(19) RÉPUBLIQUE FRANÇAISE

INSTITUT NATIONAL
DE LA PROPRIÉTÉ INDUSTRIELLE

PARIS

(11) **N° de publication :** **2 884 994**
(à n'utiliser que pour les commandes de reproduction)

(21) **N° d'enregistrement national :** **05 51036**

(51) Int Cl[8] : **H 04 L 9/00** (2006.01)

(12) DEMANDE DE BREVET D'INVENTION A1

(22) **Date de dépôt :** 22.04.05.

(30) **Priorité :**

(43) **Date de mise à la disposition du public de la demande :** 27.10.06 Bulletin 06/43.

(56) **Liste des documents cités dans le rapport de recherche préliminaire :** *Se reporter à la fin du présent fascicule*

(60) **Références à d'autres documents nationaux apparentés :**

(71) **Demandeur(s) :** *GEMPLUS Société anonyme* — FR.

(72) **Inventeur(s) :** GAUTERON LAURENT et BENOIT ALEXANDRE.

(73) **Titulaire(s) :**

(74) **Mandataire(s) :** BREESE DERAMBURE MAJEROWICZ.

(54) PROCEDE DE VERIFICATION DE PSEUDO-CODE CHARGE DANS UN SYSTEME EMBARQUE, NOTAMMENT UNE CARTE A PUCE.

(57) La présente invention concerne la vérification d'applications en langage interprété de type bytecode (pseudo-code) chargées sur des dispositifs électroniques portables, notamment une carte à puce.

La présente invention se rapporte à un procédé de vérification d'une application (31) interprétable par une machine virtuelle (42), ladite application étant chargée dans un dispositif électronique portable (1) comprenant au moins un processeur (2) et une mémoire vive (5), le procédé consistant à procéder, après le chargement de ladite application dans le dispositif et avant sa validation, à des contrôles sur le code de ladite application par un traitement mis en oeuvre par le processeur (2) caractérisé en ce qu'il comporte:

- lors d'un appel à un sous-programme, une étape de sauvegarde du contexte courant de vérification (200 à 203) dans la mémoire vive (5),
- une étape de création et d'activation d'un nouveau contexte de vérification (206 à 209) dédié au sous-programme,
- lors de la fin du sous-programme, une étape de restauration du contexte de vérification (200 à 203) précédemment sauvegardé.

FR 2 884 994 - A1

PROCÉDÉ DE VÉRIFICATION DE PSEUDO-CODE CHARGÉ DANS UN SYSTÈME EMBARQUÉ, NOTAMMENT UNE CARTE À PUCE

La présente invention concerne un procédé de vérification de cohérence de codes destinés à un système embarqué.

L'invention se rapporte plus particulièrement, mais non exclusivement, au domaine des applications en langage interprété de type *bytecode* (pseudo-code), chargées sur une carte à puce.

On considère, par la suite, le terme « système embarqué » au sens large, notamment comme système destiné à tout dispositif électronique portable, par exemple une carte à puce, dont les ressources de traitement et de mémorisation sont relativement limitées.

De même, un langage interprété est un langage non compilé dont l'exécution des lignes de code nécessite la présence de moyens auxiliaires permettant d'interpréter ce code. Un exemple d'un tel langage est le langage Java (marque déposée) largement répandu dans les solutions applicatives pour cartes à puce. L'application Java ou « *applet* » est interprétée par une machine virtuelle Java associée. Il existe également des solutions matérielles, par exemple une puce dédiée, qui implémentent l'équivalent de la machine virtuelle. Par la suite, le terme « machine virtuelle » réfèrera indifféremment à des moyens auxiliaires de type logiciel ou matériel permettant d'interpréter un langage interprété associé.

La vérification de pseudo-code (*bytecode*), par exemple et non exclusivement Java (marque déposée), est un élément

clé dans la sécurité des plateformes Java (marque déposée). Cette vérification consiste notamment à s'assurer de l'intégrité et de la conformité d'un programme *bytecode* à des propriétés, par exemple le typage des variables du code, ce programme *bytecode* étant interprété par une machine virtuelle, c'est-à-dire une machine à pile (mémoire à accès empilant et dépilant) et à registres (cases mémoires à accès indexé). Ces opérations de vérification sont relativement complexes et gourmandes en ressources (mémoire vive, temps de traitement).

Avec le développement des cartes à puce, des solutions Java (marque déposée) ont été intégrées dans celles-ci. Au cours de la vie de la carte à puce, de nouvelles applications, par exemple des *applets* Java (marque déposée), sont chargées dans cette carte afin d'être exploitées. Ces *applets* peuvent être corrompues ou trafiquées et peuvent réaliser des appels à des zones mémoires non autorisées créant des dysfonctionnements sur la machine virtuelle. Avec l'apparition des cartes à puce et l'intégration de programmes dans ces cartes, cette vérification est devenue extrêmement compliquée dans tous les systèmes embarqués eut égard au peu de ressources disponibles.

Il est fréquent que des programmes de type *bytecode* mettent en oeuvre des appels à d'autres programmes ou à des sous-programmes. Une distinction peut être faite entre les appels à des programmes partageant le même contexte d'exécution que le programme appelant et les appels à des programmes d'autres méthodes présentant un contexte d'exécution dédié spécifique. Dans le cadre de l'invention, on s'intéresse particulièrement au cas des appels à des programmes ou sous-programmes partageant le même contexte d'exécution que le programme appelant. On utilisera par la

suite le terme de « sous-programme » pour définir les parties de code qui peuvent être atteintes depuis d'autres parties de code partageant le même contexte d'exécution qu'il s'agisse indifféremment d'un programme appelé ou d'un sous-programme appelé (ensemble de lignes de code en commun avec le programme appelant). De tels appels peuvent être mis en œuvre dans des fonctions du type « Goto », « If » ou lors d'appels à des macros.

On notera, à titre d'exemple, en langage Java (marque déposée) l'existence d'un couple d'instructions JSR (*Jump to Sub-Routine* - saut de sous-routine) et RET (*return from sub-routine* - retour de sous-routine) mettant en œuvre des sous-routines ou sous-programmes. La figure 1 propose un exemple de code présentant un sous-programme (B7 à RET) avec un appel à ce sous-programme (ligne 4 : JSR B7). Lorsqu'en fin de sous-programme, on exécute une instruction RET, la machine virtuelle exécute le *bytecode* suivant le JSR ayant appelé le sous-programme. Pour mémoriser l'information du JSR appelant, l'adresse de celui-ci est enregistrée sur la pile de la machine virtuelle, mais sans instance de typage de cette information : il s'agit d'une valeur numérique dans la pile qui dépend du flot d'exécution. Le problème de cet enregistrement réside dans le fait que les vérifieurs standard travaillent sur la base des typages et n'ont pas accès aux valeurs numériques à proprement parler. Il n'est donc pas possible de déterminer statiquement les parties de code appelant le sous-programme.

En effet, ces algorithmes de vérification appliquent l'algorithme d'unification pour chaque *bytecode*, algorithme dont le principe est le suivant : dans un *bytecode*, à un point de convergence d'une même variable avec deux typages différents (provenant de deux sauts de sous-programmes différents, par exemple), la variable prend le typage du premier ancêtre commun aux deux typages (cette notion

d'ancêtre commun découle des principes d'héritage du langage orienté objet type Java (marque déposée)). Et en cas d'incompatibilité de typage, un type appelé « TOP » est affecté à la variable. Puis lors de la modélisation du *bytecode*, si le typage attendu par le *bytecode* n'est pas compatible avec celui reçu, le code est rejeté.

Dans le cas des sous-progammes, deux appels différents au même sous-programme peuvent être réalisés alors même qu'une variable n'a pas le même typage. Ainsi il peut y avoir une erreur de vérification (typages incompatibles) alors qu'il n'y a pas de problème de typage (s'agissant de deux contextes différents, les deux typages ne pourront interférer pendant l'exécution du code par la machine virtuelle).

Pour les cartes Java (marque déposée), la vérification de pseudo-codes assure qu'aucune manipulation illégale sur le typage des éléments utilisés par le *bytecode* n'est réalisée. Deux propriétés sont à vérifier :

- pour chaque *bytecode* la hauteur de la pile est toujours la même quelque soit le chemin d'exécution,
- pour chaque *bytecode*, il existe un typage des variables (registres) et des étages de pile qui est compatible avec le *bytecode* quel que soit le chemin d'exécution.

Pour ce faire, on explore tous les chemins d'exécution possibles de façon statique. Il s'agit d'une exécution abstraite du *bytecode*.

Pour chaque ligne de *bytecode*, la vérification de l'intégrité requiert la mémorisation de beaucoup d'informations. Il a été montré alors qu'il suffit d'effectuer cette mémorisation uniquement pour les cibles de sauts. De plus, l'algorithme a besoin de stocker des informations complémentaires comme le pointeur d'instruction

ou « *program counter* » (pointeur sur la ligne de code au point de vérification courant), la *worklist* (liste des lignes de codes à vérifier par la suite) et la *frame* courante (ensemble des typages des registres et de la pile au point qui est en train d'être examiné, enregistrés dans la mémoire vive du dispositif).

On connaît des solutions de vérification externes - comme la solution de SUN MICROSYSTEMS (marque déposée) - dans lesquelles le *bytecode* est initialement vérifié lors d'un traitement hors carte. Une fois qu'il est validé, il est chargé sur la carte à puce. L'inconvénient de ces solutions réside en ce qu'entre la vérification du *bytecode* et le chargement dans la carte, il existe une possibilité de trafiquer le code. Ces solutions ne garantissent donc pas l'intégrité entre le code initial et le code finalement chargé sur la carte puis exécuté.

On connaît également le vérifieur SUN MICROSYSTEMS (marque déposée) où la vérification est faite hors carte en milieu sécurisé et qui permet de signer le programme. La carte n'a plus qu'à vérifier la signature à la réception du programme.

Le portage de cette solution dans la carte présente des inconvénients, notamment une trop forte consommation de RAM.

On connaît également la vérification avec un composant de preuve (*Proof Carrying Code*). Un composant de preuve est calculé hors carte puis rajouté au programme lors de sa transmission sur la carte. Il s'agit d'insérer des infos de typage dans le code. De ce fait, la vérification sur la carte est grandement facilitée et ne requiert que très peu de mémoire vive RAM (*Random Access Memory*).

L'inconvénient de cette solution réside dans la nécessité d'un prétraitement hors carte : le calcul de la preuve ; et dans la taille plus importante des données (*bytecode* et preuves) à transmettre et à stocker : allongement des temps de transmission, augmentation de la consommation de bande passante.

On connaît également le vérifieur Trusted Logic (marque déposée) protégé par le brevet FR 2 815 434. Les registres utilisés par la machine virtuelle sont éclatés de façon monomorphe, c'est-à-dire que chaque registre a un unique typage de variable. Les besoins en RAM en sont ainsi réduits. L'inconvénient de cette solution est qu'il faut effectuer un calcul hors carte afin de modifier les méthodes pour qu'elles vérifient les deux propriétés supplémentaires requises.

La littérature tend à indiquer que certaines vérifications embarquées de *bytecode* sont infaisables. Notamment la publication « *Java bytecode verification : algorithms and formalisations* » (http://pauillac.inria.fr/~xleroy/publi/bytecode-verification-JAR.pdf) précise que les algorithmes de vérification conventionnels polyvariants ne peuvent être mis en œuvre sur des équipements à faibles capacités de traitement comme les cartes Java (marque déposée).

La présente invention entend remédier aux inconvénients de l'art antérieur en proposant un procédé de vérification de cohérence de codes destinés à un système embarqué limitant la mémoire vive consommée lors des traitements. L'invention ne requiert également aucun prétraitement à l'extérieur du dispositif électronique portable, par exemple la carte à puce, et sans ajout

d'éléments dans le code de l'application. L'invention vise également à optimiser cette gestion de la mémoire vive (RAM) pendant la phase de vérification.

L'invention réalise une vérification de *bytecode* par contexte (polyvariant) avec une gestion particulière des *frames* courantes afin de consommer moins de mémoire vive. On fait en sorte de se trouver, lors d'une vérification en sous-programme, dans la situation d'une vérification en méthode. Pour ce faire, on procède à un changement de contexte, en sauvegardant l'état des données courantes (comprenant notamment la *stack frame* aux cibles de saut, la *frame* courante et la *worklist*). En sortie de sous-programme, on unifie la *frame* courante du contexte courant du sous-programme avec la *frame* correspondant au successeur du *bytecode* appelant (JSR par exemple) du contexte précédent, puis on restaure le contexte précédent. L'objectif de cette réalisation est de pouvoir gérer les appels à des sous-programmes, par exemple JSR/RET, ceux-ci pouvant être imbriqués.

Également, l'invention optimise la consommation mémoire lors de la sauvegarde des *frames* aux cibles de saut en établissant au fil de l'eau un dictionnaire de *stack frames* (par exemple, les *frames* courantes aux cibles de saut qui est mémorisé lors du changement de contexte) sur lesquelles pointent ces sauvegardes. Dans l'utilisation classique, un grand nombre d'informations supplémentaires et parfois redondantes doit être stocké pour gérer ces appels à des sauts. Cette masse d'informations n'est pas compatible avec la taille mémoire disponible dans la majorité des dispositifs électroniques portables. L'avantage du dictionnaire est de ne pas multiplier inutilement le nombre de *stack frames* identiques sauvegardées dans la mémoire et permet ainsi d'obtenir un bon taux de compression en raison des propriétés des *bytecodes* et des compilateurs. L'intérêt

du dictionnaire est de faire tenir la mémoire consommée dans les dispositifs électroniques portables à ressources limitées, et dans le cas des objets disposant de large mémoire RAM, de permettre l'utilisation d'un objet électronique portable ayant une mémoire RAM plus réduite et d'engendrer ainsi un gain sur les coûts.

L'invention s'intègre facilement à des solutions déjà existantes afin d'en améliorer les performances par une gestion appropriée de la mémoire vive.

À cet effet, l'invention concerne dans son acception la plus générale, un procédé de vérification d'une application interprétable par une machine virtuelle, ladite application étant chargée dans un dispositif électronique portable comprenant au moins un processeur et une mémoire vive, le procédé consistant à procéder, après le chargement de ladite application dans le dispositif et avant sa validation, à des contrôles sur le code de ladite application par un traitement mis en œuvre par le processeur caractérisé en ce qu'il comporte :

- lors d'un appel à un sous-programme, une étape de sauvegarde du contexte courant de vérification dans la mémoire vive,
- une étape de création et d'activation d'un nouveau contexte de vérification dédié au sous-programme,
- lors de la fin du sous-programme, une étape de restauration du contexte de vérification précédemment sauvegardé.

Dans un mode de réalisation, lors de ladite étape de création et d'activation, le nouveau contexte de vérification est initialisé, par exemple uniquement avec la *frame* courante.

Dans un mode de réalisation particulier, le procédé comprend, en outre, l'exécution immédiate des contrôles sur le code du sous-programme appelé lorsque l'appel à celui-ci a été détecté.

Plus particulièrement, le procédé ne comporte pas d'étape de bornage préalable du code.

Dans un mode de réalisation, le procédé ne comporte pas de recours à des ressources extérieures audit dispositif à l'exception de l'alimentation électrique.

Dans un mode de réalisation, le contexte comprend une liste de travail permettant de parcourir l'arborescence du code de l'application. Le procédé comprend, en outre, lors du contrôle d'une ligne de code de l'application, une étape de mise à jour de ladite liste de travail avec les successeurs possibles de ladite ligne de code.

Particulièrement, le contexte comprend un pointeur d'instruction et une *frame* courante.

Dans un mode de réalisation, on sauvegarde tout ou partie des *frames* identiques sous une adresse unique dans une zone de la mémoire vive appelée dictionnaire.

Dans un mode de réalisation particulier, on sauvegarde les *frames* identiques sous une adresse unique et l'on crée un dictionnaire contenant les *frames* à sauvegarder.

Plus précisément, pour chaque nouvelle *frame* à sauvegarder dans la mémoire vive, on vérifie si tout ou partie de la nouvelle frame est présente dans le dictionnaire, et si c'est le cas, on utilise le pointeur (ptr1) associé, si ce n'est pas le cas, on l'enregistre dans le dictionnaire et on utilise le pointeur (ptr2) associé à ce nouvel enregistrement.

Dans un mode de réalisation, le dictionnaire est décomposé en sections correspondant à des parties homogènes des *frames*. Particulièrement, l'une des parties homogènes du

dictionnaire correspond aux registres. Dans une variante, l'une des parties homogènes du dictionnaire correspond à la pile. Selon une autre variante, l'une des parties homogènes du dictionnaire correspond à des registres non variables en type de *bytecode*.

L'invention concerne également une carte à puce et une carte Java comprenant au moins une machine virtuelle Java et une « *applet* » en langage interprétable Java dont la cohérence doit être vérifiée, pour la mise en œuvre de ce procédé de vérification.

On comprendra mieux l'invention à l'aide de la description, faite ci-après à titre purement explicatif, d'un mode de réalisation de l'invention, en référence aux figures annexées où :

- la figure 1 représente un exemple de *bytecodes* comportant un sous-programme ;
- la figure 2 représente un exemple d'architecture de carte Java (marque déposée) pour la mise en œuvre de la présente invention ;
- la figure 3 illustre le procédé de vérification selon la présente invention ;
- les figures 4 à 7 illustrent l'évolution de la mémoire vive lors de la mise en œuvre du procédé de vérification de l'invention ; et
- la figure 8 illustre l'utilisation d'un dictionnaire pour optimiser la gestion de la mémoire vive.

Dans les modes de réalisation fournis ci-après à titre d'exemple, les sous-routines Java (marque déposée) appelées par des JSR ne sont qu'un exemple de sous-programmes pouvant être appelés de façon générale et sur lesquels l'invention porte.

En référence à la figure 2, le module de carte 1 comprend un microprocesseur 2 pilotant une mémoire non-volatile 3 par exemple de type flash, une mémoire morte ROM 4 et une mémoire vive RAM 5.

La mémoire ROM 4 stocke les programmes informatiques vérifieurs 41 de *bytecode* et la machine virtuelle 42 permettant l'exécution du *bytecode*. On entend par machine virtuelle une machine qui, lors de l'exécution de *bytecodes*, gère en mémoire vive 5 une pile 51 et des registres 52. La pile 51 est une mémoire à accès par le haut dans laquelle on empile et on dépile les données. Les registres 52 sont des cases mémoires à accès indexé ou accès libre : on peut accéder à n'importe quelle information des registres.

Un programme ou application 31 à vérifier est stocké sous forme de fichier dans la mémoire non-volatile 3. Ce programme se présente sous la forme de pseudo-code ou *bytecodes* dont l'invention propose de vérifier l'intégrité par rapport à la machine virtuelle 42.

Le fichier 31 est un fichier CAP (*Converted APplet*) qui est le fichier chargé par la machine virtuelle 42. Ce fichier peut contenir plusieurs méthodes au sens Java (marque déposée), auquel cas la vérification du *bytecode* se fait méthode par méthode. Si une méthode M2 est invoquée dans la méthode M1, M1 étant en train d'être vérifiée, le vérifieur considère M2 déjà vérifiée ou à vérifier plus tard, et passe au *bytecode* suivant.

Illustré par la figure 1, un exemple de *bytecode* d'une méthode M est fourni. Ce *bytecode* comprend un saut de sous-routine JSR en ligne 4 et un retour RET en ligne 11. La sous-routine appelée par le JSR s'étend de B7 à RET.

En référence à la figure 3, le vérifieur de *bytecode* entame 100 la vérification de la méthode M. Le premier *bytecode* B1 est prélevé 102. Puisqu'il s'agit d'un *bytecode* traditionnel à vérifier 104, une vérification des critères d'intégrité (typage) est effectuée 106 sur le *bytecode* en comparaison avec les registres et la pile de la *frame* courante et cette *frame* courante est mise à jour en fonction du *bytecode* (changement de typage, nouvelle variable, ...). La *frame* courante est l'ensemble des typages des registres et de la pile de la machine virtuelle au point qui et en train d'être examiné. Comme il s'agit du premier *bytecode*, les données *frame* courante 200 peuvent être enregistrées en mémoire comme illustrées par la figure 4. Des données complémentaires par exemple la « liste de travail » 202 sont également sauvées en mémoire ; la liste de travail (*worklist*) comprend la liste du ou des *bytecodes* suivants à vérifier et celle-ci est mise à « ligne 2 » pour indiquer que le prochain *bytecode* à traiter est celui de la ligne 2. La *worklist* permet de parcourir l'arborescence du code et de couvrir tous les cas de figures de ce parcours en prenant en compte les successeurs multiples que peut présenter chaque ligne de code. On entend par « successeurs multiples » d'une ligne de code, les autres lignes du code qui peuvent être atteintes depuis cette ligne. L'ensemble des données de la *frame* courante et des données complémentaires (par exemple la liste de travail 202, le dictionnaire 203, la liste 201 des *stack frames* aux cibles de saut) constituent le contexte courant.

Puis l'opération de vérification est reproduite pour les *bytecodes* B2 et B3 avec la mise à jour en mémoire RAM des données de contexte, notamment la liste de travail passe successivement à « ligne 3 » puis à « ligne 4 ».

Lorsque le vérifieur rencontre 108 le saut de sous-routine JSR en ligne 4, la liste de travail est mise à jour à « ligne 5 ». Puis l'adresse de la JSR ou pointeur d'instruction, soit « ligne 4 » est stockée sur la pile de la *frame* ; c'est elle qui permet de « mémoriser » l'endroit depuis lequel le saut a eu lieu. Le vérifieur sauvegarde 110 alors le contexte courant dans la mémoire vive 5. La figure 5 illustre un mode de réalisation de la sauvegarde du contexte : on mémorise, entre autres, dans la mémoire 204 le pointeur p1 de la *frame* courante, les pointeurs p2 à p4 des données complémentaires (liste de travail, …) et le pointeur p5 de fin de contexte. Le terme « *stack frame* » ou « *frame* » correspond à la *frame* courante à la cible de saut, c'est-à-dire au moment où celle-ci est sauvegardée. Le contexte sauvegardé se compose de l'ensemble des données et structure 204 de données RAM utiles à la vérification d'une méthode : liste de travail 202, *stack frame* 200, notamment.

Un nouveau contexte de vérification est alors créé et activé 112. Illustrées par la figure 6, une nouvelle *frame* 206 et des données complémentaires 207 à 209 sont créées dans la mémoire RAM libre avec des pointeurs p'1 à p'5 correspondants. Lors de cette création, la nouvelle *frame* courante 206 est initialisée à l'identique de la *stack frame* courante 200 au moment du saut de sous-routine. On a donc, à cet instant, des registres et une pile conformes au contexte d'où l'on vient, mais dans un nouveau contexte.

L'algorithme de vérification peut alors reprendre, la liste de travail étant mise à « ligne 7 » au début de la sous-routine.

L'algorithme de vérification s'applique alors sur les *bytecodes* B7, B8 et B9, les données de contexte en mémoire étant mises à jour.

Lorsque le vérifieur rencontre le *bytecode* RET de la ligne 10, on fait une unification avec le successeur du JSR correspondant et on va chercher le prochain élément à vérifier dans la liste de travail du contexte en cours. Lorsque la liste de travail est rendue vide, le problème de bornage des sous-routines qui est omniprésent dans les solutions de vérification de *bytecode* est, ici, naturellement réalisé par l'algorithme général de vérification : la liste de travail du nouveau contexte est vide signifiant la fin de la sous-routine.

La sous-routine est alors terminée et le contexte précédent est restauré 114, comme illustré par la figure 7.

Enfin, lorsqu'il n'y a plus de *bytecode* à vérifier, la vérification de la méthode M prend fin 116.

Dans le cas où une méthode est invoquée dans la méthode M, une vérification des paramètres de la méthode est réalisée pour s'assurer de l'intégrité des typages courants avec ceux de la méthode appelée. La vérification de cette méthode est réalisée indépendamment de la méthode M comme cela a été précisé précédemment.

Dans un mode de réalisation de l'invention, un dictionnaire des *stack frames* est utilisé. En effet, dans de nombreux cas, l'évolution des *stack frames* est lente et beaucoup d'entre-elles ont le même contenu.

En référence à la figure 8, le code comprend trois appels à des sous-programmes.

Lors de l'appel à un sous-programme en B2, la *frame* courante peut être sauvegardée dans une partie de la mémoire RAM 5 que l'on appelle le dictionnaire. Dans ce cas, la

sauvegarde du contexte courant dans la mémoire vive utilise alors le pointeur ptr1 référant la *frame* courante.

Lors d'un appel suivant à un sous-programme en B6, on utilise pour la sauvegarde du contexte courant le pointeur ptr1 associé à la *stack frame* du dictionnaire identique à la *frame* courante au moment du saut.

Lorsqu'une telle *stack frame* n'est pas présente dans le dictionnaire 53, on crée une nouvelle entrée ptr2 dans le dictionnaire (cas de l'appel Bt de sous-programme).

Dans un mode de réalisation, on épure le dictionnaire des *stack frames* non utilisées (c'est-à-dire lorsque le pointeur associé n'est utilisé dans aucune sauvegarde de contexte) au fur et à mesure qu'on sort des sous-programmes.

Éventuellement, le dictionnaire peut être construit avec des entrées de *stack frames* partielles qui sont récurrentes afin s'optimiser au mieux la compression et le gain de mémoire RAM. À titre d'exemples, on peut séparer dans le dictionnaire les registres et la pile, ou séparer les registres en ensembles, particulièrement les registres non variables en « type de *bytecode* » (les variables globales déclarées en début de méthode du source Java ainsi que les paramètres et qui ne changent pas de typage au cours de la méthode) peuvent constituer un ensemble sur lequel chaque contexte dans la méthode pointera.

Il est entendu que ce dictionnaire ne se limite pas aux *stack frames* et peut contenir tout type d'entrées qui sont utilisées lors de la sauvegarde des contextes, afin de minimiser la taille de ces sauvegardes.

REVENDICATIONS

1. Procédé de vérification d'une application (31) interprétable par une machine virtuelle (42), ladite application étant chargée dans un dispositif électronique portable (1) comprenant au moins un processeur (2) et une mémoire vive (5), le procédé consistant à procéder, après le chargement de ladite application dans le dispositif et avant sa validation, à des contrôles sur le code de ladite application par un traitement mis en œuvre par le processeur (2) caractérisé en ce qu'il comporte :

- lors d'un appel à un sous-programme, une étape de sauvegarde du contexte courant de vérification (200 à 203) dans la mémoire vive (5),
- une étape de création et d'activation d'un nouveau contexte de vérification (206 à 209) dédié au sous-programme,
- lors de la fin du sous-programme, une étape de restauration du contexte de vérification (200 à 203) précédemment sauvegardé.

2. Procédé de vérification selon la revendication 1, caractérisé en ce que, lors de ladite étape de création et d'activation, le nouveau contexte de vérification (206 à 209) est initialisé.

3. Procédé de vérification selon la revendication 1, caractérisé en ce qu'il comprend, en outre, l'exécution immédiate des contrôles sur le code du sous-programme appelé lorsque l'appel à celui-ci a été détecté.

4. Procédé de vérification selon la revendication 1, caractérisé en ce qu'il ne comporte pas d'étape de bornage préalable du code.

5. Procédé de vérification selon la revendication 1, caractérisé en ce qu'il ne comporte pas de recours à des ressources extérieures audit dispositif à l'exception de l'alimentation électrique.

6. Procédé de vérification selon la revendication 1, caractérisé en ce que le contexte comprend une liste de travail (202) permettant de parcourir l'arborescence du code de l'application (31).

7. Procédé de vérification selon la revendication précédente, caractérisé en ce qu'il comprend, en outre, lors du contrôle d'une ligne de code de l'application, une étape de mise à jour de ladite liste de travail (202) avec les successeurs possibles de ladite ligne de code.

8. Procédé de vérification selon la revendication 1, caractérisé en ce que le contexte comprend un pointeur d'instruction et une *frame* courante.

9. Procédé selon la revendication 1, caractérisé en ce qu'on sauvegarde tout ou partie des *frames* identiques sous une adresse unique dans une zone de la mémoire vive (5) appelée dictionnaire (53).

10. Procédé selon la revendication précédente, caractérisé en ce que pour chaque nouvelle *frame* à sauvegarder dans la mémoire vive (5), on vérifie si tout ou partie de la nouvelle frame est présente dans le dictionnaire (53), et si c'est le cas, on utilise le pointeur (prt1) associé, si ce n'est pas le cas, on l'enregistre dans le dictionnaire (53) et on utilise le pointeur (ptr2) associé à ce nouvel enregistrement.

11. Procédé de vérification selon la revendication 9, caractérisé en ce que le dictionnaire (53) est décomposé en sections correspondant à des parties homogènes des *frames*.

12. Procédé de vérification selon la revendication 11, caractérisé en ce que l'une des parties homogènes du dictionnaire (53) correspond aux registres (52).

13. Procédé de vérification selon la revendication 11, caractérisé en ce que l'une des parties homogènes du dictionnaire (53) correspond à la pile (51).

14. Procédé de vérification selon la revendication 11, caractérisé en ce que l'une des parties homogènes du dictionnaire (53) correspond à des registres (52) non variables en type de *bytecode*.

15. Carte à puce comprenant au moins une mémoire vive (5) et un processeur (2) pour la mise en œuvre du procédé selon l'une quelconque des revendications précédentes.

16. Carte Java de type carte à puce, comprenant une mémoire vive (5), un processeur (2), une machine virtuelle Java (42) et au moins une « *applet* » Java (31) dont la cohérence du code interprétable par la machine virtuelle doit être vérifiée, pour la mise en œuvre du procédé selon l'une quelconque des revendications 1 à 14.

Figure 1

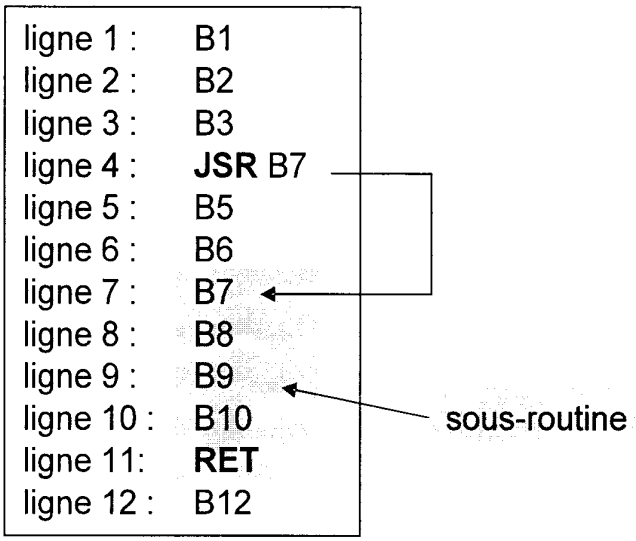


Figure 2

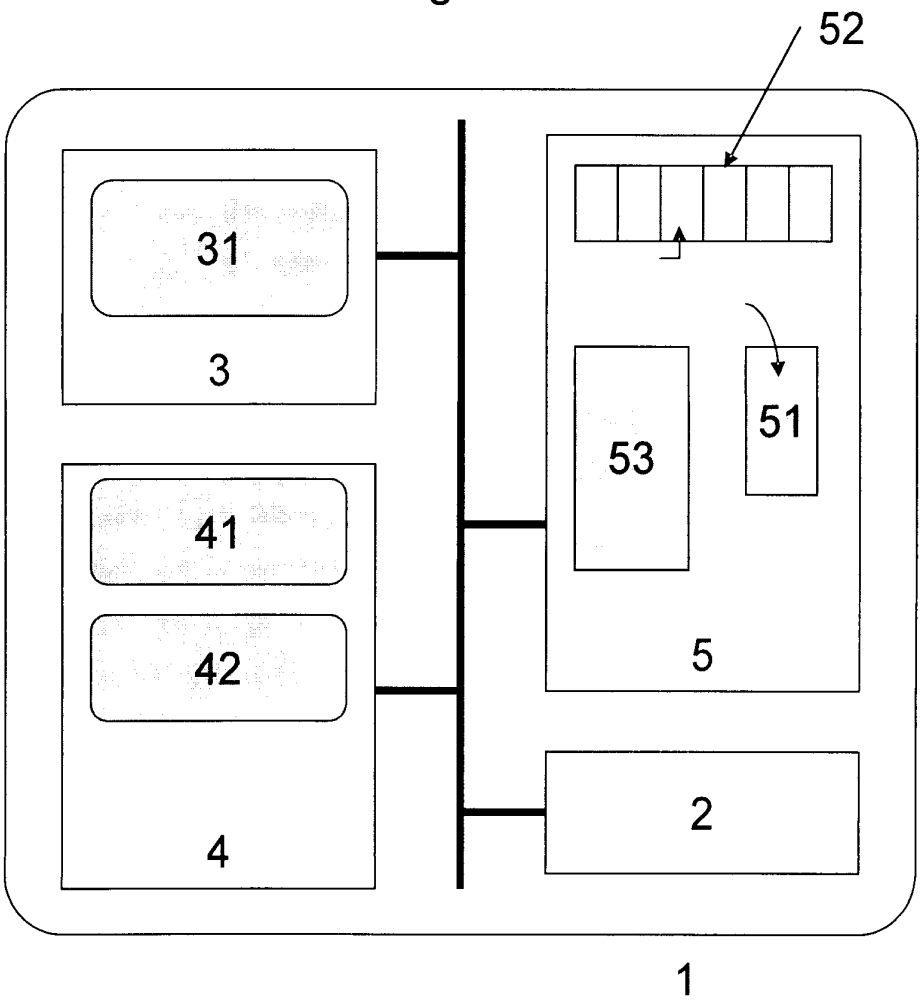


Figure 3

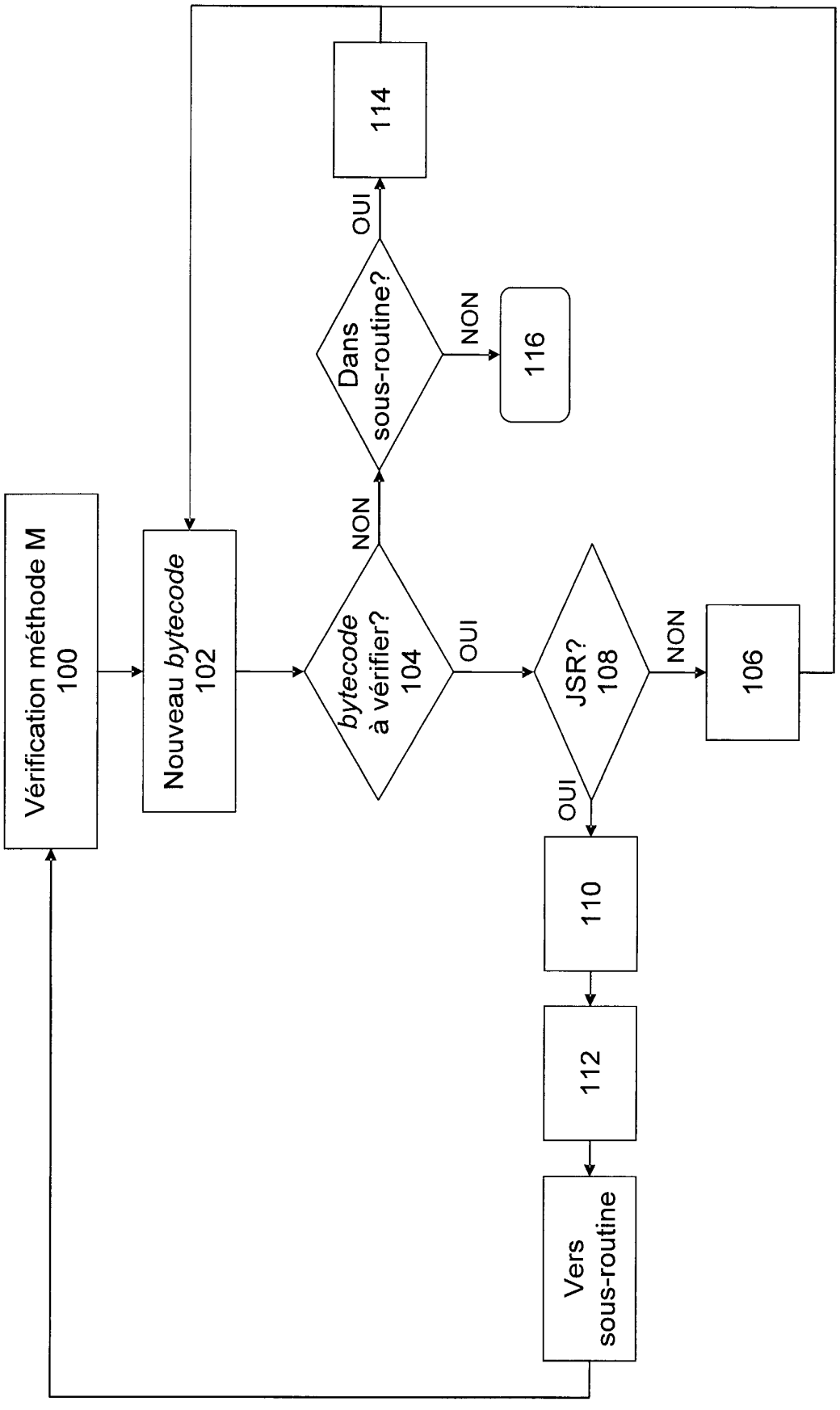

Vérification méthode M
100
Nouveau bytecode
102
bytecode
à vérifier?
104
NON
OUI
Dans
sous-routine?
OUI
NON
114
116
JSR?
108
OUI
NON
106
110
112
Vers
sous-routine

Figure 4

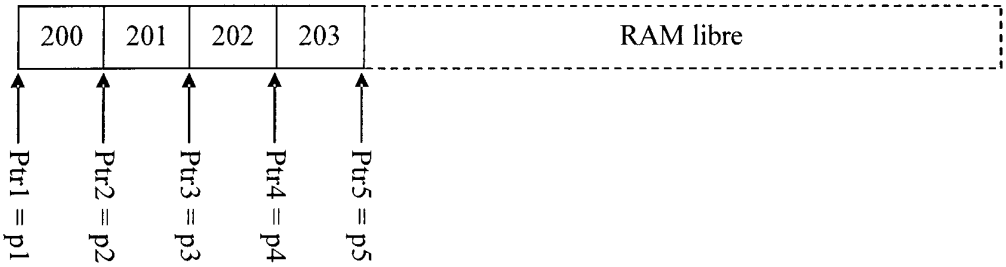


Figure 5

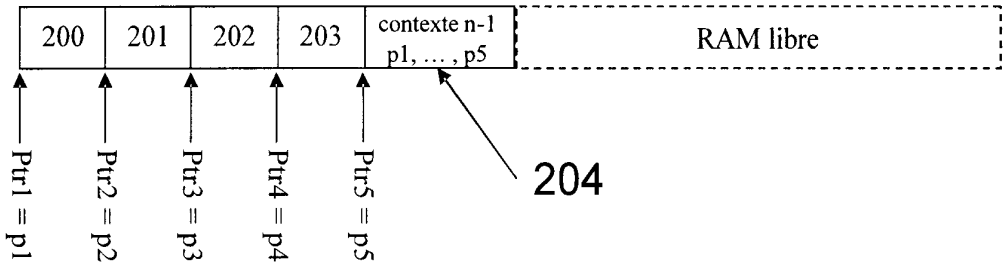


Figure 6

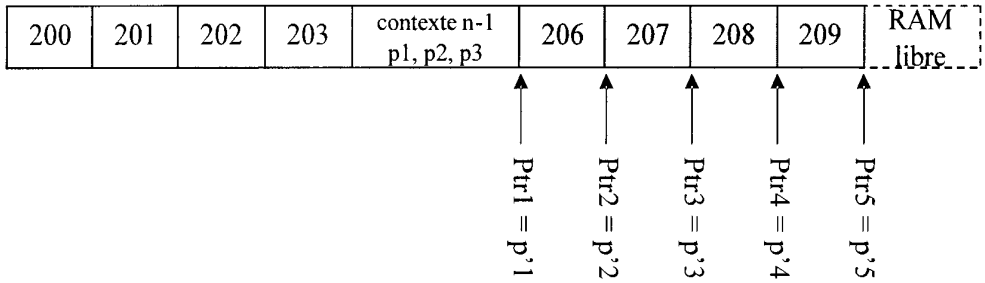


Figure 7

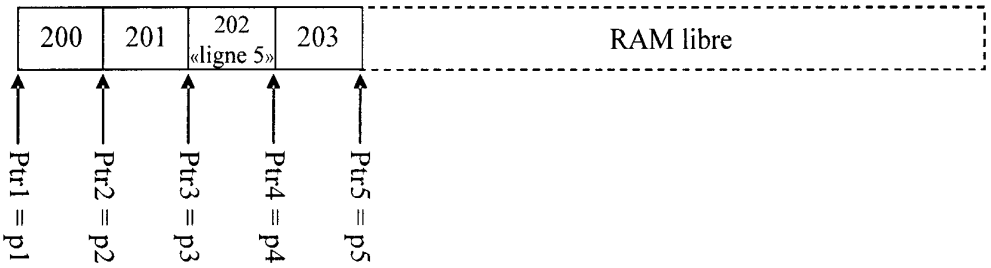
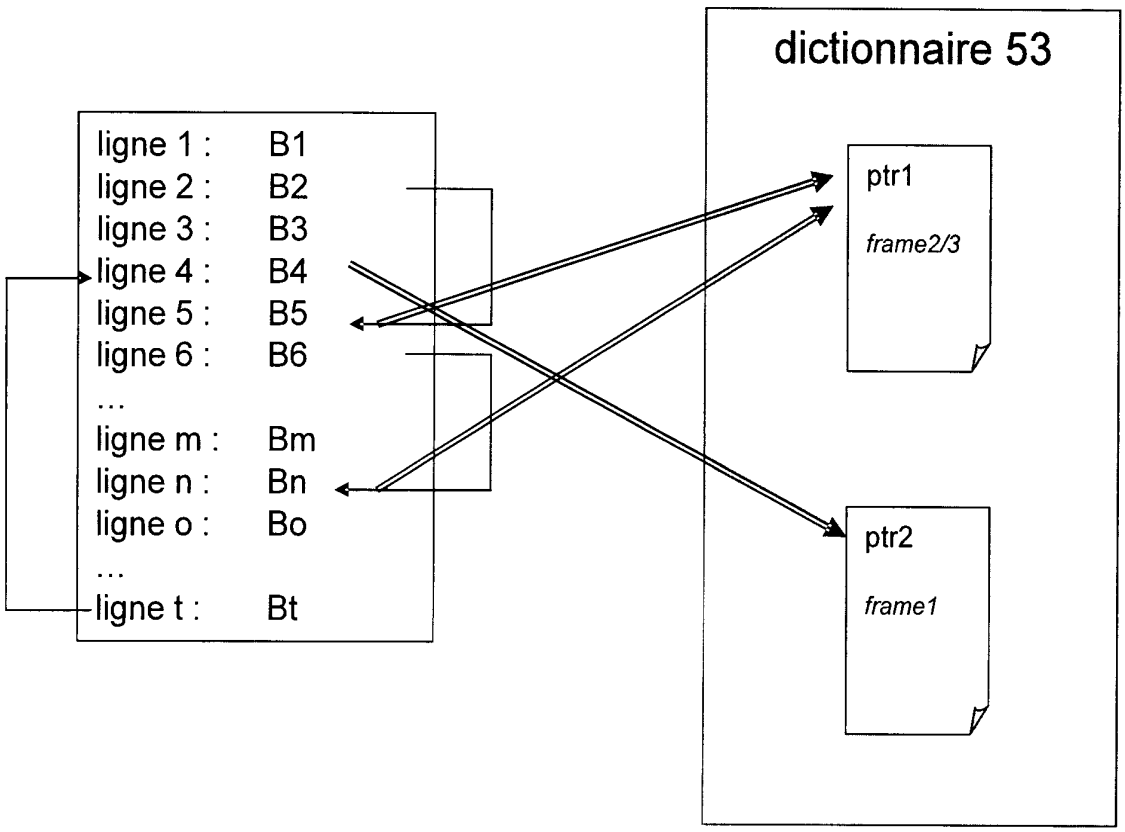


Figure 8


dictionnaire 53
ligne 1 : B1
ligne 2 : B2
ligne 3 : B3
ligne 4 : B4
ligne 5 : B5
ligne 6 : B6
...
ligne m : Bm
ligne n : Bn
ligne o : Bo
...
ligne t : Bt
ptr1
frame2/3
ptr2
frame1

RÉPUBLIQUE FRANÇAISE

2884994

INSTITUT NATIONAL DE LA PROPRIETE INDUSTRIELLE

RAPPORT DE RECHERCHE PRÉLIMINAIRE

établi sur la base des dernières revendications déposées avant le commencement de la recherche

N° d'enregistrement national

FA 663804
FR 0551036

DOCUMENTS CONSIDÉRÉS COMME PERTINENTS		Revendication(s) concernée(s)	Classement attribué à l'invention par l'INPI
Catégorie	Citation du document avec indication, en cas de besoin, des parties pertinentes		
A	A. COGLIO: "Simple verification technique for complex Java bytecode subroutines" CONCURRENCY AND COMPUTATION PRACTICE & EXPERIENCE, vol. 16, no. 7, juin 2004 (2004-06), pages 647-670, XP002353719 * page 654, alinéa 3.1 - page 658, alinéa 3.3 * * page 662, alinéa 4 - page 667 *	1-16	H04L9/00
A	FR 2 797 963 A (TRUSTED LOGIC) 2 mars 2001 (2001-03-02) * page 27, ligne 1 - page 30, ligne 8 *	1-16	
			DOMAINES TECHNIQUES RECHERCHÉS (IPC)
			G06F

Date d'achèvement de la recherche	Examinateur
11 novembre 2005	Renault, S

CATÉGORIE DES DOCUMENTS CITÉS

X : particulièrement pertinent à lui seul
Y : particulièrement pertinent en combinaison avec un autre document de la même catégorie
A : arrière-plan technologique
O : divulgation non-écrite
P : document intercalaire

T : théorie ou principe à la base de l'invention
E : document de brevet bénéficiant d'une date antérieure à la date de dépôt et qui n'a été publié qu'à cette date de dépôt ou qu'à une date postérieure.
D : cité dans la demande
L : cité pour d'autres raisons

& : membre de la même famille, document correspondant

EPO FORM 1503 12.99 (P04C14)

ANNEXE AU RAPPORT DE RECHERCHE PRÉLIMINAIRE RELATIF A LA DEMANDE DE BREVET FRANÇAIS NO. FR 0551036 FA 663804

La présente annexe indique les membres de la famille de brevets relatifs aux documents brevets cités dans le rapport de recherche préliminaire visé ci-dessus.
Les dits membres sont contenus au fichier informatique de l'Office européen des brevets à la date du 11-11-2005
Les renseignements fournis sont donnés à titre indicatif et n'engagent pas la responsabilité de l'Office européen des brevets, ni de l'Administration française

Document brevet cité au rapport de recherche	Date de publication	Membre(s) de la famille de brevet(s)	Date de publication
FR 2797963 A	02-03-2001	AT 252742 T	15-11-2003
		AU 769363 B2	22-01-2004
		AU 7015000 A	19-03-2001
		CA 2382003 A1	01-03-2001
		CN 1370294 A	18-09-2002
		DE 60006141 D1	27-11-2003
		DE 60006141 T2	26-08-2004
		EP 1212678 A2	12-06-2002
		ES 2209969 T3	01-07-2004
		WO 0114958 A2	01-03-2001
		JP 2003507811 T	25-02-2003

EPO FORM P0465