

(12) DEMANDE INTERNATIONALE PUBLIÉE EN VERTU DU TRAITÉ DE COOPÉRATION
EN MATIÈRE DE BREVETS (PCT)

(19) Organisation Mondiale de la Propriété
Intellectuelle
Bureau international



PCT



(43) Date de la publication internationale
27 juillet 2006 (27.07.2006)

(10) Numéro de publication internationale
WO 2006/077252 A2

(51) Classification internationale des brevets :
G06F 9/44 (2006.01) **G06F 9/45** (2006.01)

(21) Numéro de la demande internationale :
PCT/EP2006/050350

(22) Date de dépôt international :
20 janvier 2006 (20.01.2006)

(25) Langue de dépôt : français

(26) Langue de publication : français

(30) Données relatives à la priorité :
0500678 21 janvier 2005 (21.01.2005) FR

(71) Déposant (pour tous les États désignés sauf US) : **GEM-PLUS** [FR/FR]; Avenue du Pic de Bertagne, Parc d'activité de Gémenos, F-13420 Gémenos (FR).

(72) Inventeur; et

(75) Inventeur/Déposant (pour US seulement) : **GALLAND, Antoine** [FR/FR]; 65, rue Chape, F-13400 Marseille (FR).

(81) États désignés (sauf indication contraire, pour tout titre de protection nationale disponible) : AE, AG, AL, AM, AT,

AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, LY, MA, MD, MG, MK, MN, MW, MX, MZ, NA, NG, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SM, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) États désignés (sauf indication contraire, pour tout titre de protection régionale disponible) : ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), eurasien (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), européen (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IS, IT, LT, LU, LV, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

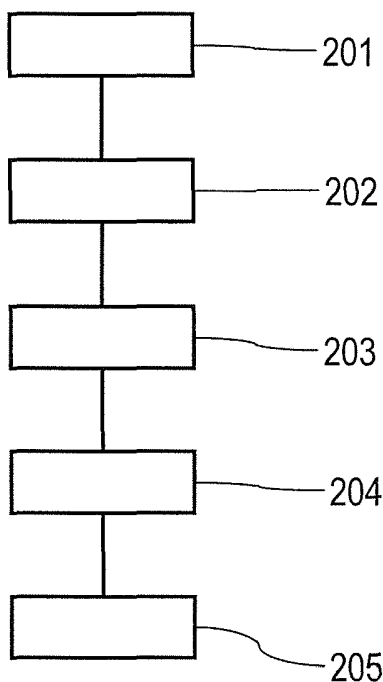
Publiée :

— sans rapport de recherche internationale, sera republiée dès réception de ce rapport

[Suite sur la page suivante]

(54) Title: ESCAPE ANALYSIS IN AN OBJECT ORIENTED LANGUAGE PROGRAM FOR A VIRTUAL STACK MACHINE

(54) Titre : ANALYSE D'ÉCHAPPEMENT DANS UN PROGRAMME EN LANGAGE ORIENTÉ OBJET POUR MACHINE VIRTUELLE À PILE



(57) Abstract: The invention relates to an escape analysis method consisting in carrying out a static escape analysis on an intermediate execution language program for a virtual stack machine (201) in a first processing device and in generating escape signatures associated to the program methods according to escape rules (202), in loading the generated escape signatures and a program (203) in a second processing device provided with a virtual stack machine, in carrying out a partial escape static analysis (204) based on the loaded signatures, in generating test escape signatures according to said escape rules and in checking (205), for a given method, the coherence between the test and loaded escape signatures thereof. Said invention makes it possible, in particular, to limit an escape analysis carried out on a second device ensuring the integrity of the loaded program.

(57) Abrégé : L'invention porte sur un procédé d'analyse d'échappement, comprenant les étapes consistant à : dans un premier dispositif de traitement, réaliser une analyse statique d'échappement sur un programme en langage intermédiaire d'exécution pour machine virtuelle à pile (201) et générer des signatures d'échappement associées à des méthodes du programme en fonction de règles d'échappement (202) ; dans un second dispositif de traitement muni d'une machine virtuelle à pile : charger les signatures d'échappement générées et le programme (203); réaliser une analyse statique d'échappement partielle (204) basée sur les signatures chargées et générer, en fonction desdites règles d'échappement, des signatures d'échappement de test; pour une méthode donnée, vérifier (205) la cohérence entre sa signature d'échappement de test et sa signature d'échappement chargée. L'invention permet notamment de limiter l'analyse d'échappement effectuée sur le second dispositif, tout en garantissant l'intégrité du programme chargé.

WO 2006/077252 A2



En ce qui concerne les codes à deux lettres et autres abréviations, se référer aux "Notes explicatives relatives aux codes et abréviations" figurant au début de chaque numéro ordinaire de la Gazette du PCT.

ANALYSE D'ÉCHAPPEMENT DANS UN PROGRAMME EN LANGAGE ORIENTÉ OBJET POUR MACHINE VIRTUELLE À PILE

L'invention concerne de façon générale les programmes informatiques, et
5 en particulier les procédés d'analyse d'échappement dans les programmes en langage orienté objet pour machine virtuelle à pile.

L'analyse d'échappement est effectuée sur un programme afin de déterminer la durée de vie d'objets. Une analyse d'échappement permet notamment des optimisations possibles lors de l'exécution, par exemple la gestion
10 mémoire en pile d'objets ou une élimination de synchronisation.

Une fonction de ramasse-miettes (désignée par le terme garbage collection en langue anglaise) est intégrée dans une machine virtuelle afin de récupérer des emplacements mémoire qui ont cessé d'être utilisés par un programme. Un ramasse-miettes balaye l'ensemble des emplacements mémoire qui ont été alloués
15 à des objets dans un espace mémoire dédié. Des emplacements devenus inutilisés sont alors identifiés. Lorsqu'un objet n'est plus pointé par un quelconque autre objet, ses emplacements sont considérés comme inutilisés. Ces emplacements mémoire sont alors marqués pour pouvoir être réalloués à un autre processus du programme.

20 La récupération de place mémoire requiert un nombre important d'opérations et la récupération de place mémoire par le ramasse-miettes peut alors occuper une partie excessive des capacités de traitement d'un dispositif à capacité de traitement limitée. Le temps requis pour la récupération de place dans une mémoire non volatile peut également être important, du fait de leur longue durée
25 d'écriture.

L'objet de l'invention consiste à adapter les analyses d'échappement de programme dans le but de mettre en œuvre ce type d'analyse au moyen d'un dispositif dont les capacités de traitement ne lui permettent pas avec les techniques connues d'analyser l'intégralité des méthodes du programme.
30 L'invention vise également à garantir l'intégrité du résultat de l'analyse d'échappement.

L'invention porte ainsi sur un procédé d'analyse d'échappement, comprenant les étapes consistant à :

- dans un premier dispositif de traitement, réaliser une analyse statique d'échappement sur un programme en langage intermédiaire d'exécution pour machine virtuelle à pile et générer des signatures d'échappement associées à des méthodes respectives du programme en fonction de règles d'échappement prédéterminées;
- dans un second dispositif de traitement muni d'une machine virtuelle à pile :
 - charger les signatures d'échappement générées et le programme;
 - réaliser une analyse statique d'échappement partielle basée sur les signatures chargées et générer, en fonction desdites règles d'échappement prédéterminées, des signatures d'échappement de test associées aux méthodes munies d'une signature d'échappement ;
 - pour une méthode donnée, vérifier la cohérence entre sa signature d'échappement de test et sa signature d'échappement chargée.

Selon une variante, le deuxième dispositif de traitement utilisé est un dispositif à capacité de traitement limitée.

Selon encore une variante, une signature d'échappement générée pour une méthode comprend des informations associées à ses paramètres tels que ses arguments, son objet courant et/ou son objet de retour.

Selon une autre variante, l'analyse statique d'échappement partielle et la vérification de cohérence sont réalisées après le chargement du programme dans le second dispositif de traitement.

Selon encore une autre variante, le programme en langage intermédiaire d'exécution est sous forme de pseudo-code binaire Java ou CLI.

D'autres caractéristiques et avantages de l'invention ressortiront clairement de la description qui en est faite ci-après, à titre indicatif et nullement limitatif, en référence aux dessins annexés, dans lesquels :

- la figure 1 illustre schématiquement deux dispositifs de traitement mis en œuvre dans l'invention ;
- la figure 2 illustre les étapes de procédé d'analyse d'un programme selon l'invention ;
- 5 - la figure 3 illustre des règles d'échappement sur des instructions d'un pseudo-code binaire Java ;
- les figures 4 à 6 illustrent un code Java et l'analyse statique d'échappement effectuée sur son équivalent en pseudo-code binaire;
- la figure 7 illustre une analyse statique et une vérification effectuées dans
10 un dispositif de traitement chargeant le programme.

On utilisera par la suite les notions suivantes :

Un objet *o* est dit non échappé (ou capturé) par une méthode *m* dans laquelle il a été alloué seulement si sa durée de vie ne dépasse pas celle de la
15 méthode *m*. Une allocation de *o* capturée peut être détruite sans risque à la sortie de cette méthode *m*. Les emplacements mémoire occupés par l'objet capturé *o* peuvent donc être libérés à la fin de l'exécution de la méthode *m*. Par opposition, un objet *o* échappé est un objet auquel il peut potentiellement être fait référence après la fin de l'exécution de la méthode *m* dans laquelle il a été alloué.

20 Un dispositif à capacité de traitement limitée désignera un dispositif de traitement dont la capacité de traitement est insuffisante pour réaliser à lui seul l'analyse statique d'échappement sur l'intégralité du programme. Un dispositif à capacité de traitement limitée présentera par exemple une mémoire vive inférieure à 32 KiloOctets et une mémoire EEPROM inférieure à 512 KiloOctets.

25

L'invention propose un procédé d'analyse d'échappement dans lequel une analyse statique d'échappement est réalisée sur un programme en langage intermédiaire d'exécution pour machine virtuelle à pile, dans un premier dispositif de traitement. Des signatures d'échappement associées aux méthodes du
30 programme sont générées selon des règles d'échappement prédéterminées. Le programme et les signatures d'échappement sont chargées dans un second

dispositif de traitement et une analyse statique d'échappement partielle, basée sur les signatures d'échappement chargées est réalisée. Des signatures de test associées aux objets du programme chargé sont générées par les mêmes règles d'échappement prédéterminées. La cohérence entre les signatures d'échappement chargées et les signatures d'échappement de test est alors vérifiée. L'intégrité du programme chargé est ainsi vérifiée. Les signatures d'échappement chargées ne peuvent donc pas être frauduleusement utilisées pour contourner les règles de sécurité sur les références d'objets.

L'invention propose un procédé d'analyse d'échappement réalisable sur une forme du programme habituellement disponible et limitant les ressources nécessaires pour effectuer la vérification de cohérence d'échappement.

La figure 1 illustre un exemple de système mettant en œuvre l'invention. Un premier dispositif de traitement DTA est formé d'un ordinateur. Le dispositif DTA est connecté à un lecteur de carte à puce LEC, par l'intermédiaire duquel il peut communiquer avec le second dispositif de traitement DCTL formé d'une carte à puce. Le dispositif de traitement DTA dispose d'un programme en langage intermédiaire d'exécution, destiné à être soumis au procédé d'analyse statique d'échappement et à un chargement dans le dispositif DCTL.

20

Un exemple de procédé mettant en œuvre l'invention est illustré en référence à la figure 2. L'analyse d'échappement sera illustrée sur les instructions de la machine virtuelle Java (en anglais nommé *bytecode* Java, décrites dans la spécification nommée « The Java Virtual Machine Specification » écrite par Tim Lindholm et Frank Yellin aux éditions The Java Series, Addison-Wesley (1ère ed.) en 1996).

A l'étape 201, le premier dispositif de traitement DTA réalise une analyse statique d'échappement sur le programme sous sa forme de pseudo-code binaire. L'analyse statique d'échappement a pour objectif de déterminer la durée de vie d'objets et vise à générer des signatures d'échappement résumant les comportements d'échappement. Pour chaque méthode *m* on génère deux

30

signatures d'échappement, l'une nommée *directe* et notée *Sem* et une seconde nommée *unifiée* et notée *[Se]m*. Les signatures se focalisent sur les paramètres d'appels d'une méthode qui sont dans l'ordre : l'objet courant « *this* », les arguments $\text{arg}_1, \dots, \text{arg}_n$ ainsi que l'objet de retour retourné par la méthode que
5 l'on nomme « *return* ».

L'analyse statique d'échappement réalisée dans le premier dispositif de traitement DTA peut être réalisée comme suit. L'analyse statique d'échappement établit un graphe du flot de contrôle du programme et parcourt cette représentation
10 abstraite du programme. Un graphe des méthodes d'appels et de leurs interactions est ainsi établi afin de réaliser une analyse inter-méthodes et un graphe d'exécution de chaque méthode est établi afin de réaliser une analyse intra-méthodes. Des procédés connus d'estimation de graphe de flot de contrôle de langages orientés objets peuvent être utilisés à cet effet.

15 L'analyse statique d'échappement coordonne une analyse intra-méthodes et une analyse inter-méthodes. En effet, pour déterminer le comportement d'une méthode *m* faisant appel à une autre méthode *m'*, il est nécessaire d'étudier également le comportement de cette autre méthode. Des hypothèses d'échappement d'objets sont adoptées en début d'analyse (les hypothèses de
20 départ supposent que tous les objets sont capturés), puis ces hypothèses sont mises à jour de façon incrémentale durant l'analyse inter-méthodes. Lorsque les hypothèses d'échappement convergent vers un point fixe, l'analyse statique d'échappement est interrompue.

25 On va par la suite détailler l'analyse intra-méthode réalisée. L'analyse utilise un interpréteur abstrait associant des règles d'évaluation pour chaque instruction du programme. L'interpréteur abstrait exécute les instructions de la machine virtuelle Java en suivant le flot de contrôle du programme. L'analyse intra-méthode évalue les instructions d'une méthode de manière isolée, méthode
30 par méthode, sans avoir connaissance du comportement des autres méthodes.

L'interpréteur peut être représenté par un ensemble de relations de transition :

$$i : (R, P) \Rightarrow (R', P')$$

où i est l'instruction évaluée, R et P représentent respectivement la valeur

- 5 des registres et de la pile avant l'exécution de i , et R' et P' leur valeur après son exécution. Par simplification les registres et les cellules de la pile manipulent des valeurs que l'on note v . Ces valeurs regroupent à la fois les types de bases (entier, booléen...) ainsi que les références d'objets ou de tableaux.

- Par exemple, l'instruction `aload_k` duplique le contenu du registre k en
10 sommet de pile :

$$\text{aload_k} : (R\{k = v\}, P) \Rightarrow (R\{k = v\}, P.v)$$

- Après exécution de l'instruction, le registre R_k et le sommet de pile contiennent donc la même valeur v . L'instruction inverse, `astore_k`, dépile la
15 valeur v en sommet de pile pour la placer dans le registre k :

$$\text{astore_k} : (R, P.v) \Rightarrow (R\{k \leftarrow v\}, P)$$

- v est ainsi défini comme étant l'ensemble des références contenues dans un registre ou une cellule de la pile en un point p de l'exécution du programme.
20 Une analyse d'alias entre les registres suit le contenu des registres en chaque point p de l'exécution du programme. Ces références r sont issues des allocations, et marquées par le point du programme qui les ont créées r_p . À chaque instruction du pseudo-code binaire Java, une règle d'échappement est associée. Une allocation en p empile sur la pile une référence définie à capturée $r_p : \perp$. (le signe $\perp$
25 désignera par la suite le plus petit élément du treillis soit une référence définie comme capturée).

$$\text{new} : (R, P) \Rightarrow (R, P.v) \text{ tel que } v = \{r_p\} \text{ et } r_p : \perp$$

Une instruction a potentiellement plusieurs prédécesseurs. Ainsi, l'état des registres et de la pile avant évaluation de i est l'union (notée $\cup$) des états des prédécesseurs de i après leur évaluation. Si on désigne $\text{instr}(p)$ pour l'instruction au point de programme p , $\text{in}(p)$ l'état avant et $\text{out}(p)$ celui après, le système d'équations de l'analyse d'échappement pour chaque point p du programme est le suivant :

$$\text{instr}(p) : \text{in}(p) \Rightarrow \text{out}(p)$$

$$\text{in}(p) = \cup(\text{out}(q)), q \text{ étant un prédécesseur de } p ;$$

avec comme point de départ pour $\text{in}(p_0)$ les données de la signature d'échappement Sem (échappement des paramètres d'appels et connexion entre paramètres d'appels).

L'unification de deux valeurs $v \cup v'$ est l'union de leur ensemble, sans doublons. Les équations sont ensuite résolues par un algorithme standard itératif jusqu'à atteinte du point fixe. Par simplification, une liste globale des valeurs L issues des allocations est maintenue pendant l'analyse (au départ de l'analyse, cette liste est définie à vide). Lorsque la référence d'une allocation s'échappe, elle est enlevée de la liste. L correspond alors à l'ensemble des références capturées $L = \{r_p \mid r_p : \perp\}$. Selon une optimisation détaillée ultérieurement, les allocations correspondantes à ces références peuvent être gérées par une gestion en pile.

En plus des registres et de la pile, on construit un graphe de connexion G qui indique les connexions mémoires existantes entre les valeurs références manipulées par le programme. La structure de données G permet de traiter les affectations. Soit V l'ensemble des valeurs, on définit G tel que $G : V \rightarrow \mathbb{P}(V)$ où $\mathbb{P}(V)$ est l'ensemble des parties de V . Le graphe est mis à jour lors des affectations `putfield` et `aastore` en établissant une connexion entre la valeur destination v_d

et la valeur affectée v_a . Cette connexion est rajoutée à celles déjà existantes depuis v_d . L'instruction `areturn` met à jour G afin d'établir la signature d'échappement sur les connexions entre paramètres pour l'objet retourné. En ce qui concerne la lecture d'un champ d'objet ou d'une cellule de tableau, instructions `getfield` et `aaload`, les valeurs pointées par v notées $G(v)$ sont montées en sommet de pile. L'échappement d'une valeur v lors de l'évaluation d'une instruction i est désignée par $\text{esc}(v) = \top$ telle que $\forall r_p \in v, r_p = \top$. De plus, l'échappement d'une valeur v fait échapper par atteignabilité toutes les valeurs accessibles depuis v telle que $\text{esc}(G(v)) = \top$.

La figure 3 illustre des règles d'échappement sur le pseudo-code binaire Java appliqué à un certain nombre d'instructions et permettant de générer les signatures d'échappement Sem et $[\text{Se}]m$.

Quelles que soient les informations de l'analyse intra-méthode, le fonctionnement de l'analyse inter-méthodes est similaire. L'analyse inter-méthodes calcule une signature pour chaque méthode ayant subi une analyse intra-méthode, cette signature étant valable quel que soit son contexte d'appel. L'analyse inter-méthodes est une analyse statique par interprétation abstraite interagissant avec celle de l'analyse intra-méthode. Les données des signatures sont des semi-treillis. Leurs structures sont issues de celles spécifiées dans l'analyse intra-méthode. Pour la *génération* des signatures, au départ, on définit la signature d'échappement au plus petit élément du treillis. Ensuite, l'algorithme inter-méthodes traite les méthodes les unes après les autres. Pour chaque méthode, on effectue une analyse intra-méthode dont le résultat permet de donner la nouvelle signature d'échappement.

Lors d'une analyse intra-méthode de m , soit $\text{in}(\text{Sem})$ sa signature en entrée et $\text{out}(\text{Sem})$ celle en sortie, si $\text{in}(\text{Sem}) \subset \text{out}(\text{Sem})$ (au moins un des paramètres

de la signature a progressé dans son semi-treillis) alors une seconde analyse intra-méthode doit être effectuée sur m ainsi que sur les méthodes appelant m . Ainsi, tant que le point fixe global des signatures n'est pas atteint, on continue l'analyse inter-méthodes. L'interaction entre les analyses inter-méthodes et intra-méthode

5 permet de générer le point fixe des signatures d'échappement Sem et $[Se]m$.

On va maintenant définir la signature d'échappement sur les paramètres d'appels. Soit Sem' la signature d'échappement de la méthode m' appelée pendant l'analyse de m . On définit cette première signature sur les paramètres d'appels

10 par :

$$\forall k \in 0 \dots n-1 \quad Sem'(v_{par_k}) = \begin{cases} \top & \text{si } v_{par_k} \text{ s'échappe par } m' \\ \perp & \text{sinon} \end{cases}$$

Le point de départ de l'analyse intra-méthode pour une méthode m se base sur sa propre signature d'échappement Sem :

$in(p_0) = ((v_{par_0}, \dots, v_{par_n-1}, v_\epsilon, \dots, v_\epsilon), \epsilon)$

15 tel que $esc(v_{par_k}) = Sem(v_{par_k})$ pour $k = 0 \dots n - 1$ et $esc(v_\epsilon) = \perp$

avec $v_{par_k} = \{r_{par_k}\}$ et $v_\epsilon = \{\}$

où r_{par_k} symbolise la référence passée en argument qui est encapsulée dans la structure de valeur afin d'effectuer l'analyse d'alias sur les registres. Au départ, chacune de ses références symboliques sur les paramètres est ajoutée à la

20 liste L . À la fin d'une passe d'analyse intra-méthode, les paramètres restant dans la liste L servent à mettre à jour la nouvelle signature. Lors d'un appel de méthode, on applique une signature d'échappement sur les arguments de la pile. Lorsque l'on connaît avec exactitude la méthode appelée (`invokespecial` et `invokestatic`) la signature calculée Sem' est appliquée sur les éléments de la

25 pile. Dans le cas des méthodes virtuelles (`invokevirtual` et `invokeinterface`),

on applique la signature unifiée notée $[Se]m''$ calculées depuis les signatures des méthodes m'' appelables telle que :

$$[Se]m'(v_{par_k}) = \bigcup_{m'' \text{ appelables}} Sem''(v_{par_k})$$

- 5 On va maintenant définir la signature d'échappement sur les connexions entre les paramètres d'appels. Cette signature sur les connexions entre les paramètres étend la signature précédente sur l'échappement des paramètres. Cette signature expose en dehors d'une méthode m les connexions entre les paramètres d'appels établies pendant l'analyse intra-méthode :

10

$$\forall k,j \in 0 \dots n-1 \quad Sem'(G(v_{par_k}, v_{par_j})) = \begin{cases} \top & \text{si il existe une connexion entre } v_{par_k} \text{ et } v_{par_j} \\ \perp & \text{sinon} \end{cases}$$

Le point de départ de l'analyse intra-méthode devient :

$$in(p_0) = ((v_{par_0}, \dots, v_{par_n-1}, v_\epsilon, \dots, v_\epsilon), \epsilon)$$

tel que $esc(v_{par_k}) = Sem(v_{par_k})$ pour $k,j = 0 \dots n-1$ et $esc(v_\epsilon) = \perp$

- 15 avec $v_{par_k} = \{r_{par_k}\}$, $v_\epsilon = \{\}$ et $G(r_{par_k}) = \{r_{par_j}\}$ lorsque

$$Sem(G(v_{par_k}, v_{par_j})) = \top$$

- 20 La figure 4 illustre un exemple de code Java dans lequel l'allocation au point 1. est échappée et dont l'allocation au point 2. est non échappée. Les figures 5 et 6 illustrent l'utilisation des règles d'analyse statique, règles décrites en liaison avec la figure 3, sur chaque instruction des méthodes du pseudo-code Java associé.

- La signature Se de la méthode $id()LC$ prise en entrée est : $\perp () \perp$ sans connexion. A la sortie, la signature Se de $id()LC$ vaut $\perp () \perp$ avec une connexion de r_{return} vers r_{this} . La colonne de droite indique l'évolution de la liste L et du graphe de connexion G .
- 25

La signature Se de la méthode `factory()`LC prise en entrée est : $\perp ()\perp$ sans connexion. L'analyse de la méthode `factory()`LC fait appel à la signature Se de `id()`LC. La signature Se de sortie de la méthode `factory()`LC vaut : $\perp ()\perp$ sans connexion, l'allocation du point p_4 est non-échappée et celle du point p_0 est
 5 échappée. En effet, au moment de l'instruction `areturn` lors de l'appel à la méthode `id()`LC au point p_{10} , la référence $r_{\text{invoke_p10}}$ s'échappe et par atteignabilité, $G(\text{invoke_p10}) = \{r_{p0}\}$ fait échapper r_{p0} .

Suite à l'analyse d'échappement, des signatures d'échappement sont ainsi
 10 obtenues pour au moins certaines méthodes à l'étape 202.

Comme détaillé dans les exemples ci-dessus, pour une méthode prédéterminée, les signatures d'échappement comprennent avantageusement des informations d'échappement concernant différents paramètres: ses arguments, son objet courant ou son objet renvoyé.

15 Une signature d'échappement peut ainsi comprendre les informations suivantes :

- des définitions d'échappement définissant si un objet est capturé par la méthode pour tous ses paramètres ;
- une référence de liaison mémoire définissant si, pour chaque
 20 combinaison de deux paramètres définis dans la signature, l'un fait référence à l'autre dans le programme. Ces informations permettent de déterminer le cas d'échappement suivant : si un objet est référencé par un objet s'échappant, alors il s'échappe également.

Les appels de méthodes virtuelles peuvent rendre la prédiction effectuée
 25 durant l'analyse statique d'échappement peu précise. Pour remédier à cela, on a généré pour chaque méthode la signature d'échappement directe Se_m et la signature d'échappement unifiée $[Se]_m$.

La signature d'échappement directe concerne une trace d'exécution effectuée durant l'analyse statique: cette signature pourra être utilisée par le

second dispositif DCTL si le type de l'objet est exact. Le type de l'objet est exact, si au cours de sa durée de vie, son type dynamique est égal à son type déclaré statistiquement.

La signature d'échappement unifiée réalise une prédiction au pire cas, par
5 unification de différentes traces d'exécution possibles : cette signature d'échappement pourra être utilisée par le second dispositif DCTL si, lors d'un appel de méthode, le type d'un objet a varié depuis sa création. La signature d'échappement unifiée est par exemple générée en réalisant l'unification des signatures d'échappement directes pour l'ensemble des méthodes appelables en ce
10 point du programme. Ainsi, même si la signature d'échappement unifiée ne correspond pas précisément aux conditions d'exécution du programme, ses indications d'échappement sont suffisamment approchées pour ne pas déterminer un objet capturé de façon erronée. La signature d'échappement unifiée est générée en parcourant l'arborescence d'héritage et la surcharge de méthode. Bien entendu,
15 l'utilisation de la signature d'échappement unifiée permet de déduire un nombre plus réduit d'objets capturés par rapport à l'utilisation de la signature d'échappement directe. Lors de l'exécution, le dispositif DCTL utilisera soit l'une ou l'autre des signatures d'échappement en fonction des appels statiques déclarés dans le pseudo-code ou en fonction d'informations de type obtenues d'un vérifieur
20 de pseudo-code, comme détaillé ultérieurement.

L'homme du métier saura limiter le nombre de méthodes à optimiser pour conserver un programme à charger de taille raisonnable. Ainsi, l'homme du métier pourra réserver la génération de signatures d'échappement aux méthodes présentant les meilleures perspectives d'optimisation lors de l'exécution.

25

L'étape 203 comprend un chargement du programme et des signatures d'échappement dans le dispositif de traitement DCTL, une analyse statique partielle d'échappement conduisant à la génération de signatures d'échappement de test et une vérification de cohérence entre les signatures d'échappement
30 chargées et les signatures d'échappement de test. L'analyse statique partielle

d'échappement utilise les mêmes règles d'analyse d'échappement que celles utilisées dans l'analyse statique du dispositif DTA.

Comme illustré à la figure 7, pour effectuer la vérification de cohérence d'échappement, les signatures d'échappement chargées sont utilisées comme
5 hypothèses pour une analyse statique d'échappement intra-méthode à l'étape 204.

Lors de l'analyse intra-méthode de la méthode m , sa signature chargée est utilisée comme sa signature en entrée $in(Sem)$ est prise en entrée et la signature $out(Sem)=Stm$ est obtenue en sortie comme signature de test. Lors d'un appel à la méthode m' , la signature Sem' chargée est utilisée pour poursuivre l'analyse
10 intra-méthode sur m .

On compare alors à l'étape 205 la signature d'échappement chargée à la signature d'échappement de test obtenue. Si l'ensemble des échappements définis dans la signature chargée est inclus dans l'ensemble des échappements de définis dans la signature de test Stm , on considère qu'il y a cohérence entre la signature
15 d'échappement chargée et la signature d'échappement de test : l'utilisation de la signature d'échappement chargée ne risque pas de déterminer de façon erronée un objet comme capturé par la méthode. On peut ainsi vérifier l'intégrité du programme et des signatures chargées.

Comme seule l'analyse intra-méthode est effectuée par le dispositif DCTL,
20 la quantité de ressources dont il doit disposer pour mettre en œuvre l'analyse statique partielle et la vérification de cohérence d'échappement peut être limitée. L'intégrité du programme peut donc être vérifiée dans un temps raisonnable même avec un dispositif à ressources limitées, tel qu'une carte à puce. Le dispositif de traitement DTA présente ainsi avantageusement des capacités de
25 traitement nettement supérieures à celles du dispositif de traitement DCTL.

Des signatures directes concernant les méthodes du programme peuvent être générées dans le dispositif DTCL par un vérifieur de pseudo-code binaire disponible le cas échéant. La vérification peut alors être effectuée en prenant en compte soit la signature d'échappement directe chargée, soit la signature
30 d'échappement unifiée chargée.

Lorsqu'une incohérence est détectée lors de la vérification, on peut prévoir que la machine virtuelle bloque la poursuite du chargement du programme, bloque l'exécution ultérieure du programme ou bloque la réalisation d'une optimisation du programme basée sur les signatures d'échappement. On peut également
5 envisager de limiter l'optimisation aux méthodes pour lesquelles la vérification de cohérence a donné un résultat favorable.

Bien qu'il soit possible de réaliser la vérification de cohérence d'échappement lors de l'exécution du programme, on privilégiera en pratique une vérification de cohérence d'échappement réalisée durant ou juste après le
10 chargement du programme dans le dispositif de traitement DCTL.

Bien qu'on ait représenté un chargement du programme depuis le dispositif de traitement réalisant la première analyse statique d'échappement, le chargement sera généralement effectué depuis une autre machine sur une ligne de fabrication.
15

A l'issue du procédé selon l'invention, la machine virtuelle du dispositif de traitement du dispositif DCTL exploite les signatures d'échappement chargées afin de déterminer des optimisations de l'exécution du programme.

La machine virtuelle peut notamment déterminer, sur la base des
20 signatures d'échappement chargées, les objets capturés qui pourront être gérés en mémoire avec une fonction différente de la gestion par ramasse-miettes de la machine virtuelle. Afin d'alléger la gestion mémoire, on peut notamment prévoir de gérer ces objets par allocation en pile. L'allocation et la désallocation des objets capturés en pile suit la pile des méthodes d'appel. L'allocation de chaque
25 objet capturé se fait en sommet de pile. La désallocation est effectuée de façon globale à la sortie de la méthode dans laquelle les objets capturés ont été alloués.

La machine virtuelle peut encore être utilisée pour l'élimination de synchronisations à partir des signatures d'échappement. Un objet capturé par une méthode étant accédé par un unique processus, il ne nécessite alors pas de
30 mécanismes de synchronisation d'accès (gestion de verrous et de mécanismes d'exclusion mutuelle). L'élimination de synchronisation élimine les éléments de

synchronisation inutiles afin d'optimiser la gestion de l'ordonnancement et de la concurrence d'accès dans le dispositif DCTL.

La machine virtuelle réalise l'exécution du programme en prenant en
5 compte les optimisations déterminées précédemment.

REVENDICATIONS

1. Procédé d'analyse d'échappement, caractérisé en ce qu'il comprend les étapes consistant à :
 - 5 - dans un premier dispositif de traitement, réaliser une analyse statique d'échappement sur un programme en langage intermédiaire d'exécution pour machine virtuelle à pile (201) et générer des signatures d'échappement associées à des méthodes respectives du programme en fonction de règles d'échappement
10 prédéterminées (202) ;
 - dans un second dispositif de traitement muni d'une machine virtuelle à pile :
 - charger les signatures d'échappement générées et le
 programme (203);
 - 15 - réaliser une analyse statique d'échappement partielle (204)
 basée sur les signatures chargées et générer, en fonction
 desdites règles d'échappement prédéterminées, des
 signatures d'échappement de test associées aux méthodes
 munies d'une signature d'échappement ;
 - 20 - pour une méthode donnée, vérifier (205) la cohérence
 entre sa signature d'échappement de test et sa signature
 d'échappement chargée.
2. Procédé selon la revendication 1, dans lequel le deuxième dispositif de
25 traitement utilisé est un dispositif à capacité de traitement limitée.
3. Procédé selon la revendication 1 ou 2, dans lequel une signature
 d'échappement générée pour une méthode comprend des informations
 associées à ses paramètres tels que ses arguments, son objet courant et/ou
30 son objet de retour.

4. Procédé selon l'une quelconque des revendications précédentes, dans lequel l'analyse statique d'échappement partielle et la vérification de cohérence sont réalisées après le chargement du programme dans le second dispositif de traitement.

5

5. Procédé selon l'une quelconque des revendications précédentes, dans lequel le programme en langage intermédiaire d'exécution est sous forme de pseudo-code binaire Java ou CLI.

1/3

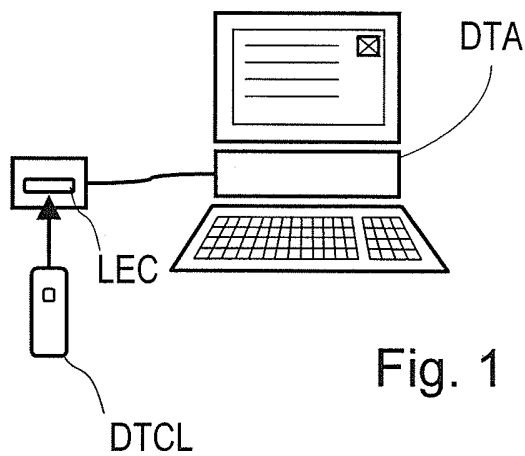


Fig. 2

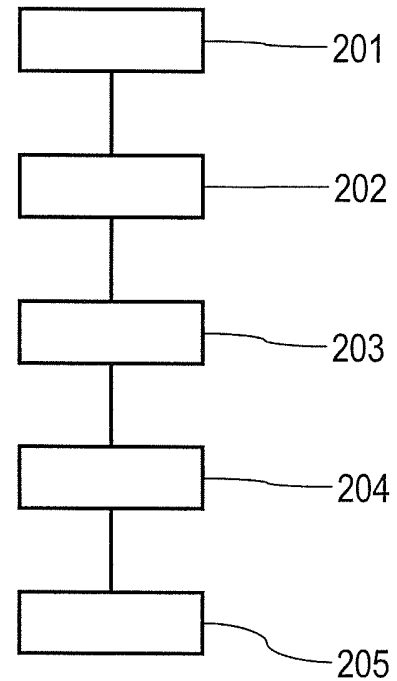


Fig. 3

athrow		$(R, P.v) \Rightarrow (R, P)$ and $v = T$
putstatic	:	$(R, P.v_{obj}) \Rightarrow (R, P)$ and $v_{obj} = T$
putfield	:	$(R, P.v_{obj_1}.v_{obj_2}) \Rightarrow (R, P)$ and $G(v_{obj_1}) \cup \{v_{obj_2}\}$ if $v_{obj_1} = T$ then $esc(v_{obj_2}) = T$
getfield	:	$(R, P.v_{obj_1}.) \Rightarrow (R, P.G(v_{obj_1}))$
aastore	:	$(R, P.v_{array}.i.v_{obj}) \Rightarrow (R, P)$ and $G(v_{array}) \cup \{v_{obj}\}$ if $v_{array} = T$ then $esc(v_{obj}) = T$
aaload	:	$(R, P.v_{array}.i) \Rightarrow (R, P.G(v_{array}))$
invoke m'		$(R, P.v_{this}.v_{arg_1} \dots v_{arg_n}) \Rightarrow (R, P.v_{invoke})$ with $\forall$ parameter par_k, par_j , if $\exists$ one method m'' invocable as m' such that: - $Sem''(v_{par_k}) = T$ then $esc(v_{arg_k}) = T$ - $Sem''(G(v_{par_k}, v_{par_j})) = T$ then $G(v_{arg_k}) \cup \{v_{arg_j}\}$
areturn		$(R, P.v) \Rightarrow (R, \epsilon)$ and $G(v_{areturn}) \cup \{v\}$ with $\forall r \in v$, if r is not an argument, then $esc(v) = T$

Fig. 4

```
class C {  
  
    public C factory()  
    {  
        C c = new C();    // 1. escaped allocation  
        new C().id();      // 2. captured allocation  
        return c.id();  
    }  
  
    private C id()  
    {  
        return this;  
    }  
}
```

Fig. 5

.method private id()LC;		Registre R ₀	Pile P ₀	
		{r _{this} }		L={r _{this} , r _{areturn} }
p ₀	aload_0	{r _{this} }	{r _{this} }	
p ₁	areturn	{r _{this} }		G(r _{areturn})={r _{this} }
				L={r _{this} , r _{areturn} }
.end method				

.method public factory()LC;		Registre R ₀	Registre R ₁	Pile P ₀	Pil e P ₁	
		{rthis}	{}			L={rthis, rreturn}
p ₀	new C	{rthis}	{}	{rp ₀ }		L={rthis, rreturn, rp ₀ }
p ₁	dup	{rthis}	{}	{rp ₀ }	{rp ₀ }	
p ₂	invokespecial C/<init>()V	{rthis}	{}	{rp ₀ }		
p ₃	astore_1	{rthis}	{rp ₀ }			
p ₄	new C	{rthis}	{rp ₀ }	{rp ₄ }		L={rthis, rreturn, rp ₀ , rp ₄ }
p ₅	dup	{rthis}	{rp ₀ }	{rp ₄ }	{rp ₄ }	
p ₆	invokespecial C/<init>()V	{rthis}	{rp ₀ }	{rp ₄ }		
p ₇	invokespecial C/id()LC;	{rthis}	{rp ₀ }	{rinvoke _p ₇ }		G(rinvoke_p ₇)={rp ₄ }
p ₈	pop	{rthis}	{rp ₀ }			
p ₉	aload_1	{rthis}	{rp ₀ }	{rp ₀ }		
p ₁₀	invokespecial C/id()LC;	{rthis}	{rp ₀ }	{rinvoke _p ₁₀ }		G(rinvoke_p ₁₀)={rp ₀ }
p ₁₁	areturn	{rthis}	{rp ₀ }			L={rthis, rreturn, rp ₄ }
						L={rthis, rreturn, rp ₄ }
.end method						

Fig. 6

Fig. 7

