RENPAR'15 / CFSE'3 / SympAAA'2003
La Colle sur Loup, France, 15 au 17 octobre 2003

Économiser l'or du banquier

Antoine Galland[§, *], Mathieu Baudet[‡, †]

[§] Gemplus Research Labs,
La Vigie, Avenue du Jujubier, ZI Athelia IV,
13705 La Ciotat Cedex - France
antoine.galland@research.gemplus.com

[‡] LSV/CNRS UMR 8643 & INRIA Futurs, projet SECSI & ENS Cachan,
61, avenue du Président Wilson,
94235 Cachan Cedex - France
mathieu.baudet@lsv.ens-cachan.fr

Résumé
Cet article étudie le problème de la disponibilité des ressources dans le contexte des codes mobiles pour de petits systèmes embarqués tels que la carte à puce. Il présente une architecture dédiée permettant de contrôler et d'optimiser l'usage d'une ressource dans un système multitâche. Celle-ci s'appuie sur un algorithme d'ordonnancement de tâches qui minimise la quantité de ressource requise par le système. Notre architecture comporte deux parties. La première calcule statiquement la quantité de ressource nécessaire à l'aide d'un treillis spécifique, en se plaçant dans le cadre de l'interprétation abstraite. La seconde garantit la disponibilité des ressources à l'exécution grâce à un algorithme efficace d'évitement d'interblocage, dans la lignée de l'*algorithme du banquier* de Dijkstra. Notre algorithme peut être considéré comme une généralisation de l'algorithme de Gold [14] à une algèbre de processus. Nous détaillons son utilisation pratique sur une architecture Java pour carte à puce.

Mots-clés : Contrôle de ressource, carte à puce, évitement d'interblocage, algèbre de processus, interprétation abstraite.

1. Introduction

La carte à puce est un objet fortement contraint et normalisé, aux ressources de calcul limitées : processeur ($4,7 - 100$ MHz), communication ($9600 - 192000$ Bauds), mémoire (typiquement $1 - 4$ ko RAM, $32 - 128$ ko ROM et $16 - 64$ ko Flash RAM). Cependant, ses mécanismes de sécurité physique (*tamper resitance*) en font l'un des objets pour code mobile les plus sûrs. Ainsi, depuis l'émergence [25] de la carte à puce jusqu'à aujourd'hui [10], intégrer un haut niveau de sécurité avec si peu de ressources physiques a toujours été le défi principal des fabricants de cartes à puce. Il faut donc garder à l'esprit qu'optimiser les ressources physiques ou logiques est primordial dans un tel système embarqué.
Aujourd'hui, les cartes à puce modernes offre l'opportunité de charger du code dans la carte chez l'utilisateur final (principe de *post issuance*). Il a donc fallu résoudre les problèmes de sécurité liés au chargement de code mobile. Dans les machines virtuelles Java Card [4], plusieurs solutions ont été proposées afin d'intégrer un vérifieur de code dans la carte dans le but d'assurer que les applications soient bien typées au moment du chargement [22, 2, 11]. Dans cette démarche d'une carte à puce toujours plus sûre, le contrôle des ressources apparaît comme l'étape suivante [13]. En effet, un fournisseur de services pour cartes à puce voudrait s'assurer qu'une application a toujours suffisamment de ressources pour s'exécuter correctement.

[*] Étudiant en thèse à l'université Pierre et Marie Curie, Laboratoire LIP6.
[†] Travail réalisé chez Gemplus dans le cadre du DEA Programmation et de l'École Nationale Supérieure des Télécommunications.

L'objectif de ce papier est de proposer une architecture qui résolve ces deux problèmes, à savoir, optimiser l'usage d'une ressource tout en garantissant sa disponibilité dans un environnement multiapplicatif. La section 2 étudie différents modèles de contrôle de ressource et introduit le problème d'évitement d'interblocage. Puis, la section 3 présente notre cadre d'étude basé sur l'analyse de code et l'évitement d'interblocage sur une ressource. Ensuite, la section 4 fournit les détails d'une architecture pour carte à puce. Finalement, la section 5 présente des résultats d'expérimentation et la section 6 conclut.

2. État de l'art

2.1. Le contrôle des ressources

Au démarrage d'une application, rien ne garantit qu'elle aura suffisamment de ressources (mémoire, CPU, espace disque...) pour s'exécuter normalement. Dans les plates-formes pour code mobile, une des solutions retenues est l'approche par contrat [15]. Dans cette approche, l'application déclare ses besoins dans un contrat et la plate-forme d'accueil, une fois le contrat accepté, s'engage à le respecter et à fournir les ressources demandées. Cependant, il est nécessaire de surveiller l'application pendant son exécution afin de voir si elle ne consomme pas plus que la quantité autorisée. Cette surveillance (*monitoring*) est en pratique assez coûteuse. De plus, une fois le contrat annulé, il est généralement trop tard pour reprendre une exécution normale même si des mécanismes de reprise (*call-back*) sont envisageables [8].

Afin de réduire le surcoût lié à la surveillance, il serait préférable de vérifier une fois pour toute que l'application va respecter son contrat. Cela implique de pouvoir déterminer et borner la consommation d'un programme par analyse statique ou à l'aide d'un système de type [7, 17]. Ces procédés, souvent complexes, sont difficiles à embarquer dans de petits systèmes. Néanmoins, des techniques de vérification de preuve de type « Proof-Carrying Code » peuvent être utilisées afin de vérifier en ligne (c.-à-d. sur la plate-forme d'accueil) le résultat du calcul hors ligne [24, 23].

Une fois le contrat accepté entre les parties, la plate-forme doit faire en sorte de fournir les ressources réclamées. Une solution simple consiste à réserver dès le départ la quantité de ressource demandée par chaque application. C'est le cas des Java Card 2.1 [4] avec la gestion mémoire. Le développeur d'application Java Card doit regrouper toutes ses allocations mémoires dans une méthode[3] d'installation.

L'inconvénient de cette solution est le gaspillage de ressource quand on multiplie les applications. En effet, celles-ci n'utilisent pas forcément tout leur quota à chaque instant. Au contraire des « pics » de consommation de ressource apparaissent ponctuellement. De plus, même si ce n'est pas le cas, on peut retarder une ou plusieurs tâches afin d'éviter cette situation. En résumé, avec cette solution l'usage de la ressource est loin d'être optimisée.

Partant de ce constat [13], notre objectif est donc de garantir la disponibilité d'une ressource (dès le démarrage) pour chaque application tout en minimisant la quantité globale de ressource dans le système (afin d'économiser de la ressource). Les différentes approches que nous venons de voir ne résolvent pas simultanément ces deux problèmes. Le modèle de gestion de ressource exposé dans ce papier permet (i) de garantir qu'une application aura suffisamment de ressource pour s'exécuter (dès le démarrage) ; (ii) de minimiser la quantité de ressource du système. Le paragraphe suivant détaille l'architecture retenue.

2.2. Le lien avec l'évitement d'interblocage

Dans la plupart des systèmes, quand un programme demande plus de ressources que celles disponibles, une erreur est soulevée et le programme s'arrête. Considérant notre objectif, en gardant à l'esprit que nous sommes dans un système multitâche, une solution plus économe est de suspendre temporairement le programme demandeur dans l'espoir que des ressources soient libérées plus tard par une autre tâche. Dans ce cas les méthodes d'allocation et de désallocation sont équivalentes aux fameuses primitives de synchronisation P « Puis-je » et V « Vas-y » de Dijkstra [12] avec comme sémaphore la ressource.[4]

Désormais on s'intéresse donc à une architecture dans laquelle une allocation insatisfiable suspend l'exécution du processus en cours. Cette approche nous conduit aux problèmes classiques en théorie de la concurrence, à savoir :

- la *famine*, situation dans laquelle un ou plusieurs processus sont bloqués indéfiniment, tandis que

[3] `javacard.framework.Applet.install()`.

[4] Le nom des primitives provient de l'Hollandais *Passeren* pour P et *Vrygeven* pour V.

les autres continuent de s'exécuter ;

– l'*interblocage* (ou *deadlock*), qui intervient lorsque tous les processus sont mis en attente faute de ressource.

Le problème de la famine peut être éludé si l'on considère des processus qui terminent en un temps fini, ce qui est généralement le cas des programmes pour carte à puce. Dans la suite nous nous intéressons donc plus particulièrement à l'évitement des interblocages.

Le problème de prévention des interblocages a été étudié en premier par Dijkstra [12]. Ce dernier a introduit une représentation simple du problème de manière géométrique appelé *graphe d'avancement* (voir Figure 1). Sur cette figure, les axes représentent la progression de chacune des tâches au cours

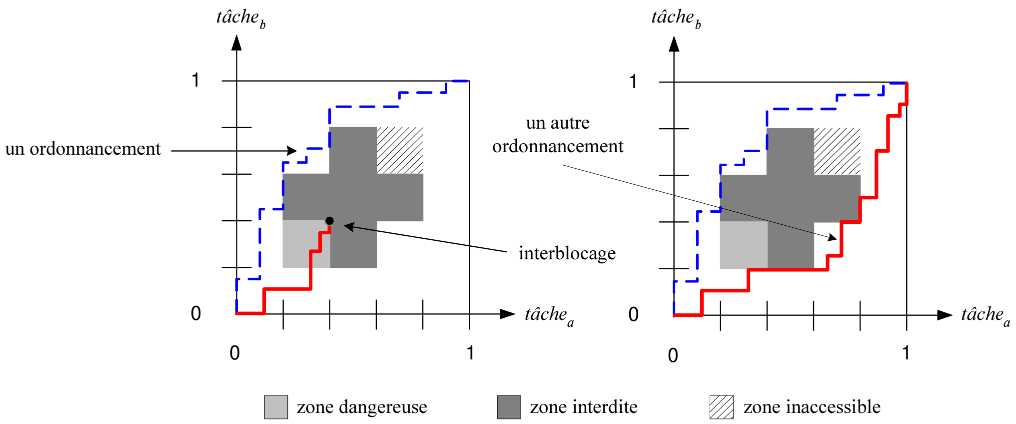

FIG. 1 – *Graphe d'avancement et évitement d'interblocage.*

du temps. On peut considérer un *ordonnancement valide* comme un chemin évitant les *zones interdites* – celles par définition où la somme des consommations instantanées des processus excède la quantité totale de ressource disponible. Un état est *sûr* s'il existe un ordonnancement à partir de cet état qui permette de terminer toutes les tâches en cours. A l'inverse, les états non-sûrs (ou *dangereux*) conduisent nécessairement à un interblocage. Une stratégie d'évitement d'interblocage consiste à vérifier avant chaque étape d'exécution que le système reste dans un état sûr. Ces algorithmes cherchent à détecter ou prédire les états non sûr afin de les éviter.

Le premier algorithme d'évitement d'interblocage a été introduit par Dijkstra dans [12] sous le nom d'*algorithme du banquier*. Généralisé par Haberman dans [16], cet algorithme donne un critère de sûreté polynomial pour un système concurrent à plusieurs ressources renouvelables. Depuis lors, l'évitement d'interblocage a été étudié dans de nombreux systèmes d'allocation de ressource, et notamment pour des chaînes de fabrication industrielles[5] (voir par exemple [21]). Dans [14], Gold montre la NP-complétude du problème dans le cas général, et donne une classification de la complexité de différents sous-problèmes. Le cas d'une seule ressource est déjà identifié comme étant polynomial. L'algorithme présenté à la section 3.1 peut être considéré comme une généralisation de l'algorithme de Gold à une algèbre de processus.

3. L'évitement d'interblocage sur une ressource

3.1. Algèbre de processus

Nous allons en premier lieu considérer un langage de processus simple permettant de modéliser un ensemble de tâches accédant une seule ressource. Ceci s'avérera très utile par la suite. Les termes du

[5] en anglais sous le titre *Flexible Manufacturing Systems (FMS).*

langage, appelé *processus* ou *programmes* sont définis inductivement dans la grammaire suivante :

$$p \ ::= \ \epsilon \mid x \mid (p_1\, p_2) \mid (p_1 \parallel p_2)$$

où x est un entier, positif ou négatif. Intuitivement, la signification de cette syntaxe peut être décrite comme suit :

- ϵ constitue le processus vide (c.-à-d. terminé) ;
- x représente une instruction qui alloue une quantité x de ressource (lorsque x est négatif il s'agit donc d'une désallocation) ;
- $(p_1\, p_2)$ désigne une séquence de processus : p_1 puis p_2 ;
- $(p_1 \parallel p_2)$ est l'exécution concurrente de p_1 et p_2.

Pour formaliser la sémantique du langue, nous choisissons un système de transitions étiquetées décrites dans le Tableau 1. Les ϵ-transitions servent à simplifier des termes, tandis que les transitions normales *émettent* une allocation x. Une transition $\xrightarrow{x}$ consiste donc intuitivement en une série de transitions structurelles, suivie d'une allocation x en position d'évaluation, puis d'une autre série de transitions structurelles.

$$\begin{array}{ccccl}
\dfrac{}{(\epsilon\, p) \xrightarrow{\epsilon} p} & \dfrac{}{(\epsilon \| p) \xrightarrow{\epsilon} p} & \dfrac{}{(p \| \epsilon) \xrightarrow{\epsilon} p} & & \text{(règles structurelles)} \\[2ex]
\dfrac{}{x \xrightarrow{x} \epsilon} & \dfrac{p \xrightarrow{\epsilon *} \xrightarrow{x} \xrightarrow{\epsilon *} p'}{p \xrightarrow{x} p'} & & & \text{(règles d'évaluations)} \\[2ex]
\dfrac{p_1 \xrightarrow{x} p_1'}{(p_1\, p_2) \xrightarrow{x} (p_1'\, p_2)} & \dfrac{p_1 \xrightarrow{x} p_1'}{(p_1 \parallel p_2) \xrightarrow{x} (p_1' \parallel p_2)} & \dfrac{p_2 \xrightarrow{x} p_2'}{(p_1 \parallel p_2) \xrightarrow{x} (p_1 \parallel p_2')} & & \text{(règles de contextes)}
\end{array}$$

TAB. 1 – *Sémantique << à petits pas >> à l'aide de transitions étiquetées.*

Suivant les conventions habituelles des automates, les séquences de transitions émettent une liste de nombres entiers : $p \xrightarrow{x_1 x_2 \ldots x_n} p'$ si $p \xrightarrow{x_1}\xrightarrow{x_2} \cdots \xrightarrow{x_n} p'$. Il est facile de prouver, en utilisant notre sémantique, que chaque programme termine après avoir émis toutes ses allocations. Ainsi, nous pouvons définir les traces d'exécution d'un programme p comme un ensemble de listes l tel que $p \xrightarrow{l} \epsilon$. Par exemple, les traces d'exécution du programme $((1\,2) \| {-3})$ sont : $1\,2\,(-3)$, $1\,(-3)\,2$ et $(-3)\,1\,2$.
Pour une trace d'exécution, nous pouvons facilement définir la quantité de ressource nécessaire. Nous l'appellerons le *coût* d'une trace.

$$\mathcal{C}(x_1 x_2 \ldots x_n) \stackrel{def}{=} \max_{0 \le i \le n} \left(\sum_{1 \le j \le i} x_j \right) \quad (\ge 0)$$

Grâce à lui, nous pouvons maintenant formuler un critère simple de sûreté. Du fait qu'un état est décrit par un terme p de l'algèbre de processus, si M est la quantité de ressource disponible, un état p est dit sûr si et seulement s'il existe une trace l telle que : $p \xrightarrow{l} \epsilon$ et $\mathcal{C}(l) \le M$. Étant donné que nous cherchons à minimiser la quantité M de ressource du système, le critère de sûreté pour un état p est donc : $\min\{\mathcal{C}(l), p \xrightarrow{l} \epsilon\} \ \le \ M$.
Ce que nous allons montrer dans la prochaine section, c'est une manière efficace de calculer ce minimum que nous appellerons *coût d'un processus* p :

$$\mathcal{C}(p) \stackrel{def}{=} \min\{\mathcal{C}(l), p \xrightarrow{l} \epsilon\}$$

3.2. Calcul du coût d'un processus à l'aide de listes normalisées

Une première classe d'algorithmes possibles pour calculer $\mathcal{C}(p)$ consiste à explorer le système de transitions entre p et ϵ. Avec des techniques de programmation dynamique, de tels algorithmes demandent un temps de calcul linéaire en fonction du nombre d'états du système. Malheureusement ce nombre croît de manière exponentielle en fonction du nombre de processus (phénomène d'*explosion combinatoire*), ce qui rend ces techniques inutilisables dans notre cas.

Nous allons donc tenter de simplifier le problème en réduisant le nombre d'états à considérer, l'objectif étant de conserver une solution exacte. L'idéal serait de pouvoir calculer $\mathcal{C}(p)$ récursivement à travers p en utilisant le fait qu'une seul ressource est considérée ici.

Un premier pas dans cette direction est présenté dans le Tableau 2 où l'algorithme calcule récursivement une paire d'entiers $B(p)$. On peut montrer que, pour chaque p, la première coordonnée de $B(p)$ est une

$B(\epsilon)$	$=$	$(0,\,0)$
$B(x)$	$=$	$(\max(x,0),\,x)$
$B(p_1\,p_2)$	$=$	$B(p_1)\cdot B(p_2)$
$B(p_1 \,\|\, p_2)$	$=$	$B(p_1)\times B(p_2)$
avec :		
$(c_1,\delta_1)\cdot(c_2,\delta_2)$	$=$	$(\max(c_1,\delta_1+c_2),\ \delta_1+\delta_2)$
$(c_1,\delta_1)\times(c_2,\delta_2)$	$=$	$(\min(\max(c_1,\delta_1+c_2),\max(c_2,\delta_2+c_1)),\ \delta_1+\delta_2)$

TAB. 2 – *Un premier algorithme avec les blocs.*

sur-approximation de $\mathcal{C}(p)$ tandis que la seconde calcule la somme de toute les allocations. En utilisant cet algorithme sur l'exemple précédent, on obtient $B((1\,2)\|{-3}) = B(1\,2)\times(0,-3) = (3,3)\times(0,-3) = (0,0)$. Ce qui est ici la valeur exacte pour $\mathcal{C}((1\,2)\|{-3}) = 0$.

En fait, les paires d'entiers de la forme (c,δ) avec $c \geq \max(0,\delta)$ n'ont pas été choisies innocemment. Nous les définissons comme des *blocs*. Elles sont équivalentes aux *requêtes partielles* définies par Gold [14] avec comme paire $(c, c-\delta)$.

Bien que rapide et simple, l'algorithme du Tableau 2, qui emploie des blocs, donne malheureusement des résultats très approchés sur des entrées plus compliquées. Il y a deux raisons à cela :

- Premièrement, l'opérateur $\times$ ne reflète pas l'associativité de la composition parallèle. Par exemple, $B[((2,0)\|(1,1))\,\|\,(0,-1)] = (2,1)\times(0,-1) = (1,0)$ tandis que $B[(2,0)\,\|\,((1,1)\|(0,-1))] = (2,0)\times(0,0) = (2,0)$;
- Deuxièmement, l'opérateur $\cdot$ cause une perte significative d'information, en masquant les étapes internes d'une séquence. Par exemple, bien que le coût de $(-1\,1)\|(2\,-1)$ est **1**, l'algorithme ne trouve pas la trace optimale $(-1\,2\,-1\,1)$, en effet $B((-1\,1)\|(2\,-1)) = B((-1\,1))\times B((2\,-1)) = (0,0)\times(2,1) = (\mathbf{2},1)$.

Pour résoudre le premier problème, une possibilité est de généraliser l'opérateur $\times$ afin de traiter un ensemble de blocs. Une manière naturelle de faire ceci est de définir :

$$\begin{aligned}(c_1,\delta_1)\times\ldots\times(c_n,\delta_n) &\stackrel{def}{=} \prod_{\sigma\in\Sigma_n}(c_{\sigma 1},\delta_{\sigma 1})\cdot\ldots\cdot(c_{\sigma n},\delta_{\sigma n})\\ &= \left(\min_{\sigma\in\Sigma_n}\max_{1\leq i\leq n}\left(c_{\sigma i}+\sum_{1\leq j<i}\delta_{\sigma j}\right),\ \sum_{1\leq i\leq n}\delta_i\right)\end{aligned}$$

où Σ_n représente l'ensemble des permutations sur $\{1\ldots n\}$ et $\sqcap$ est la borne inférieure[6] sur les paires de nombres entiers. Après avoir fait cela, nous sommes face à un problème de minimisation non trivial.

[6] greatest lower bound.

Heureusement il peut être résolu en temps polynomial, et même en temps $O(n\log(n))$, en triant les blocs selon le pré-ordre total suivant :

$$(c_1,\delta_1) \preceq (c_2,\delta_2) \text{ ssi: } \quad \begin{array}{r|ll} & \delta_1 \leq 0 \text{ et } \delta_2 \geq 0 & (1) \\ \text{ou} & \delta_1 < 0, \delta_2 < 0 \text{ et } c_1 \leq c_2 & (2) \\ \text{ou} & \delta_1 > 0, \delta_2 > 0 \text{ et } \delta_1 - c_1 \leq \delta_2 - c_2 & (3) \end{array}$$

On peut montrer en effet que la relation $\preceq$ est réflexive, transitive et totale et que $(c_1,\delta_1)\times\ldots\times(c_n,\delta_n) = (c_{\sigma 1},\delta_{\sigma 1})\cdot\ldots\cdot(c_{\sigma n},\delta_{\sigma n})$ dès que $(c_{\sigma 1},\delta_{\sigma 1}) \preceq \ldots \preceq (c_{\sigma n},\delta_{\sigma n})$. La définition de $\preceq$ repose sur les mêmes idées que la généralisation de l'algorithme du banquier par Gold [14] : par exemple la règle (1) signifie << les producteurs (de ressource) avant les consommateurs >> ; la règle de (2) signifie << les producteurs les moins exigeants en premier >> ; enfin la règle (3) peut être vue comme une image du (2) en inversant le temps.

Nous allons maintenant nous intéresser au second problème: la perte d'information causé par l'opérateur $\cdot$. L'exemple précédent $(-1\,1)\|(2\,-1)$ montre que fusionner les blocs implique la perte de nombreux ordonnancements valides. En fait, les traces d'exécution perdues sont celles qui *entrelacent* les processus entre eux.

Pour cette raison, nous allons travailler avec des *listes de blocs* plutôt qu'avec des blocs, afin de garder plus d'information sur le comportement des programmes. Ainsi, l'opérateur $\cdot$ ne va plus systématiquement fusionner les blocs comme précédemment. Nous décidons par exemple que $(0,-2)\cdot(2,2) = (0,-2)(2,2)$, ce qui est plus précis que simplement $(0,0)$. En effet $(0,-2)(2,2)$ en parallèle avec $(2,0)$ peut avoir comme entrelacement $(0,-2)(2,0)(2,2)$ ce qui nous donne un coût de 0, tandis que $(0,0)$ en parallèle avec $(2,0)$ a un coût de 2.

Définir $\cdot$ comme la concaténation habituelle des listes nous mènerait à une structure de taille comparable à celle du programme d'origine ce qui n'est pas acceptable, pour des raisons de place et d'efficacité. La suite de ce paragraphe présente une procédure de *normalisation* des listes qui répond à ce problème. À cet effet nous définissons une relation entre les blocs appelée *relation de simplification* et notée $\mathcal{S}$, qui déterminera quels blocs adjacents peuvent être fusionnés :

$$(c_1,\delta_1)\mathcal{S}(c_2,\delta_2) \text{ ssi: } \quad \begin{array}{r|l} & \delta_2 \leq 0 \text{ et } c_1 \geq c_2 + \delta_1 \\ \text{ou} & \delta_1 \geq 0 \text{ et } c_1 \leq c_2 + \delta_1 \end{array}$$

Nous pouvons alors définir la règle de réécriture suivante sur les liste de blocs :

$$\begin{array}{rcl} (c_1,\delta_1)(c_2,\delta_2) & \rightarrow & (\max(c_1,\delta_1+c_2),\delta_1+\delta_2) \text{ quand } (c_1,\delta_1)\mathcal{S}(c_2,\delta_2) \\ (0,0) & \rightarrow & \epsilon \end{array}$$

où ϵ représente la liste vide. On peut démontrer formellement que ce critère de simplification ne perd pas d'information et qu'il est optimal [1]. Intuitivement, deux blocs sont fusionnés lorsque tout ordonnancement qui intercalerait un bloc entre les deux peut être transformé en un ordonnancement équivalent (ou moins coûteux) qui laisse les deux blocs adjacents. On montre également que ces règles sont confluentes et fortement normalisantes ; c'est à dire que $\rightarrow$ termine toujours et donne pour chaque liste une unique *forme normale*.

Une propriété remarquable des listes normalisées est qu'elles sont triées suivant le critère d'ordre $\preceq$ mentionné précédemment (et même strictement croissantes). Grâce à cette propriété et aux règles de normalisation, nous pouvons généraliser directement la définition de $\times$ aux *listes* de blocs (normalisées). Au lieu donc de trier les blocs, nous *fusionnons* les listes normalisées au sens du *tri-fusion* (voir par exemple [19]), le critère d'ordre utilisé étant toujours $\preceq$.

Finalement le Tableau 3 décrit un algorithme calculant récursivement une liste de blocs $L(p)$ << équivalente >> à p. Le coût de p est alors égal au coût de $L(p)$, défini par :

$$\mathcal{C}(\,(c_1,\delta_1)\ldots(c_n,\delta_n)\,) = \max_{1\leq i\leq n}\left(c_i + \sum_{1\leq j<i}\delta_j\right) \text{ et } \mathcal{C}(\epsilon) = 0$$

$L(\epsilon)$	$=$	ϵ
$L(x)$	$=$	$(\max(x,0),\ x)$ si $x \neq 0$, ϵ sinon
$L(p_1\, p_2)$	$=$	$L(p_1) \cdot L(p_2)$
$L(p_1 \parallel p_2)$	$=$	$L(p_1) \times L(p_2)$
avec:		
$l_1 \cdot l_2$	$=$	$normalise(l_1 l_2)$
$l_1 \times l_2$	$=$	$normalise(fusionne(l_1, l_2))$

TAB. 3 – *Algorithme final avec les listes de blocs.*

À ce stade, nous disposons donc d'un algorithme (Tableau 3) qui calcule le coût *exact* d'un processus et non plus seulement une majoration (Tableau 2), sachant que le critère d'ordre $\preceq$ et que la relation de simplification $\mathcal{S}$ sont optimaux.

3.2.1. Complexité de l'algorithme

Étant donné la complexité linéaire de la normalisation et des opérations de fusion, les calculs de $L(p_1) \cdot L(p_2)$ et de $L(p_1) \times L(p_2)$ s'effectuent tout deux en temps $O(n_1 + n_2)$ où n_i est la taille de la liste l_i. Si m est la profondeur de l'arbre syntaxique (binaire) de p et n sa taille, la traduction de p s'effectue donc en temps $O(nm)$. En pratique, les opérations de fusion diminuent très notablement la taille des listes à manipuler (voir l'étude expérimentale section 5.1) de sorte que l'algorithme de traduction a un comportement quasi-linéaire.

3.3. Analyse statique de programme par interprétation abstraite

Jusqu'ici nous avons expliqué comment calculer rapidement le coût d'un système et donc déterminer s'il est sûr ou non, en supposant que l'on pouvait le modéliser à l'aide de l'algèbre de processus. Nous avons besoin maintenant d'une manière pour représenter les systèmes qui exécutent de vrais programmes. Ceci est fait en analysant les programmes statiquement, et en les annotant avec l'information extraite. De cette façon, un *module d'évitement d'interblocage* pourra déterminer l'état du système pendant l'exécution, et détecter des zones non sûres avant qu'elles se produisent.

Étant donné l'algorithme précédent pour calculer les coûts de processus, les listes normalisées apparaissent comme une structure de donnée naturelle pour représenter les demandes de ressource pendant l'analyse. Cependant, les instructions de branchement (`if`, `while`...) ne sont pas représentées directement dans l'algèbre de processus, ce qui est pénalisant si l'on souhaite utiliser notre algorithme sur des langages plus réalistes.

Nous allons voir maintenant comment mettre en œuvre les *listes normalisées* en présence de structures de contrôle. L'ingrédient essentiel est que ces listes peuvent être équipées d'une structure de *treillis*. Ce résultat important nous permet d'utiliser la théorie de l'*interprétation abstraite* [6] en guise d'analyse statique de programme. Dans ce treillis, l'ordre $\sqsubseteq$ sur les listes (réflexif, transitif, anti-symétrique) représente la perte d'information d'une manière naturelle. Par exemple $l_1 \sqsubseteq l_2$ ssi $\forall l, \mathcal{C}(l_1 \times l) \leq \mathcal{C}(l_2 \times l)$. De plus $\sqsubseteq$ est stable par passage au contexte : $l_1 \sqsubseteq l_2$ implique que $\forall l$, $(l_1 \times l) \sqsubseteq (l_2 \times l)$, $(l_1\, l) \sqsubseteq (l_2\, l)$ et $(l\, l_1) \sqsubseteq (l\, l_2)$.

La structure de treillis des listes normalisées est décrite dans le Tableau 4. La borne supérieure $\sqcup$ nécessite une fonction auxiliaire *join* décrite dans le Tableau 5. Les opérations arithmétiques sur $\mathbb{Z}$ sont étendues à $\mathbb{Z} \cup \{\pm\infty\}$ de la manière usuelle, avec la convention $(+\infty) + (-\infty) = +\infty$.

La section suivante détaille l'utilisation pratique de ces opérations à travers l'implémentation d'une architecture Java pour carte à puce afin d'analyser et exécuter des applications Java.

4. Application à Java pour carte à puce

Nous allons maintenant décrire une architecture spécialisée pour carte à puce qui analyse des programmes Java (au niveau du *bytecode*) et contrôle ensuite l'accès à une ressource. Étant donné la puissance de calcul limité des cartes, toutes les opérations ne sont pas faites dans la carte. Ainsi, nous utilisons une architecture scindée en deux parties afin de déléguer les calculs complexes hors de la carte.

Relation d'ordre	: $l_1 \sqsubseteq l_2$ ssi $\mathcal{C}(l_1 \times -l_2) = 0$
Borne supérieure	: $l_1 \sqcup l_2 = normalise(join(0, l_1, l_2))$
Borne inférieure	: $l_1 \sqcap l_2 = -((-l_1) \sqcup (-l_2))$
Le plus petit élément (*Bottom*)	: $\bot = (0, -\infty)$
Le plus grand élément (*Top*)	: $\top = (+\infty, +\infty)$
L'opposé	: $-(\,(c_1, \delta_1) \ldots (c_n, \delta_n)\,) =$
$normalise(\,(0, -c_1)(\max(-\delta_1, 0), -\delta_1) \ldots (0, -c_n)(\max(-\delta_n, 0), -\delta_n)\,)$	

TAB. 4 – *Treillis des listes normalisées.*

$join(x, \epsilon, \epsilon)$	$=$	ϵ
$join(x, (c_1, \delta_1)q_1, \epsilon)$	$=$	$shift(x, c_1, \delta_1)\, join(x + \delta_1, q_1, \epsilon)$
$join(x, \epsilon, (c_2, \delta_2)q_2)$	$=$	$shift(-x, c_2, \delta_2)\, join(x - \delta_2, \epsilon, q_2)$
$join(x, (c_1, \delta_1)q_1, (c_2, \delta_2)q_2))$	$=$	$shift(x, c_1, \delta_1)\, join(x + \delta_1, q_1, l_2)$ si $(c_1, \delta_1) \preceq (c_2, \delta_2)$
$join(x, (c_1, \delta_1)q_1, (c_2, \delta_2)q_2))$	$=$	$shift(-x, c_2, \delta_2)\, join(x - \delta_2, l_1, q_2)$ sinon
$shift(x, c, \delta)$	$=$	$(\max(c, x) - \max(0, x),\ \max(\delta, x) - \max(0, x))$

TAB. 5 – *Calcul de la borne supérieure.*

La partie hors carte est en charge de la prédiction de ressource sur des applications Java tandis que la partie dans la carte prévient l'interblocage sur la ressource grâce à un serveur de ressource. La section suivante explique plus en détails le rôle de chaque partie.

4.1. Analyse statique de programmes Java (hors carte)

Le programme d'analyse comprend trois parties. Premièrement, nous résumons le graphe de flot de contrôle de chaque application ; puis nous extrayons et résumons les besoins en ressource à l'aide de des listes normalisées ; enfin ces informations sont utilisées pour annoter les fichiers Java (`.class`).

4.1.1. Graphe de flot de contrôle

L'objectif de cette partie est de donner la représentation la plus précise possible du graphe de flot de contrôle de l'application. Cette étape construit le graphe des méthodes d'appels (inter-méthodes) et le graphe d'exécution pour chaque méthode (intra-méthodes).

Les appels de méthodes virtuelles, les exceptions et la création de tâches (*threads*) dynamique en Java rendent ce calcul difficile à faire statiquement. Nous avons pour cela repris les techniques d'estimation de graphe de flot de contrôle des langages orientés objets [9]. Dans notre cas, nous traitons les appels virtuels en prenant la consommation au pire cas (par unification) des classes parentes. De même, pour les exceptions, une évaluation approchée est de calculer l'ensemble des points d'arrivés possibles (*exception handlers*) [3]. Ces solutions demandent un ensemble fixe de classes Java à analyser (API Java et applications cartes). Nous ne traitons pas pour l'instant le chargement de classe dynamique. Concernant la durée de vie des *threads* Java[7], pour que la destruction soit prise en compte, nous avons besoin de déterminer son point de création (associé) statiquement. Afin d'optimiser la partie suivante de l'analyse, nous ne gardons que les instructions de contrôle (branchements, appels de fonctions...) et les opérations d'allocation/désallocation sur la ressource.

4.1.2. Interprétation abstraite en utilisant le treillis de listes normalisées

Nous pratiquons une analyse arrière par interprétation abstraite [6] en utilisant les opérations du treillis du Tableau 4. L'opérateur de borne supérieur $\sqcup$ s'utilise pour les branchements et les boucles. Dès qu'une « fuite » de ressource est détectée au niveau d'une boucle, nous mettons $\top$ comme résultat d'analyse pour cette portion de code. Étant donné que chaque méthode est analysée indépendamment des autres (ignorant le contexte d'appel), l'algorithme d'évitement d'interblocage à besoin d'être informé

[7] `thread.start()` et `thread.join()`.

du contexte réel d'appel lors de l'exécution. Pour cela, des *annotations* sont rajoutées au programme lorsque l'on entre et ressort d'une méthode (idem pour les *threads*). La Figure 2 donne une illustration des annotations ajoutées automatiquement au bytecode Java.[8] Le ≪ contrat global ≫ par exemple, correspond à : $(5,2)(2,1) = \underbrace{L(1)}_{\texttt{alloc(1)}} \cdot (\underbrace{(L(4) \cdot L(-3))}_{\texttt{thread.start()}} \times (\underbrace{(L(-2) \sqcup L(2))}_{\texttt{foo(args)}} \cdot \underbrace{L(-1)}_{\texttt{alloc(-1)}}))$.

Avant

```java
class SimpleExample implements Executable {

  int [] getGlobalAnnotation() {
    return null;
  }

  void run(String[] args){
    Server.alloc(1)
    SimpleThread thread = new SimpleThread();

    thread.start();

    foo(args);

    Server.alloc(-1);

  }

  void foo(Object obj) {
    if (obj == null) {
      Server.alloc(-2);
    } else {
      Server.alloc(2);
    }

  }

  static class SimpleThread extends Thread {
    public void run() {
      Server.alloc(4);
      Server.alloc(-3);

  }
}
```

Après

```java
class SimpleExample implements Executable {

  int [] getGlobalAnnotation() {
    return [(5,2),(2,1)]; // contrat global
  }

  void run(String[] args){
    Server.alloc(1,[(4,1)(2,1)]);
    SimpleThread thread = new SimpleThread();
    Server.fork([(2,1)], thread, [(4,1)]);
    thread.start();
    Server.call([(2,2)], [0,-1]);
    foo(args);
    Server.discard();
    Server.alloc(-1,[]);
    Server.end();
  }

  void foo(Object obj) {
    if (obj == null) {
      Server.alloc(-2,[]);
    } else {
      Server.alloc(2,[]);
    }
    Server.end();
  }

  static class SimpleThread extends Thread {
    public void run() {
      Server.alloc(4, [(0,-3)]);
      Server.alloc(-3, []);
      Server.end();
  }
}
```

FIG. 2 – *Exemple de Java annoté utilisant l'API du serveur de ressource.*

Des optimisations peuvent être faites en réduisant les appels non nécessaires au serveur. Par exemple, quand une méthode ne ≪ manipule ≫ pas une ressource, il n'est pas utile de l'entourer des méthodes `Server.call(..)` et `Server.discard()`. Une fois le fichier annoté, il peut être directement utilisé et chargé par le serveur de ressource dans la carte.

4.2. Phase dynamique d'évitement d'interblocage (dans la carte)

Nous avons implémenté le serveur de ressource intégralement en Java, au dessus d'une machine virtuelle Java (JVM) spécialisée dans les petits systèmes embarqués et notamment les cartes à puce de prochaine génération [20]. Cette JVM expérimentale est plus proche de la spécification J2ME [18] que du standard Java Card actuel [4]. Elle propose notamment une API de *threads* nécessaire au module de ressource.

Le serveur de ressource est informé de chaque changement de contexte grâce aux annotations (allocation sur la ressource, désallocation, création de thread, lancement d'une nouvelle application... que nous appellerons *requêtes*). Pour chaque requête au serveur, l'algorithme d'évitement d'interblocage simule la transaction. Si elle conduit à un état sûr, la transaction est validée, sinon (état dangereux détectés), le thread de l'application demandeuse est suspendu. Il sera réveillé quand l'état global du serveur de ressource aura changé. Nous utilisons uniquement l'API des threads Java (`wait()` et `notifyAll()`) pour suspendre et réveiller les threads.

[8] Par commodité, nous présentons ce résultat en Java où les listes normalisées sont entre crochets.

L'algorithme d'ordonnancement est distribué entre l'ordonnanceur interne de la JVM et le serveur de ressource. Le premier est en charge de l'équité de partage du processeur entre les threads (via un algorithme de *round-robin* par exemple) et le second garantit l'évitement d'interblocage sur la ressource (par dessus l'ordonnanceur interne de la JVM). L'avantage de cette architecture est que la JVM et son ordonnanceur interne n'ont pas besoin d'être modifiés. Grâce aux API de threads Java, le serveur de ressource est écrit uniquement en Java. La section suivante présente les résultats expérimentaux de notre architecture.

5. Résultats expérimentaux

5.1. Étude expérimentale de la normalisation

Pour se faire une idée du comportement statistique de la normalisation, on a étudié expérimentalement la longueur après normalisation sur des échantillons de 1000 listes tirées au sort. Le tirage au sort consistait à générer un vecteur de taille $n+1$ dont les coordonnées étaient choisies uniformément entre $-m$ et $+m$, puis à traduire ce vecteur en n blocs élémentaires. Pour $m = 100$, le critère de simplification $\mathcal{S}$ a donné:

valeur de n	minimum	moyenne	maximum
10	1	2.47	5
100	1	4.21	9
1000	1	4.43	9
10000	1	4.41	9

À m constant, la taille moyenne des listes normalisées tend donc expérimentalement vers une limite extrêmement basse. Ce résultat dépend évidemment du modèle de tirage au sort des listes, mais il laisse à penser que la taille des listes manipulées dépassera rarement la dizaine de blocs en pratique.

5.2. Validation expérimentale

Nous avons testé le serveur de ressource sur une ressource abstraite (un simple compteur). Le but étant de valider ce modèle, et d'évaluer les coûts et bénéfices de notre architecture avant de l'appliquer sur une ressource réelle. Les résultats présentent trois jeux de tests ($\mathtt{Test}_i$). Chaque test comporte plusieurs threads qui modélisent un groupe d'applications concurrentes. Le compteur de ressource est manipulé intensivement afin tester la réactivité et la correction du serveur. Le Tableau 6 compare les ressources nécessaires suivant trois modèles de ressource différents et discute du niveau d'équité du processeur entre les applications (c.-à-d. du temps de partage du processeur entre les threads) :

- Le premier modèle réserve toutes les ressources nécessaire pour les applications au démarrage. La quantité réservée représente la consommation au pire cas pour l'ensemble du test. Il n'y a pas d'optimisation sur la ressource mais chaque application pour être sélectionnée par l'ordonnanceur à tout moment donc l'équité entre les applications est optimale ;
- Le deuxième modèle exécute une version approchée de l'algorithme sur les *blocs*. Bien que des entrelacements soient possibles, ils ne sont pas envisagés lors du calcul de l'algorithme. Ainsi, la prédiction d'interblocage est sur-approximée et l'équité entre les applications est faible ;
- Le troisième modèle est celui présenté dans ce papier où l'évitement d'interblocage est basé sur les *listes normalisées*. Les threads sont suspendus quand un blocage est détecté. Il s'agit d'une *prédiction* à cause notamment des instructions de branchements. En pratique l'équité est forte car les applications sont rarement suspendues pour un long moment, l'ordonnanceur interne de la JVM élisant prioritairement les threads ayant le plus patienté afin d'équilibrer le partage.

Ces tests montrent qu'un large gain sur la ressource peut être obtenu (`Test2` et `Test3` du Tableau 6). Dans certains cas toutefois, il n'est pas si avantageux d'utiliser ce modèle (`Test1`). Il est important de noter que le gain (ou non) peut être déterminé à l'avance dès le chargement de l'application. Ainsi, on peut déterminer quels modèles choisir avant le lancement de l'application. S'il n'y pas de gain sur la ressource « possible » , il est alors plus judicieux de choisir un modèle de ressource plus simple.

Le fichier binaire (`.class`) d'une application annotée est 47% plus gros que le code original. Cela représente l'accroissement au pire cas possible étant donné que nos tests ne font *que* des manipulations

	Ressource nécessaire par modèle		
	Sans optimisation	Blocs	Listes Normalisées
`Test1` (2 threads)	18	16	16
`Test2` (4 threads)	12	6	2
`Test3` (2 threads)	7	4	4
Équité (en pratique)	optimale	faible	forte

TAB. 6 – *Ressource nécessaire et équité suivant chaque modèle.*

sur la ressource. De plus cet accroissement est dû en grande partie à l'insertion de code (requêtes au serveur) et non pas aux données (listes normalisées). Un codage plus approprié des appels au serveur permettrait de réduire cette taille.

Concernant le comportement dynamique du serveur, nos mesures de partage du processeur ont montré qu'un ordonnancement *mixte* entre le serveur de ressource et l'ordonnanceur interne était tout à fait adapté à la fois à une bonne équité et à l'évitement d'interblocage. Le calcul de l'évitement d'interblocage ne causant pas de ralentissement notoire en pratique grâce essentiellement à nos listes normalisées de faible taille (*cf.* section 5.1).

6. Conclusion

Réserver l'ensemble des ressources nécessaires au démarrage est une solution simple et efficace pour garantir la disponibilité des ressources aux applications. L'inconvénient est que cette solution conduit à un gaspillage quand on généralise ce modèle dans un contexte multiapplicatif. L'architecture que nous avons exposée permet à la fois de garantir la disponibilité d'une ressource et d'optimiser son utilisation grâce à un algorithme d'évitement d'interblocage. Notre contribution est une généralisation des travaux de Gold dans le cas à une ressource à l'aide d'un langage de processus simple qui permet d'appréhender une classe de système plus complexe. Notre algorithme, basé sur une structure de donnée appelée *listes normalisées* prend un temps de calcul linéaire en pratique et est donc parfaitement adapté à l'ordonnancement dynamique. Grâce à la structure de treillis de ces listes normalisées, nous pouvons utiliser les techniques de l'interprétation abstraite et ainsi calculer un algorithme d'évitement d'interblocage efficace pour des langages de programmation. Nous avons appliqué avec succès notre architecture à un sous-ensemble réaliste du bytecode Java en soulignant qu'aucune modification du JVM n'est nécessaire et que nos fichiers annotés restent des fichiers Java (`.class`) valides. Le modèle de contrôle de ressource présenté dans ce papier est destiné à économiser le plus de ressource possible. L'économie obtenue se fait au prix d'une architecture et d'un algorithme non trivial. Nous pensons cependant que ce modèle est approprié pour d'autres systèmes embarqués fortement contraints.

De manière théorique, plusieurs améliorations peuvent être explorées. La première consiste à gérer les programmes qui ne terminent pas en temps fini (boucle infinie, entrées-sorties). La deuxième est d'étendre ce modèle afin de gérer plusieurs ressources. Néanmoins, ce problème étant NP-complet en général, cela devrait se faire au prix d'une approximation importante. Enfin, étant donné que l'analyse statique est faite hors carte (c.-à-d. dans un milieu non sûr), il est nécessaire de vérifier la cohérence des annotations (vérification dynamique, signature cryptographique par une tiers de confiance, preuve de code...).

De manière pratique, plusieurs études sont actuellement en cours. La principale est d'appliquer ce modèle sur une ressource réelle, parmi les candidats : la mémoire, les fichiers, les tampons de communication. Le problème principal est de retrouver dans les programmes les allocations/désallocations et de déterminer la quantité de ressource requise. Si on prend la mémoire, ressource que les fabriquants de carte essayent le plus d'économiser, ce problème devient difficile surtout dans les langages utilisant une libération automatique de la mémoire (langage à *Garbage Collector* notamment). Nous projetons d'utiliser une adaptation des techniques d'*analyse d'échappement* [5] afin de déterminer statiquement les libérations sur la mémoire des applications Java. L'objectif étant de déterminer les conditions pertinentes d'utilisation de ce modèle de ressource.

Bibliographie

1. Mathieu Baudet. Contrôle de ressource et évitement des interblocages sur la mémoire, September 2002. Mémoire de DEA (Programmation : Sémantique, Preuves et Langages), Gemplus Research Labs. `http://www.lsv.ens-cachan.fr/~baudet/publis/memoire_dea.ps.gz`.
2. Ludovic Casset, Lilian Burdy, and Antoine Requet. Formal Development of an embedded verifier for Java Card Byte Code. In *the IEEE International Conference on Dependable Systems & Networks (DSN)*, pages 51–58, Washington, D.C., USA, June 2002.
3. Byeong-Mo Chang, Jang-Wu Jo, Kwangkeun Yi, and Kwang-Moo Choe. Interprocedural Exception Analysis for Java. In *16th ACM Symposium on Applied Computing (SAC)*, LasVegas, USA, March 2001.
4. Zhiqun Chen. *Java CardTM Technology for Smart Cards : Architecture and Programmer's Guide*. The JavaTM Series. Addison Wesley, 2000.
5. Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape Analysis for JavaTM. In *ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 2–19, Denver USA, November 1999.
6. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *the 4th Annual ACM SIGPLAN-SIGACT*, pages 238–252, Los Angeles, California, 1977.
7. Karl Crary and Stephanie Weirich. Resource Bound Certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 184–198, Boston, MA, January 2000.
8. Grzegorz Czajkowski and Thorsten von Eicken. JRes: A Resource Accounting Interface for Java. In *the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 21–35, Vancouver, British Columbia, Canada, October 1998.
9. Greg DeFouw, David Grove, and Craig Chambers. Fast Interprocedural Class Analysis. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 222–236, San Diego, CA, USA, January 1998.
10. Damien Deville, Antoine Galland, Gilles Grimaud, and Sébastien Jean. Smart Card Operating Systems: Past, Present and Future. In *the 5th USENIX/NordU Conference*, Västerås, Sweden, February 2003.
11. Damien Deville and Gilles Grimaud. Building an "impossible" verifier on a Java Card. In *2nd USENIX Workshop on Industrial Experiences with Systems Software (WIESS)*, Boston, USA, December 2002.
12. Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.
13. Antoine Galland, Damien Deville, Gilles Grimaud, and Bertil Folliot. Contrôle des ressources dans les cartes à microprocesseur. In *1er congrès Logiciel Temps Réel Embarqués (LTRE)*, Toulouse, France, 2002.
14. E. Mark Gold. Deadlock prediction: Easy and difficult cases. *SIAM Journal on Computing*, 7(3):320–336, 1978.
15. Frédéric Guidec and Nicolas le Sommer. Towards resource consumption accounting and control in Java: a practical experience. In *16th European Conference on Object-Oriented Programming(ECOOP), Workshop on Resource Management for Safe Language*, Malaga, Spain, June 2002.
16. Arie Nicolaas Habermann. Prevention of system deadlocks. *Communications of the ACM*, 12(7):373–377, 1969.
17. Martin Hofmann. A Type System for Bounded Space and Functional In-Place Update – Extended Abstract. In Gert Smolka, editor, *Programming Languages and Systems: 9th European Symposium on Programming (ESOP)*, volume 1782 of *LNCS*, Berlin, Germany, March 2002. Springer-Verlag.
18. JavaTM 2 Platform Micro Edition (J2METM). Connected Limited Device Configuration (CLDC) Specification Version 1.0. `http://java.sun.com/j2me/`.
19. Donald Ervin Knuth. *The Art of Computer Programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., 1998.
20. Laurent Lagosanto. Next-Generation Embedded Java Operating System for Smart Cards. In *Gemplus Developer Conference (GDC)*, Singapore, November 2002.
21. M. Lawley and S. Reveliotis. Deadlock Avoidance for Sequential Resource Allocation Systems: Hard and Easy Cases. *The Journal of FMS*, 13(4):385–404, 2001.
22. Xavier Leroy. Bytecode Verification for Java smart card. *Software Practice & Experience*, 32:319–340, 2002.
23. Mobile Resource Guarantees (MRG). European Project IST-2001-33149, 2002. `http://www.dcs.ed.ac.uk/home/mrg/`.
24. Aloysius K. Mok and Weijiang Yu. TINMAN: A Resource Bound Security Checking System for Mobile Code. In D. Gollmann, G. Karjoth, and M. Waidner, editors, *7th European Symposium on Research in Computer Security (ESORICS)*, volume 2502 of *LNCS*, Zurich, October 2002. Springer-Verlag.
25. Jean-Jacques Quisquater. The adolescence of smart cards. *Future Generation Computer Systems*, 13:3–7, 1997.