(19) RÉPUBLIQUE FRANÇAISE

INSTITUT NATIONAL
DE LA PROPRIÉTÉ INDUSTRIELLE

PARIS

(11) **N° de publication :** **2 840 084**
(à n'utiliser que pour les commandes de reproduction)

(21) **N° d'enregistrement national :** **02 06445**

(51) Int Cl[7] : **G 06 F 11/36,** G 06 K 19/07

(12) DEMANDE DE BREVET D'INVENTION A1

(22) **Date de dépôt :** 27.05.02.

(30) **Priorité :**

(43) **Date de mise à la disposition du public de la demande :** 28.11.03 Bulletin 03/48.

(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.

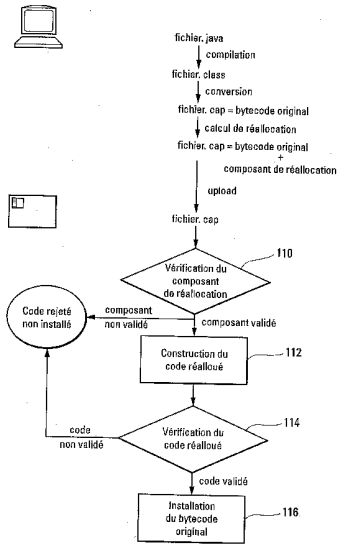
(72) **Inventeur(s) :** GREZES JEAN FRANCOIS et BENOIT ALEXANDRE.

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

(74) **Mandataire(s) :** GEMPLUS.

(54) PROCEDE DE VERIFICATION DE CODES POUR MICROCIRCUITS A RESSOURCES LIMITEES.

(57) L'invention concerne un procédé de vérification de code intermédiaire exécutable par un microcircuit à ressources limitées connecté à un système de traitement de données externe, comportant une étape de modification du code intermédiaire comprenant la réallocation des registres réels de type r à des registres virtuels v de type monomorphe, et de construction d'un code réalloué dont les instructions PC font référence aux registres virtuels v, et une étape de vérification du code réalloué dans le microcircuit à ressources limitées, caractérisé en ce que, en cas de succès de la vérification du code réalloué dans le microcircuit, l'on installe le code intermédiaire original dans le microcircuit à ressources limitées pour exécution.



FR 2 840 084 - A1

Procédé de vérification de codes pour microcircuits à ressources limitées

La présente invention concerne un procédé de vérification de codes utilisés avec des microcircuits à ressources limitées, tels que utilisés dans des cartes à puce, notamment pour vérifier l'intégrité et assurer l'innocuité du code.

Les cartes à microcircuit électronique, dites cartes à puce, sont utilisées comme support informatique mobile pour des applications très diverses et exigeant pour la plupart d'entre elles un haut niveau de sécurité, notamment les opérations bancaires, les paiements sécurisés, l'accès aux bâtiments ou à des zones sécurisées et les télécommunications. L'utilisation répandue des cartes à puce et la diversité de leurs applications tendent actuellement à l'utilisation de plateformes ouvertes, telle la plateforme connue sous le nom de "Java", qui permet le chargement, dans la mémoire d'une machine de traitement de données électroniques, d'applications compatibles avec la plateforme. Une plateforme tel que Java présente deux avantages, la standardisation du code intermédiaire, dit "bytecode", et son indépendance par rapport à la machine. Tout programme Java peut donc être compilé en une série d'instructions bytecode universellement comprises par toute machine basée sur une telle plateforme.

Etant donné que les cartes à puce ou d'autres systèmes embarqués sont mobiles et qu'elles sont utilisées en relation avec des systèmes informatiques étrangers, ou qu'elles exécutent un bytecode d'origine soit inconnue, soit non-sécurisée, il est nécessaire de

vérifier l'intégrité et l'innocuité du programme utilisé avec la carte ou avec d' autres systèmes embarqués.

Un programme de vérification de code intermédiaire de type Java est décrit dans le brevet US 5,740,441. Le procédé de vérification décrit dans ce brevet est adapté aux systèmes informatiques tels que des ordinateurs PC dont la puissance de calcul de leur processeur et la capacité de leur mémoire vive RAM (Random Access Memory) sont nettement supérieures à celles d'une carte à puce. La puissance d'un processeur et la capacité de la mémoire vive ou de la mémoire persistante ROM (Read Only Memory) sont fonction de la surface de silicium encarté. Les cartes à puce devant d'une part respecter des contraintes mécaniques, en particulier la torsion et la flexion, et d'autre part garantir une durée de vie raisonnable, leur surface de silicium ne dépasse généralement pas 25 mm^2. La mémoire d'une carte à puce a typiquement une capacité d'environ 8 kilo-octets, par conséquent bien inférieure à la capacité de RAM dont dispose l'ordinateur PC le moins performant proposé actuellement sur le marché.

Il existe plusieurs solutions permettant de vérifier le bytecode dans des dispositifs dont la capacité de mémoire est très limitée.

L'une de ces approches consiste à ajouter au code le résultat d'un algorithme de pré-calcul qui sera ensuite repris par le vérifieur du bytecode. Il peut s'agir par exemple de la méthode PCC (Proof Carrying Code) ou d'une variante de cette dernière qui permet au vérifieur encarté d'utiliser des algorithmes plus simples. Un programme, tel qu'un applet Java quelconque, ne comprennent pas le code supplémentaire PCC, ne pourra donc pas être vérifié par le microcircuit. La sphère

d'utilisation des microcircuits utilisant ces méthodes est donc limitée.

Une autre approche consiste à modifier le code intermédiaire du programme à vérifier à l'extérieur du dispositif de telle sorte que le procédé de vérification soit plus aisé sans entraîner pour autant une diminution de fiabilité, telle que décrite dans la demande de brevet internationale WO 0114958 A2. Dans cette demande, on décrit un procédé de vérification d'un programme de type Java dans lequel l'on modifie le code intermédiaire par un procédé dit "de normalisation" dans un système de traitement de données externe avant de transférer ce code modifié dans un système embarqué. Lors de cette normalisation, le registre réel des types est réalloué à un registre virtuel, chaque case de ce registre ne définissant qu'un type, c'est-à-dire que le registre virtuel est monomorphe. Cette modification permet donc de réduire fortement la consommation en mémoire nécessaire pour stocker les différents types d'un registre polymorphe lors du procédé de vérification. Si le procédé de vérification de ce code modifié se termine avec succès, le code modifié est ensuite exécuté par le système embarqué. L'un des désavantages de ce procédé est que l'on ne peut pas être absolument sûr que le code modifié dans le système externe correspond au code original et sera exécuté correctement par le système embarqué. D'autre part, le fait d'exécuter un code modifié selon le procédé décrit dans la demande de brevet précitée, limite l'utilisation de certaines techniques d'optimisation de calcul du programme.

De manière générale, les procédés conventionnels de vérification du code intermédiaire (bytecode) Java destinés à être utilisés dans des microcircuits à

ressources limitées peuvent également avoir les inconvénients d'augmenter la quantité d'informations à transmettre au microcircuit, de nécessiter l'implémentation et l'exécution d'un logiciel relativement complexe dans le système informatique externe en communication avec la carte à puce, ou de limiter l'utilisation de certaines techniques d'optimisation de calcul du programme.

Tenant compte de ce qui précède, l'un des objectifs de l'invention est de fournir un procédé de vérification de code intermédiaire destiné à être exécuté dans un objet à microcircuit à ressources limitées, fiable et économe en termes d'occupation des ressources de mémoire du microcircuit, notamment la mémoire volatile ou d'autres mémoires accessibles en écriture et lecture du microcircuit.

Il est avantageux de fournir un procédé de vérification destiné à être exécuté dans un objet à microcircuit à ressources limitées qui ne compromet pas l'exécution du code intermédiaire et qui ne limite pas l'utilisation des techniques d'optimisation de calcul du code intermédiaire.

Il est avantageux de fournir un procédé de vérification destiné à être exécuté dans un objet à microcircuit à ressources limitées, ayant un large spectre d'utilisation pour une plateforme donné, tel que le Java ou des variantes telles que le Javacard.

Il est en outre avantageux de réduire l'utilisation des ressources de calculs du microcircuit de l'objet.

Il est également avantageux de réduire le temps nécessaire pour effectuer la vérification d'un programme à vérifier.

Des buts de l'invention sont réalisés par un procédé selon la revendication 1.

Dans la présente invention, un procédé de vérification de codes intermédiaires exécutable par un microcircuit à ressources limitées connecté à un système de traitement de données externe, comporte une étape de modification du code intermédiaire comprenant la réallocation de registres réels de types à des registres virtuels de types monomorphes, et de construction d'un code réalloué dont les instructions font référence aux registres virtuels, et une étape de vérification du code réalloué dans le microcircuit à ressources limitées, caractérisée en ce que, en cas de succès de la vérification du code réalloué dans le microcircuit, l'on installe le code intermédiaire original dans le microcircuit à ressources limitées pour exécution.

Dans une première forme d'exécution, la modification du code intermédiaire original peut être effectuée dans un système de traitement de données externe avant son chargement dans le microcircuit à ressources limitées, la modification comprenant la génération d'un composant de réallocation incluant un tableau de réallocation définissant la réallocation des données du type d'un registre réel à des données du type d'un registre virtuel monomorphe. Dans cette forme d'exécution, le procédé de vérification comprend la vérification du composant de réallocation et, en cas de succès, la vérification du code réalloué, le code réalloué étant construit soit lors du procédé de vérification du composant de réallocation, soit après vérification du composant de réallocation. Si la vérification du code réalloué se termine avec succès, le bytecode original est installé pour exécution, le bytecode original étant soit stocké dans une mémoire

persistante accessible en lecture et en écriture du microcircuit, soit chargé depuis le système externe. Dans ce dernier cas, afin d'assurer que le bytecode original installé depuis le système externe correspond au code original qui a été vérifié lors du premier chargement du bytecode original, un calcul de hachage du code intermédiaire original est effectué et comparé avec le résultat du calcul de hachage du code intermédiaire original à réinstaller après le procédé de vérification.

Dans une autre forme d'exécution, les étapes de calcul de la réallocation et de la construction du code réalloué peuvent être effectuées dans le microcircuit à ressources limitées, le code intermédiaire original étant chargé dans le microcircuit avant l'exécution du procédé de vérification et stocké dans une mémoire du microcircuit, par exemple dans une mémoire persistante accessible en lecture et en écriture, du type EEPROM (Electrical Erasable Programmable Read Only Memory).

Avantageusement, la génération d'un code réalloué pour vérification permet de générer un registre du type monomorphe réduisant la consommation de la mémoire volatile du microcircuit lors de la vérification du code intermédiaire. L'installation du code intermédiaire original pour exécution, après le procédé de vérification, permet toutefois d'éviter des problèmes éventuels liés à l'exécution d'un code intermédiaire modifié et permet également l'utilisation de techniques de calcul optimisées.

D'autres caractéristiques avantageuses de l'invention ressortiront des revendications, de la description détaillée de formes d'exécution de l'invention donnée ci-après et des dessins annexés, dans lesquels:

la Fig. 1 montre un organigramme illustrant l'enchaînement des étapes d'un procédé de vérification d'un code intermédiaire selon une première forme d'exécution de l'invention;

la Fig. 2 est une représentation simplifiée d'instructions du code intermédiaire (bytecode) et d'un tableau de réallocation associé aux instructions du code intermédiaire;

la Fig. 3 est une représentation simplifiée d'un exemple de code intermédiaire avec son tableau de réallocation selon une forme d'exécution de l'invention;

la Fig. 4 montre un organigramme illustrant l'enchaînement des étapes pendant le procédé de vérification du tableau de réallocation;

la Fig. 5 est une représentation simplifiée de l'affectation et de la mise à jour du tableau de réallocation courante pour une instruction du code intermédiaire et du type "*def r*", qui est une instruction définissant une valeur dans le registre réel associé à l'instruction;

la Fig. 6 est une représentation simplifiée de l'affectation et de la mise à jour du tableau de réallocation courante pour une instruction du type "*use r*", qui est une instruction utilisant une valeur *r* dans un registre associé à l'instruction courante;

la Fig. 7 est une représentation simplifiée de l'affectation et de la mise à jour du tableau de réallocation courante pour une instruction du code intermédiaire du type "*brch*", qui est une instruction de branchement à au moins deux instructions;

la Fig. 8 est une représentation simplifiée de l'affectation et de la mise à jour du tableau de

réallocation courante pour une instruction du type "*nop*", qui est une instruction qui ne comporte pas d'opération sur le registre associé à l'instruction courante;

la Fig. 9 est une représentation simplifiée de l'affectation et de la mise à jour du tableau de réallocation courante pour une instruction du type "*return*";

la Fig. 10 montre un organigramme illustrant l'enchaînement des étapes dans un procédé de vérification de code intermédiaire selon une deuxième forme d'exécution de l'invention.

Faisant référence à la Fig. 1, les étapes d'un procédé de vérification d'un code intermédiaire, tel qu'un bytecode Java ou Javacard, selon une première forme d'exécution de l'invention, sont montrées. Un microcircuit à ressources limitées notamment pour un système embarqué tel qu'une carte à puce, est connecté à un système de traitement de données externe (ci-après "système externe") pour le chargement et l'exécution d'un programme tel qu'un applet Java, ou Javacard, dans le microcircuit. Le système externe effectue la compilation et la conversion du programme Java en un fichier *.cap*, c'est-à-dire le code intermédiaire (bytecode) exécutable par le microcircuit.

Dans cette première forme d'exécution, l'on ajoute au code intermédiaire original un composant de réallocation comprenant un tableau de réallocation T, tel que montré dans les Figures 2 et 3. Ce composant de réallocation est généré par le système externe par une méthode du calcul de réallocation connue, telle que la méthode de "coloration de graph".

Le calcul de réallocation par coloration de graph est bien connu en tant que tel et est utilisé par exemple

dans le procédé de vérification décrit dans la demande internationale WO 0114958 A2. Toutefois, dans la demande précitée, le code intermédiaire original est modifié par un procédé dit de "normalisation", Ainsi, d'une part, les instructions du code intermédiaire et le registre sont modifiés, de sorte que le registre est monomorphe, c'est-à-dire que chaque registre ne reçoit des données représentant un seul type et, d'autre part, à chaque instruction de branchement, la pile est vide. Cette transformation permet de rendre le procédé de vérification linéaire et d'éviter la forte consommation en mémoire volatile (RAM) que nécessite le stockage de registres pouvant accepter des données représentant différents types, c'est-à-dire les registres polymorphes.

Dans la méthode connue précitée, le bytecode Java ainsi modifié est utilisé pour l'exécution du programme après vérification. L'exécution du code modifié peut d'une part avoir des conséquences sur la fiabilité de l'exécution, d'autre part elle limite les possibilités d'utiliser les techniques d'optimisation connues du bytecode Java, utilisées par exemple pour réduire le temps d'exécution ou le temps de communication avec le système externe.

Dans la présente invention, le code intermédiaire (bytecode) original ainsi que le composant de réallocation est chargé dans le microcircuit à ressources limitées. Le code intermédiaire original est stocké en mémoire du microcircuit, par exemple dans une mémoire persistante accessible en écriture et en lecture de type EEPROM et le composant de réallocation est chargé dans une mémoire volatile (RAM) du microcircuit. Après chargement, le composant de réallocation est vérifié (étape 110) et, en cas de succès, l'on procède à la

construction du code réalloué (étape 112) et finalement à une étape de vérification du code réalloué (étape 114) avant l'installation du code intermédiaire original stocké dans la mémoire persistante (étape 116) pour exécution. En cas d'échec du procédé de vérification du composant de réallocation ou du procédé de vérification du code réalloué, le code intermédiaire est rejeté et n'est pas installé pour exécution. Il est à remarquer que la construction du code réalloué peut être effectuée lors du procédé de vérification du composant de réallocation, c'est-à-dire que les étapes 110 et 112 peuvent être confondues.

La procédure de vérification du composant de réallocation sera décrite plus en détail ci-après, en faisant référence aux Figures 2 à 9:

La Fig. 2 montre, de manière simplifiée, une série d'instructions PC du code intermédiaire ainsi qu'un composant de réallocation T du registre réel de données de type. Les instructions du code intermédiaire peuvent être classées parmi les cinq classes d'instructions suivantes en fonction de leur effet sur la réallocation du registre de données de type correspondant.

"*def x*" : On définit la valeur de la variable *x* dans le registre (par exemple les instructions "*sstore*", "*astore*")

"*use x*" : On utilise la variable *x* dans le registre (par exemple les instructions "*sload*", "*aload*")

"*nop*" : On n'effectue aucune opération sur les valeurs du registre (par exemple les instructions "*sxor*", "*iadd*")

"*return*" : On sort de la méthode (par exemple les instructions "*areturn*", "*sreturn*")

"*brch x y* : On branche à des instructions cibles *x* ou *y* (par exemple les instructions "*ifeq*", "*if-scmpeq*")

Faisant référence plus particulièrement aux Figures 2 et 3, le composant de réallocation T comprend un sous-tableau D contenant la réallocation pour chaque instruction qui définit le type d'une variable x dans le registre r_x, c'est-à-dire une instruction de la classe "*def x*", et un tableau de réallocation F ayant le même nombre de colonnes *k* que de registres réels et le même nombre de lignes *S* que d'instructions PC. Le tableau F est un tableau résultant de la réallocation des registres réels de données de type r_x à des registres virtuels v_y. Chaque registre r_x de même type est réalloué à un seul registre virtuel v_y correspondant à ce type. Les registres réels dont le type peut changer, c'est-à-dire les registres polymorphes, sont réalloués à différents registres virtuels correspondants aux différents types de ces registres. Comme chaque registre virtuel ne définit qu'un seul type, les registres virtuels sont monomorphes, c'est-à-dire les variables pour lesquelles il existe un type restent valables tout au long du procédé de vérification du code intermédiaire.

L'algorithme de vérification est beaucoup plus simple dans le cas des variables monomorphes, du fait qu'il convient simplement de trouver pour chaque variable le type de la variable qui sera valable tout au long de la procédure de vérification. Il s'agit en fait d'un calcul de point fixe où le type des variables est spécialisé jusqu'à ce qu'il reste inchangé. Un tel algorithme échoue si on lui présente des variables polymorphes comme étant des variables monomorphes.

Le procédé de vérification du composant de réallocation utilise un tableau courant F' (voir Fig. 3) contenant le type de chaque variable pour l'instruction (PC) courante, de sorte que les types contenus dans ce tableau courant F' correspondent à l'évolution de la réallocation courante lors de la vérification du composant de réallocation *T*. Le tableau courant F' peut donc être représenté par une seule ligne et un nombre de colonnes égal au nombre *k* de registres réels.

Au début du procédé de vérification, c'est-à-dire à l'instruction PC_1, les données du tableau de réallocation courante sont initialisées (étape 402 de la Fig. 4) avec le type "*nul*", telles que montrées dans la Fig. 3, qui est le plus petit élément du treillis des types, au lieu du type "*top*" qui réunit tous les types, c'est-à-dire le plus grand élément du treillis des types. On lit ensuite la première instruction (étape 404 de la Fig. 4) et en fonction de la classe d'instruction du code intermédiaire, l'on effectue une vérification suivie par une mise à jour du tableau de réallocation courante F' ou simplement une mise à jour du tableau de réallocation courante.

Pour les instructions sans opération sur le registre "*nop*" et de retour "*return*", l'algorithme de vérification effectue simplement une mise à jour du tableau de réallocation (étapes 412 respectivement 414 de la Fig. 4), telle que montrée dans les Figures 8 respectivement 9. Dans le cas d'une instruction de type "*nop*", la mise à jour consiste simplement en un report des données de type ($f'_{\alpha, i}$) dans le tableau de réallocation courante, c'est-à-dire que les données de type sont conservées sans changement pour la prochaine instruction $PC_{□+1}$. Dans le cas d'une instruction de classe "*return*", la mise à jour

consiste en l'affectation des données de type ($f'_{\alpha+1,i}$) du sous-tableau de réallocation F de l'instruction suivante $PC_{\square+1}$ dans le tableau de réallocation courante F', telle que montrée dans la Fig. 9.

Dans le cas d'une instruction de définition "*def*", l'on affecte les données de type (d_α) dans le sous-tableau D contenant la réallocation pour l'instruction ($PC_\square$) en question, au registre correspondant du tableau de réallocation courante F', c'est-à-dire si l'instruction courante $PC_\square$ définit la valeur d'un registre ($f_{\square,x}$) la réallocation ($d_\alpha=v_y$) définie dans le tableau D est affectée et met à jour le registre ($f'_{\square,x}$) du tableau de réallocation courante F'. Lors de cette mise à jour, les autres valeurs ($f'_{\square,i}$ à $f'_{\square,x-1}$ et $f'_{\square,x+1}$ à $f'_{\square,k}$) du tableau de réallocation courante sont conservées, telles que montrées dans la Fig. 5.

Si l'instruction est de classe "*use* r_x", l'algorithme de vérification vérifie simplement si la valeur du registre correspondant (la colonne x) du tableau de réallocation courante F' n'est pas égale à la valeur "*nul*", qui est la plus petite valeur du treillis des types. La mise à jour pour cette instruction est simplement la conservation des données de type ($f'_{\square,i}$) du tableau de réallocation courante F', telle que montrée à la Fig. 6.

Dans le cas d'une instruction de branchement, la vérification consiste à comparer les données de type ($f'_{\square,i}$) dans le tableau dans le tableau de réallocation courante F' correspondant à l'instruction courante PC_α avec les données de type ($f'_{\square,i}$) du tableau de réallocation F aux instructions cibles PC_β et PC_γ et, en cas d'inégalité, la procédure de vérification se termine

avec un échec. Le tableau de réallocation courante F' est ensuite mis à jour en affectant les données de type du tableau de réallocation F à l'instruction suivante $PC_{\alpha+1}$ au tableau de réallocation courante F'.

La prochaine étape (416) consiste à vérifier s'il reste encore des instructions à lire et, dans l'affirmative, d'incrémenter le pointeur de l'instruction courante (étape 418) et de répéter la boucle de lecture de l'instruction (étape 414) et de vérification et de mise à jour de la réallocation courante jusqu'à ce qu'il n'y ait plus d'instructions à lire. Le programme de vérification passe ensuite à l'étape de construction du code réalloué (étape 112) et à la vérification du code réalloué (étape 114). La vérification du code réalloué suit les procédés de vérification connus. Il est toutefois à remarquer que le code réalloué peut être construit lors de la vérification du composant de réallocation (étape 110) en changeant la variable sur laquelle agit l'instruction à sa valeur réallouée. Par exemple, si une instruction de définition "*def* r_1" (voir Fig. 3) agit sur le registre r_1 qui est réalloué au registre virtuel v_1, alors l'instruction du code réalloué devient "*def* v_1", de sorte qu'elle agit sur le registre virtuel de données de type au lieu du registre réel de données de type original.

On constate donc que les instructions sont lues linéairement de la première à la dernière instruction, chaque instruction étant exécutée par l'interpréteur de type.

Pour les instructions d'utilisation d'une variable ("*use*"), sans opération sur les registres ("*nop*") ou de retour ("*return*"), le comportement de l'algorithme de

vérification est similaire à celui d'un vérifieur conventionnel.

Cependant on doit garantir lors d'une instruction d'utilisation "*use x*" que la variable "*x*" sur laquelle agit cette instruction a été bien définie auparavant, alors que la variable peut avoir été initialisée dans une autre branche du programme avant cette instruction. Ce problème est résolu en vérifiant que la valeur stockée dans le sous-tableau D pour une instruction de définition "*def x*" spécialise le type précédent de la variable *x* dans le tableau de réallocation courante F'. Pour cette raison il est important qu'au début du programme, les variables du tableau F' sont initialisées à *"nul"* pour leur permettre d'être spécialisées.

On remarque que sur une instruction de branchement "*brch*", l'algorithme de vérification n'effectue aucun changement au tableau de réallocation courante F' et avance d'une instruction.

Une fois arrivé à la fin du code intermédiaire, le programme recommence au début jusqu'à ce que le type de chaque variable reste inchangé. L'algorithme de vérification échoue quand on lui présente un code avec des variables non monomorphes, car il vérifie sur les instructions de définition "*def*" que les données de type dans le sous-tableau D spécialisent le type déjà présent dans le tableau de réallocation courante F'. Un code qui passe la vérification monomorphe et initialise les données de type de ses variables doit être correctement typé, puisque chaque registre virtuel (v_i) ne peut contenir qu'un seul type. En d'autres mots, l'interpréteur de types vérifie que toutes les utilisations possibles d'une variable sont conformes avec son type et ceci est donc assuré par le fait que les

opérations des instructions PC sur le tableau de réallocation F passent une vérification monomorphe.

Une réallocation est valide si dans le code intermédiaire original, toute instruction d'utilisation "*use x*" utilise la même réallocation que toutes les instructions de définition "*def x*" qui ont pu la définir. C'est à dire que dans le code généré pour la vérification, si une instruction "*use x*" s'est transformée en "*use y*", alors l'instruction "*def x*" s'est transformée en "*def y*".

La validité de la proposition précédente peut être démontrée par un raisonnement par l'absurde. Soit une réallocation non valide pour laquelle l'algorithme se termine sur la fin du code. Si la réallocation n'est pas valide, il existe alors une instruction "*use x*" dans le champ d'une instruction "*def x*" pour laquelle la réallocation de la variable x est la variable z, alors que la réallocation de la variable x pour l'instruction "*def x*" est la variable y, où la variable y est différente de la variable z. Il existe une séquence d'instructions du code intermédiaire original "*def x*", "*i-1*", "*i-2*", ...," *i-k*", "*use x*" qui conduit de l'instruction "*def x*" à "*use x*" par un flot d'exécutions correctes. Il n'y a pas d'autre instruction "*def x*" à l'intérieur de la séquence, sinon ce serait elle qui définit l'instruction "use x". Toutes les autres instructions préservent la réallocation courante du registre x lors du déroulement de l'algorithme. En effet, les instructions "*nop*", "*return*" et "*use*" ne changent pas la réallocation courante et l'instruction de branchement "*brch*" s'assure que la réallocation après le branchement est la même que celle avant.

La séquence d'instructions précitée est donc en contradiction avec la proposition par l'absurde, ce qui démontre la validité de la proposition initiale, objet de la démonstration.

On peut également démontrer par le raisonnement suivant que si un code intermédiaire, généré à partir d'un code intermédiaire original et de son tableau de réallocation valide, passe un procédé de vérification, alors le code intermédiaire original passe aussi le procédé de vérification. Supposons que dans le procédé de vérification du code intermédiaire on stocke en mémoire volatile (RAM) les types des variables pour chaque cible de branchement ainsi qu'un tableau courant et une liste de travail. Les tableaux sont initialisés avec le type "*top*", le plus grand élément du treillis des types. Le tableau courant est également initialisé avec le type "*top*" pour chaque variable non paramétrée et avec les types de la signature pour les variables paramétrées. L'algorithme de vérification commence avec l'instruction associée au point d'entrée de la méthode dans la liste de travail. On enlève la première instruction de la liste de travail. On calcule le nouveau tableau courant après l'exécution de cette instruction à partir de l'interpréteur de type et on l'unifie à tous les tableaux qui correspondent aux instructions qui peuvent succéder. On rajoute à la table de travail les instructions dont les tableaux ont changé. Le code intermédiaire est vérifié avec succès si l'on aboutit à une liste de travail vide. Le procédé de vérification rejet un algorithme si une unification de types est impossible ou si l'interpréteur de types rencontre une instruction incompatible avec les données de type dans le tableau des registres.

En prenant une étape quelconque de la vérification du code intermédiaire original, l'on a comme hypothèse que la vérification s'est bien déroulée jusqu'à présent, en on a à disposition le tableau courant et le contenu des registres. On part aussi de l'hypothèse que l'étape correspondante de la vérification du code intermédiaire avec réallocation se déroule bien et on sait que la réallocation est valide. Pour terminer cette démonstration, on distingue plusieurs cas selon l'instruction en cours :

"*nop*" : L'instruction ne concerne pas les variables, on avance à l'instruction suivante sans rien modifier. Le procédé de vérification ne peut pas échouer sur une instruction "*nop*".

"*def x*" : Mise à jour du tableau courant avec le nouveau type. Le procédé de vérification ne peut pas échouer sur un "*def*" car il prend un argument dans la pile, mais on n'est pas concerné par les types dans la pile.

"*return*" : La vérification se poursuit normalement. Le procédé de vérification ne peut pas échouer ici.

"*use x*" : Le procédé de vérification peut échouer si l'interpréteur de type décide que le type de la variable x n'est pas compatible avec son utilisation. La validité de la réallocation nous assure que la réallocation de la variable x n'a pas changé entre cette instruction "use" et les instructions "*def*" qui lui correspondent. Comme la vérification du code réalloué a réussi jusqu'ici, le type de la variable x est compatible avec son

utilisation. Donc, le procédé de vérification passe avec succès sur l'instruction.

"*brch x y*" : Le procédé de vérification peut échouer dans l'unification du tableau courant avec celui correspondant au registre *x* ou avec celui correspondant au registre *y*. Or, la validité de la réallocation nous assure que la réallocation courante au moment de l'instruction de branchement "*brch*" est la même que celle en *x* et en *y*. De plus, la vérification du code réalloué réussit pour cette instruction, la vérification réussit donc aussi pour le code intermédiaire non réalloué.

En ce qui concerne le traitement des exceptions dans le code intermédiaire, si l'on se trouve sur une partie de code protégée par un "handler" d'exception, chaque instruction doit être considérée comme un branchement potentiel vers la portion de code intermédiaire qui traite l'exception, c'est à dire que la réallocation courante doit être la même que celle donnée dans le tableau de réallocation de l'exception. Si ce traitement des exceptions permet d'assurer la correction du typage, il est par contre trop restrictif dans le sens où il ne permet pas de changer la réallocation d'une variable à l'intérieur d'un bloc protégé par un même handler d'exception. Exemple :

PC 1 : *def* r_1 comme entier: réallocation $r_1 \rightarrow v_1$

PC 2 : *def* r_1 comme référence: réallocation $r_1 \rightarrow v_2$

Si les deux instructions "*def*" sont protégées par le même "handler" d'exception et que la variable r_1 n'est pas utilisée dans le traitement de l'exception, alors le

typage est respecté, mais l'algorithme de vérification du tableau de réallocation échoue car la variable r_1 ne peut avoir qu'une seule réallocation à l'entrée du traitement de l'exception. La solution est de créer un registre factice (appelons le Top) qui interdit l'utilisation du registre réel réalloué en Top. Dans l'exemple, il suffit de réallouer à Top le registre r_1 dans le tableau de réallocation du traitement de l'exception. Une réallocation à Top se transmet par les branchements et peut apparaître dans un tableau de réallocation qui correspond au début du traitement d'une exception.

En ce qui concerne les appels de sous-routines, on doit tenir compte du fait qu'elles font perdre le lien entre les instructions "*use*" et les instructions "*def*" des variables. Pour rendre compatible les réallocations avec les appels de sous-routines JSR/RET, il faut ajouter un système de contexte de réallocation.

En faisant référence à la Fig. 10, dans une deuxième forme d'exécution de l'invention, le calcul de la réallocation et du code réalloué (étape 1010) est effectué entièrement dans le microcircuit à ressources limitées, suivi d'une étape 1012 de vérification du code réalloué dans le microcircuit et, en cas de succès, l'installation du code intermédiaire original pour exécution (étape 1016). Le calcul de la réallocation et du code réalloué dans le microcircuit peut être effectué selon les méthodes connues, toutefois sans qu'il soit nécessaire d'effectuer une optimisation du code réalloué du fait qu'il n'est pas utilisé pour l'exécution, mais uniquement pour la vérification de l'intégrité et de l'innocuité du code intermédiaire.

Le code intermédiaire peut soit être stocké en mémoire persistante accessible en écriture et en lecture,

par exemple de type EEPROM, lors du premier chargement depuis le système externe, soit être réinstallé après vérification, en appliquant une fonction de hachage au premier chargement, en stockant le résultant dans une mémoire du microcircuit et en le comparant avec la valeur de hachage calculée à partir du code intermédiaire chargé après vérification.

Revendications

1. Procédé de vérification de code intermédiaire exécutable par un microcircuit à ressources limitées connecté à un système de traitement de données externe, comportant une étape de modification du code intermédiaire comprenant la réallocation des registres réels de type *r* à des registres virtuels *v* de type monomorphe, et de construction d'un code réalloué dont les instructions PC font référence aux registres virtuels *v*, et une étape de vérification du code réalloué dans le microcircuit à ressources limitées, caractérisé en ce que, en cas de succès de la vérification du code réalloué dans le microcircuit, l'on installe le code intermédiaire original dans le microcircuit à ressources limitées pour exécution.

2. Procédé selon la revendication 1, caractérisé en ce que l'étape de la construction du code réalloué est effectuée dans le microcircuit à ressources limitées.

3. Procédé selon l'une des revendications précédentes, caractérisé en ce que le code intermédiaire original est chargé dans le microcircuit à ressources limitées avant l'exécution du procédé de vérification et stocké dans une mémoire du microcircuit.

4. Procédé selon l'une des revendications précédentes, caractérisé en ce que, dans l'étape de modification du code intermédiaire, l'on crée un composant de réallocation T définissant la réallocation des registres réels de type aux registres virtuels monomorphes.

5. Procédé selon la revendication précédente, caractérisé en ce que le composant de réallocation T

contient un sous-tableau de réallocation D contenant la réallocation pour chaque instruction de définition "*def*" du code intermédiaire définissant le type d'une variable dans un des registres réels, et un tableau de réallocation F contenant la réallocation des registres réels de type aux registres virtuels monomorphes.

6. Procédé selon la revendication précédente, caractérisé en ce que, lors du procédé de vérification du composant de réallocation T, l'on vérifie si le sous-tableau de réallocation D pour les instructions de définition "*def*" comporte une valeur de réallocation et, s'il ne comporte pas de valeur de réallocation, le code intermédiaire est rejeté.

7. Procédé selon l'une des revendications précédentes, caractérisé en ce que, lors du procédé de vérification du tableau de réallocation F, pour une instruction du code intermédiaire qui utilise une variable, l'on vérifie si la variable utilisée dans le tableau de réallocation courante n'est pas "*nul*" et, si elle est "*nul*", le code intermédiaire est rejeté.

8. Procédé selon l'une des revendications précédentes, caractérisé en ce que, lors du procédé de vérification du tableau de réallocation dans le cadre d'une instruction de branchement "*brch*" à des instructions cibles, l'on vérifie l'identité du tableau de réallocation courante F' avec les lignes du tableau de réallocation F correspondant aux instructions cibles et, en cas de différences, le code intermédiaire est rejeté.

9. Procédé selon l'une des revendications 1 à 3, caractérisé en ce que l'étape de modification du code intermédiaire est effectuée dans le microcircuit à ressources limitées.

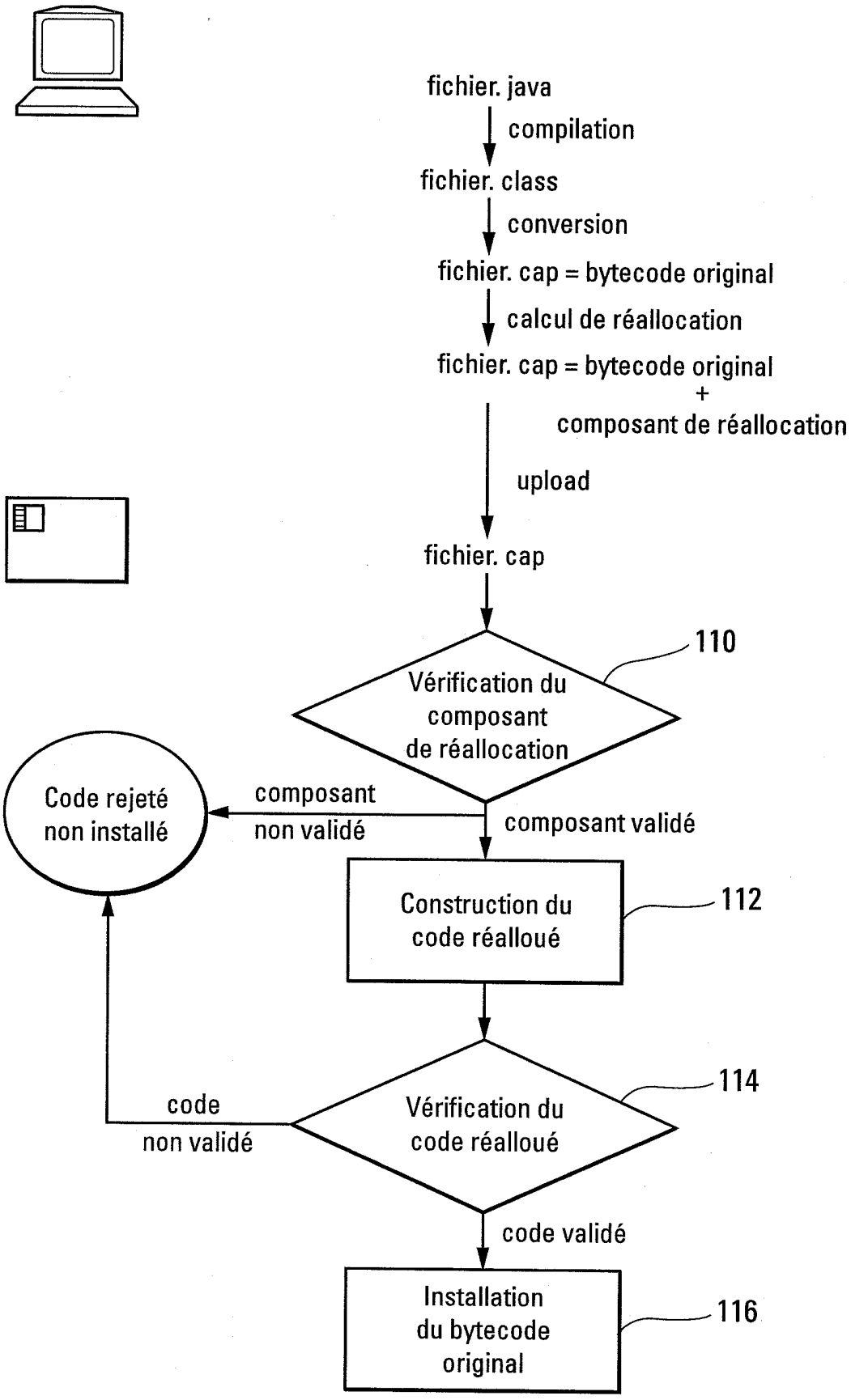
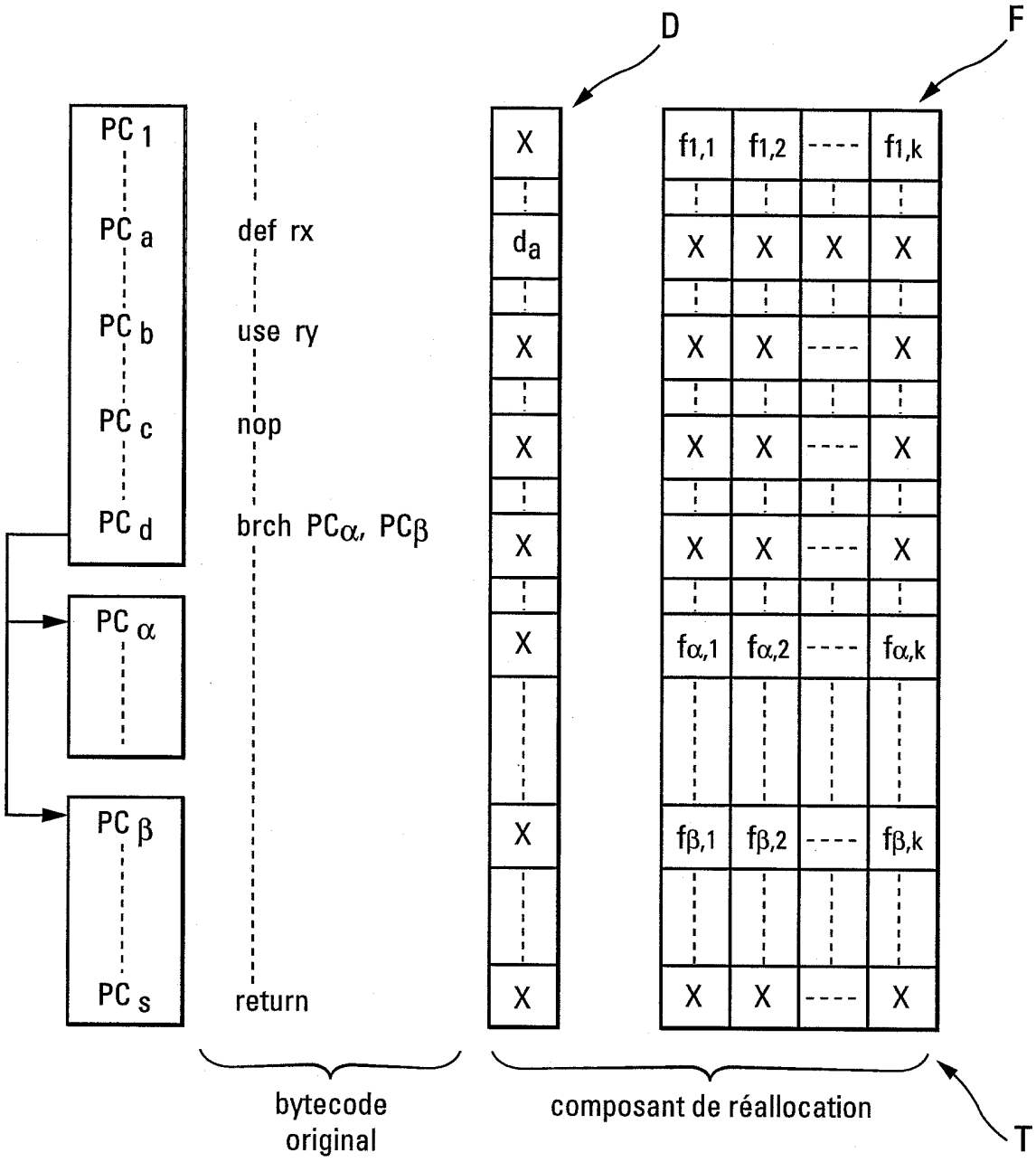

fichier. java
compilation
fichier. class
conversion
fichier. cap = bytecode original
calcul de réallocation
fichier. cap = bytecode original
+
composant de réallocation
upload
fichier. cap
110
Vérification du composant de réallocation
Code rejeté non installé
composant non validé
composant validé
Construction du code réalloué
112
114
Vérification du code réalloué
code non validé
code validé
Installation du bytecode original
116

Fig. 1



Légende : X case vide d'une table

Fig. 2

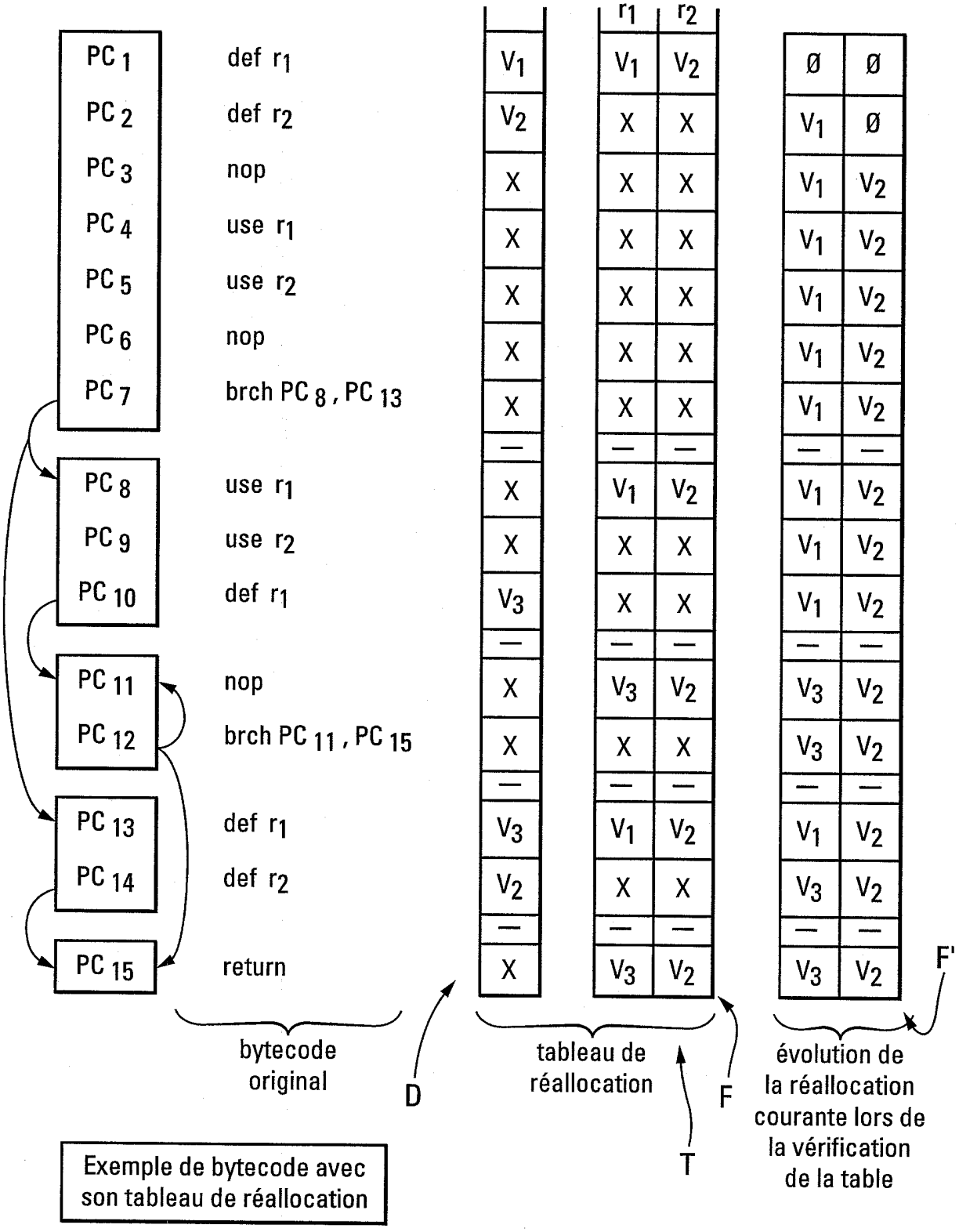

registres réels = r1 , r2
registres virtuels = Ø , v1 , v2 , v3
PC = PC 1 , PC 2 , ... , PC 15
PC 1 def r1
PC 2 def r2
PC 3 nop
PC 4 use r1
PC 5 use r2
PC 6 nop
PC 7 brch PC 8 , PC 13
PC 8 use r1
PC 9 use r2
PC 10 def r1
PC 11 nop
PC 12 brch PC 11 , PC 15
PC 13 def r1
PC 14 def r2
PC 15 return
r1 r2
bytecode original
D
tableau de réallocation
F
T
évolution de la réallocation courante lors de la vérification de la table
F'
Exemple de bytecode avec son tableau de réallocation

Fig. 3

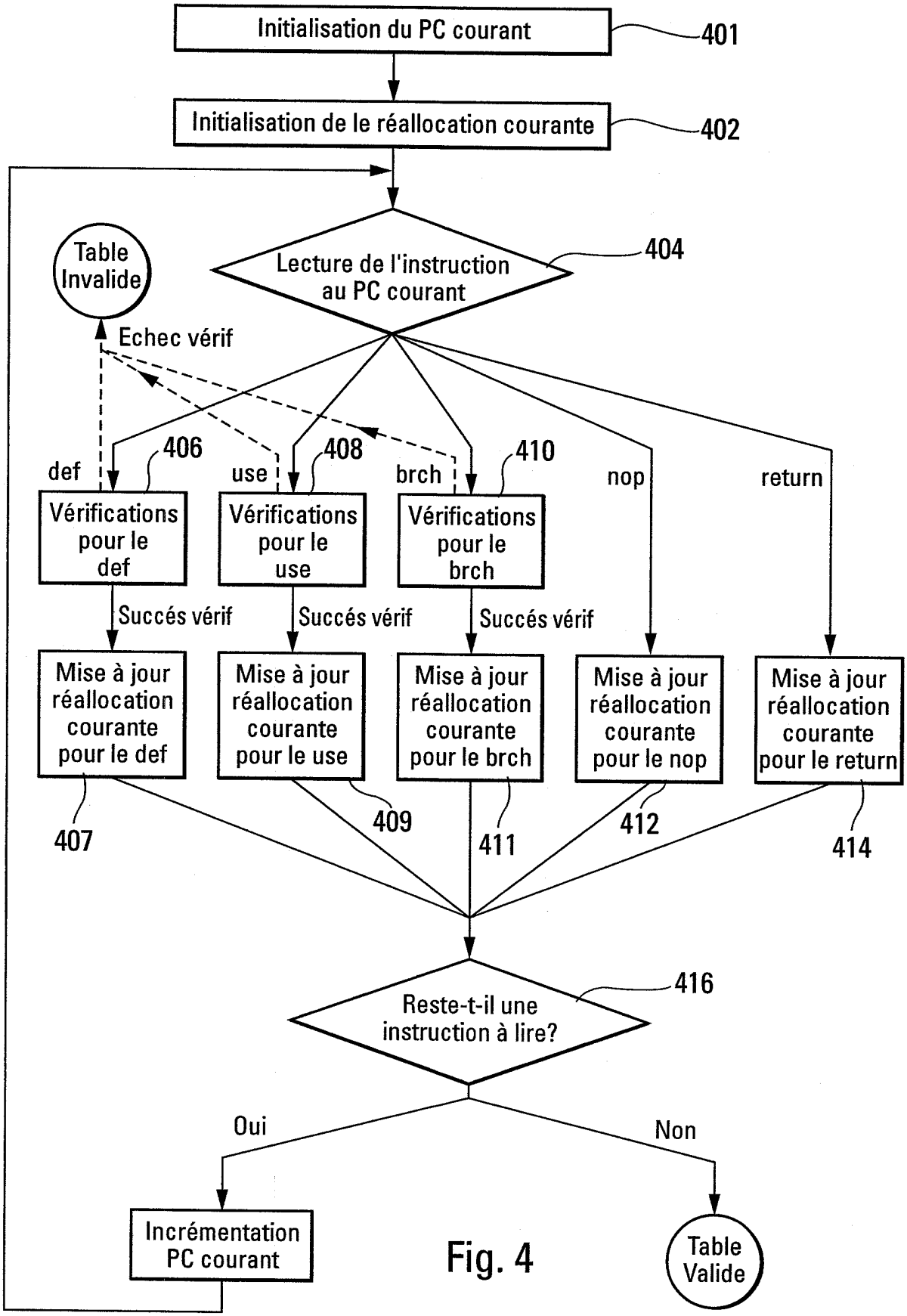

Initialisation du PC courant
401
Initialisation de le réallocation courante
402
Table Invalide
Lecture de l'instruction au PC courant
404
Echec vérif
def
406
use
408
brch
410
nop
return
Vérifications pour le def
Vérifications pour le use
Vérifications pour le brch
Succés vérif
Succés vérif
Succés vérif
Mise à jour réallocation courante pour le def
Mise à jour réallocation courante pour le use
Mise à jour réallocation courante pour le brch
Mise à jour réallocation courante pour le nop
Mise à jour réallocation courante pour le return
407
409
411
412
414
Reste-t-il une instruction à lire?
416
Oui
Non
Incrémentation PC courant
Table Valide

Fig. 4

Algorithme de vérification de la table, cas du def

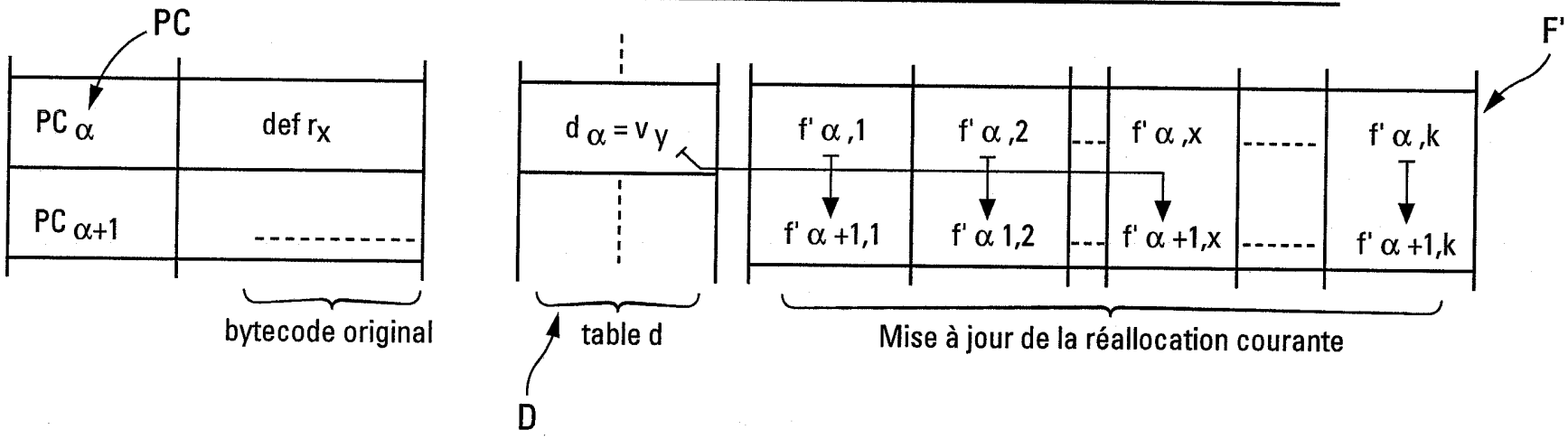


Fig. 5

cas du nop

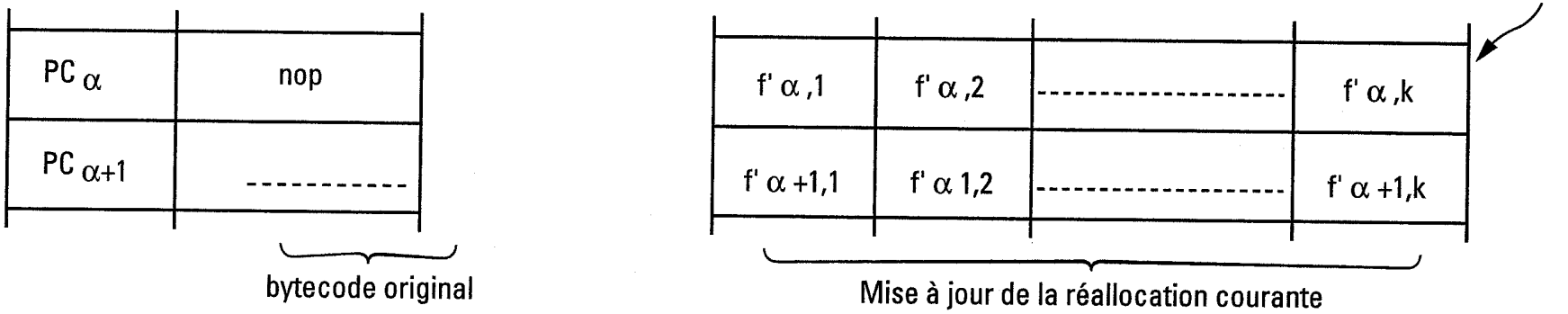
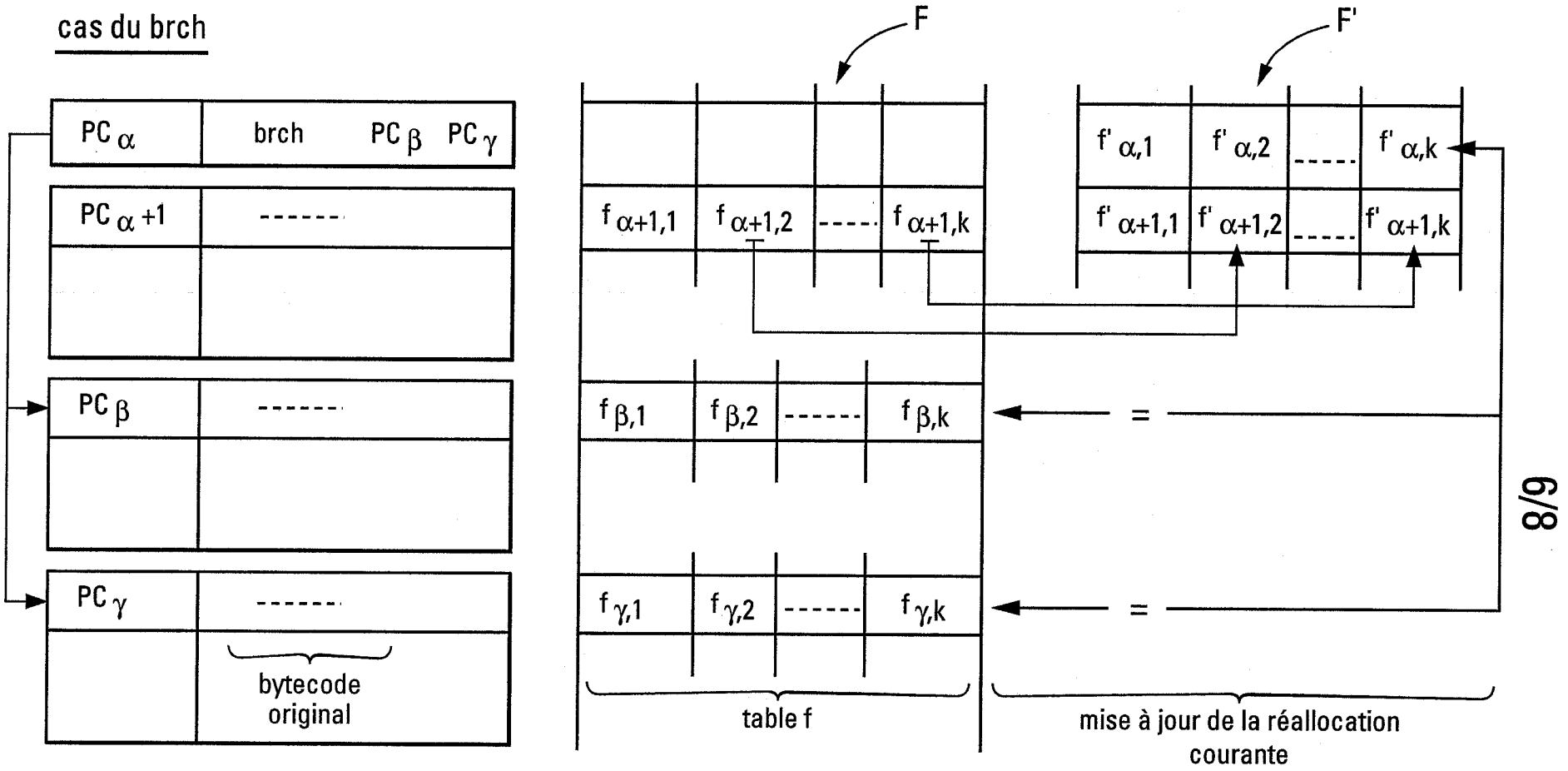


Fig. 8

Légende : ⟼ affectation



Vérifier que $f'_{\alpha,1} = f_{\beta,1} = f_{\gamma,1}$, $f'_{\alpha,2} = f_{\beta,2} = f_{\gamma,2}$, ... , $f'_{\alpha,k} = f_{\beta,k} = f_{\gamma,k}$

Légende : ⟼ affectation

⟵ = ⟶ Test d'égalité de deux vecteurs lignes

Fig. 7

2840084

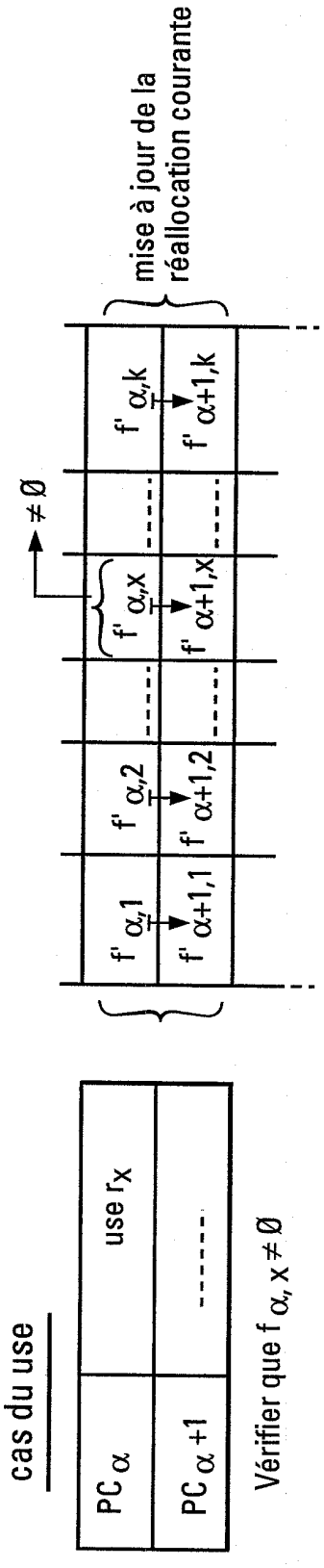

cas du use
PC α
use rx
PC α +1
Vérifier que f α, x ≠ Ø
≠ Ø
f' α,1
f' α+1,1
f' α,2
f' α+1,2
f' α,x
f' α+1,x
f' α,k
f' α+1,k
mise à jour de la réallocation courante

Fig. 6

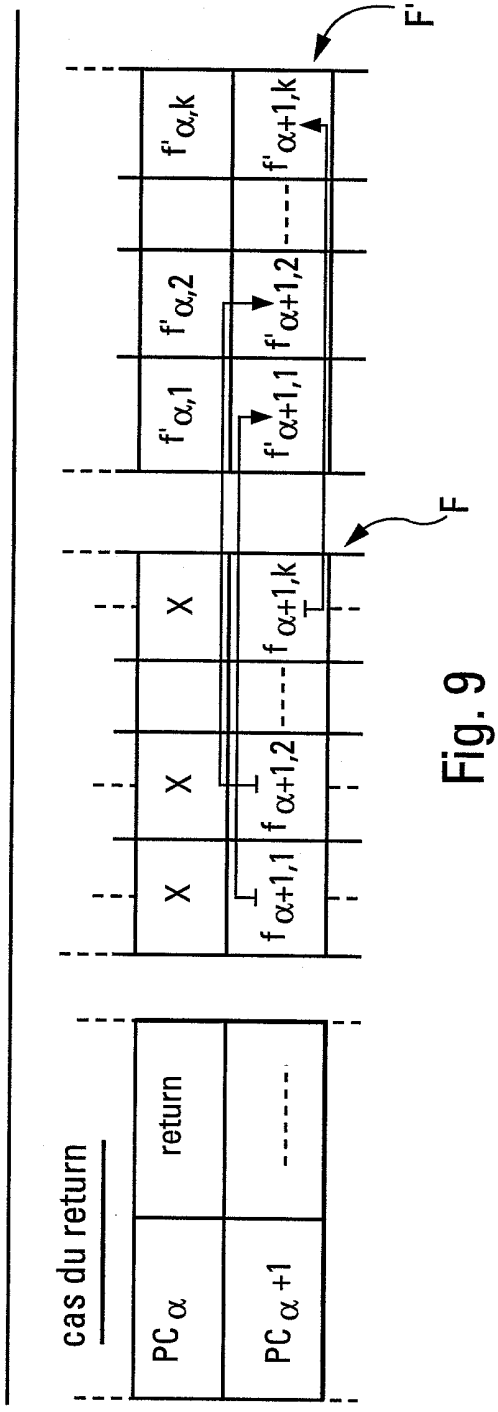

cas du return
PC α
return
PC α +1
X
X
X
f α+1,1
f α+1,2
f α+1,k
F
f'α,1
f'α,2
f'α,k
f'α+1,1
f'α+1,2
f'α+1,k
F'

Fig. 9

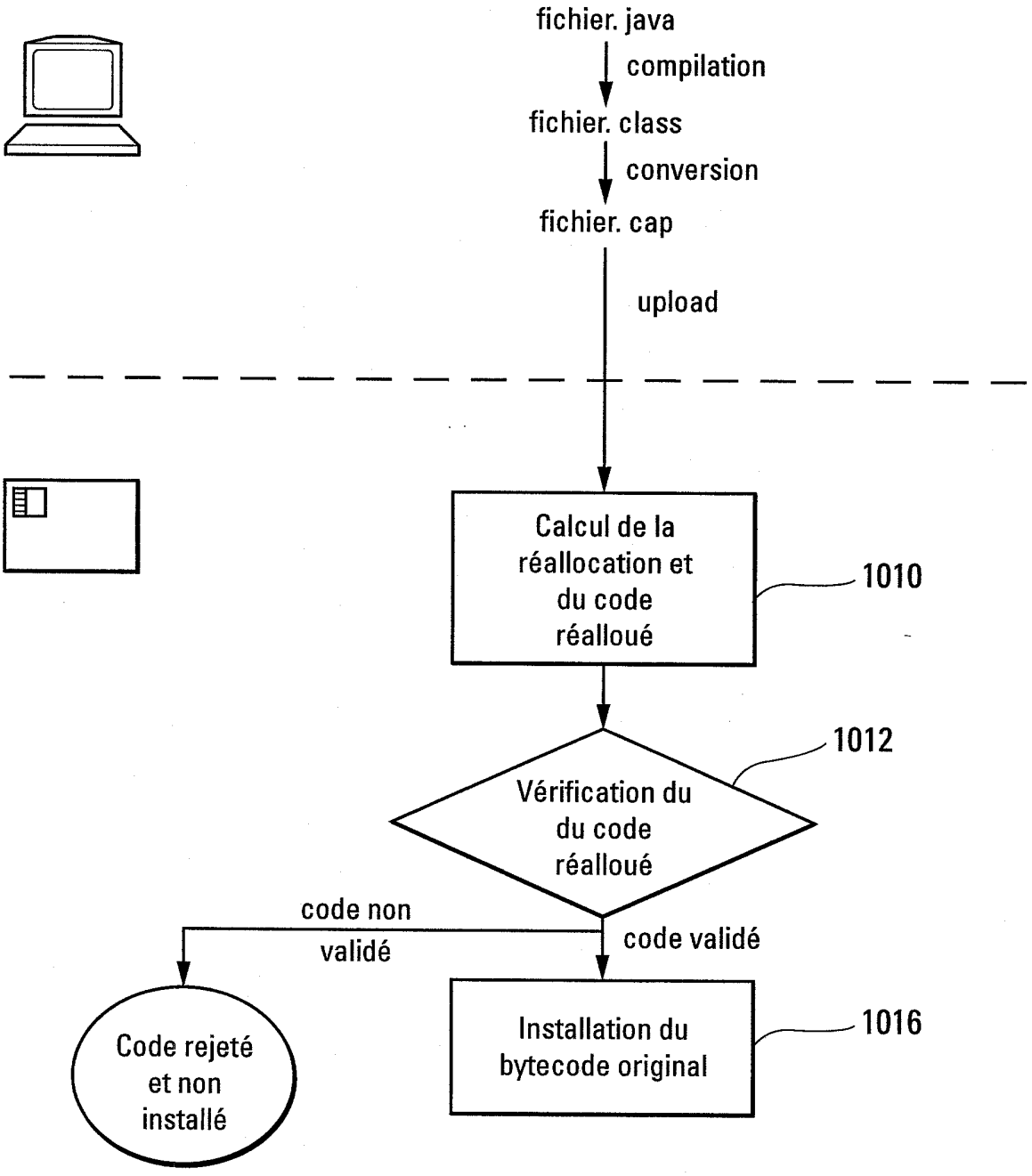

fichier. java
compilation
fichier. class
conversion
fichier. cap
upload
Calcul de la réallocation et du code réalloué
1010
1012
Vérification du du code réalloué
code non validé
code validé
Code rejeté et non installé
Installation du bytecode original
1016

Fig. 10

INPI
INSTITUT NATIONAL DE LA PROPRIETE INDUSTRIELLE

2840084

N° d'enregistrement national

FA 620278
FR 0206445

RAPPORT DE RECHERCHE PRÉLIMINAIRE

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

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		
X	LEROY X: "BYTECODE VERIFICATION ON JAVA SMART CARDS" SOFTWARE PRACTICE & EXPERIENCE, JOHN WILEY & SONS LTD. CHICHESTER, GB, vol. 32, no. 4, 10 avril 2002 (2002-04-10), pages 319-340, XP001081545 ISSN: 0038-0644	1,3	G06F11/36 G06K19/07
A	* alinéa '0004! *	2,4-9	
A	FR 2 797 963 A (TRUSTED LOGIC) 2 mars 2001 (2001-03-02)		
			DOMAINES TECHNIQUES RECHERCHÉS (Int.CL.7)
			G06F

Date d'achèvement de la recherche	Examinateur
20 janvier 2003	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 0206445 FA 620278

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 20-01-2003
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	FR 2797963 A1	02-03-2001
		AU 7015000 A	19-03-2001
		CN 1370294 T	18-09-2002
		EP 1212678 A2	12-06-2002
		WO 0114958 A2	01-03-2001

EPO FORM P0465