels. Oberthur Card Systems propose des solutions à base de cartes à puce dans touts les domaines :banque, téléphonie, etc.

Tableau 1.2 Les principaux fabricants de carte à puce [1]

Lecteur	Caractéristiques
 Lecteur USB :GEM PC SL	▪ Interface/connexion : USB ▪ Norme/spécification : - PC/SC - ISO/IEC 7816-1,2,3,4: IC carte avec contact - EMV level 1, EMV2000 version 4.0 - CCID - Chip card Interface device 1.0 ▪ Cartes lues: ISO 7816 ▪ OS compatible: - Windows 98, 98SE, Me, 2000,Xp et Vista 32 bits - Windows Xp, Serveur & Vista 64 bits - Linux Redhat WS3.0, WS4.0, Suse Professional 9.2, DEBIAN "Sarge"
 Lecteur de carte SIM : GEMPC Key USB	▪ Interface/connexion : USB 2.0 ▪ Norme/spécification : - PC/SC, ISO/IEC 7816-1, 2, 3, 4 carte avec contact - Windows Logo Program WLP 2.0 - CCID - Chip Card Interface Device 1.0 ▪ Cartes lues: Cartes SIM ISO7816-1, 2, 3, 4 ▪ OS compatible: - Windows 98SE, Me, 2000, XP, Serveur 2003 - Windows XP 64bits et serveur 64 bits

 Lecteur des ordinateurs portables : CARDMAN 4321 Express card	▪ Interface/connexion : ExpressCard 54mm ▪ Norme/spécification : - PC/SC 2.01, CT-API, Sync-API - RoHS, WEEE ▪ Cartes lues: - ISO 7816 ▪ OS compatible: - Windows 98/ME, Windows 2000, XP, 2003 Server.
 Lecteurs RFID : CARDMAN 5321 USB	▪ Interface/connexion : USB 2.0 ▪ Norme/spécification : - PC/SC 2.01, CT-API, Sync-API, RoHS, WEEE ▪ Cartes à contact lues : ISO 7816 ▪ Cartes sans contact lues : - ISO14443A, ISO14443B, ISO15693 ▪ OS compatible: - Windows 98/ME, Windows 2000, XP, 2003 Server.

Tableau 1.3 Exemple de lecteurs de carte à puce [2].

1.12 Conclusion

La carte à puce est aujourd'hui le support de stockage de données le plus sûr, grâce à des mécanismes sécuritaires implémentés dans la carte, tel que : le code PIN, la cryptographie…etc. Pour garantir l'interopérabilité entre les cartes et les lecteurs, des standards et des spécifications ont été définie, par exemple L'ISO 7816 a défini des normes pour la carte à puce à contact pour standardiser son utilisation.

Les cartes à puce possédant un système d'exploitation natif ont prouvé leur limites à cause de la difficulté et la lenteur du développement ainsi que l'impossibilité de charger et de mette à jour des applications après l'émission de la carte. Pour pallier à ses inconvénients, de nouvelles cartes à OS ouvert ont vu le jour. Dans le chapitre suivant, nous allons étudier une catégorie de cette nouvelle technologie qui est : la JavaCard.

CHAPITRE II

Java Card

Chapitre II

Java Card

2.1 Introduction

Toutes les cartes à puces possèdent des ressources très limitées disponibles pour l'exécution d'applications. 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 à des experts. Il a donc fallut développer un langage qui soit à la fois fiable, robuste, peu gourmand en ressources et bien sûr simple. C'est en 1996 que Sun Microsystems Inc propose une solution après des essais menés par Schlumberger : le JavaCard. Le JavaCard est un système de programmation de cartes à puces basé sur le langage Java et la Java Virtual Machine. JavaCard est une version épurée du langage Java s'adaptant aux environnements réduits des cartes à puces. L'environnement minimum nécessaire au fonctionnement de l'API JavaCard est un processeur 300 KIP, 12 KO ROM, 4 KO EEPROM, et 512 Octets de RAM. Un certain nombre de composants, par exemple les tableaux multidimensionnels, les Threads et le garbage collector ..., ont été enlevés.

2.2 Historiques

En **Novembre 1996**, un groupe d'ingénieurs du centre de production de Schlumberger à Austin au Texas cherche à simplifier la programmation des cartes à puce tout en préservant la sécurité. Le langage de programmation Java apparaît comme la solution.

En **Février 1997**, Bull et Gemplus se joignent à Schlumberger pour co-fonder le Java Card Forum. Le but de ce consortium industriel est d'identifier et de résoudre les problèmes de a technologie Java Card en proposant des spécifications à JavaSoft (la division de Sun microsystems à qui appartient Java Card). Il a également pour objectif de promouvoir des APIs Java Card afin de permettre son adoption par l'industrie de la carte à puce. Aujourd'hui, le Java Card Forum regroupe les fabriquant de cartes, Sun et des utilisateurs. À partir de ce moment, avec un large soutien de l'industrie, Sun microsystems va acquérir la société Integrity Arts afin de développer la technologie Java Card comme plate-forme de la technologie Java pour les cartes à puce et les périphériques comportant peu de mémoire. Cette

orientation va être un gros avantage pour Gemplus qui était alors spécialisé dans le développement de machine virtuelle et de systèmes d'exploitation pour les cartes à puce.

En **Novembre 1997**, Sun microsystems présente les spécifications Java Card 2.0 qui consistent en un sous-ensemble du langage et de la machine virtuelle Java. Elles définissent des concepts de programmation et des APIs très différentes de celles de la version 1.0. Il n'y a encore rien sur le format des applets téléchargeables.

En **Mars 2001** sort la version 2.1 des spécifications Java Card. Elle se compose en fait de trois spécifications :

- la Java Card 2.1 *API Specification*.
- la Java Card 2.1 *Runtime Environment Specification*.
- la Java Card 2.1 *Virtual Machine Specification*.

Cette version de la spécification apporte un remaniement des API tels que la cryptographie et la gestion des exceptions. L'environnement d'exécution des applets est standardisé. Mais le changement le plus significatif de cette version est la définition explicite de la machine virtuelle Java Card (JCVM) et du *capfile* (fichier binaire contenant l'application java compilée), permettant ainsi une vraie interopérabilité.

En **juin 2002**, Sun Microsystems publie la version 2.2 des spécifications dont les principales nouveautés sont la prise en charge des canaux logiques ainsi que l'invocation de méthodes à distance (RMI voir [Gro02]). Elles rajoutent quelques éléments sur l'installation et l'effacement des applications sur la carte et quelques nouveaux algorithmes cryptographiques.

Depuis cette version, les spécifications se poursuivent ; ainsi en **octobre 2003**, la version 2.2.1 est publiée (voir [Sun03a ; Sun09a ; Sun03b]), cette dernière corrige l'API de la version précédente et apporte quelques clarifications. Puis lui succède en **mars 2006** la version 2.2.2 qui n'apporte pas de changement majeur mis à part quelques nouveaux algorithmes cryptographiques et une clarification des API. Après cette version arrive la branche 3.0 des spécifications qui apporte une révolution dans les spécifications, car elle introduit deux types de spécification Java Card :

- **Java Card 3 Classical Edition** qui est juste une évolution de la Java Card 2.2.2 faite pour fonctionner avec des cartes ayant de faibles ressources. Elle contient les fonctionnalités nécessaires aux supports des applets.

- **Java Card 3 Connected Edition** qui introduit de vraies nouveautés, car en plus des applets classiques, elle supporte un nouveau type d'application : les servlets. Les servlets sont des applications web qui nécessitent d'avoir un serveur web embarqué dans la carte. Elle nécessite donc des cartes avec des contraintes de ressources moins fortes (cartes modernes haut de gamme). Elle introduit un quatrième sous-ensemble dans la spécification celui des servlets. Ainsi pour cette édition, on a :

  - *Java Card Runtime Environment Specification, Connected Edition* [Sun09c].
  - *Java Card Application Programming Interface Specification, Connected Edition* [Sun09b].
  - *Java Card Virtual Machine Specification, Connected Edition* [Sun09f].
  - *Java Card Servlet Specification, Connected Edition* [Sun09d].

2.3 Les caractéristiques de Java Card

Dû à la faible capacité de leur mémoire, le langage utilisé sur les plateformes Java Card est un sous ensemble du langage Java. Il est adapté à l'écriture de programmes sur des Smart Cards et autres petits composants, mais préserve toutefois les caractéristiques principales de Java et notamment la programmation orienté objet.

Caractéristiques Java non supportées	Caractéristiques Java non supportées
➢ Type simple de données de petite taille: boolean, byte, short ➢ Tableux à une dimention ➢ Paquetages Java, classes, interfaces et execptions ➢ Caractéristiques orientées objets: héritage, méthodes virtuelles, surcharge et création dynamique d'objets, contrôle d'accées ➢ Le mot clé int et le support des entiens sur 32 bits sont optionnels	➢ Type simple de données de grosse taille : long, double, float ➢ Tableaux à plusieurs dimensions ➢ Caractères et chaines ➢ Sécurity Manager ➢ Ramasse miettes et finalisation ➢ Threads ➢ Sérialisation d'objet ➢ Clonage d'objet

Tableau II.1 Les caractéristiques Java supportées et non supportées.

2.3.1 Caractéristiques Java non supportées

● Chargement dynamique des classes

Une implémentation de la plateforme Java Card ne doit pas charger les classes dynamiquement. Les classes sont soit inclues dans le masque à la fabrication soit installées au travers d'un processus d'installation sur la carte après qu'elle ait été fournie. Les programmes qui s'exécutent sur la carte doivent seulement se référer aux classes déjà existantes sur la carte. Il n'y a aucune façon de télécharger des classes durant une exécution normale du code d'une application.

● Security Manager

Le gestionnaire de sécurité sur la plateforme Java Card diffère significativement de celui de la plateforme Java. Dans la plateforme Java, il y a une classe Security Manager (*java.lang.SecurityManager*) responsable des caractéristiques sécuritaires d'implémentation. Dans la plateforme Java Card les règles de sécurités du langage sont implémentées par la machine virtuelle. Il n'y a pas de classe Security Manager qui fabriquent des règles de décision pour autoriser des opérations.

● Ramasse-miettes et finalisation

La technologie Java Card ne requière pas de ramasse-miettes et n'autorise pas non plus la désallocation explicite des objets pour garder la contrainte de *pointer-safety*. C'est pourquoi le développeur d'application ne peut pas s'assurer que les objets qu'il alloue seront toujours désalloués. L'espace de stockage d'objets inaccessibles ne pourra pas nécessairement être réclamé. Quelques Java Card avancées fournissent un mécanisme de ramasse-miettes qui permet la destruction d'objet. La finalisation n'est pas nécessaire. *finalize()* n'est pas nécessairement appelé par la JCVM, et le programmeur ne doit pas dépendre de ce comportement.

● Threads

La JCVM ne peut pas supporter de multiple thread de contrôle. Un programme Java Card ne peut pas utiliser la classe Thread ou n'importe quel mot clé relatif aux threads du langage Java.

● **Clonage d'objet**

La plateforme Java Card ne supporte pas le clonage d'objet. La classe Object de l'API Java Card n'implémente pas la méthode clone, et ne fournit pas l'interface Cloneable.

● **Le contrôle d'accès des paquetages Java**

Le langage Java Card supporte le contrôle d'accès aux paquetages définit dans le langage Java. Pourtant, il y a des cas non supportés tels que les suivants :

- Si une classe implémente une méthode avec un accès de visibilité package, une sous-classe ne peut pas surcharger la méthode et changer l'accès de visibilité de la méthode en *protected* ou *public*.
- Une interface définie avec un accès de visibilité package ne peut pas être héritée par une interface avec un accès de visibilité *public*.
- Une méthode *public* ou *protected* d'une classe public ne peut pas contenir un paramètre formel de type référence à une classe de visibilité package.
- Une classe de visibilité package qui peut être héritée par une classe public ne peut pas définir de méthodes *public* et *protected*.

2.3.2 Caractéristiques Java supportées

● **Paquetages**

La plateforme Java Card suit les règles standards des paquetages de la plateforme Java. Les classes des APIs Java Card sont écrites comme des fichiers sources Java qui inclus les désignations des paquetages. Les mécanismes de paquetages sont utilisés pour identifier et contrôler les accès aux classes, aux champs statiques et aux méthodes statiques. Excepter la remarque faite sur le contrôle d'accès, les paquetages sur la plateforme Java Card sont utilisés exactement comme sur la plateforme Java.

● **Création dynamique d'objet**

Les programmes de la plateforme Java Card supporte la création dynamique d'objet, les instances de classes et les tableaux. Cette opération est réalisée comme d'habitude par l'opérateur *new*. Les objets sont alloués en dehors du tas. Comme souligné dans la remarque plus haut, la JCVM n'a pas nécessairement besoin de ramasse-miettes. Tous les objets créés

par la machine virtuelle peuvent continuer à exister et à consommer des ressources même après qu'ils soient devenus inaccessibles.

● **Méthodes virtuelles**

Comme les objets Java Card sont des objets du langage de programmation Java, l'invocation de méthodes virtuelles sur des objets dans un programme écrit pour la plateforme Java Card est exactement la même que celle d'un programme écrit pour la plateforme Java. L'héritage est supporté, incluant l'utilisation du mot clé *super*.

● **Interfaces**

Les classes Java Card peuvent définir ou implémenter des interfaces comme dans le langage de programmation Java. L'invocation de méthodes sur les types d'interfaces fonctionne comme attendu. La vérification de type et l'opérateur *instanceof* fonctionne aussi correctement avec les interfaces.

● **Exceptions**

Les programmes Java Card peuvent définir, lancer et attraper des exceptions, comme dans les programmes Java. La classe *Throwable* et ses sous classes importantes sont supportées. Quelques sous classes de *Exception* et *Error* sont omises, celles dont les exceptions ne peuvent pas apparaître dans la plateforme Java Card.

2.4 Les avantages apportés par JavaCard

Voici les principaux avantages du langage qui justifient le choix de ce standard en ce qui concerne la programmation des cartes à puces.

- **Langage de haut niveau orienté objets :** il possède donc tous les avantages de ce type de langage (encapsulation des données, héritage...). Il permet une structuration plus naturelle du code et donc plus compréhensible. Le concept d'héritage permet la réutilisation du code. La création de librairies est simplifiée : les packages. Auparavant, le développement de programmation carte était réalisé en assembleur ou en C. Cela était moins souple et plus technique. La mise au point de programmes sera plus rapide et les temps de développement seront sensiblement réduits. Ce langage accroît la sécurité : gestion strict de la mémoire, pas

d'arithmétique de pointeur, « access scope » (vérification des accès) des membres d'un objet, langage fortement typé.

- **Write Once, Run Anywhere :** le code d'un programme JavaCard est universel au sens où il est exécutable sur n'importe quelle machine virtuelle JavaCard. Le JavaCard n'introduit rien de spécifique et respecte la norme ISO-7816 dans le dialogue avec un lecteur de carte.

- **Plate-forme multi applicative :** JavaCard possède un modèle de sécurité qui permet à plusieurs applications de coexister en sécurité sur la même carte. Chaque applet possède son espace de mémoire propre. Une carte peut contenir une application bancaire, un accès pour télévision satellite, un carnet de santé… .

- **Partage de données entre applications :** les différentes applications présentes sur une carte peuvent accéder à des données d'une autre application si le programme a été envisagé. Seul le JCRE peut accéder à toutes les données. Une applet peut contenir les identifiants de sécurité sociale qui peuvent être utile pour une applet mis par la mutuelle du détenteur.

- **Sécurité des données :** Comme nous l'avons vu, le JCRE gère strictement la mémoire de chaque applet de façon étanche. L'API JavaCard intègre les outils de base pour des applications d'authentifications et de signatures. Les données sensibles sont détenues par le JCRE : certificat, code PIN… Nul ne peut les lire directement.

- **Souplesse :** Le fait que les applets soient chargées dans une EEPROM, permet de les mettre à jour en effaçant l'ancienne version et en chargeant la nouvelle. Cette évolutivité rend une carte pleinement réutilisable, alors qu'auparavant les applications étaient directement incluses dans le masque et donc non modifiables. Les modifications et chargement d'applet font appels à des mécanismes sécurisés par le « CardManager ».

2.5 Architecture d'une java card

D'un point de vue global, une Java Card peut être représentée comme un système comportant deux zones différentes. La première est figée : elle est constituée du système d'exploitation et des extensions éventuellement fournies par les fournisseurs de la carte. Elle contient le JCRE et le système d'exploitation sous-jacent (les services natifs). La deuxième est modifiable. Elle est constituée des applets chargées dans la carte.

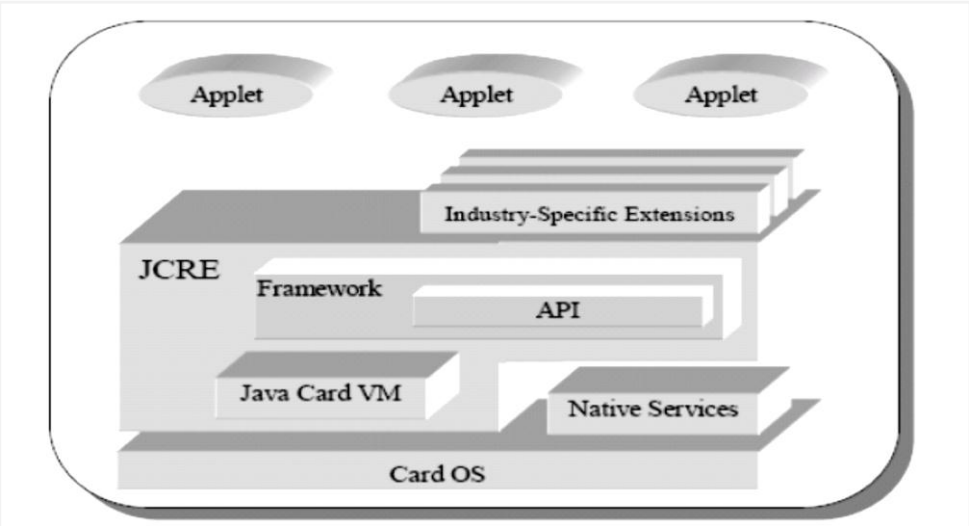

Figure II.1 Structure d'une JavaCard [NTK07].

L'application se situe exclusivement dans la zone modifiable, et ne communique qu'au travers du JCRE. En conséquence, la seule connaissance de l'interface du JCRE doit suffire pour permettre de développer des applets.

2.5.1 L'environnement d'exécution Java Card

L'environnement d'exécution Java Card (JCRE) gère les ressources de la carte, la communication avec le réseau, l'exécution des applets et leur sécurité. En fait c'est le système d'exploitation des smart cards. Le JCRE est situé au sommet de l'architecture d'une smart card. Il consiste en la machine virtuelle Java (l'interpréteur de bytecode), la structure des classes d'applications Java Cards (APIs), des extensions industrielles spécifiques, et les classes systèmes. Le JCRE sépare habilement les applets de la propriété de la technologie des smart cards des vendeurs. Les applets sont ainsi plus faciles à écrire et sont indépendantes de l'architecture privée des smart cards.

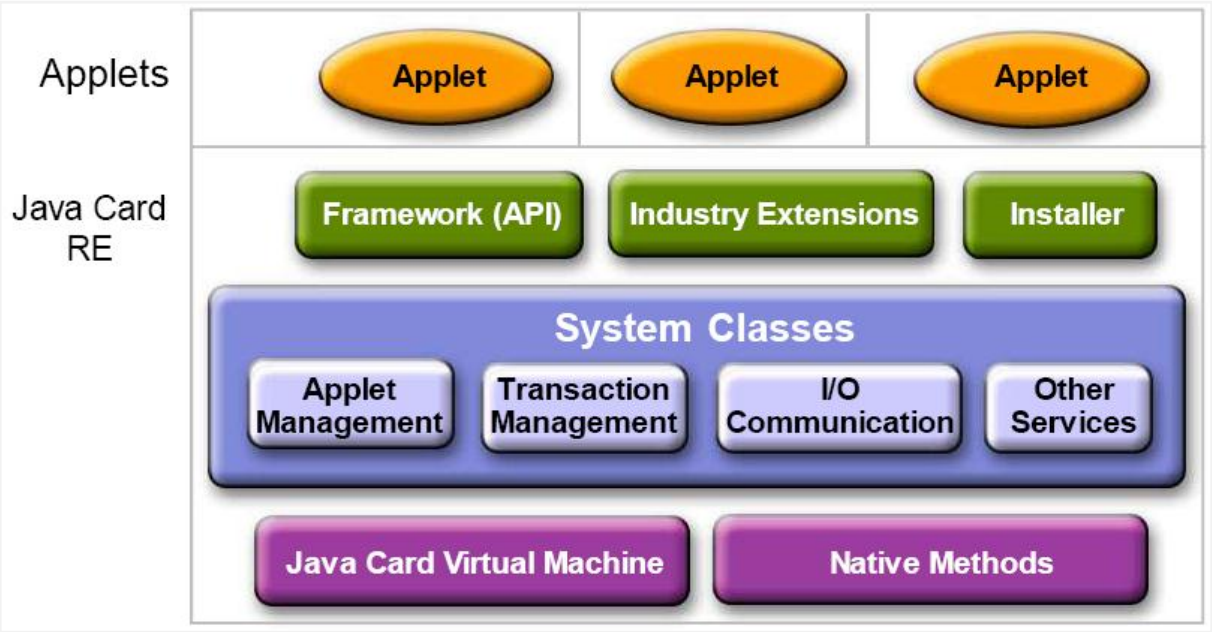

Figure II.2 La structure du JCRE [Lan08].

La période pendant laquelle la carte est insérée dans une machine est appellée session CAD (Card Acceptance Device). Durant une session CAD, le JCRE opère comme une smart card en établissant une communication APDU (Application Protocol Data Unit) d'entrée-sortie avec un serveur. Les APDUs sont des paquets de données échangés entre des applets et une application hôte. Chaque APDU contient soit un ordre du client à l'applet, soit une requête de l'applet au client.

Après l'initialisation du JCRE, celle-ci entre dans une boucle, en attente d'un ordre APDU du client Celui-ci envoie des ordres APDUs à la Java Card par l'intermédiaire d'une communication utilisant les points de contacts d'entrée-sortie. Quand une commande arrive, le JCRE sélectionne soit une applet à exécuter comme indiqué dans la commande, soit envoie directement la commande à une applet déjà sélectionnée. L'applet sélectionnée prend alors le contrôle et exécute la commande APDU. Une fois la commande exécutée, l'applet envoie une réponse à l'application hôte et redonne le contrôle au JCRE. Le procédé se répète à l'identique quand une nouvelle commande arrive.

Le cycle de vie du JCRE est le même que le cycle de vie de la SmartCard. Lorsqu'il n'y a pas de tension électrique, la machine virtuelle est suspendue, l'état du JCRE et des objets créés sur la carte est préservée. À la prochaine remise sous tension, le JCRE relance l'exécution de la machine virtuelle en chargeant les données depuis la mémoire morte. Durant la phase de

Reset (où la tension est enlevée), si une transaction n'a pas été terminée, le JCRE réalise tous les nettoyages nécessaires pour garder le JCRE dans un état constant (connu).

2.5.2 La Java Card Virtual Machine

La JCVM (JavaCard Virtual Machine) est composé en deux parties : le convertisseur, situé sur l'ordinateur, et l'interpréteur de byte code, situé dans la JavaCard. Ce découpage est dû principalement à la capacité mémoire limitée de la carte, notamment en mémoire volatile, ce qui permet de gagner de l'espace en mémoire et de la consacrer uniquement à l'exécution des applications

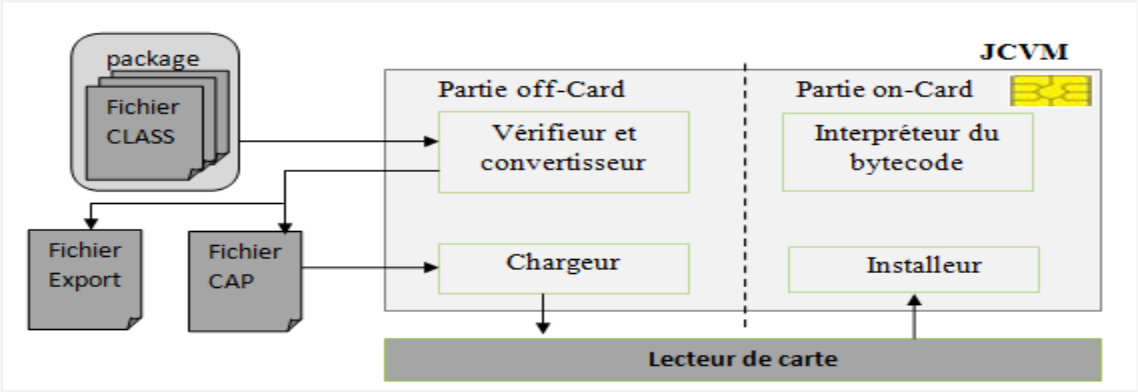

Figure II.3 L'organisation de la JCVM [Tsa 07].

Plusieurs applications pouvant exister sur une même JavaCard, la JVM se charge du partage des données. Certaines données peuvent être communes à toutes les applications hébergées mais une donnée privée à une application particulière ne pourra être accédée par aucune autre application. La JVM offre de plus le support du langage et masque les spécificités liées à la technologie de la carte puisqu'elle est située au dessus du système d'exploitation et des méthodes natives.

2.5.2.1 Le convertisseur

Contrairement à la JVM qui ne traite qu'une classe à la fois, l'unité de conversion du convertisseur est un package. Les fichiers *.class* sont produits par le compilateur java à partir du code source. Ensuite, le convertisseur prétraite tous les fichiers *.class* contenu dans un package et les convertit en un fichier CAP (Converted Applet). Le fichier CAP est ensuite chargé sur une Java Card 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 Java Card.
- 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.

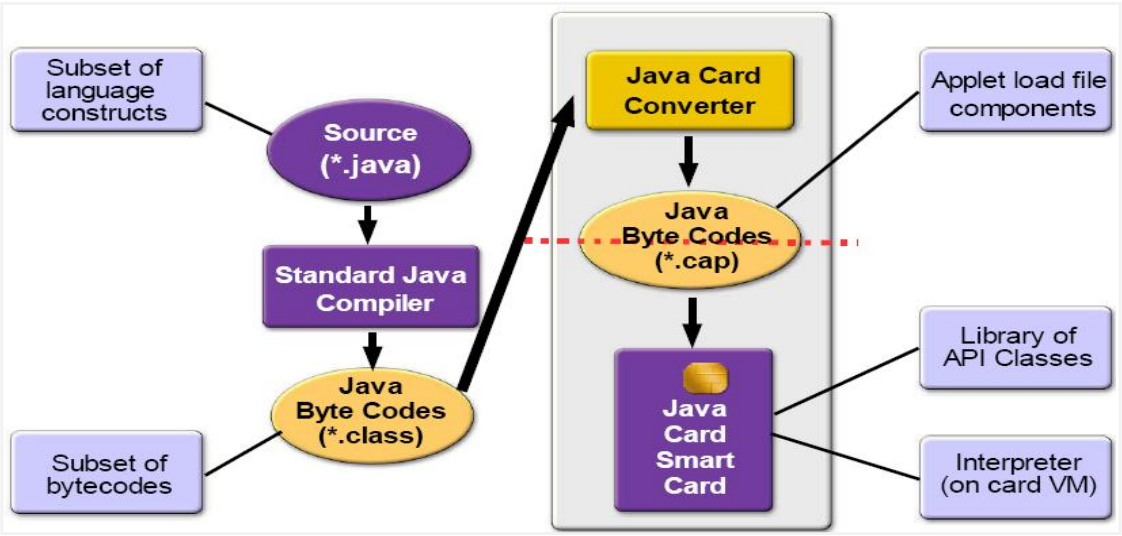

Figure II.4 Le convertisseur [Lan08].

2.5.2.2 L'interpréteur de byte code

L'interpréteur, en fournissant un support d'exécution du langage Java, permet aux applets d'être indépendantes du matériel. Il exécute en particulier les tâches suivantes :

– Il exécute les instructions du bytecode et des applets.

– Il contrôle l'allocation mémoire et la création d'objets.

– Il sécurise le temps d'exécution.

La machine virtuelle Java Card a été décrite comme comprenant le convertisseur et l'interpréteur. Elle est toutefois définie informellement comme étant la partie située sur la carte, à savoir l'interpréteur actuellement décrit. Ainsi concernant les Java Cards, on peut faire l'analogie entre l'interpréteur et la machine virtuelle, contrairement à la machine virtuelle Java qui elle, inclue le convertisseur et l'interpréteur.

2.5.2.3 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 de la carte.

2.6 Formats de fichier JavaCard

La technologie JavaCard introduit deux nouveaux formats binaires de fichiers qui permettent l'exécution d'applications JavaCard indépendamment de la plate-forme :

2.6.1 Fichier d'exportation **[Sun 1]**

Les fichiers d'exportation ne sont pas chargés sur les JavaCards et ne sont donc pas directement utilisés par l'interpréteur. Ils sont produits et consommés par le convertisseur et peuvent être utilisés plus tard pour la conversion d'un autre package qui importe les classes du package associé. On peut considérer les fichiers d'exportation comme les fichiers .h du langage C ; ils ne contiennent en effet que des informations sur les APIs publiques des classes et des signatures de méthodes et de champs. Ils peuvent donc être laissés en accès libre aux utilisateurs de l'applet sans pour autant en révéler son code source.

2.6.2 Fichier CAP (Converted APplet)

Le fichier .CAP contient une représentation binaire exécutable qui contient les classes, les méthodes, les *import* d'un package Java. Le format CAP est le format dans lequel les applets sont chargés sur la Java Card **[Sun 1]**.

Il repose sur la notion de composants. Onze composants standards définissent le fichier CAP. Ils sont, dans l'ordre de chargement :

1. le *Header* contient des informations générales sur le paquetage contenu dans le fichier CAP : son nom, les versions de la machine virtuelle nécessaires pour ce paquetage,

2. le *Directory* contient la taille de tous les composants contenus dans le fichier CAP. Il indique si le paquetage importe des librairies et s'il définit une applet. Il contient aussi la liste et la taille des composants personnalisés,

3. l'*Import* contient la liste des paquetages extérieurs utilisés par les classes définies dans ce paquetage,

4. l'*Applet* contient la liste des applets définies dans ce paquetage,

5. le *Class* contient la description des classes et des interfaces définies dans ce paquetage,

6. le *Method* contient le code de chaque méthode définie dans ce paquetage,

7. le *StaticField* contient toutes les informations requises pour créer et initialiser une image des champs statiques définis dans ce paquetage,

8. l'*Export* liste tous les éléments statiques qui peuvent être importés par des classes définies dans d'autres paquetages,

9. le *Constant Pool* liste les différentes entrées pour chaque classe, méthode ou champ référencé par élément dans le composant *Method*,

10. le *Reference Location* représente la liste des décalages dans le composant *Method* vers des éléments qui ont des points d'entrées dans le composant *Constant Pool*,

11. le *Descriptor* contient les informations pour parcourir et vérifier tous les éléments du fichier CAP.

Le fichier CAP n'a pas été vraiment pensé pour la vérification. En effet, même dans la spécification de Sun, le composant *Descriptor* qui est le seul à pouvoir permettre la vérification de typage, est indiqué comme un composant optionnel. Dans les faits, ce composant n'est jamais chargé dans les Java Card actuelles. Cela se comprend car il n'est pas utile à l'exécution de l'application, seulement à sa vérification. En marge des onze composants, le standard Java Card propose un système de personnalisation de composants, permettant d'ajouter des composants. Ces composants n'étant pas standards, leur développeurs ne peuvent pas s'attendre à ce que d'autres implémentations de Java Card puissent les reconnaître.

2.7 Les API Java Card

Les APIs Java Card consistent en un ensemble de classes spécialisées dans la programmation d'applications de smart card conformément à la norme ISO 7816. Les APIs contiennent trois packages principaux et un package d'extension. Les trois packages principaux sont les

packages *java.lang, javacard.framework,* et *javacard.security*. Le package d'extension est *javacardx.crypto*.

Les classes dans ces APIs sont compactes et succinctes. Elles incluent des classes adaptées à la plate-forme 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.

- **Le package java.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 Java Card 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 Java Card. Plus important, il définit la classe de base Applet qui donne le cadre d'exécution et d'interaction des applets durant leur fonctionnement. Son rôle est similaire à celui de la classe Applet d'un browser Internet. Une autre classe importante de ce package est la classe APDU qui sert aux protocoles de transmissions.

- **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 cryptographies et des interfaces dont les spécifications sont régulées par les Etats-Unis. Il définit la classe abstraite Cipher pour l'encryptions et la décryptions des données.

❖ **Le package java.io**

Ce package est un sous ensemble strict de celui de la plate-forme Java. Il ne difinit qu'une exception IOException pour indiquer un problème de communucation. Celle-ci est n'est présente que pour les besoin de JCRMI qui en hérite au traves de la classe RemoteExeception du package java.rmi.

❖ **Le package java.rmi**

Ce package définit l'interfaces Remote qui identifie les interfaces dont les méthodes peuvent être appelées par des applications clientes du CAD. Il définit également l'exception RemoteException.

Dans le développement d'application à base d'une 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.

2.8 Le firewall d' applet [Sun 2]

Toute implémentation du JCRE devrait supporter l'isolation des contextes d'applets. L'isolation signifie qu'une applet ne peut accéder aux champs des objets d'une autre applet, à moins que cette dernière fournisse une interface d'accès (interface *shareable*). Cette isolation est possible grâce au mécanisme du firewall.

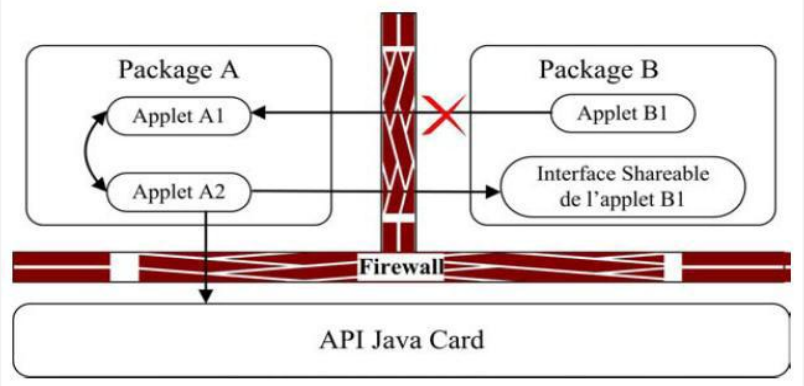

Figure II.5 Le firewall et contexte JavaCard [Kar 08].

Le firewall est une protection spécifique à JavaCard, il est indépendant de la technologie Java et procure 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.

2.9 Les applets

2.9.1 Flux de données

Voici un schéma représentant les différentes étapes pour créer une applet et l'exécuter.

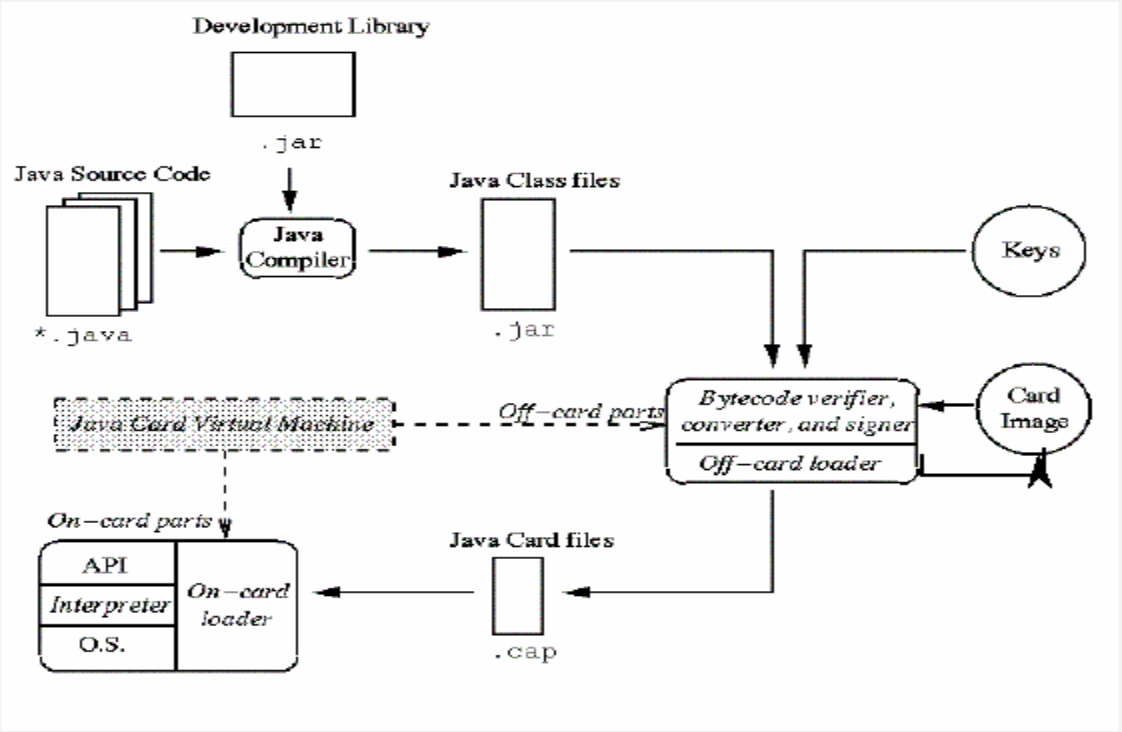

Figure II.6 Flux de données [NTK07].

Au départ, les applets .java et les librairies sont compilées pour créer des fichiers .class (équivalent .jar). Ensuite, ce fichier est convertit en fichier CAP (« converted Applet ») après vérification des bytecodes. Ce .CAP est ensuite téléchargé dans la carte à puce pour être exécuté.

2.9.2. Types d'applets

Une JavaCard, à l'origine, contient le JCRE et tout le système du constructeur de la carte. Le JCRE est « écrit » dans la ROM de la carte : c'est le masque de la carte. Il est donc inviolable. Le code des applets est stocké sur la carte généralement en EEPROM. Pour des besoins

spécifiques, des applets peuvent être placées dans la ROM en même temps que le JCRE. Ces applets sont dites prés issus, les autres sont dites post issus.

a) Applet pré issue

Les applets pré issus sont différentes des autres car elles ne sont pas soumises aux mêmes restrictions. Elles peuvent déclarer des méthodes natives et les utiliser dans leur code. Ces méthodes natives sont écrites en C ou en assembleur. Ces méthodes natives ne sont pas soumises aux dispositifs de sécurité du JCRE. Les instances de ces applets sont obligatoirement stockées en EEPROM, tout comme celles des applets post issus. La création de ces instances peut être fait à l'initialisation du JCRE ou plus tard.

b) Applet post issue

L'installation d'une tel applet se fait alors que le JCRE tourne. Cela se passe en deux phases : chargement du code de l'applet (CAP file) sur la carte puis création d'une instance de l'applet. Pour installer l'applet, le « off card » installeur prend le fichier .CAP et le transforme en une séquence d'APDU qui télécharge le contenu du fichier sur la carte.

Le « on card » installeur écrit le contenu dans l'EEPROM et fait les liens entre les classes dans le fichier .CAP et les classes résidant sur la carte (Standard Class Libraries dans la ROM). Il crée et initialise les données qui seront utilisées par le JCRE en interne pour se servir de l'applet. Si l'applet nécessite plusieurs packages, chaque fichier .CAP correspondant à un package sera chargé sur la carte. Ensuite, il est possible de créer une instance d'applet et de l'enregistrer auprès du JCRE. L'applet ne peut être activée avant ces deux opérations.

Pour créer et enregistrer une applet, le JCRE fait un appel automatique de la méthode *install ()*, qui ne sera exécutée qu'une seule fois. Cette méthode doit être surchargée pour pouvoir y mettre les déclarations des variables ou objets qui existeront pendant la durée de vie de l'applet. La méthode *register ()* doit être absolument appelée car elle sert à enregistrer l'applet dans l'environnement d'exécution JavaCard. C'est le « card manager » qui enregistre grâce à cette méthode l'instance de l'applet.

2.9.3 Méthode de développement

On peut séparer l'écriture d'une applet JavaCard en plusieurs phases :

• Spécifications des fonctions de l'applet

• Attribution des identifiants d'application (AIDs) à l'applet et au package contenant ses classes.

• Elaboration de la structure des classes de l'applet

• Définition de l'interface entre l'applet et l'application hôte.

2.9.3.1. Spécification des fonctions de l'applet

Nous allons donner l'exemple d'un porte-monnaie électronique qui doit pouvoir traiter le crédit, le débit, la vérification du solde et un code PIN. L'utilisateur de la Smart Card doit donc par exemple pouvoir ajouter de l'argent, en retirer et demander le montant de son capital. L'applet doit également être sécurisée pour empêcher tout autre utilisateur de se servir de la carte. L'utilisateur devra donc entrer un code PIN sur le clavier connecté au CAD. Le code PIN devra être vérifié avant toute transaction.

2.9.3.2 AIDs de l'applet

Les classes Java de l'applet sont définies dans un seul package. Ci dessous sont définies les AIDs (Applet Identifier) de l'applet du porte-monnaie électronique. Une AID est constituée de deux parties : un RID et un PIX. Le RID est celui de Sun MicroSystems. On doit en demander un à l'International Standards Organization (ISO). Le PIX peut lui être choisi directement par nous même.

Champs	Valeur	Taille
RID	0xA0 0x00 0x00 0x00 0x62	5 octets
Package PIX	0x00 0x01 0x0D 0x06	4 octets
Applet PIX	0x00 0x01 0x0D 0x06 0x01	5 octets

Tableau II.2 AIDs de l'applet

2.9.3.3. Définitions des fonctions et de la structure de classe

Une classe implémentant une applet JavaCard doit dériver de la classe javacard.framework.Applet. De plus, une applet JavaCard doit supporter quelques méthodes indispensables (install, register, process, select, deselect).

- **install ()** : La fonction *install ()* est appelée par le JCRE pour créer et enregistrer (par la méthode *register()*) une instance d'applet auprès de celui-ci. Les AID servent d'identificateur pour le JCRE. Chaque AID d'une applet présent sur la carte doit être unique et correspondre à la norme ISO-7816. Une applet est sélectionnable seulement après avoir été enregistré par le JCRE.

- **process ()** : La fonction *process ()* est la fonction principale de l'applet, c'est celle qui va collecter toutes les APDU entrantes. Elle envoie ces APDU vers les méthodes qui correspondent à l'octet d'instruction (INS). Le principal travail d'un programmeur se situe dans la gestion du dialogue entre la carte et le lecteur. C'est dans cette méthode qu'il doit être géré.

- **select ()** : La fonction *select ()* est appelée par le JCRE pour rendre active et initialiser une instance d'applet. Comme définie dans la norme ISO-7816, toutes les applets doivent rester sélectionnables.

- **deselect ()** : La fonction *deselect ()* est appelée par le JCRE pour désactiver l'applet en-cours et faire les possibles opérations nécessaires avant la désélection (Ex : Clôture d'une transaction).

- **register ()** : La fonction Register () permet l'enregistrement de l'applet dans l'environnement d'exécution JavaCard.

```
import javacard.framework.APDU;
import javacard.framework.Applet;
import javacard.framework.ISOException;

public class applet extends Applet {

      private applet() {
      }
      public boolean select () {
            return (true);
      }
      public void deselect () {
      }

public static void install(byte bArray[], short bOffset,byte
bLength)

                  throws ISOException {
            new applet().register();
      }

      public void process(APDU apdu) throws ISOException {

      }
}
```

Figure II.7 Code de base d'une applet.

Donc, pour développer une applet, cela revient à :

1. Créer une sous classe qui hérite de la classe *Applet***.**
2. Implémenter les méthodes *select, deselect, install* et *process*.
3. Déclarer et initialiser les octets CLA et INS (instruction). Ces deux variables sont de type *final static*, car leurs valeurs sont inchangées. A la réception d'une commande APDU, la méthode *process* exécute la fonction dont l'INS correspond à celui de l'APDU.
   Exemple :

```
public static final byte CLA_MONAPPLET = (byte) 0xB0;
public static final byte INS1 = (byte)0x01 ;
public static final byte INS2 = (byte)0x02 ;
.
.

public void process(APDU apdu) throws ISOException {

 byte[] buffer = apdu.getBuffer();

  if (this.selectingApplet()) return; /*indique si l'applet
est déjà sélectionnée*/

      switch (buffer[ISO7816.OFFSET_INS]) {
      case INS1:
       fonction1;
       break;
       .
       .
       case INS2:
       fonction2;
       break;
      default :

      ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);

                        }
      }
```

Figure II.8 Exemple d'implémentation de la méthode process.

4. Il est préférable d'allouer toutes les ressources dans le constructeur de l'applet. La création d'objet par l'opérateur *new* en Runtime est techniquement possible mais vivement déconseillée car les java cards n'embarquent pas de *garbage collector* (ramasse miette), ce qui fait que toute ressource allouée (par *new*) sera sauvegardée en mémoire persistante (EEPROM) et seule la suppression de l'applet pourra les libérer.

5. Vérifier la longueur des données entrantes. Dans le cas échéant, soulever une exception.

6. Caster toutes les valeurs non typées, par exemple : la valeur 5 = *(byte)* 5 ou *(short)* 5.

7. Récupérer la commande *apdu* dans un buffer (un tableau de *byte*).

2.10 Les applications courantes utilisant la JavaCard [Tsa 07]

On peut estimer le nombre de JavaCards fonctionnelles à plus d'un milliard. Elles sont utilisées dans plusieurs domaines d'application :

- Monétique :

- ✓ Carte bancaire.
- ✓ Porte monnaie (ex : Moneo en France).

- Identification :

- ✓ Carte d'identité nationale (ex : eiD en Belgique).
- ✓ E-passeport.

- **Enseignement :**

- ✓ Carte d'étudiant

- **Téléphonie mobile :**

- ✓ Carte SIM

- **Secteur médical :**

- ✓ Carte Vitale (en France)
- ✓ Carte SIS (en Belgique)
- ✓ Carte Chifa (Algérie)

- **Transport :**

- ✓ Passe Navigo (à Paris)
- ✓ Oyster (à Londres)

- **Sécurité informatique** (authentification forte et signature électronique) : Dans ce cas, la carte contient un crypto-processeur pour la génération des clés et le stockage de la clé privée.

2.9.11 Conclusion

Les JavaCards sont des cartes à puce possédant une plate-forme ouverte basée sur la machine virtuelle, ce qui rend les applications embarquées indépendantes du hardware. En plus de leur interopérabilité, elles définissent une plate-forme sécurisée qui met le développement d'applications à base de cartes à puce à la disposition de n'importe quel programmeur. S'ajoute à cela, la possibilité de charger et de mettre à jour plusieurs applications potentiellement communicantes.

Après avoir présenté un état de l'art sur la Java Card et les cartes à puce, nous allons décrire dans le chapitre suivant l'approche réactive au-dessus de la plateforme javaCard.

CHAPITRE III

Implémentation de l'approche réactive au-dessus de Java Card

Chapitre III

L'approche réactive au-dessus de Java

3.1 Introduction

Les insuffisances des systèmes transformationnels et des systèmes interactifs ont conduit à l'introduction de la notion de système réactif **[HP]**. Les systèmes transformationnels utilisent un ensemble de données d'entrée, pour effectuer un traitement sans interaction ave le monde extérieur. Après avoir fourni un résultat ils sont prêts pour un autre cycle. Les systèmes interactifs interagissent avec leur environnement mais le font à leur propre rythme. Par exemple, les interfaces graphiques et la plupart des systèmes d'exploitation sont des systèmes interactifs. Des événements trop rapprochés peuvent être perdus si le système est surchargé. Les systèmes réactifs, quant à eux, réagissent au rythme des événements émis par l'environnement. Tous les événements extérieurs sont pris en compte et c'est au système de choisir ceux qui ne seront pas traités, s'il est surchargé. Il fonctionne alors en mode dégradé.

La programmation utilisant les "threads" classiques, en Java par exemple, est extrêmement difficile et sujette à erreurs, pour une raison essentielle qui est que le programmeur ne maitrise pas leur ordonnancement. La sémantique du programme apparaît donc comme non-déterministe et oblige le programmeur à installer des verrous dans son code pour protéger la manipulation des données partagées. Le code ne peut donc être modulaire – on ne peut réutiliser du code qui n'est pas conçu pour le "multithreading" –, il est aussi inefficace – à cause des protections sur les données –, et les risques d'inter-blocage sont grands. De plus, la coordination entre les threads est illusoire, car le résultat dépend en fait de l'ordonnancement choisi par le "scheduler".

La programmation réactive offre une alternative au "multi-threading" classique : le parallélisme y est déterministe, la communication par évènements diffusés rend le code plus modulaire.

Notre objectif est d'adapter le noyau réactif issu du projet de Frédiric Bussni à INRA pour prendre en compte le cas de la Java Card qui est caractérisé, comme les systèmes embarqués en générale, par des ressources extrêmement limitées en termes de mémoire et de puissance de calcul, ce qui nous conduit à faire des choix sur les aspects de cette approche de programmation à prendre en compte , dans notre cas la manipulation du parallélisme et des évènements.

3.2 Approche Synchrone

L'Approche Synchrone est fondue sur l'hypothèse que le système réagit en temps zéro, c'est-à-dire que les sorties sont simultanées avec les entrées qui les ont provoquées. Ceci peut engendrer des problèmes de causalité, c'est-à-dire des programmes incohérents qui, détectant l'absence d'un événement lors d'un instant, réagissent en générant ce même événement au cours du même instant **[RA01]**.

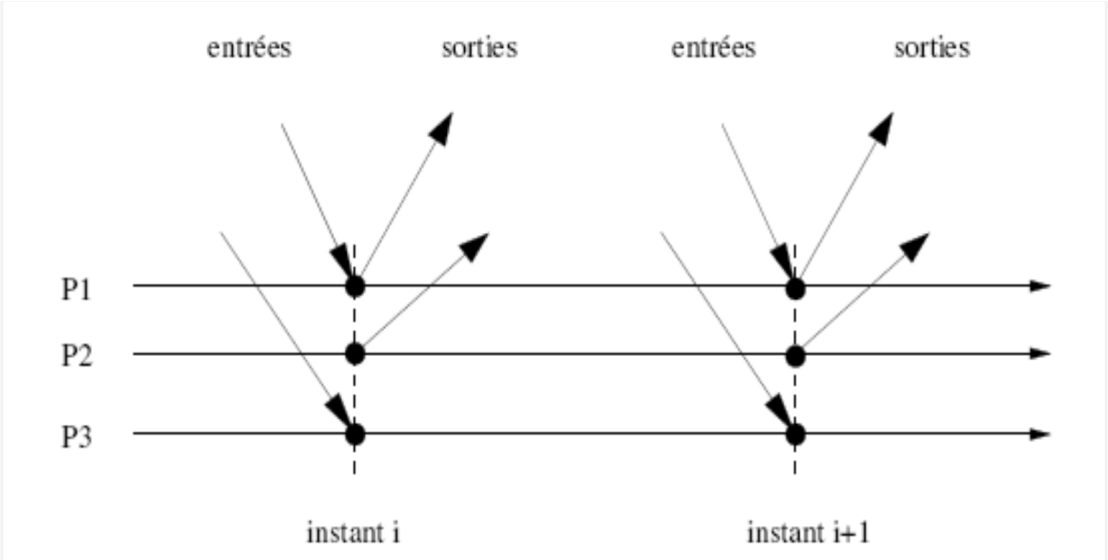

Figure III.1 L'approche Synchrone [RA03].

3.2.1 Langages synchrones

En utilisant les concepts réactifs synchrones, il a été créé un ensemble important des langages qui utilisent la réactivité. Ces langages réactifs synchrones peuvent être classés selon le paradigme de programmation utilisé:

ESTEREL est un langage impératif qui implémente la notion d'instant (de durée nulle), la diffusion instantanée d'événements et la réaction instantanée à l'absence d'un événement. Ce dernier concept provoque l'apparition des problèmes de causalité qui sont rejetés par ESTEREL. Par ailleurs, la création dynamique des nouveaux composants est interdite en Esterel. **SL** est une variante d'ESTEREL dans laquelle la réaction à l'absence d'un évènement est retardée à l'instant suivant, pour éviter les problèmes de causalité. **Reactive-C** qui est une extension de **C** a été utilisé pour implémenter SL. **Lustre**, développé au laboratoire *Verimag de Grenoble*, est un langage de type flot de données. Ce langage adopte un style déclaratif dans lequel les systèmes sont décrits par un ensemble d'équations. **Signal**, développé par l'*Irisa*, est un langage de type flot de données comme Lustre. **StateCharts**, développé par *David Harel* est un formalisme pour spécifier graphiquement les systèmes réactifs. Argos s'inspire de StateCharts avec comme différence principale l'utilisation d'un opérateur de composition hiérarchique. **SyncCharts** qui s'inspire de StateCharts est développé à l'*Université de Nice* et produit du code **ESTEREL**. Parmi les langages fonctionnels, on trouve **Lucid Synchrone** qui est un langage flot de données qui combine les caractéristiques de Lustre et la programmation fonctionnelle de ML. Il existe également une **Extension réactive à Standard ML**. Pour les langages orientés objets, en particulier Java, on trouve des bibliothèques comme les **SugarCubes** ou **Junior** et des extensions Java comme. Finalement, pour les langages interprétés, on trouve les **Reactive Scripts** qui ont été implémentés au-dessus des SugarCubes.

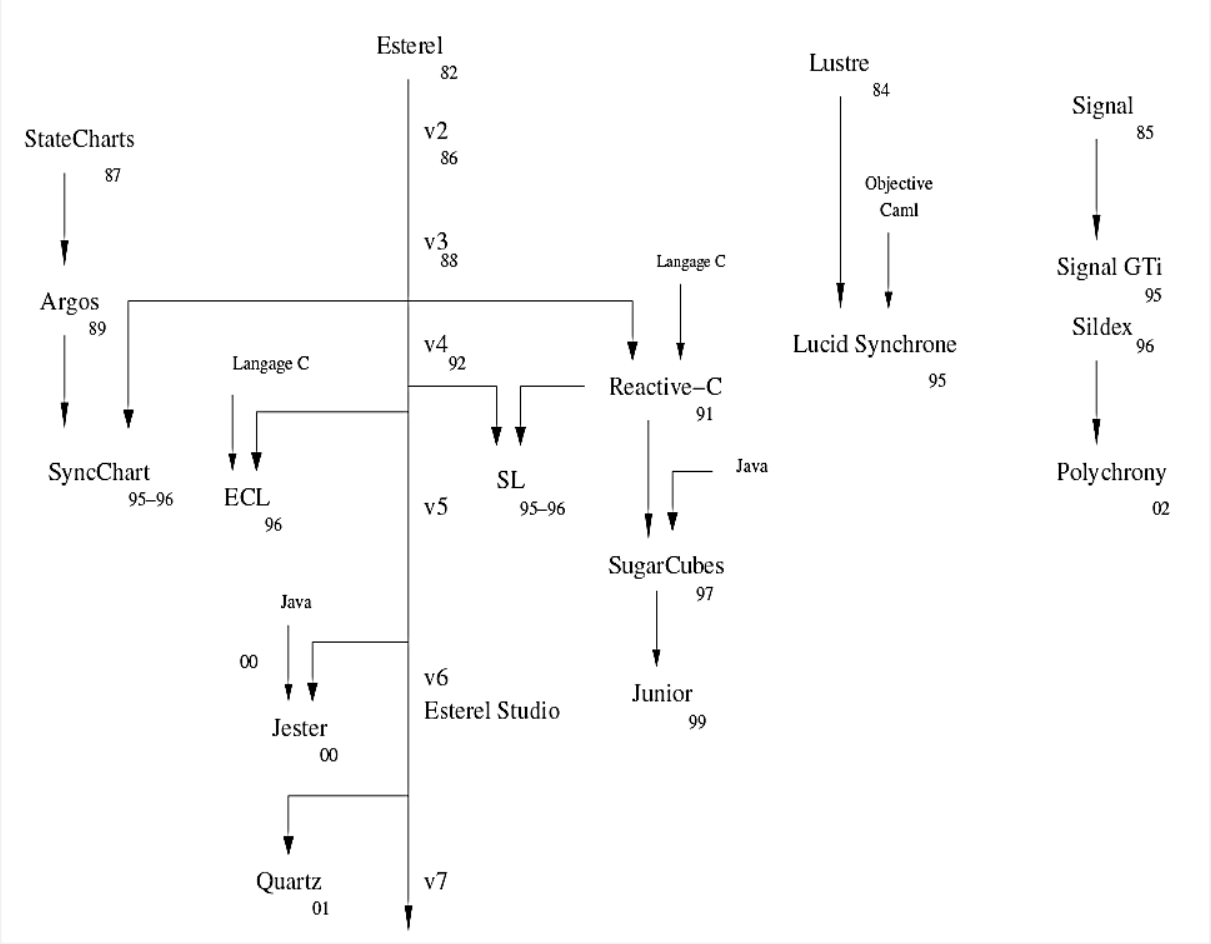

Figure III.2 L'évolution des langages synchrones [RA03]

3.3 Approche réactive

En 1985, D. Harel et A. Pnueli [HP] ont proposé un nouveau paradigme: les systèmes réactifs. Il s'agit de systèmes qui ne sont pas censés terminer pour produire des résultats contrairement à ceux de l'approche classiques présentée précédemment. Ces systèmes réagissent de façon continue, à toute modification de leur environnement, en influant en retour sur ce même environnement. Ce sont donc des programmes cycliques qui entretiennent une interaction étroite et continue avec leur environnement.

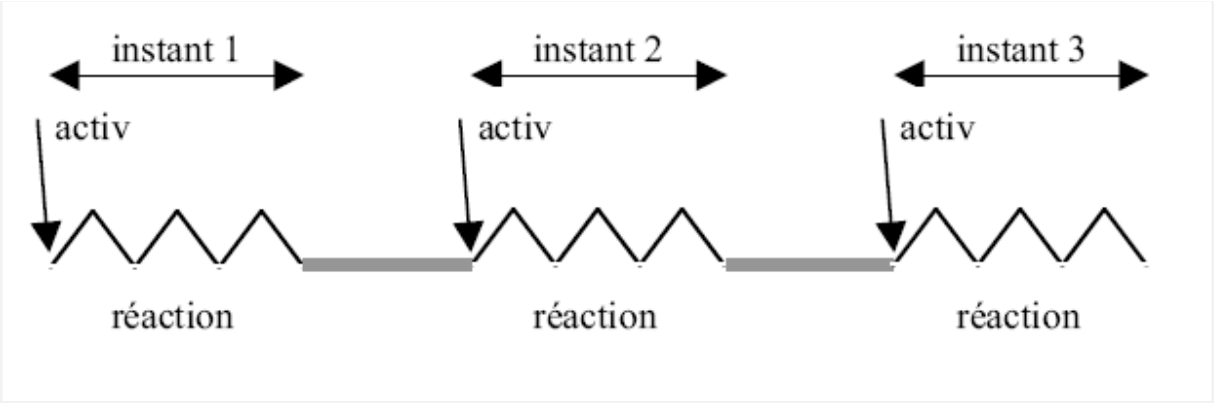

Figure III.3 L'évolution d'un système réactif [RA00].

Les systèmes réactifs se conçoivent naturellement dans un cadre concurrent ou parallèle. Chaque composant, s'exécutant en parallèle, réagit à une perturbation de son environnement d'exécution en produisant à son tour une modification de l'environnement qui fera réagir d'autres composants. Le développement des formalismes synchrones s'inscrit dans cette optique. **[JFS01]**

L'Approche Réactive que nous allons considérer dans ce document est définie par F.Boussinot dans l'ouvrage "La programmation réactive". 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. Dans ces conditions, la souplesse obtenue, grâce au caractère dynamique des systèmes ainsi construits, se paye par une perte d'expressivité (plus ou moins dommageable en fonction du type de systèmes considérés) et l'impossibilité d'établir des preuves sur les aspects dynamiques.

L'approche réactive=Instant+ Concurrence + Diffusion d'événement + Dynamisme

L'Approche Réactive poursuit les objectifs suivants :

• 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.

3.4 Définitions

3.4.1 La notion d'instant [CB04]

Chaque activation du système réactif définit une étape dans son évolution. On appelle ces étapes « instants ». Les instants définissent ainsi une base de temps abstraite (une sorte d'horloge virtuelle) pour le système. Il est important de noter que, à ce niveau, la durée effective des instants peut être quelconque. La durée d'un instant est donc 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. L'évolution d'un système réactif au cours du temps est représentée sur la figure suivante :

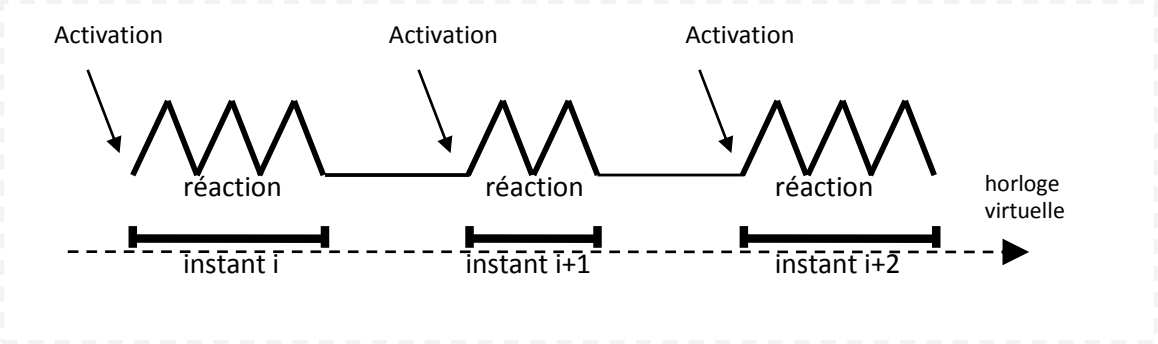

Figure III.4 La notion d'instant.

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

3.4.2 Concurrence

Comme pour l'*Approche Synchrone*, la notion d'instant fournit un moyen de définir une sémantique claire de la concurrence ainsi chaque composant en parallèle exécute un bout de son programme au cours d'un instant. Mais, le fait que ces instants aient une durée permet d'appréhender le problème sous une forme différente, c'est-à-dire, la fin d'un instant est effective lorsque tous les composants parallèles du système ont terminé pour l'instant courant. Ainsi, tous les composants se synchronisent à chaque fin d'instant **[JFS01]**.

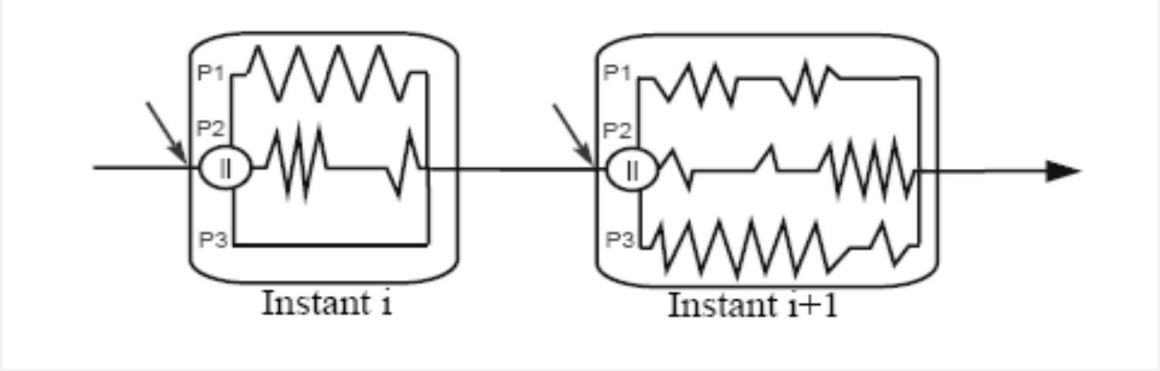

Figure III.5 La concurrence [CB04].

Dans la figure **III.5**, P1, P2 et P3 représentent chacun un des composants qui sont exécutés en parallèle dans le système réactif. Chacun des composants évolue à sa manière aux modifications apportées à l'environnement d'exécution du système. La découpe en micro-étapes peut aussi être appliquée à l'exécution de chacun des composants exécutés en parallèle **[CB04]**.

3.4.3 Diffusion d'événement

La notion d'instant logique permet de définir un puissant moyen de communication entre les composants parallèles d'un système réactif ; la communication par diffusion instantanée de signaux ou d'événements (l'Approche Réactive utilise en général le terme d'événement). Ce type particulier de communication a été introduit par le langage Esterel et présente les propriétés suivantes:

- **instantanéité**: un événement est un message échangé entre composants parallèles. Cet événement est en fait un élément de l'environnement d'exécution du système réactif dont la durée de vie (présence dans l'environnement) est celle d'un instant. Il s'agit d'une donnée non persistante entre instants dont l'état sera redéfini à chaque nouvel instant.

- **diffusion**: cet événement est vu à l'identique par l'ensemble des composants s'exécutant en parallèle dans le système à l'instant de sa génération. Autrement dit, pour le même instant, deux composants parallèles ne peuvent avoir deux informations différentes sur le même événement; c'est-à-dire que l'on ne peut avoir un composant qui voit un événement présent, tandis qu'un autre le voit absent au cours du même instant.

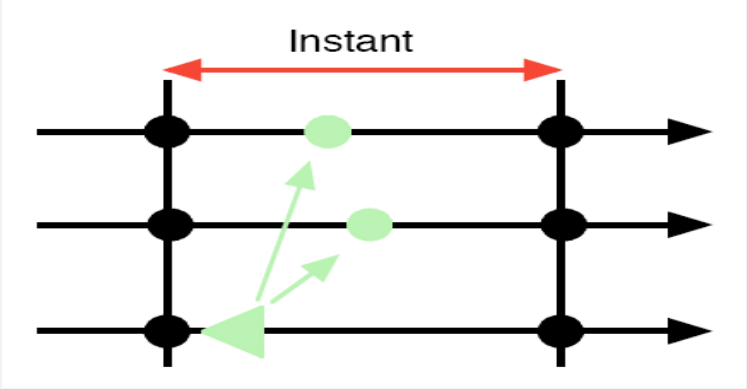

Figure III.6 La diffusion d'événement [JFS01].

3.4.4 Dynamisme

Le dynamisme concerne la possibilité d'ajout et/ou de retrait de composants en cours d'exécution. Les ajouts et les retraits de composants ne sont pas des opérations instantanées. La notion d'instant permettant de garantir la stabilité du système à la fin de celui-ci, ces opérations sont uniquement effectuées entre deux instants, ainsi aucun problème de causalité ne peut apparaître dans la combinaison de plusieurs composants en parallèle, donc dans l'ajout de composants **[CR04]**.

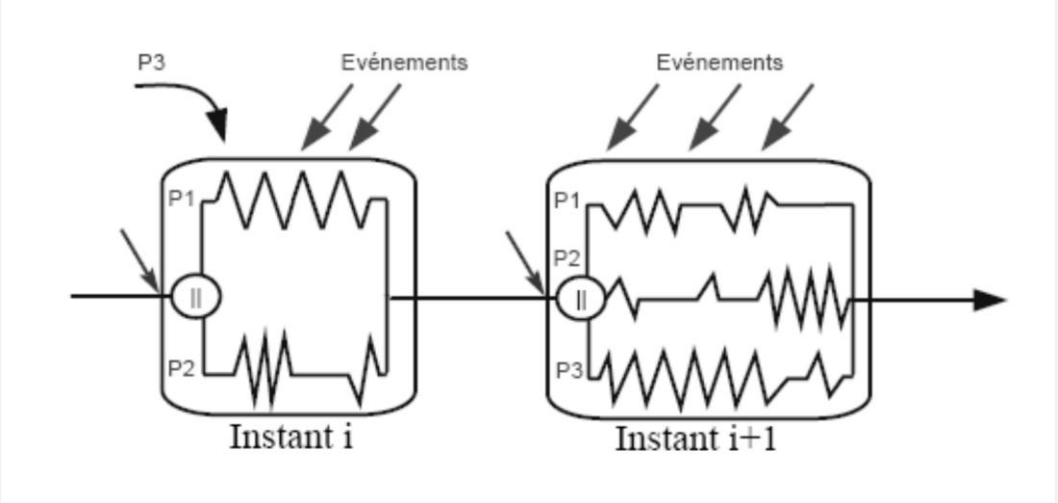

Figure III.7 Le dynamisme [CB04].

Le retrait d'un composant peut être considéré comme une préemption de ce composant. À la différence d'Esterel, le modèle réactif ne permet pas la préemption forte, c'est-à-dire qu'il n'est pas possible d'interrompre définitivement l'exécution d'un comportement qui n'a pas atteint un état stable au moment ou un événement est généré. Le modèle réactif permet uniquement la préemption faible et ne permet pas de stopper immédiatement l'exécution d'un programme au moment ou un événement est généré. Le programme doit atteindre un état stable pour l'instant courant avant d'être préempté, c'est-à-dire qu'il doit avoir fini l'exécution de son comportement pour l'instant courant. Le modèle réactif propose deux mécanismes de retrait de composant :

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

Un système réactif peut ainsi être reconfiguré dynamiquement sans aucun problème entre deux réactions consécutives par ajout ou retrait de composants parallèles en cours d'exécution. Cette caractéristique dynamique est une particularité de l'Approche Réactive qui la rend adaptée à la programmation des systèmes présentant par nature un fort caractère dynamique, tels que les systèmes distribués, les systèmes à base d'agents ou les interfaces graphiques **[JFS01]**.

3.5 Les formalismes réactifs au dessus de java

L'approche réactive découle des travaux menés sur les langages synchrones, principalement Esterel. Les premiers formalismes engendrés par cette approche ont été initialement construits aux dessus de langages procéduraux tels que **C** et **Tcl-Tk**. Les principaux formalismes qui ont été développés dans ce cadre sont : **Reactive-C** ; ce langage ajoute un ensemble de primitives réactive, au langage C, Les **Objets Réactifs** ; Il s'agit d'avantage ici d'un modèle de programmation, visant à intégrer les concepts de la programmation objet avec les principes de l'Approche Réactive, Les **Objets Réactifs Distribués** ; il s'agit d'une variante du modèle **ROM**, dans laquelle on considère des systèmes distribués, formés d'îlots synchrones communiquant entre eux de manière asynchrone, Les **Scripts Réactifs**. Ils apportent toute la souplesse et le dynamisme d'un langage interprété à l'Approche Réactive, aussi les programmes exécutés en parallèle par l'interprète sont ajoutés dynamiquement en cours d'exécution, et en fin, les **Icobjs** et en particulier la **programmation graphique par Icobjs** constituent un formalisme particulier, permettant d'exprimer graphiquement la programmation par *combinaison de comportements élémentaires.*

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, le langage JAVA est certainement l'un des premiers langages de programmation orienté objet, largement répandu, capable de prendre en considération les problèmes de concurrence directement au niveau du langage. Il possède :

- un modèle objet simple et facile à appréhender (héritage simple, tout est classe et objet…).

- Une *API* très riche permettant un développement rapide en se concentrant sur l'essentiel. Notamment il dispose des *API* nécessaire à la programmation de la JavaCard, ce qui est adapté à notre cas.
- est un langage à bytecode qui en fait un langage particulièrement portable et ce quelques soient les plates-formes considérées.
- Etc.

Cependant, le modèle de concurrence proposé ici reste basé sur un modèle de threads il ne constitue pas un cadre de programmation idéal 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 à l'exemple de **JavaTriveni, Reactive Java, Jester**, les **sugarcubes**…, etc.

Nous allons voir dans cette section les formalismes réactifs au-dessus de java de faciliter l'interfaçage entre les formalismes réactifs et les objets du langage Java et d'exprimer dans un programme la concurrence et la communication entre activités parallèles sans recourir aux threads. Ainsi, nous présentant leurs avantages et leurs inconvenants puis nous allons choisir la meilleur implémentation que l'on pourra adapter à la plate forme JavaCard.

3.5.1 Les SugarCubes :

Les SugarCubes forment une bibliothèque Java permettant l'implémentation de systèmes réactifs dynamiques et modulaires. Cette bibliothèque apporte une alternative au mécanisme standard des threads Java dans la programmation de systèmes concurrents.

Ils proposent une implémentation originale de l'Approche Réactive au-dessus de Java dans laquelle les primitives réactives sont implémentées par des objets. Il ne s'agit donc pas à proprement parler d'un langage à part entière avec une grammaire spécifique. Chaque primitive est décrite par une classe Java particulière. En effet, 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.

En SugarCubes, on parlera d'activation pour décrire l'exécution d'une micro-étape, tandis que l'on parlera de réaction pour décrire l'exécution d'un instant logique du système pouvant mettre en jeu plusieurs micro-étapes successives.

En effet, dans l'Approche Réactive, l'exécution d'une réaction (c'est-à-dire d'un instant) du système peut nécessiter un certain nombre de micro-étapes afin d'aboutir à un état stable en fin de réaction. L'Approche Réactive considère ainsi des micro-étapes (micro-steps) successives qui composent une réaction (un instant ou macro-step) du système. Ces micros étapes permettent au cours d'une activation de faire progresser le système vers un état stable marquant la fin de la réaction en cours.

3.5.1.1 Les instructions

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

TERM indique que l'instruction a complètement terminé son exécution pour l'instant courant et les suivants. Si l'on réactive une instruction qui a renvoyé le statut TERM, il ne se passera plus rien et elle renverra toujours le statut TERM.

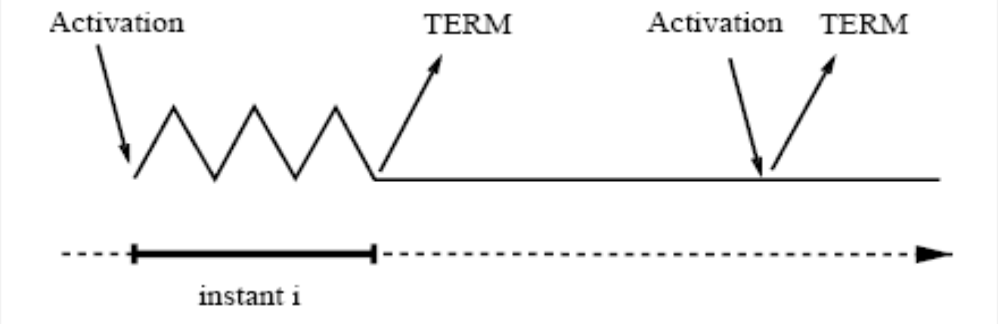

Figure III.8 Comportement qui rend TERM [RA03].

STOP (pour 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.

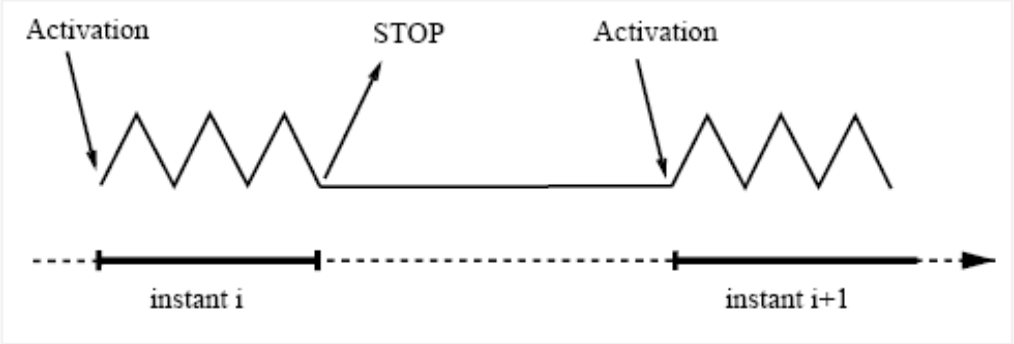

Figure III.9 Comportement qui rend STOP [RA03]

SUSP (pour 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. L'exécution de l'instruction se suspend momentanément pour laisser à d'autres composants s'exécutant en parallèle l'opportunité de générer cet événement. L'instruction suspendue est réactivée jusqu'à ce qu'elle atteigne un état stable (voir machine d'exécution).

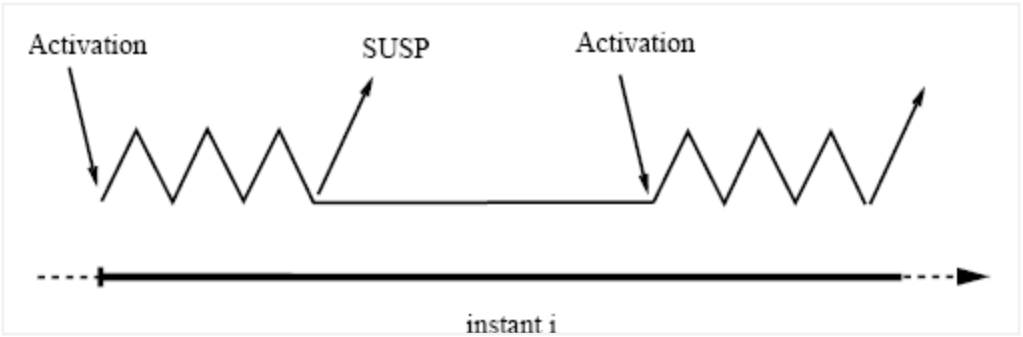

Figure III.10 Comportement qui rend SUSP [RA03]

WAIT a été rajouté dans la version Storm de Junior pour des questions d'efficacité. Ce statut est renvoyé par les instructions événementielles pour indiquer qu'elles sont en attente d'un événement et qu'il n'est pas nécessaire de les réactiver tant que l'événement n'est pas présent ou tant que la fin d'instant n'a pas été déclarée. Cela évite de repasser à chaque micro-étape sur l'ensemble des instructions en attente d'un événement, et d'exécuter seulement celles dont l'événement déclencheur a été généré.

3.5.1.2 Les principale primitives

Chaque primitive réactive des **SugarCubes** est définie par dérivation de la classe *Instruction* ou de sous-classes comme les classes *BinaryInstruction* ou *UnaryInstruction.* La figure III.11 propose un diagramme représentant la hiérarchie des *classes-instructions* **SugarCubes**. Les classes grisées sont des classes abstraites qui implémentent des services communs à plusieurs primitives réactives.

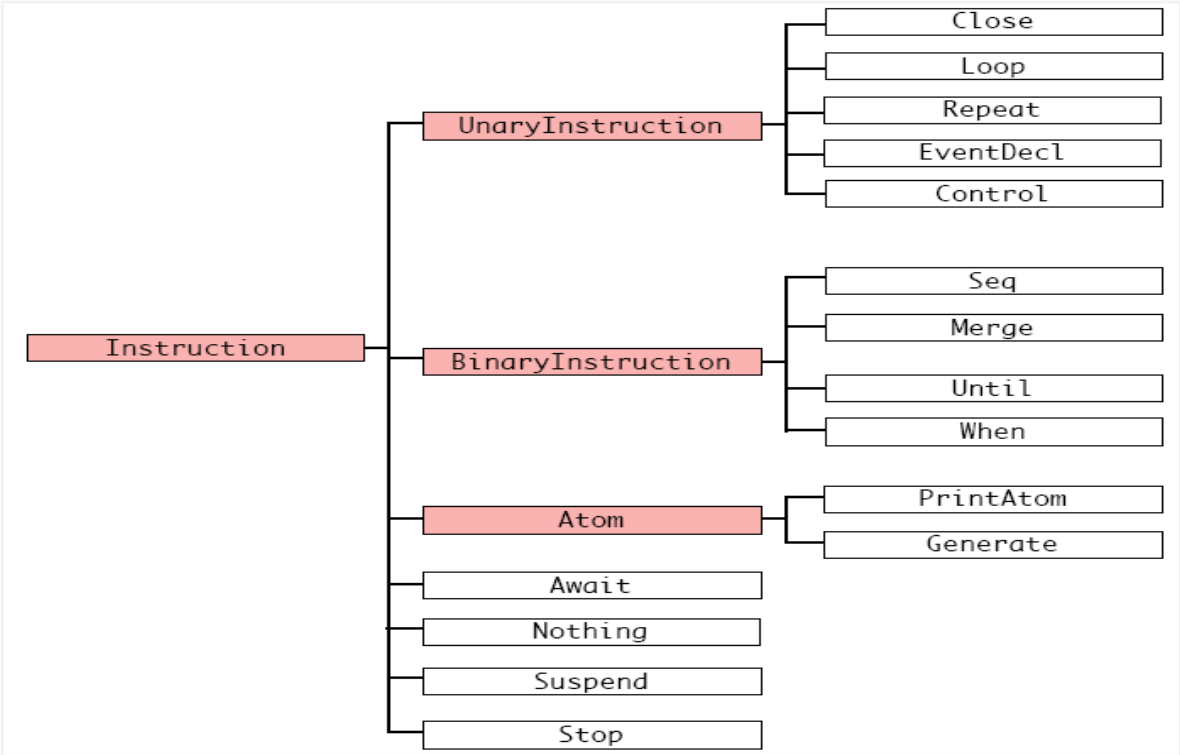

Figure III.11 hiérarchie des classes-instructions SugarCubes [JFS01].

3.5.2 Junior

Junior est né des expériences réalisées avec la première implémentation du modèle réactif synchrone en Java, les SugarCubes. En particulier Junior est né avec l'objectif de donner une

sémantique formelle au modèle réactif synchrone; la sémantique permet une meilleure compréhension du modèle et de valider les différentes implémentations **[RA03]**. **Junior** est donc un ensemble de classes Java permettant d'implémenter des systèmes réactifs. Les classes **Junior** définissent des primitives réactives permettant d'écrire un programme réactif dans un style impératif "*à la*" **SugarCubes**. Ces instructions réactives sont exécutées par des machines d'exécution. Les machines d'exécution sont des objets capables d'accepter dynamiquement de nouveaux comportements en parallèle **[JFS01]**.

3.5.2.1 API de junior

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

Nous distinguons plusieurs types d'instructions réactives :

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

a. Interfaçage avec Java

a.1 Les Wrappers

Les wrappers en Junior ont été définis pour palier l'absence de pointeurs en Java qui sont nécessaires dans la construction d'un programme réactif. Ils sont donc 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.

a.2. Les instructions

• Jr.Atom(Action a)

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

• Jr.Link(ObjectWrapper objWrap, Program body)

Cette instruction associe un objet Java à l'exécution d'un programme réactif. L'objet Java associé est obtenu après évaluation du wrapper d'objet. En fait, cette instruction place l'objet Java dans l'environnement d'exécution pour que le programme réactif exécuté puisse interagir avec lui.

b. Les instructions de base

• Jr.Nothing()

Cette instruction ne fait rien et termine immédiatement, ce qui correspond à retourner TERM à chaque activation.

• Jr.Stop()

Cette instruction termine l'exécution d'une séquence d'instructions pour l'instant courant. Cela correspond à retourner STOP à la première activation et TERM à toutes les suivantes.

• Jr.Seq(Program first, Program second)

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

• Jr.Par(Program left, Program right)

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

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

Tableau 3.1 Status retourné par l'instruction Par.

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

Jr.Loop(Program body)

Cette instruction exécute en boucle le programme body. Dès que body se termine, il est réinitialisé puis immédiatement ré-exécuté. Cela signifie que les wrappers qui avaient été évalués le seront à nouveau à la prochaine exécution. Ceci est une optimisation du système car il est moins couteux de réinitialiser les instructions entre deux exécutions plutôt que de recréer, à chaque itération, une nouvelle instance du programme. L'utilisation de cette instruction peut cependant poser le problème de boucle instantanée. En effet, si l'exécution de

body est instantanée, l'instruction Loop ne peut converger vers un état stable en temps fini, puisque body sera immédiatement ré-exécuté.

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

Jr.Repeat(IntegerWrapper num, Program body)

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

• Jr.If(BooleanWrapper cond, Program thenInst, Program elseInst)

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

c. Evénements et configurations événementielles

c.1 Les événements

L'approche réactive dispose d'un moyen de communication puissant en l'occurrence la diffusion d'événement. Ainsi la représentation d'un événement en junior se fait en utilisant un type particulier. Si on choisissait, par exemple, le type String on serait limité dans le pouvoir expressif (et aussi dans certain cas dans l'efficacité du programme) des types d'événements que l'on pourrait générer. Pour ne pas avoir cette limitation et pouvoir générer des événements d'un type quelconque, Junior définit l'Identifier. La seule chose dont on besoin pour généraliser la notion d'événement est d'avoir un mécanisme qui nous permet de savoir si deux événements sont égaux. Un événement en Junior donc 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. Cependant, dans l'environnement JavaCard, le concept de tables de hachages n'est pas supporté, il sera du fait remplacé par un tableau simple.

c.2 Les configurations événementielles

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

• Jr.Presence(IdentifierWrapper idWrap)

Cette configuration teste la présence d'un événement dont l'identificateur est obtenu à partir de l'évaluation du wrapper d'événement. L'évaluation du wrapper est effectuée au moment du premier test de présence de l'événement.

• Jr.And(Configuration c1, Configuration c2)

Cette configuration exprime la conjonction de deux configurations événementielles.

• Jr.Or(Configuration c1, Configuration c2)

Cette configuration exprime la disjonction de deux configurations événementielles.

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

c.3 Les instructions événementielles

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

• Jr.Generate(IdentifierWrapper idWrap)

•Jr.Generate(IdentifierWrapper idWrap, ObjectWrapper objWrap)

Cette instruction génère un événement qui est alors considéré comme présent dans l'environnement d'exécution pour l'instant courant. L'identificateur d'événement est obtenu après évaluation du wrapper d'événement. La seconde version de l'instruction permet de générer un événement et de lui associer une valeur. Cette valeur est obtenue après évaluation du wrapper d'objet Java. Cette instruction termine immédiatement après avoir modifié l'environnement d'exécution.

• Jr.Await(Configuration c)

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

• Jr.When(Configuration c, Program thenInst, Program elseInst)

Cette instruction correspond à l'instruction If, excepté qu'au lieu d'évaluer un BooleanWrapper, une configuration événementielle est évaluée. Si cette configuration événementielle est satisfaite, alors le programme *thenInst* est exécuté, sinon c'est le programme *elseInst*. L'exécution de *thenInst* ou *elseInst* peut être reportée à l'instant suivant en fonction de la configuration événementielle. En effet, la décision de satisfaction ou de non-satisfaction de la configuration événementielle peut être reportée à la fin de l'instant.

L'évaluation de la configuration événementielle n'est faite qu'au premier instant où l'instruction est exécutée.

• Jr.Until(Configuration c, Program body, Program handler)

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

• Jr.Freezable(IdentifierWrapper idWrap, Program body)

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

• Jr.Control(IdentifierWrapper idWrap, Program body)

Cette instruction conditionne l'exécution de *body*, c'est-à-dire que body ne sera exécuté qu'aux instants où un événement sera présent. Cet événement est obtenu après évaluation du Wrapper, à la première exécution de *Control*.

• Jr.Local(Identifier id, Program body)

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

d. Machine et environnement d'exécution

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

La machine réactive Junior propose deux types de machines réactives : la safe-machine et l'unsafe-machine. La différence entre ces deux types de machines est que la safe-machine est conçue pour fonctionner dans un environnement multi-threadé, ce qui n'est pas le cas pour un environnement javaCard. Les événements générés dans une unsafe-machine sont perdus à partir du moment où la fin d'instant est déclarée. La machine finit l'instant courant à la fin de l'instant que lorsque toutes les instructions parallèles du programme qu'elle contient sont soit totalement terminées soit stoppées (c'est-`a-dire qu'il n'y a plus d'instructions suspendues). Sinon, la machine active cycliquement le programme tant qu'il reste des instructions suspendues. A la fin de chaque activation, la machine teste si un nouvel événement a été généré durant l'activation; s'il n'y en a pas, alors la situation ne peut plus évoluer et la fin de l'instant courant est décidée. Dans ce cas, un drapeau est levé et une dernière activation du programme permet aux instructions suspendues de stopper. Ces deux types de machine sont accessibles à travers l'API par les deux méthodes suivantes :

• Jr.Machine(Program p)

Cette méthode crée une unsafe-machine exécutant le programme p.

• Jr.SafeMachine(Program p)

Cette méthode crée une safe-machine exécutant le programme p. Pour interagir avec la machine, l'utilisateur dispose des méthodes suivantes :

• void add(Program p)

Cette méthode ajoute un programme en parallèle à ceux qui sont déjà chargés dans la machine. Cet ajout est pris en compte à l' instant suivant.

• boolean react()

Cette méthode fait progresser d'un instant le programme chargé dans la machine. La méthode retourne un booléen qui indique s'il reste encore quelque chose à faire à l'instant suivant. Cependant, en cas d'appels ré-entrants, c'est-à-dire que si une action atomique appelle la méthode react sur la machine qui l'exécute, cette dernière renverra immédiatement false sans rien exécuter.

• void generate(Identifier id)

• void generate(Identifier id, Object obj)

Ces deux méthodes génèrent, respectivement, un événement sans ou avec une valeur dans la machine sans passer par l'instruction Generate. Une génération dans la safemachine se fait à l'instant courant si la fin d'instant n'a pas été déclarée. Si elle a été déclarée, la génération est prise en compte à l'instant suivant. A l'inverse, pour l'unsafemachine, l'événement est perdu si la génération est faite après que la fin d'instant soit déclarée.

• Program getFrozen(Identifier id)

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

3.5.2.2 Construction et exécution d'un programme Junior

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

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

Un exemple concret

Dans cette exemple ci-dessous, nous avons crée deux programmes, l'un est le programme principal qui va être ajouté à la machine d'exécution tout au début, soit à la toute première réaction de celle-ci. Le programme principal (*Figure III.13*) met en séquence deux composants, le premier exécute une action atomique (*Figure III.15*) affichant un message dans ce cas <<Event generated ?>>, quant au second, il consiste à attendre la génération d'un

événement <<monevent>> à l'aide de l'instruction *Jr.Control* pour afficher le message <<Fin !>> soit la fin du programme principale.

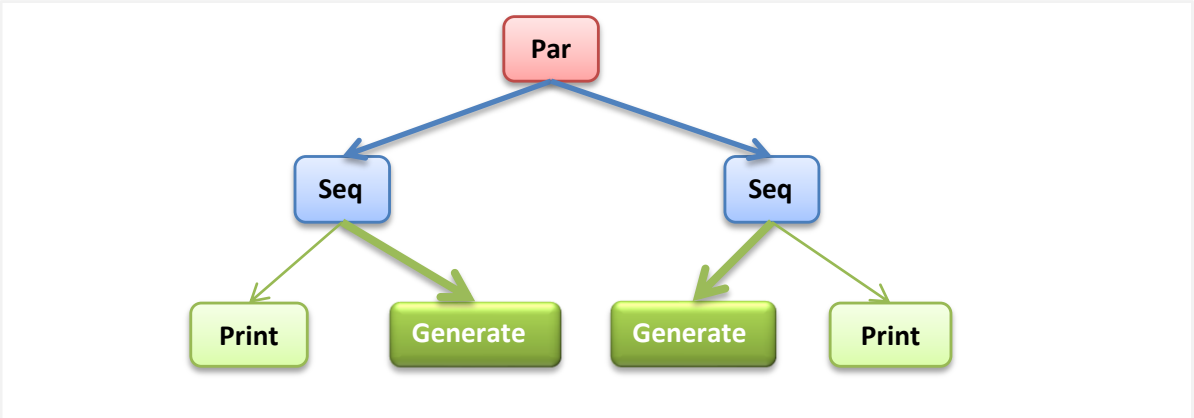

Figure III.12. Arbre syntaxique du programme.

Dans la suite du programme, on fait réagir la machine un certain nombres de fois et à la septième plus exactement en ajoute dynamiquement un programme destiner a générer l'événement attendue par le programme principal, ceci à l'aide de la méthode *add(Program p)* de la machine d'exécution. Le programme (*Figure III.13*) qui va générera l'événement est composé d'une séquence de deux programmes, le premier génère l'évènement <<monevent>> en exécutant l'instruction *Jr.Generate(IdentifierWrapper iw)* qui prend comme paramètre un IdentifierWrapper et celui-ci doit être implémenté (*Figure III.14*), le deuxième programme affiche le message <<Even generated>>.

Après avoir lancé cet exemple dans un IDE, on peux voir la sortie du programme sur la console, la (*Figure III.16*), d'abord l'affichage du message <<Event generated ? >> au premier instant puis rien dans les 5 prochains instants et au 7ème instant l'affichage de <<Event generated>> synonyme que l'événement à bien été générer par la machine d'exécution et en fin le programme se termine en affichant <<Fin !>> toujours dans le 7ème instant.

```java
public class ExempleJunior {

    public static final byte monevent = 1;
    public static void main(String[] args) {

        // Création du programme qui gènére l'événement monevent
        Program genererEvent = Jr.Seq (
                Jr.Generate (new MonEventWrap(new MonEvent(monevent)) ) ,
                Jr.Atom(new MonAtom ("Even generated")));

        // Création du programme principal qui attend la génèration de monevent pour terminer
        Program programPrincipal = Jr.Seq(Jr.Atom(new MonAtom("Even generated?")) , Jr.Control(
                new MonEventWrap((MonEvent)new  MonEvent(monevent))
                ,
                Jr.Atom(new MonAtom("Fin!"))
                ));

        // Création d'une Machine réactive et ajout du programme dans cette machine
        Machine m = Jr.Machine(programPrincipal);

        // Faire réagir la machine m un nombre quelconque de fois
        int t =1;
        do{
            System.out.println("instant "+t+": ");
            if(t==7) {m.add(genererEvent);} // ajout dynamique du programme qui génére monevent
            t++;
        }while(!m.react()); // tanque le programme n'est pas terminer
    }//endOfMain
}//endOfExempleJuinior
```

Figure III.13. Création et exécution du programme réactif.

```java
//Implimentaion de l'interface pour créer un événement
class MonEvent implements Identifier{
    public byte id ;
    public MonEvent ( byte i) { id = i;}
    public int hashCode ( ) { return id;}
    public boolean equals(Object object){
        if( object instanceof MonEvent ) return id == ((MonEvent)object).id?true:false;
        return false ;}
}//endOfMOnEvent

// Implimentaion de l'interface IdentifierWrapper pour ponter vers un événement
 class  MonEventWrap implements IdentifierWrapper{
    Identifier identifier;
    public MonEventWrap(Identifier id) {this.identifier = id;}
    public Identifier evaluate(Environment environment){return this.identifier; }
}//endOfMonEventWrap
```

Figure III.14 Implémentation des interfaces : Identifiées et IdentifierWrapper.

```
57 class MonAtom implements Action{
58     String msg;
59     public MonAtom(String msg) {this.msg = msg;}
60     public void execute ( Environment env ){System.out.println(" -> "+this.msg) ; }
61 }//endOfMonAtom
```

Figure III.15 Implémentation de l'instruction Action.

```
<terminated> ExempleJunior [Java Application] C:\Program Files\Java\jre1.6.0_07\bin\javaw.exe
instant 1:
 -> Even generated?
instant 2:
instant 3:
instant 4:
instant 5:
instant 6:
instant 7:
 -> Even generated
 -> Fin!
```

Figure III.16 Sorties du programme après exécution.

3.5.3 Sémantique

Junior formalise la sémantique des opérateurs de l'Approche Réactive a travers un ensemble de règles SOS (Sémantique Opérationnelle Structurelle) très simples.

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, 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 défini 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.

b) La Syntaxe

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

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

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

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

La sémantique repose sur un nombre réduit de règles 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*).

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

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:

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

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

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

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:

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

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

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:

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

L'instruction If :

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

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

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

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

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

- *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:

$$sat(C, E) \equiv fixed(C, E) \text{ and } eval(C, E)$$

- la fonction de non-satisfaction:

$$unsat(C, E) \equiv fixed(C, E) \text{ and } eval(C, E) = false$$

- la fonction de configuration non encore déterminée:

$$unknown(C, E) \equiv fixed(C,E) = false$$

La primitive Generate :

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

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

Exemple :

1 Await(‘‘Initialiser’’)

2 Atom([somme =0;])

| |

3 Stop

4 Generate(‘‘Initialiser’’)

Instant 1 : 1, 3

Instant 2 : 1, 4, 1, 2

Dans cet exemple, le programme s'exécute en deux instants : 1) d'abord la première branche exécute Await qui arrête la séquence, puis c'est la deuxième branche qui s'arrête immédiatement avec l'exécution de Stop, 2) dans le deuxième instants l'instruction 4 (Generate) va générée l'évènement ‘‘Initialiser’’ qui permettra l'exécution de l'instruction 2.

Voici le programme : (*MonEvenWrap* voir Figure III.14)

```
public static final byte Initialiser = (byte) 1;
Program programme =
new Par(new Seq(new Await((new Presence(new  MonEvenWrap(Initialiser)))),
new Atom(new Action(){public void execute(Environment environment)
{somme=(short)0;}} )), new Seq(new Stop(),new Generate(new
MonEvenWrap(Initialiser))));
Machine m = new Machine(programme);
while(!m.react());
```

L'instruction When :

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

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

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

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:

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

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

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

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

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

Exemple :

```
Program programme = new Loop(  new Seq(
new When(new And((new Presence(new MonEvenWrap(A))),
new Not(new Presence(new MonEvenWrap(B)))),
new Atom(new Action(){public void execute(Environment environment) {
somme=somme+10; // incrémentation de 10
// Envoi de la commande apdu
byte[] buffer = apdu.getBuffer();
short bytesRead = apdu.setIncomingAndReceive();
if (bytesRead!=2) ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
Util.setShort(apdu.getBuffer(), ISO7816.OFFSET_CDATA, somme);
apdu.sendBytes((short) ISO7816.OFFSET_CDATA, (short)2);
}}),
new Atom(new Action(){
public void execute(Environment environment) {
      somme++; // incrémenter la somme.
}})),new Stop()));

        Machine m = new Machine(programme);
         for(byte i=1; i<=5; i++) {
               System.out.println(i);
               if(i==2) m.generate(B);
               if(i==3) m.generate(A);
               m.react();
               }}

1 :
2 : // somme vaut 1
3 :
4 : somme= 11
```

Dans cet exemple, le programme s'exécute 4 fois, sans se terminer vue que le programme est contenu dans une boucle. 1) dans le premier instant, When évalue la configuration et elle est indéterminé dans ce cas elle renvoie Stop comme drapeau, mais dans l'instant 2 elle exécute le programme (la branche Else) car la configuration n'est pas satisfaite (B est déclarer présent juste au début de l'instant n^{o} 2) de ce faite, elle incrémente la somme de **1**. Dans les instants 3 et 4 ; une fois A généré dans l'instant 3, la configuration ne peut être satisfait qu'à l'instant suivant pour déclarer l'absence de B puis incrémenter de la somme de 10 et afficher le résultat.

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:

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

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

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

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

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

Exemple : idem que exemple précédent.

L'instruction Control :

L'instruction Control exécute son corps uniquement aux instants où la configuration de contrôle est satisfaite. Si la configuration d'événements est satisfaite, alors *Control* exécute son corps :

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

Si la configuration est insatisfaite, alors Control a fini pour l'instant et renvois STOP comme drapeau:

$$\frac{unsat(C,E)}{Control(C,t), E \xrightarrow{stop} Control(C,t'), E}$$

Par contre, si la configuration ne peut pas être évaluée, c.-à-d. qu'il manque des informations au moment de l'évaluation et que ce n'est pas la fin de l'instant, ainsi en est dans un état appelé état inconnue (unknown), dans ce cas Control se suspend pour pouvoir être ré exécuté au cours de ce même instant :

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

Exemple :

```
Program programme =
new Loop(new Seq(
new Par( // 1er composant:
new Control((Presence)(new Presence(new MonEvenWrap(A))),
new Atom(new Action(){public void execute(Environment environment)
{somme=somme+10;
//Envoi de la commande apdu
byte[] buffer = apdu.getBuffer();
short bytesRead = apdu.setIncomingAndReceive();
if(bytesRead!=2)ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
Util.setShort(apdu.getBuffer(), ISO7816.OFFSET_CDATA, somme);
apdu.sendBytes((short) ISO7816.OFFSET_CDATA, (short)2);
}})), // 2éme composant:
new.Loop(new.Seq(new.Atom(new Action(){public void execute(Environment
environment) {somme++;}}),Jr.Stop()))
, new Stop()));
Machine m = new Machine(programme);
for(byte i=1; i<=3; i++) {
if(i==3) m.generate(A);
m.react();
}}
```

```
1 : // vaut somme 1
2 : // vaut somme 2
3 : somme = 12
    //somme vaut 13, s'exécute  même après avoir envoyé l'apdu.
```

Dans cet exemple le programme exécute en parallèle deux composants, le premier ne sera activé qu'à l'instant 3 directement par la machine ; il ajoute 10 à la somme puis affiche le tout, tandis que le deuxième composant, quand à luis, il incrémente la somme à chaque instants.

L'instruction Until :

L'instruction *Until* permet de préempter un programme sur satisfaction d'une configuration événementielle. C'est une préemption faible, donc, si la configuration est satisfaite, il faut attendre la fin de l'exécution, pour l'instant courant, du programme à préempter et *Until* se comporte comme son corps si celui-ci ne retourne pas STOP.

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

Si le corps retourne STOP, *Until* se comporte comme la forme auxiliaire *Until** :

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

Les règles décrivant le comportement de la forme auxiliaire Until* sont les suivantes :

Le handler de l'opérateur Until est immédiatement exécuté si la configuration de préemption est satisfaite avant la fin d'instant :

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

Until* retarde l'exécution du handler à l'instant suivant si la configuration de préemption n'est satisfaite qu'à la fin de l'instant :

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

Si la configuration n'est pas satisfaite, Until* retourne STOP et se réécrit dans la forme standard Until :

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

Enfin, tant que la configuration reste indéterminée, Until* se suspend :

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

Exemple

```
Program programme =
new Until((new Presence (new MonEvenWrap(A))),
new Loop(new Seq( new Atom(new Action(){
public void execute(Environment environment) {somme++;
}}),Jr.Stop())),
new Atom(new Action(){public void execute(Environment environment) {
somme=somme+10; // incrémentation
byte[] buffer = apdu.getBuffer();
short bytesRead = apdu.setIncomingAndReceive();
if (bytesRead!=2) ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
Util.setShort(apdu.getBuffer(), ISO7816.OFFSET_CDATA,somme);
apdu.sendBytes((short) ISO7816.OFFSET_CDATA, (short)2);
 }}));

1: // somme vaut 1
2: // somme vaut 2
3: // somme vaut 3 : termine d'exécuté Body puis le handler.
      somme= 13 //affiche la somme
```

Dans cet exemple le programme se termine au bout de trois instants, le premier : étant la configuration non satisfaite, le programme du « Else » s'exécute et incrémente la somme de 1.

Même chose pour le deuxième instant. En fin, le troisième instant, l'événement « A » est générer dans l'environnement au début de cette instant, dans ce cas ***Until*** exécute ou termine l'exécution du programme « Else » puis le programme « Then » et de se termine après ça.

3.5.4. 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.

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 **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.

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, 3 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.

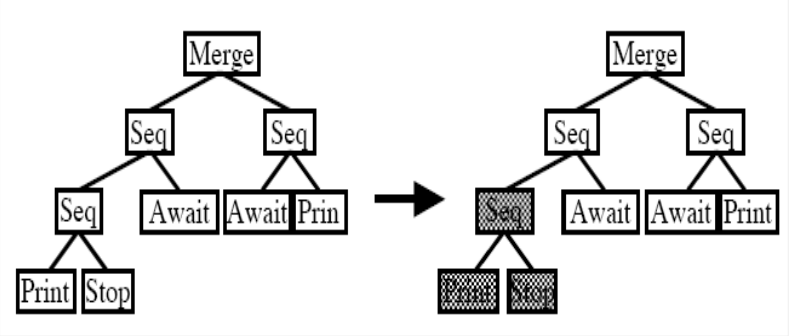

Figure III.17.En Replace, le programme est conservé [JFS01].

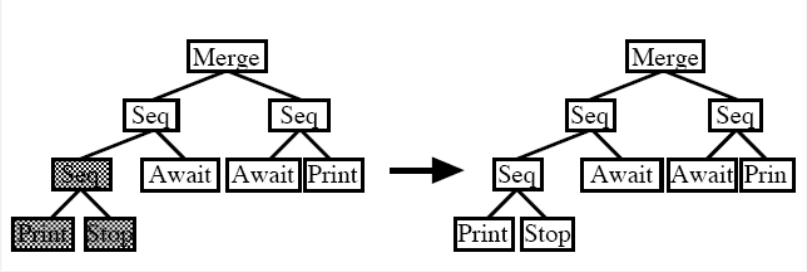

Figure III.18. La fonction reset permet de remettre le programme dans son état initial [JFS01].

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.

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'´événement sont réveillées par un mécanisme appelé **précurseur**. Il est chargé de libérer un chemin entre l'instruction réveillée et la racine de l'arbre.

L'exécution d'une branche est bloquée par une ou plusieurs instructions **Par**. La libération d'un chemin entre l'instruction réveillée et la racine de l'arbre consiste donc à modifier l'état des instructions **Par** bloquant l'exécution de l'instruction en passant le statut de **WAIT** à **SUSP**.

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);
...
}
```

5) Glouton [RA03]

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.-à-d. 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

6) 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.

7) 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.2 : Comparaison entre Replace et les implémentations de Storm [JF01].

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

3.6. 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**. De plus, le choix de **Replace** repose sur la réécriture de l'état associer à l'opérateur, ainsi la structure du programme n'est pas altérée, et contrairement à **Rewrite,** des branches '' mortes'' d'un arbre continuerons à subsister et il faudra parfois interroger l'état d'un opérateur, pour s'apercevoir qu'une branche est terminer et ne doit être plus exécuter. Cependant, la gestion de la mémoire est plus efficace et les pertes de temps

passé à tester les états des opérateurs sont très largement compensé, ceci dit, le cas des branches ‘‘mortes’’ peut être résolue en adoptant un style de programmation, par exemple, en encapsuler le programme concerné dans une instruction **Loop** qu'elle a la particularité de réinitialisé l'état du programme a chaque activation, celle-ci (**Loop**), pourra être contrôlée par la réception d'un événement donné. En outre par rapport à Strom et Simple Replace est plus optimisé en terme de lignes de code.

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 et en adaptant d'autres pour afin qu'elles tiennent compte des restrictions lier à un environnement Java Card c'est notamment le cas des événements où en ne retiendra pas les événements values, 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. Et la carte à puce ne contient pas d'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); }
}
```

2. **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.

3. **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**.

4. **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**.

5. **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.

6. **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 : **Par** (**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 sur l'implémentation Replace de Junior.

7. **Les fonctions *resudual(), linkedObject()* et *equal()* sont supprimer dans toutes les instructions** :

Pour la premiere c'est le fait qu'elle est utilisée à dégeler des programmes déjà gelés par l'instruction Frezable qui supprimée, de même, *linkedObject()* car elle sert à lier des objets entre les composant par le bais des événements valués, les quelle ne sont pas considérer dans ce travail, enfin la méthode *equal()* qui à mon sens, n'est pas primordial pour le fonctionnement du noyau et vue que celui-ci dois être optimisé au maximum pour pouvoir tourné sur une carte à puce.

8. **D'autres instructions sont supprimées** à limage de **Jr** qui n'a pour but que la simplification de la syntaxe des programmes surtout quant en sais quelle a une taille de 4ko, le choix est vite fait. Ou les Wrapper ; IntegerWraper qui est remplacer par un *short* dans L'instruction **Repeat** et ObjectWrapper du fait qu'il n'y pas d'événement valué, ainsi que d'autre class sensée les gérer comme EvenData et EvenDataImpl.

3.6.1 L'environnement de développement

Afin de pouvoir développer des applications **JavaCard** et des applications clientes, 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**.

3.6.2 Implémentations

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. Ceci afin de dégager un noyau réactif capable de tourner dans un environnement aussi contraignait en ressources que les Java Card et avec un langage moins puissant en terme de mots clé et de concept de programmation que offre celui qui tourne dans

un environnement classique, vue que lui même à subi des élagages pour pouvoir répondre au contraintes de tel environnements.

Ce noyau sera composé de :

- Les instructions de base comme les instructions Stop, Seq, Loop, etc…
- Les instructions de gestion des évènements tels Generate, Control, Presence, etc et bien sûr, le type d'événement retenu ici ne concerne que les événements purs, c.-à-d. sans valeurs.

3.6.2.1 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'y a 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 {
    public static final byte STOP = 0;
    public static final byte TERM = 1;
    public static final byte 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 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 byte rewrite(){
     if (terminated == false){ action.execute(env); terminated = true; }
     return TERM;
  }
}
```

– **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:

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

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

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

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 program1, Program program2) {
            super(program1, program2);
            leftTerminated = false;
  }
  public void reset(){ ssuper.reset(); leftTerminated = false; }

  public byte rewrite(){

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

– **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 program) {super(program); }
    public byte rewrite(){
        byte etat = super.body.rewrite();
        if(etat != TERM)
        { return etat;} else {
            super.body.reset();
```

```
            return rewrite();
      }}}
```

– **L'instruction Par :**

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. **Par** 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 **Par** 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 = super.right.rewrite();
      else{
          leftFlag = super. left.rewrite();
              rightFlag = super. right.rewrite();}
      return result(); } }
```

3.6.2.2 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 toad;
   public boolean somethingToAdd;
   public boolean terminated;

public MachineImpl(Program program){
        toAdd =   new Nothing();   //Le programme initiale de la  machine
somethingToAdd = false;
        terminated = false;
        buildEnvironment();
        instant = new Instant (program);
        instant.bind(env); }
// building the environment
public void buildEnvironment(){ env = new EnvironmentImpl(); }
protected void performAddings(){
if (somethingToAdd == false) return;
instant.body = (Instruction)new Par(toAdd, (UnaryInstruction)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; }
  // Responsable de générer un événement dans l'environemet qui est un byte
  // exemple machine.generate((byte) 1) ; puis en la fait réagir avec la
méthode machine.react() ;
  public void generate(byte identifier) {env.generate(identifier); }
}
```

Ici, la méthode *generate()* nous permet de générer un événement sans passer par l'instruction Generate se qui a pour effet la non création d'un nouveau objet, surtout en **runtime.**

- **L'environnent**

Il s'occupe de la gestion des instants, c.-à-d. c'est lui qui dit quand un instant est terminé ou qu'il devra être repoussé, il se charge également de la gestion des événements.
Voici sont code :

```
public class EnvironmentImpl implements InternEnvironment{
   public short instant;
    public boolean eoi;
    public boolean move;
    public short[] eventSet;  // la liste des évènements de l'environnent
    public EnvironmentImpl(){
    instant   = 1; //le premier instant
    eventSet = new short[127]; // le nombre max d'évènement pendant un
instant
    }
    public boolean eoi(){return eoi;}
```

```
    // Déclare la fin de l'instant si il n y pas d'évènements nouveaux
    public void setEoiIfNoMove() { if(move) move = false;  else  eoi =
true; }
    public void resetEoi(){eoi = false;}
    // Déclare un nouveau instant
    public void newInstant(){
    instant++;
    resetEoi();
    move = false; }
   //Génération d'un événement
    public void generate(byte eventName){
    // Si il n'est pas généré avant, la fin de l'instant est repousser
    // pour donner un chanse aux composant concerner de s'exécuter
      if(!isGenerated(eventName)) move = true;
   // En l'enregistre l'instant ou cet événement et généré pour pouvoir
   // y être comparer la prochaine fois qu'il soit régénérer
      eventSet[eventName] = instant;}
    public boolean isGenerated(byte eventName){
      return instant == eventSet[eventName]; //si il est déjà généré
      } }
```

3.6.2.3 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 Control**

Le corps de la primitive Control est exécuté si l'événement de contrôle est présent:

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

Control retourne ***STOP*** si l'événement n'est pas là (ce qui d'ailleurs ne peut être connu qu'à la fin d'instant) :

$$\frac{\boldsymbol{unsat(C,E)}}{\boldsymbol{Control(C,t),E \xrightarrow{stop} Control(C,t'),E}}$$

Tant que l'événement reste indéterminé, *Control* se suspend:

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

Et voici sont code :

```
public class Control extends UnaryInstruction{
      public Presence presence;
    public Control(Presence presence1, Program program) {
             super(program);presence = presence1; }
    public void reset(){ super.reset(); presence.reset(); }
    public byte rewrite(){
        if(presence.sat(super.env))
            return super.body.rewrite();
        return ((byte)(!presence.unsat(super.env) ? SUSP : STOP));}
}
```

- **L'instruction Generate**

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

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

Voici son code :

```
public class Generate extends Instruction {
public final IdentifierWrapper idwrap// Le pointeur vers l'événement à
générer
public boolean terminated;
public Generate(IdentifierWrapper identifierwrapper) {
    terminated = false; idwrap = identifierwrapper;}
public void reset(){terminated = false;}
public byte rewrite(){
  if(!terminated){
  byte identifier = idwrap.evaluate(super.env);
  super.env.generate(identifier); //génère l'évènement dans l'environnement
            terminated = true;}
   return TERM;
 }}
```

- **L'instruction Await**

Voici son code :

```
public class Await extends Instruction{
public Config config;
    public boolean terminated;
    public Await(Configuration configuration){
       terminated = false;
       config = (Config)configuration; }
    public void reset(){
        terminated = false;
        config.reset();}
    public byte rewrite(){
        if(terminated) return TERM;
        if(config.sat(super.env)){
            terminated = true;
            return ((byte)(super.env.eoi() ? STOP: TERM));}
        return ((byte)(!config.unsat(super.env) ? SUSP: STOP));}}
```

- **L'instruction Until**

Voici son code :

```
public class Until extends BinaryInstruction{
    static short count =0 ;
    public Config config;
    public boolean runHandler;
    public boolean bodyStopped;
    public Until(Configuration configuration,Program body, Program
handler){
     super(body, handler);
     runHandler = false;
        bodyStopped = false;
        config = (Config)configuration;}
    public void reset(){
        super.reset();
       config.reset();
        runHandler = bodyStopped = false;}
    public byte rewrite(){
        if(runHandler)return super.right.rewrite();
        if(bodyStopped) return rewriteStar();
        byte byte0 = super.left.rewrite();
        if(0 != byte0){
            return byte0;} else{s
            bodyStopped = true;
            return rewriteStar();}}
    public byte rewriteStar(){
        if(config.sat(super.env)){runHandler = true;
           return super.env.eoi() ? 0 : super.right.rewrite();}
        if(config.unsat(super.env)){
            bodyStopped = false; return 0;} else{return 2;}}}
```

- **L'instruction When**

  Voici son code :

```
public class When extends BinaryInstruction
{
  public Config config;
  public boolean evaluated = false, value = false;
  public When(Configuration config,Program thenInst, Programe
lseInst){
    super(thenInst,elseInst); this.config = (Config)config;}
  public void reset(){ super.reset(); config.reset(); evaluated =
false; }
  public byte rewrite(){
     if (evaluated) return (value ? left : right).rewrite();
     if (config.sat(env)){
        evaluated = true;
        value = true;
          if (super.env.eoi()) return STOP; else return
left.rewrite(); }
     if (config.unsat(env)){
           evaluated = true;
           value = false;
         if (super.env.eoi()) return STOP; else return
right.rewrite();  }
     return SUSP;}}
```

Fini de dégager le noyau réactif (RJC) et de l'exporter sous formats **.jar**, ce fichier jar à une taille de **18,2 Ko** soit (**18 686 octets**). Une taille en dessous de 24 Ko que dispose la ROM.

Ce qui est une bonne opération d'optimisation d'espace mémoire, surtout sur une carte à puce.

3.6.4 Exemple

On va présenter un exemple qui montre l'utilisation du noyau **RJC** sous une plateforme **JavaCard** et de l'exécuter. Cet exemple mit en jeux un certain nombre d'instruction réactive de base tels **Seq, Stop, Loop** et la mise en parallèle de composants réactif avec **Par** et ceux-là sont activer par la réception d'un événement diffèrent chaque un. Puis il y a aussi la communication entre certain composant par l'envois et réception d'évènement.

Puis de puis le programme client en interagir avec l'applet et chaque réception d'une commande, en génère un événement qui réveillera le composant approprié. L'arbre syntaxique de l'exemple est le suivant :

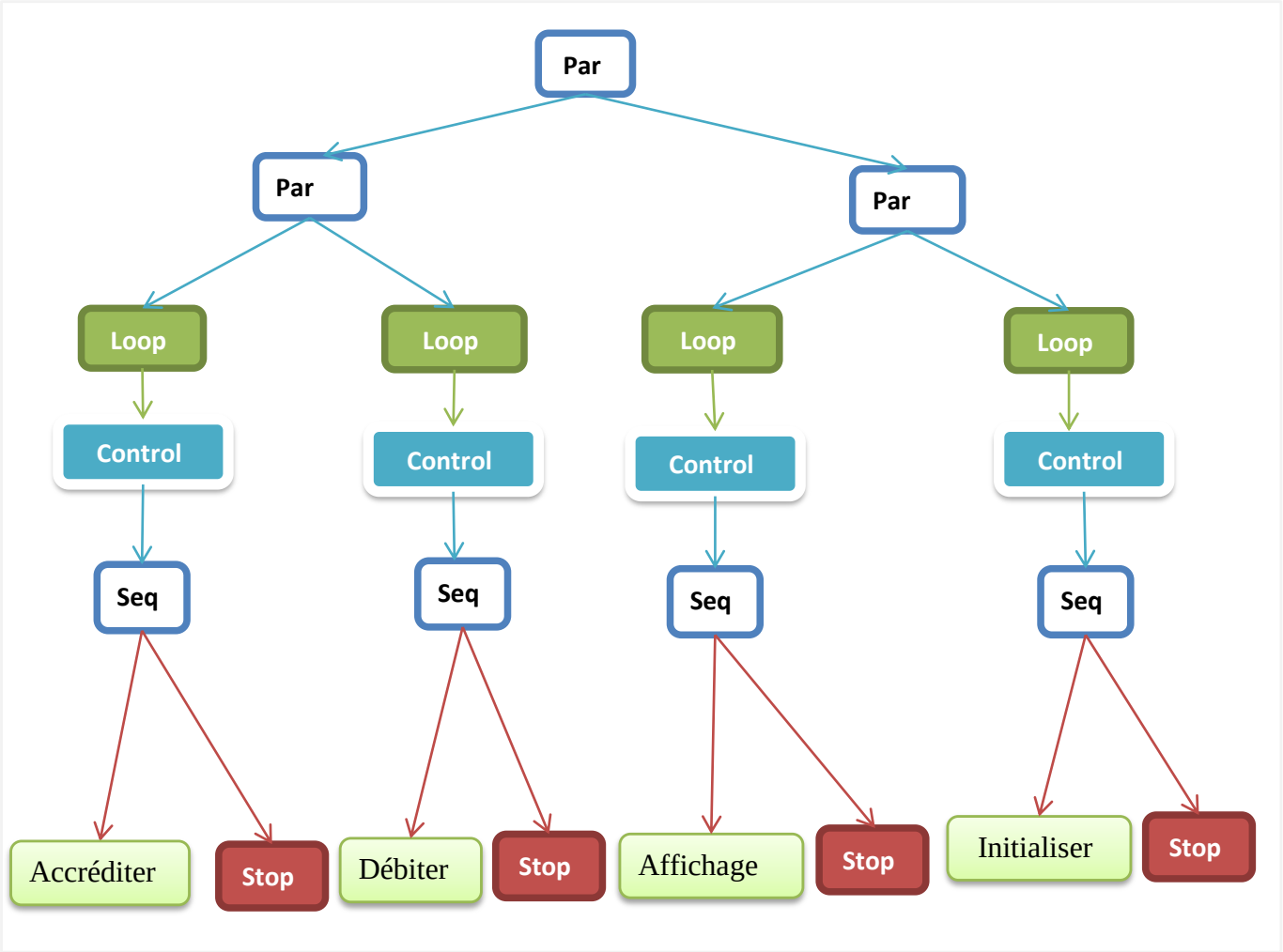

Figure III.19 Arbre Syntaxique du programme réactif.

Voici le code du programme réactif :

```
import junior.Action;
```

```
import junior.Atom;
import junior.Control;
import junior.Environment;
import junior.IdentifierWrapper;
import junior.Jr;
import junior.Loop;
import junior.Machine;
import junior.MachineImpl;
import junior.Par;
import junior.Presence;
import junior.Seq;
import junior.Stop;
import javacard.framework.APDU;
import javacard.framework.Applet;
import javacard.framework.ISO7816;
import javacard.framework.ISOException;
import javacard.framework.Util;
public class MasterCardApplet extends Applet {

      // Constantes
      public static final byte  CLA_MONAPPLET           = (byte) 0xB0;
      // INSTRUCTIONS
      public static final byte  INS_CREDIT              = (byte) 0x01;
      public static final byte  INS_DEBIT               = (byte) 0x02;
      public static final byte  INS_AFFICHAGE           = (byte) 0x03;
      public static final byte  INS_DETTE               = (byte) 0x04;
      // VALEURS  DE LA BALANCE
      public static final short MAX_BALANCE             = (short) 0x7fff;
      public static final short MAX_TRANSACTION_AMOUNT  = (short) 0x3fff;
      private static short      balance; // MONTANT DE LA BALANCE
      private static short      dette;   // MONTANT DE LA DETTE
      private static short      debit; // MONTANT DEBITER
      // EXCEPTIONS
      public static final short SW_INVALID_AMOUNT       = (short) 0x6A83;
      public static final short SW_AMOUNT_EXCEED_MAX    = (short) 0x6A82;
      public static final short SW_NEGATIVE_BALANCE     = (short) 0x6A84;

      // AFFICHAGE DU SOLDE
      public static byte        print                   = (byte) 0x05;
      public static final byte  valeur_balance          = (byte) 0x05;
      public static final byte  valeur_dette            = (byte) 0x06;
      public static final byte  valeur_debit            = (byte) 0x07;
      // Attributs
      private static APDU       apdu;

   //Constructeur
      private MasterCardApplet () {
            super();
            balance=(short) 2020; //monatant de dépare
      }

      public static void install(byte bArray[], short bOffset, byte
bLength)
                  throws ISOException {
            new MasterCardApplet ().register();
      }

    //Programme réactif
      Machine machine =
      new MachineImpl(//Machine
```

```
                new Par(//Par1
                            new Par(//Par2
                                        new Loop(//loop1
                                                    new Seq(////seq2
                                                                new
Control(//control1

      new Presence(

               new MonEvenWrap(INS_CREDIT))
//*****************************{    CREDITER    }**************************
               new Atom(new Action(){//Atom     Credit

              public void execute(Environment environment) {

                byte[] buffer = apdu.getBuffer();
                short bytesRead = apdu.setIncomingAndReceive();
                if (bytesRead!=2)
ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
                short montant= Util.makeShort(buffer[ISO7816.OFFSET_CDATA],
              buffer[ISO7816.OFFSET_CDATA+1]);

                if (montant>MAX_TRANSACTION_AMOUNT) {
                    ISOException.throwIt(SW_AMOUNT_EXCEED_MAX);}
                else
                    if ((short)(montant+balance)>MAX_BALANCE){
                              ISOException.throwIt(SW_INVALID_AMOUNT);}
                    else {
                              balance+=montant;
                              // APPEL DU COMPOSANT DEBITER
                              if(dette>0)  machine.generate(INS_DEBIT);
                         }
               }})//endOfAtomcREDITER
//*****************************{    CREDITER    }**************************

               )//control1
               ,
               new Stop()
               )//seq1
               )//loop1
               ,
               new Loop(//loop2
                           new Seq(////seq2
                                       new Control(//control2
                                                   new Presence(new
MonEvenWrap(INS_DEBIT))
      ,
//*****************************{    DEBITER     }**************************
      new Atom(new Action(){//AtomDebit

      public void execute(Environment environment) {
            byte[] buffer = apdu.getBuffer();
            // DEMANDE DE DEBIT DU CLIENT
            if(buffer[ISO7816.OFFSET_INS] == INS_DEBIT) {

                  // MONTANT A DIBITER:
                  short bytesRead = apdu.setIncomingAndReceive();
                  if (bytesRead!=2)
ISOException.throwIt(ISO7816.SW_WRONG_LENGTH );
                  short montant=
Util.makeShort(buffer[ISO7816.OFFSET_CDATA],
```

```
                    buffer[ISO7816.OFFSET_CDATA+1]);

                if (montant > MAX_TRANSACTION_AMOUNT ){
ISOException.throwIt(SW_AMOUNT_EXCEED_MAX);}
                else{
                 // BALANCE NEGATIVE
                  if ((short)(balance-montant)<0){
                    dette += (short)(montant-balance);
                    debit = balance;
                        balance = (short)0;
                      }
                 // BALANCE SUPERIEUR OU EGALE
                  else{
                          balance-= (short) montant;
                          debit = (short) montant;
                    }
                }
             //---------------------- demande du client -----------------
---
            }else
            // DEMANDE DE SERVICE DE LA PART D'UN COMPOSANT
            {
                  if((short)(balance-dette)>=0)
                  {
                        balance -= (short) dette;
                        debit  = (short) dette;
                        dette    = (short) 0;
                  }else{
                        dette -=balance;
                        debit  = (short) balance;
                        balance =(short)0;
                        }
            }// ---------------demande de service---------------

            // AFFICHAGE DU MONTANT DIBITER
           if(debit > 0 )
           print = valeur_debit;
             machine.generate(INS_AFFICHAGE);
    }})//endOfAtom
  // ****************************{ DEBITER  }**************************

      )//control2
      ,
      new Stop()
      )//seq2
      )//loop2
      )//Par2
      ,
      new Par(//Par3
                  new Loop(//loop3
                  new Seq(////seq3
                              new Control(//control3
                  new Presence(new MonEvenWrap(INS_DETTE))
      ,
//************************{   EFFACER LA DETTE }**************************

      new Atom(new Action(){//ActionSolde

      public void execute(Environment environment) {
            dette = (short) 0;
            print = valeur_dette;
```

```
            machine.generate(INS_AFFICHAGE);
      }})//endOfActionGetBalance
      )//control3
      ,
      new Stop()
      )//seq3
      )//loop3
      ,
      new Loop(//loop4
                  new Seq(////seq4
                              new Control(//control4
                                          new Presence(new
MonEvenWrap(INS_AFFICHAGE))
      ,
      Jr.Atom(new Action(){//AtomInitialise
// *********************{ AFFICHAGE }***********************************

      public void execute(Environment environment) {
            short le = apdu.setOutgoing();
               if (le != 2) ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
               else{
                     apdu.setOutgoingLength((byte) 2);
                     short valeur = (short) 0x0000;
                     switch(print) {
                     case valeur_balance:
                           valeur = (short) balance;
                           break;
                     case valeur_dette:
                           valeur = (short) dette;
                           print = valeur_balance;
                           break;
                     case valeur_debit:
                           valeur = (short) debit;
                           print = valeur_balance;
                           break;
                     default :
                           valeur = (short) balance;
                           break;
                     }
                     Util.setShort(apdu.getBuffer(), ISO7816.OFFSET_CDATA,
valeur);
                  apdu.sendBytes((short) ISO7816.OFFSET_CDATA, (short)2);
               }
      }})//endOfAtomInitialise
      )//control4
      ,
      new Stop()
      )//seq4
      )//loop4
      )//Par3
      )//Par1
      );//endOfMachine

      public void process(APDU arg0) throws ISOException {
            apdu = arg0;

            // Si l'applet est sélectionnée
            byte[] buffer  = apdu.getBuffer();
            if (this.selectingApplet()) return;

            // Vérification de la classe de l'applet
```

```
            if (buffer[ISO7816.OFFSET_CLA] != CLA_MONAPPLET) {
            ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
            }

        // Tester instruction choisie
            switch (buffer[ISO7816.OFFSET_INS]) {

            case INS_CREDIT :
                  machine.generate(INS_CREDIT);
                  machine.react();
                  break;

            case INS_DEBIT:
                  machine.generate(INS_DEBIT);
                  machine.react();
                  break;

            case INS_AFFICHAGE:
                  machine.generate(INS_AFFICHAGE);
                  machine.react();
                  break;

            case INS_DETTE:
                  machine.generate(INS_DETTE);
                  machine.react();
                  break;

            default: ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
                     break;
            }//endOfSwitch
      }//endOfProcess


      public boolean select(){
            return true;
      }//endOfSelect

      public void deselect() {}//endOfDeselect

}//endOfMasterCardApplet

//Implimentaion de l'interface IdentifierWrapper pour ponter vers un
événement
class  MonEvenWrap implements IdentifierWrapper{
      byte identifier;
      public MonEvenWrap(byte id) {this.identifier = id;}
      public byte evaluate(Environment environment){return this.identifier;
}

}//endOfMonEventWrap
//*/
```

La capture du programme l'or d'un jeu d'essai :

```
Implémentation de l'approche réactive sur une plateforme Javacard
-----------------------------

1 - Crediter
2 - Debiter
3 - Solde
4 - Quitter

-----------------------------
Votre choix ?
3
CLA: b0, INS: 03, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 00, SW1: 00, SW2: 00
CLA: b0, INS: 03, P1: 00, P2: 00, Lc: 00, Le: 02, 07, e4, SW1: 90, SW2: 00
Votre balance = 2020
-----------------------------
Votre choix ?
2
CLA: b0, INS: 02, P1: 00, P2: 00, Lc: 02, 03, e8, Le: 02, 00, 00, SW1: 00, SW2: 00
CLA: b0, INS: 02, P1: 00, P2: 00, Lc: 02, 03, e8, Le: 02, 03, e8, SW1: 90, SW2: 00
Votre balance vient d'être debiter de 1000
-----------------------------
Votre choix ?
2
CLA: b0, INS: 02, P1: 00, P2: 00, Lc: 02, 03, e8, Le: 02, 00, 00, SW1: 00, SW2: 00
CLA: b0, INS: 02, P1: 00, P2: 00, Lc: 02, 03, e8, Le: 02, 03, e8, SW1: 90, SW2: 00
Votre balance vient d'être debiter de 1000
-----------------------------
Votre choix ?
2
CLA: b0, INS: 02, P1: 00, P2: 00, Lc: 02, 03, e8, Le: 02, 00, 00, SW1: 00, SW2: 00
CLA: b0, INS: 02, P1: 00, P2: 00, Lc: 02, 03, e8, Le: 02, 00, 14, SW1: 90, SW2:
Votre balance vient d'être debiter de 20
-----------------------------
Votre choix ?
3
CLA: b0, INS: 03, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 00, SW1: 00, SW2: 00
CLA: b0, INS: 03, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 00, SW1: 90, SW2: 00
Votre balance = 0
-----------------------------
Votre choix ?
1
CLA: b0, INS: 01, P1: 00, P2: 00, Lc: 02, 03, e8, Le: 02, SW1: 00, SW2: 00
CLA: b0, INS: 01, P1: 00, P2: 00, Lc: 02, 03, e8, Le: 02, 03, d4, SW1: 90, SW2:
Votre balance vient d'être debiter de 980
-----------------------------
Votre choix ?
1
CLA: b0, INS: 01, P1: 00, P2: 00, Lc: 02, 03, e8, Le: 02, SW1: 00, SW2: 00
CLA: b0, INS: 01, P1: 00, P2: 00, Lc: 02, 03, e8, Le: 00, SW1: 90, SW2: 00
-----------------------------
Votre choix ?
3
CLA: b0, INS: 03, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 00, SW1: 00, SW2: 00
CLA: b0, INS: 03, P1: 00, P2: 00, Lc: 00, Le: 02, 03, fc, SW1: 90, SW2: 00
Votre balance = 1020
```

Figure III.20. Capture d'écran du jeu d'essai du programme réactif sur le simulateur.

Les étapes suivies pour le développement sont :

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. Donc pas d'ajout dynamique de programme en runtime.
- Mettre le projet dans une librairie **JavaCard**, afin d'obtenir un fichier **.jar**

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.
- Implémenter le programme client.
- Interagir avec la carte par le bais du 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 :

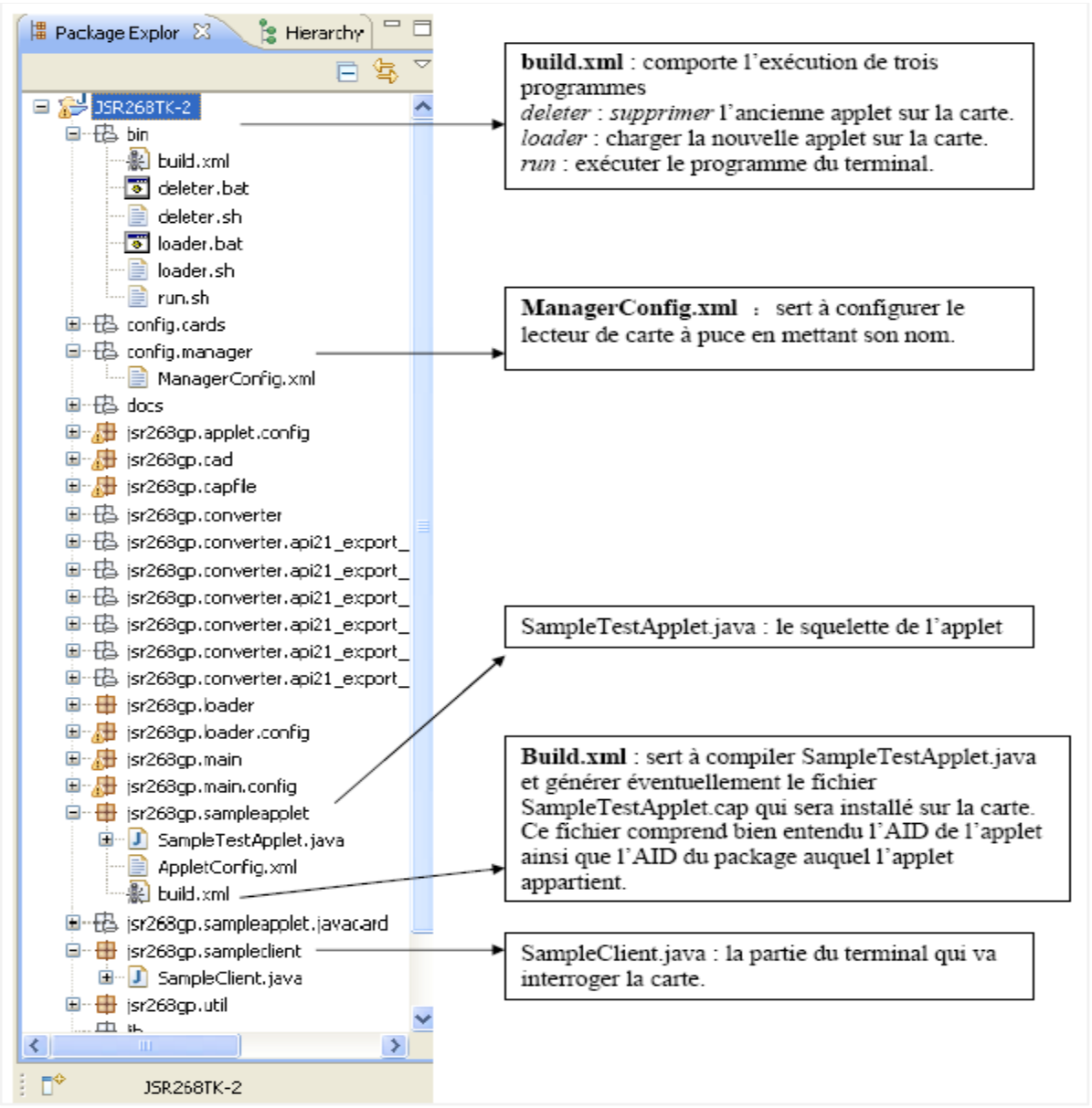

Figure III.21.Vue Eclipse après importation de la JSR-268.

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 explicité 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 bien adapté aux systèmes à faibles ressources notamment les cartes à puce. Concernant **Simple**, **Storm**, et **Glouton** elles sont plus évoluer dans leurs gestions des événements et des composants parallèles mais nécessitons des mécanismes qui n'existe pas sur la plateforme JAVACARD ou ils coûtent extrêmement chère en espace mémoire

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**, qui prend en charge le parallélisme et la communication par événements non values, ce dernier, a une taille de 16.4 Ko.

Bibliographie

Bibliographie

[HP] *D. Harel, A. Pnueli, "*On the Development of Reactive Systems*",* Proc. NATO ASI Series F, Vol.13 477-498, Springer-Verlag ;1985.

[Ser10] Ahmadou Al Khary SÉRÉ, Thèse DOCTEUR DE L'UNIVERSITÉ DE LIMOGES Discipline / Spécialité : Informatique et Applications présentée et soutenue par23 Sep 2010, Tissage de contremesures pour machines virtuelles embarquées

[197] "IDENTIFICATION SYSTEM". Brev. US Patent 3,641,316. 1972.

[Sta87] International Organization for Standardization. ISO7816. Rap. tech. ISO, 1987.

[Mor76] R. Moreno. "Methods of data storage and data storage systems". Brev. US Patent 3,971,916. 1976.

[Mor77] R. Moreno. "Data-transfer system". Brev. US Patent 4,007,355. 1977.

[Mor78] R. Moreno. "Systems for storing and transferring data". Brev. US Patent 4,092,524.
1978.

[Mor83] R.C.D. Moreno. "Method and apparatus for electrically connecting a removable article, in particular a portable electronic card". Brev. US Patent 4,404,464. 1983.

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

[Ran 07] Wolfgang Rankl « *Smart Card Applications Design Models for using and programming smart card***s** »; ISBN 978-0-470-05882-4; 2007.

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

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

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

[RA00] Raul ACOSTA BERMEJO, REactive Java Objects-Reactive Operating System,

Rapport de stage RSD, 2000

[RA03] Raul ACOSTA BERMEJO, Rejo : Langage d'Objets Réactifs et d'Agents, , thése de doctorat, 2003.

[JFS01] Jean-Ferdinand SUSINI, L'Approche Réactive au dessus de Java : sémantique et implémentation des SugarCubes et de Junior, thèse de doctorat, Sep 2001.

[CR04] Christian BRUNETTE Construction et simulation graphiques de comportements : le modèle des Icobjs, thèse de Doctorat, 2004.

[Gro02] William Grosso. *Java RMI*. O'Reilly & Associates, 2002.

[Sun09c] Sun Microsystems. *Java CardTM 3.0.1 Runtime Environment (JCRE) Specification, Connected Edition*. Sun Microsystems. 2009.

[Sun09d] Sun Microsystems. *Java CardTM 3.0.1 Servlet Specification, Connected Edition*. Sun Microsystems. 2009.

[Sun09e] Sun Microsystems. *Java CardTM 3.0.1 Specification, Classic Edition*. Sun Microsystems.2009.

[Sun09f] Sun Mycrosystems. *Java CardTM 3.0.1 Virtual Machine (JCVM) Specification, Connected Edition*. Sun Microsystems. 2009.

[Ott05] M. Otto. "Fault attacks and countermeasures". Thèse de doctorat. Université de Paderborn, 2005.

[Fre02] Jonas Fredriksson, Mikael Wahlberg « *SCOS Smart Card Operating System* »; School of Mathematics and Systems Engineering, MSI Växjö University, Report 02141, ISSN 1650-2647; November 2002.

[DEI09] Dimensions économique et industrielle des cartes à puces, Rapport final, Pôle interministériel de prospective et d'anticipation des mutations économiques, France, Novembre 2009.

[MU] Michel UGON, carte à puces, Techniques de l'Ingénieur, traité Électronique, réf E 3 520

[NTK07] NGUYEN Trong Khanh (B), La carte à puce, Rapport de TIPE, 2007.

[BS09] Samia Bouzefrane, Maître de Conférences "la technologie java card" CEDRIC – CNAM, France ; 2009.

[Lan08]. Jean-Louis Lanet , Rapport : Évaluation de mécanismes de détection d'attaques en fautes sur le tas statique d'une Java Card, 2008

Webographie

[1] www.cartesapuce.fr.

[2] http://www.scardshop.com/boutique/liste_rayons.cfm

Annexes

ANNEXE 1.

Les paramètres d'une commande APDU définie par l'ISO

Valeurs du CLA :

Valeur	Classe d'instruction
0x0n, 0x1n	Instruction selon l'ISO 7816-4, accès aux données et opérations de sécurité
De 0x20 à 0x7F	Réservés
0x8n ou 0x9n	Format utilisé pour le développement d'applications standard
0xAn	Instructions spécifiques aux vendeurs ou pour applications standard
De B0 à CF	Format utilisé pour le développement d'applications standard
De D0 à FE	Instructions spécifiques aux vendeurs ou pour applications standard
FF	Réservé

Dans la pratique, on utilise plutôt les des codes CLA supérieurs à 0x80.

Valeurs de l'INS quand le CLA est compris entre 0x00 et 0x7F :

Valeur	Description de la commande
0E	Erase Binary
20	verify
70	Manage Channel
82	External Authenticate
84	Get Challenge
88	Internal Authenticate
A4	Select File
B0	Read Binary
B2	Read Record(s)
C0	Get Response
C2	Envelope
CA	Get Data
D0	Write Binary
D2	Write Record

D6	Update Binary
DA	Put Data
DC	Update Record
E2	Append Record

Valeurs de SW :

Valeur SW1 SW2	Type d'erreur	Code d'erreur
61 00	Response bytes remaining	SW_BYTES_REMAINING_00
67 00	Wrong length	SW_WRONG_LENGTH
69 82	Security condition not satisfied	SW_SECURITY_STATUS_NOT_SATISFIED
69 83	File invalid	SW_FILE_INVALID
69 84	Data invalid	SW_DATA_INVALID
69 85	Condition of use not satisfied	SW_CONDITION_NOT_SATISFIED
69 86	Command not allowd	SW_COMMAND_NOT_ALLOWD
69 99	Applet selection failed	SW_APPLET_SELECTION_FAILED
6A 80	Wrong data	SW_WRONG_DATA
6A 81	Function not supported	SW_FUNC_NOT_SUPPORTED
6A 82	File not found	SW_FILE_NOT_FOUND
6A 83	Record nor found	SW_RECORD_NOT_FOUND
6A 84	Not enough memory space in file	SW_FILE_FULL
6A 86	Incorrect parameters P1,P2	SW_INCORRECT_P1P2
6B 00	Wrong parameters P1,P2	SW_WRONG_P1P2
6C 00	Correct expected length (le)	SW_CORRECT_LENGTH
6D 00	INS value not supported	SW_INS_NOT_SUPPORTED

6E 00	CLA value not supported	SW_CLA_NOT_SUPPORTED
6F 00	No precise diagnosis	SW_UNKNOWN

Valeurs de SW renvoyées par le CardManager dans le cas d'une carte Java Card :

62 83	Selected applet blocked
63 00	Cryptogram not verified
63 10	More data available
65 81	Memory failure
67 00	Incorrect length
6A 88	No element matches the search criteria
6C XX	Only XX bytes of data are available
6F 80	Stack overflow
90 00	Command process without error
92 40	Memory problem
94 02	Acces out of range
94 84	Algorithm not supported
94 85	Invalid key check value
9F XX	XX bytes of data available

ANNEXE 2.

Le gestionnaire de fichier des cartes à puce

La gestion des fichiers est la tâche principale d'un système d'exploitation d'une carte à puce. Le rôle d'un gestionnaire de fichiers n'est pas seulement de fournir les accès en lecture et/ou en écriture aux fichiers, ou bien la création ou la suppression des fichiers, mais aussi d'accorder des accès privilégiés. Le gestionnaire de fichiers est particulièrement important parce que la plupart des applications de la carte à puce sont basées sur des fichiers **[Ran 07]**.

La norme ISO/CEI 7816-4 constitue la spécification qui définit le fonctionnement logique de la carte à puce, c'est-à-dire l'arborescence des fichiers, la structure des messages échangés entre le lecteur et la carte, le contenu et les moyens de sécurisation des fichiers et leurs droits d'accès.

1. La structure interne d'un fichier :

Les systèmes de gestion de fichiers modernes des cartes à puce ont une structure orienté-objet, cela veut dire que toutes les informations concernant un fichier sont sauvegardées dans ce même fichier. Dans de tels systèmes, les fichiers sont divisés en deux parties :
La première partie est appelée l'en-tête. Elle contient les informations concernant la disposition et la structure du fichier ainsi que les conditions d'accès. Les données modifiables de l'utilisateur sont sauvegardées dans la deuxième partie appelé le corps du fichier qui est relié par un pointeur à l'en-tête **[Ran 03].**
Les entêtes sont stockés sur une page mémoire, les données sont hébergées sur une autre page.

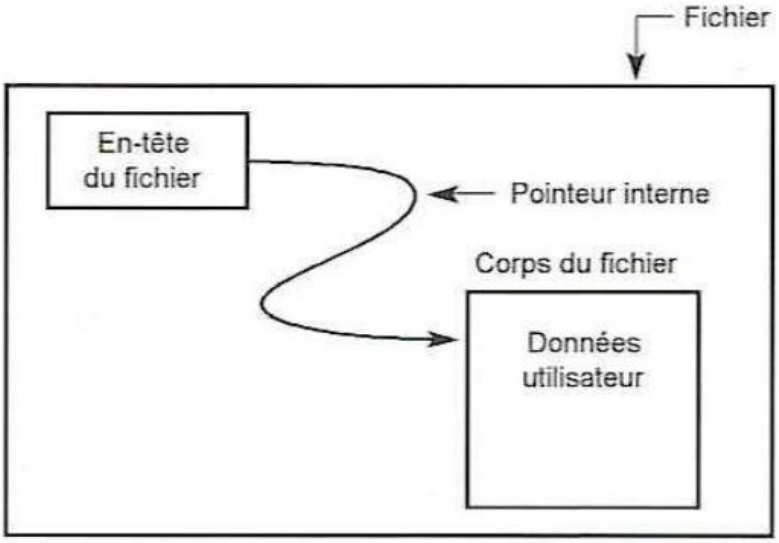

Figure 2.1 La structure interne d'un fichier dans les cartes à puce [Ran 03].

2. Les types de fichiers [Fre 02] :

Il existe deux types de fichiers supportés par le standard ISO 7816 :

✓ Fichier dédié (Dedicated File : DF) : c'est un répertoire de dossiers qui peut contenir d'autres répertoires et/ou des fichiers élémentaires de données. il ne peut pas contenir directement des données autres que l'identificateur et les références des fichiers qu'il commande. chaque DF est associé à une application particulière.

✓ Fichier élémentaire (Elementary File : EF) : il ne peut pas contenir d'autres fichiers mais seulement les données des applications et les données du système d'exploitation. Les EFs sont toujours localisés dans un répertoire. Il existe deux sous-types:

- Les fichiers élémentaires internes (Internal Elementary File : IEF) : ils sont utilisés par le système d'exploitation pour sauvegarder des données internes. Il existe plusieurs types de fichiers internes :

  - les fichiers de clés : stocke des clés servant à construire les clés de session selon le protocole utilisé.

- les fichiers de codes secrets : stocke les codes confidentiels utilisés pour contrôler l'accès aux fichiers des applications.

- les fichiers de porte-monnaie : pour chaque porte-monnaie, le fichier indique le solde maximal, le débit limite pour chaque transaction et le solde courant.

- les fichiers de certificats : ils sont utilisés dans le cas d'un chiffrement à clé publique.

3. L'arborescence des fichiers :

L'organisation logique des fichiers dans une carte à puce est un arbre dont le répertoire racine est appelé MF (Master File), il est analogue au répertoire 'C :' dans un ordinateur. Il est défini d'une façon unique dans l'arborescence et créée automatiquement à la personnalisation de la carte (après le RESET).

Trois ou quatre niveaux de l'arbre sont utilisés par les applications actuelles. Le système d'exploitation des cartes à puce supporte rarement plus de huit niveaux.
Toutefois, cette arborescence est rigide et elle ne convient pas aux situations dynamiques dans lesquelles des fichiers sont créés et éliminés au fur et à mesure que des applications sont mises à jour.

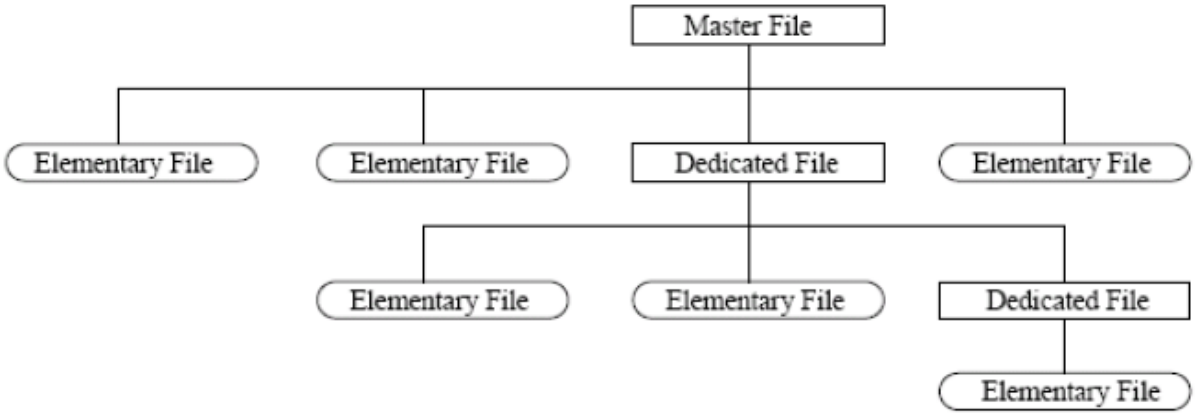

Figure 2.2 Système de fichier.

4. Les noms de fichiers [Ran 07] :

Chaque fichier possède un nom appelé FID (File IDentifier) codé sur deux octets. Le FID du MF est fixé à une valeur 3F00, par contre les autres FIDs peuvent être choisis librement. Chaque répertoire DF possède un nom supplémentaire en plus de son FID appelé DF name, et il peut être adressé par ce dernier dans l'arbre de fichiers. A chaque EF est assigné un AID, et un SFI (Short File Identifier) qui peut être transmis comme un paramètre dans une commande de lecture ou d'écriture affin de sélectionner le EF directement.

5. La structure d'un EF:

Il existe différentes structures de stockage de données définies dans le standard ISO 7816. La plus populaire est la structure transparente dans laquelle un fichier est vu comme une séquence d'unités de données de taille maximale 255 octets. Les autres variantes sont des structures à enregistrement dans lesquelles un fichier est une séquence d'enregistrements individuellement identifiables. Les méthodes structurées définissent 3 types de fichiers **[Fre 02]** :

✓ Des fichiers à enregistrements séquentiels de taille fixe : Chaque enregistrement est identifié par un numéro d'enregistrement (dans l'ordre de création). On peut stocker jusqu'à 255 enregistrements de 255 octets chacun.

✓ Des fichiers à enregistrements séquentiel de taille variable : Chaque enregistrement est identifié par un numéro d'enregistrement (dans l'ordre de création) ;

✓ Des fichiers à enregistrement cyclique de taille fixe : chaque enregistrement est identifié par un numéro d'enregistrement (dans l'ordre inverse de création).

Il existe une grande différence entre la méthode cyclique et les deux méthodes séquentielles. Dans la première, lorsque le dernier enregistrement est atteint et quand on avance d'un pat, on reboucle vers le premier enregistrement ce qui n'est pas le cas dans les méthodes séquentielles où on s'arrête au derniers enregistrement.

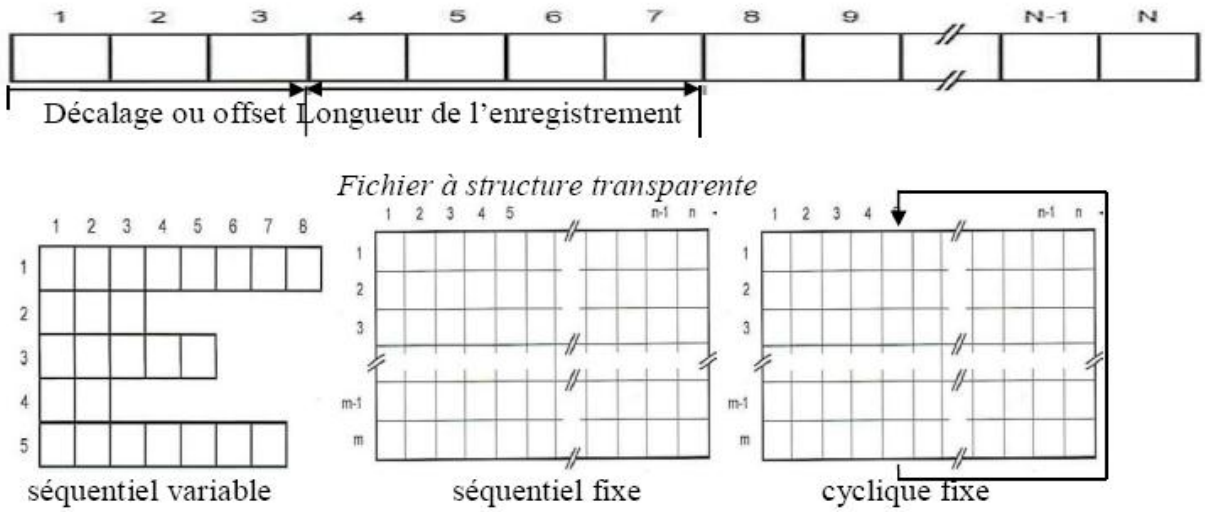

Figure 2.3 Les différentes structures d'un fichier élémentaire.

6. les attributs des fichiers [Ran 03]:

Les fichiers dans les cartes à puce peuvent avoir aussi plusieurs attributs selon le système d'exploitation. L'attribut le mieux connu est partageable et non partageable, il est utilisé afin de spécifier pour chaque fichier s'il autorise ou pas les accès concurrents en lecture/écriture. Il existe plusieurs autres attributs, mais ils ne sont pas standardisés.

Glossaire

Glossaire

Applet : Une Applet est une mini application qui ne peut tourner qu'à l'intérieur d'un navigateur. Il est sécuritaire, petit et ne peut faire que des choses limitées.

Bytecode : Dans un environnement interprété, le code Java n'est conçu pour aucune plate-forme spécifique mais une sorte de 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 Java Virtual Machine.

CAD : *Card Acceptance Device* - Lecteur de cartes à puce.

CAP : *Converted Applet* - Le fichier CAP contient une représentation binaire exécutable de toutes les classes d'un package Java. Le format CAP est le format dans lequel les logiciels sont chargés sur la JavaCard.

Classe : Notion de base en programmation objet, il s'agit du modèle générique d'objet (composante modulaires d'un programme) qui ne différent que par la valeur de leurs attributs. En Java, qui est un langage objet, il n'est possible de créer une application qu'à partir des classes.

EEPROM : *Electrical Erasable Programmable Read-Only* - Comme la ROM, elle peut conserver les données quand la mémoire n'est plus alimentée. La différence est que l'on peut modifier les données stockées dans la mémoire.

ISO : International Standards Organization.

JCRE : JavaCard Runtime Environment - 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 c'est le système d'exploitation des smart cards.

JCVM : JavaCard Virtual Machine - Les contraintes imposées par les SmartCard (la taille de la JVM et du système de runtime ne devant pas dépasser 16 Ko) ont

contraint à définir un système demandant le moins de ressource mémoire possible. La JVM a donc été réduite : JCVM

JVM : Java Virtual Machine - La JVM est l'équivalent d'un ordinateur virtuel résidant entièrement en mémoire. Cette JVM doit être implémentée sur chaque plate-forme. La JVM fournit un niveau d'abstraction et d'indépendance entre le programme Java compilé et une plate-forme matérielle.

Objet : Instance de classe créée à partir d'une classe lorsque le programme est lancé, 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 déboguage, leurs propriétés d'héritage et leurs droits d'accès.

PIN : *Personnal Input Number* - Le PIN est un code personnel, exemple le code d'un téléphone portable, d'une carte bancaire, … .

PIX : Le PIX est une partie de l'AID. Sa valeur est libre et donc choisi par le programmeur.

RAM : *Random Access Memory* - mémoire utilisé comme espace temporaire pour modifier est stocker les données. Son contenu n'est pas préservé à l'arrêt de l'alimentation.

RID : Le RID est une partie de l'AID. Sa valeur est fixer par de Sun MicroSystems, il en faire une demande à l'ISO.

ROM : *Read Only Memory* - Mémoire utilisée pour stocker les programmes fixes de la carte, les données permanentes, … . On ne peut ainsi ni les modifier ni les supprimer. Aucune alimentation n'est nécessaire pour conserver les données.