Consolidation de correspondances

Sommaire

11.1 De l'intérêt de la consolidation de correspondances

La recherche de facteurs répétés de sous-arbres frères telle que présentée dans le chapitre précédent montre en pratique ses limites, même avec l'usage de certains profils d'abstraction. Si certains profils permettent de gérer des cas d'obfuscation comme la réécriture d'expressions (abstraction des petits sous-arbres) ou la modification de structures de contrôle (suppression des nœuds représentant ces structures), d'autres opérations d'éditions conduisent à l'obtention de valeurs de hachage différentes pour un sous-arbre et son clone obfusqué.

11.1.1 Ajout, suppression ou transposition de code

En particulier, l'ajout ou la suppression de code inutile n'est pas systématiquement décelable lors d'une étape de normalisation de l'arbre de syntaxe. Il en est de même pour certaines opérations de transposition de code. Ces opérations conduisent à l'ajout, la suppression ou le déplacement de sous-arbres de volume plus ou moins important.

Ajout/suppression de code Ainsi, par exemple, pour une fratrie de sous-arbres $C_1C_2\cdots C_n$ et son homologue clone $C'_1C'_2\cdots C'_{i-1}DC'_{i+1}\cdots C'_n$ auquel est ajouté un sous-arbre D, deux facteurs répétés distincts $\{C_1\cdots C_{i-1}, C'_1\cdots C'_{i-1}\}$ et $\{C_{i+1}\cdots C_n, C'_{i+1}\cdots C'_n\}$ seront relevés. Il apparaîtrait plus légitime, pour un évaluateur humain, de consolider ces deux correspondances en une seule en mentionnant l'ajout du sous-arbre D si celui-ci est de volume négligeable comparé à celui de l'ensemble de la fratrie.

Transposition de code Considérons maintenant le cas de la structure conditionnelle basée sur l'arbre de syntaxe $A = \text{si}(\text{cond}, \text{alors}, \text{sinon})$. Une version obfusquée A' est réalisée en négativant la condition et intervertissant les sous-arbres alors et sinon : $A' = \text{si}(\neg\text{cond}, \text{sinon}, \text{alors})$. Même l'utilisation d'un profil d'abstraction éliminant les structures conditionnelles ($\text{abstr}(A) = (\text{cond}, \text{alors}, \text{sinon}$, $\text{abstr}(A') = (\neg\text{cond}, \text{sinon}, \text{alors})$ ne permet pas de consolider les sous-arbres alors et sinon en une unique correspondance à cause de la transposition des deux sous-arbres et de la négation de la condition.

11.1.2 Factorisation ou développement de fonctions

Un ensemble d'instructions inter-dépendantes d'une fonction peut être externalisée dans une nouvelle fonction qui sera appelée depuis la fonction source. L'externalisation (ou factorisation) se matérialise sur l'arbre de syntaxe par la migration des instructions de la fonction source vers la fonction externalisée avec l'introduction de paramètres de fonction correspondant aux

variables locales utilisées dans la fonction source ; l'ajout d'instructions de retour est également probable.

L'opération réciproque de développement consiste à remplacer l'appel d'une fonction par le corps de ladite fonction. Le corps de la fonction nécessite généralement quelques adaptations au contexte local : changement de noms de variables locales, remplacement des instructions de retour par des affectations, remplacement des identificateurs des paramètres par leur valeur d'argument (cf figure 7.1)... Ces petites modifications ne permettent souvent pas d'obtenir une correspondance exacte entre le corps développé et le corps original : des fossés de non-correspondance sont présents. Il s'agit de regrouper ces correspondances séparées par des fossés en une unique correspondance.

11.1.3 Travaux antérieurs

Des méthodes d'extension de correspondances ont déjà été proposées pour le raccordement de facteurs de séquences de lexèmes : celles-ci ont été abordées brièvement en 5.4.3.

L'extension de correspondances sur des arbres de syntaxe est quant à elle un sujet qui été peu exploré. On pourra citer la méthode de propagation de correspondances utilisée par CloneDr [91]. Celle-ci consiste, lorsque deux arbres (ou chaînes d'arbres frères) ont été reportés comme similaires, à tenter de prolonger systématiquement la correspondance à leurs parents. Les sous-arbres ayant pour racine respective chacun des deux parents sont donc comparés en utilisant une méthode d'alignement sur arbres comme celles décrites en section 5.5. Si cette approche permet la détection en tant que clones de deux arbres possédant au moins un sous-arbre identique avec des modifications sur les autres sous-arbres frères, plusieurs niveaux de propagation pourraient être nécessaires pour permettre la mise en valeur de certains clones. D'autre part, le coût de détermination de distance d'édition entre deux arbres peut devenir prohibitif pour de grands sous-arbres.

Cobena [42] a également utilisé des méthodes de propagation à des nœuds parent pour la recherche de paires de sous-arbres correspondants dans des arbres de documents XML. La problématique demeure néanmoins assez différente de la notre car consistant à déterminer un delta entre différentes versions d'un même document XML, chaque sous-arbre ne pouvant correspondre au plus qu'avec un seul sous-arbre de l'autre version. Dans le cadre de la recherche de correspondances sur des sous-arbres de syntaxe provenant d'analyse de code source, l'utilisation de clones chevauchants n'est pas sans intérêt, une même zone de code pouvant être dupliquée au sein d'un même projet, voire d'une même unité de compilation.

11.1.4 Aperçu de la méthode de consolidation

Nous proposons dans ce chapitre une heuristique de consolidation de paires de correspondances en s'aidant de l'arbre de syntaxe des unités étudiées. Étant donné un arbre de syntaxe requête et une base d'arbres indexés, nous recherchons dans un premier temps les classes d'équivalence de sous-arbres, selon un profil d'abstraction donné, contenant au moins un exemplaire d'un sous-arbre de l'arbre requête. Nous en déduisons l'ensemble des paires de sous-arbres similaires dont un exemplaire provient de l'arbre requête et l'autre d'un arbre de la base. Ces

paires constituent des germes qui serviront de base à l'extension des correspondances. L'heuristique de consolidation de correspondances utilise alors deux étapes principales. L'étape de fusion consiste à regrouper les correspondances dont l'exemplaire sur l'arbre requête est proche. La seconde, l'étape de propagation, consiste à étendre une correspondance couvrant un volume important d'une fratrie de sous-arbres d'un arbre de la base à son sur-arbre parent. Ces deux étapes sont menées itérativement depuis les germes de forte profondeur des arbres de la base vers les germes de faible profondeur : elles permettent d'agglutiner des germes au sein de correspondances de volume croissant. Les étapes de fusion et propagation permettent non seulement la consolidation de sous-arbres frères, ce qui était déjà proposé d'une certaine manière par [91] ou en 10.4.4 lorsque les sous-arbres étaient consécutifs dans la fratrie, mais également le regroupement de sous-arbres cousins plus ou moins éloignés.

Nous nous intéressons ensuite à la consolidation des correspondances à travers les graphes d'appels afin de gérer des opérations d'obfuscation par développement ou factorisation de fonctions. Ainsi, si une correspondance occupe un volume conséquent d'une fonction, celle-ci pourra être propagée à l'ensemble de la fonction et par la suite à ses fonctions appelantes.

11.2 Extension à travers les arbres de syntaxe

11.2.1 Quelques définitions préalables

Définition 11.1. *(Correspondance élémentaire)* Une correspondance élémentaire (U, V) entre un arbre requête A et un arbre B de la base $\mathcal{B}$ met en relation une chaîne de sous-arbres (consécutifs) de la même fratrie de A (U) avec une chaîne de sous-arbres de la même fratrie de B (V). U et V sont égales selon un profil d'abstraction p donné.

Définition 11.2. *(Correspondance consolidée)* Une correspondance consolidée (U, V) entre un arbre requête A et un arbre B de la base $\mathcal{B}$ met en relation un ensemble de sous-arbres d'une même fratrie de A (U) avec un ensemble de sous-arbres d'une même fratrie de B (V). Cette correspondance est caractérisée par les propriétés suivantes :

- L'opération de consolidation ayant conduit à l'obtention de la correspondance.
- Les correspondances C (correspondance consolidée ou élémentaire) ayant servi de base à la création de la correspondance à l'aide de l'opération de consolidation.
- Le volume de U et de V ($\mathcal{V}(U)$ et $\mathcal{V}(V)$)[1].

Nous notons qu'il est possible, à partir d'une correspondance consolidée racine, d'en déduire un arbre de consolidation, chaque correspondance ayant pour correspondances enfants les correspondances C ayant été utilisées pour sa création. Les correspondances feuilles de l'arbre sont les correspondances élémentaires (ou germes).

Chaque correspondance (U, V) est, pour les besoins de l'algorithme, rattachée au nœud parent de V dans la base. Ceci nous permet de connaître les correspondances auxquelles participent les sous-arbres enfants d'un sous-arbre de la base, un sous-arbre enfant pouvant participer à plusieurs correspondances liées à des sous-arbres distincts de l'arbre requête mais présentant une similarité entre-eux.

[1]Dans un premier temps, à des fins de simplification, nous confondons le nombre de nœuds de l'arbre et son volume.

La connaissance des volumes des sous-arbres manipulés sur l'arbre de requête et chacun des sous-arbres de la base est nécessaire afin de tester les conditions des opérations de fusion et propagation. Les volumes de tous les sous-arbres de l'arbre requête sont calculables en temps linéaire si ils s'assimilent au nombre de nœuds. Ceux des sous-arbres de la base peuvent être enregistrés lors de leur indexation.

11.2.2 Récupération des paires de correspondances

Préalablement à l'extension, il est nécessaire de déterminer l'ensemble des correspondances élémentaires (germes) qui seront utilisées pour la constitution de correspondances consolidées. Ainsi par un parcours en largeur, nous examinons chacun des sous-arbres $A[t]$ de l'arbre requête A afin de déterminer s'il existe une classe d'équivalence sur la base avec le profil d'abstraction considéré correspondant à $A[t]$. On se reportera au chapitre 10 pour plus de détails sur cette opération. Nous obtenons pour chaque sous-arbre $A[t]$ l'ensemble c des sous-arbres de la classe d'équivalence correspondante sur la base. Nous notons que si un autre arbre β appartient à c, ses sous-arbres enfants appartiennent aux même classes d'équivalence que les enfants $A[t]_i$ de $A[t]$ sans toutefois être référencés explicitement dans ces classes.

Le processus de consolidation utilise des 2-correspondances et non des groupes de cardinalité supérieure à 2 : à partir d'une k-correspondance comportant k' composantes sur A ($k' \geq 1$), nous extrayons les $k'(k-k')$ 2-correspondances composées exactement d'une composante de A.

11.2.3 Fusion

Objectif

L'opération de fusion de correspondances consiste à regrouper plusieurs correspondances dont les composantes sur la base sont frères et dont les composantes sur l'arbre requête sont suffisamment proches. Son objectif principal vise à rassembler dans l'arbre requête, que l'on suppose être une version obfusquée d'une portion d'arbre de la base, des sous-arbres qui auraient été frères (dans la base) mais dispersés dans l'arbre requête. Cette situation peut être rencontrée, par exemple, dans un cas d'obfuscation où une portion de code serait extraite d'une structure de contrôle, et donc séparée de ses frères, pour être remontée dans l'arbre de syntaxe. Pour la fusion, nous utilisons une heuristique gloutonne avec validation.

Création d'une correspondance par fusion

Lorsque k correspondances $(U_1, V_1), (U_2, V_2), \cdots, (U_k, V_k)$ sont fusionnées, une nouvelle correspondance (α, β) est créée. β est constituée de l'ensemble des composantes sur la base : $\{V_1, V_2, \cdots, V_k\}$. Nous distinguons deux types de fusion :

1. La fusion entre composantes frères et de même ordonnancement dans l'arbre requête et l'arbre de la base. Un ensemble de correspondances $(U_1, V_1), (U_2, V_2), \cdots, (U_k, V_k)$ (triées par position de leur composante sur A) satisfait cette condition ssi pour tout j, U_j et U_{j+1} sont contigüs (chaînes ou arbres unitaires) ainsi que leur homologues sur la base V_j et V_{j+1}. Une nouvelle correspondance $(U_1U_2 \cdots U_k, V_1V_2 \cdots V_k)$ est alors créée.
2. La fusion entre composantes non-frères ou d'ordonnancement différent. Dans ce cas, pour les composantes sur l'arbre requête, nous recherchons leur plus petit ancêtre commun

(Lowest Common Ancestor : LCA) sur l'arbre requête A. Il s'agit concrètement de déterminer le plus petit sous-arbre $L = lca(\{U_1, U_2, \cdots, U_k\})$ de A contenant $U_1, U_2, \cdots, U_k$. Une nouvelle correspondance $(U = L, V = \{B_1, B_2, \cdots, B_k\})$ est alors créée issue de la fusion des correspondances. Pour pouvoir être fusionnées, les correspondances doivent concerner des composantes de la base (composantes frères) totalement distinctes[2].

Calcul du *LCA* Le calcul du LCA de deux nœuds s'effectue naïvement par récupération du premier nœud commun sur leurs branches ascendantes menant à la racine en temps linéaire en la profondeur cumulée des deux nœuds ($\Theta(h)$ pour un arbre de hauteur h). De nombreux calculs de LCA étant requis lors des opérations de fusion, un pré-traitement de l'arbre afin de pouvoir déterminer le LCA en $o(h)$ est envisageable. En utilisant la méthode de Berkman et Vishkin [5], l'arbre requête fait d'abord l'objet d'un parcours en profondeur. Nous obtenons alors la séquence des nœuds parcourus (deux fois pour chaque nœud) associés à leur profondeur dans l'arbre : trouver le LCA de deux nœuds équivaut alors à déterminer le nœud de plus faible profondeur entre les deux nœuds de la séquence de parcours. Ceci revient à trouver l'entier minimal sur un intervalle d'une suite d'entiers (*Range Minimum Query* : RMQ). Le problème du RMQ est résolvable en temps constant après un pré-traitement linéaire. Fisher et Heun proposent une implantation [9] nécessitant seulement $2n + o(n)$ bits pour le pré-traitement d'une suite de n entiers. Le LCA de k nœuds peut être calculé récursivement en déterminant le LCA de $k-1$ de ces nœuds avec le nœud restant : après pré-traitement, cette opération est réalisée en $O(k)$.

Problématique Nous souhaitons déterminer sur les κ 2-correspondances dont les composantes de base sont frères une partition de ces 2-correspondances dont chaque groupe représenterait une correspondance issue de la fusion de ses membres. Il s'agit de minimiser le nombre de groupes tout en optimisant le ratio de couverture du sous-arbre LCA de chaque groupe par ses composantes germes constituantes sur l'arbre requête. Nous notons ce ratio ρ ; en supposons des correspondances non-chevauchantes $(U_1, V_1), \cdots, (U_k, V_k)$ envisagées pour la fusion, nous avons $\rho = \frac{\mathcal{V}(U_1)+\cdots+\mathcal{V}(U_k)}{\mathcal{V}(\text{lca}(U_1,\cdots,U_k))}$. Les deux objectifs de minimisation des groupes et d'optimisation des ratios de couverture étant antagonistes, un compromis doit être réalisé : nous proposons à cet effet une heuristique de partition en deux étapes décrite ci-après.

Heuristique de sélection des correspondances à préfusionner Nous nous proposons de fusionner progressivement les κ correspondances par une heuristique gloutonne en commençant par les fusions de ratio de couverture ρ les plus élevés. Nous maintenons un tas comportant l'ensemble des 2-correspondances consolidées avec priorité aux correspondances de plus fort ratio de couverture ρ. Il est initialisé par les κ 2-correspondances germes frères sur un arbre de la base avec $\rho = 1$. À chaque itération, nous associons la correspondance c de valeur ρ maximale avec une autre correspondance du tas telle que la correspondance issue des deux précédentes soit de ratio de couverture ρ maximal. Ceci est accompli en calculant les ratios de couverture de c avec chacune des autres correspondance du tas. Les deux correspondances germes sont supprimées du tas et la nouvelle correspondance créée y est insérée. L'itération de ce processus $k-1$ fois conduirait à l'obtention d'une large correspondance issue de toutes les correspondances germes initiales et dont le LCA global représenterait un sous-arbre de l'arbre

[2]Il est possible qu'un nœud de la base soit en correspondance avec plusieurs nœuds de l'arbre requête : ceux-ci ne sont donc pas fusionnés en leur LCA

requête de volume potentiellement important. Le nombre de correspondances obtenues serait minimisé mais le ratio de couverture serait faible. Nous introduisons donc une condition de préfusion : deux correspondances peuvent être associées si la correspondance résultante est de ratio de couverture $\rho \geq t_m$ où t_m est un seuil fixé. La condition d'arrêt des itérations est caractérisée par l'inexistance de couple de correspondances du tas dont l'association serait de ratio de couverture $\rho \geq t_m$.

Validation des correspondances préfusionnées Chaque correspondance préfusionnée est de la forme $(\{U_1, U_2, \cdots\}, \{V_1, V_2, \cdots\})$ issue de la préfusion d'au moins deux correspondances (U_1, V_1) et (U_2, V_2). Nous avons $L = lca(U_1, U_2, \cdots)$ avec le ratio de couverture ρ associé. Si $\rho \geq t_M$, valeur seuil pour la fusion définitive (avec $t_M \geq t_m$), la correspondance préfusionnée est définitivement fusionnée. Si la correspondance préfusionnée ne satisfait pas la condition de fusion définitive, l'opération de préfusion la plus récente est annulée : nous obtenons les deux correspondances préalablement à la préfusion sur lesquelles nous testons la condition de fusion définitive. L'opération est récursivement répétée sur chaque correspondance issue d'une préfusion si celle-ci ne peut être fusionnée définitivement ou alors lorsqu'une correspondance simple non issue de préfusion est obtenue.

Intérêt de la préfusion L'utilisation d'un seuil de préfusion t_m plus faible que le seuil de fusion définitive t_M réside dans la possibilité de préfusionner dans l'arbre requête des structures de proximité importante dont le ratio de couverture par rapport au *lca* est trop faible pour atteindre t_M, mais qui pourraient, après plusieurs préfusions successives atteindre ce ratio. Considérons par exemple un sous-arbre $B = a(\alpha, \beta, \gamma)$ de la base avec $\mathcal{V}(\alpha) = \mathcal{V}(\beta) = \mathcal{V}(\gamma) = 1$. Imaginons une version obfusquée de cet arbre où chaque sous-arbre aurait été encapsulé dans une structure conditionnelle systématiquement exécutée (que nous noterons c) avec effet neutre sur le volume : $A = a(c(\alpha'), c(\beta'), c(\gamma'))$. Si $t_M > \frac{2}{3}$, la fusion directe des correspondances (α', α) et (β', β) de LCA B serait impossible ($\rho = \frac{2}{3} < t_M$) : cependant avec $t_m \leq \frac{2}{3}$, une préfusion serait possible. Après deux préfusions, nous obtenons $(A, \{\alpha, \beta, \gamma\})$ avec $\rho = 1$: la fusion peut être validée.

Choix du sur-arbre du *LCA* Afin de gérer certains cas d'obfuscation consistant à entourer un bloc de code par un ou plusieurs niveaux de structures inutiles (entourage par une structure conditionnelle invariablement vraie, une boucle d'itération unique, ...), plutôt que de choisir le LCA des composantes de l'arbre requête pour la fusion de correspondances, nous sélectionnons le parent direct ou indirect. Ainsi, si le ratio $\rho' = \frac{\mathcal{V}(L)}{\mathcal{V}(\mathcal{P}(L))}$ du volume de l'arbre L conservé par la fusion (initialement L est le LCA des composantes de l'arbre de requête) sur le volume de son sur-arbre parent est supérieur à un seuil $1 - \epsilon$, L est substitué par $\mathcal{P}(L)$. Le procédé est itéré jusqu'à ce que la condition de sélection du parent ne soit plus remplie.

11.2.4 Propagation

Une fois réalisées les fusions de correspondances associées à une fratrie de sous-arbres d'un arbre de la base se pose la question du devenir de ces correspondances fusionnées et de celles restantes n'ayant pas fait l'objet de fusion. Il peut être intéressant de les associer à leur grand parent sur l'arbre de la base afin de pouvoir les fusionner avec des correspondances tantes issues elle-mêmes de correspondances cousines. L'opération de propagation permet de *remonter* une correspondance dans l'arbre de syntaxe de la base.

Données : Correspondances à fusionner : $(U_1, V_1), (U_2, V_2), \cdots, (U_k, V_k)$
Données : t_m : ratio minimal de couverture pour la préfusion
Données : $t_M > t_m$: ratio minimal de couverture pour la fusion
Résultat : Ensemble des correspondances fusionnées R

début
  Initialisation de la file de priorité des paires de correspondances candidates à la fusion ;
  $F \longleftarrow \emptyset$;
  pour $(U_i, V_i) \in$ *correspondances germes considérées pour la fusion* **faire**
    $F \longleftarrow F + (A_i, B_i)$;
  Itérations de préfusion ;
  $G \longleftarrow \emptyset$ *(Correspondances non-associables)* ;
  tant que $F \neq \emptyset$ **faire**
    $c \longleftarrow \max_\rho(F)$;
    Initialisation de la correspondance à associer ;
    $c' \longleftarrow \emptyset$;
    pour $d \in F / d \neq c$ **faire**
      si $\rho(\text{fusion}(c, d)) > \rho(\text{fusion}(c, c')) > t_m$ **alors**
        $c' \longleftarrow d$
    si $c' \neq \emptyset$ **alors**
      $F \longleftarrow F - c - c' + \text{fusion}(c, c') + G$;
      $G \longleftarrow \emptyset$;
    sinon
      $G \longleftarrow G + c$;
  Validation des préfusions ;
  $F' \longleftarrow \emptyset$;
  tant que $F \neq \emptyset$ **faire**
    $c \longleftarrow F[1]$;
    $F \longleftarrow F - c$;
    si $\rho(c) \geq t_M$ **alors**
      $F' \longleftarrow F' + c$;
    sinon
      $(c_1, c_2) \longleftarrow$ défusion(c) ;
      $F \longleftarrow F + c_1 + c_2$;
  retourne *les correspondances fusionnées* F'
fin

Algorithme 6 : Heuristique gloutonne de fusion de correspondances

L'opération de propagation sur une correspondance (U, V) associée à son nœud parent de la base $\mathcal{P}(V)$ transforme ladite correspondance en $(U, \mathcal{P}(V))$ désormais associée à $\mathcal{P}(\mathcal{P}(V))$.

Pour chaque correspondance (U, V) associée au sous-arbre B de la base, la propagation est décidée si le ratio $\rho'' = \frac{\mathcal{V}(U)}{\mathcal{V}(\mathcal{P}(V))}$ est supérieur à un seuil de propagation t_p. Nous notons que si B concerne des enfants de la racine d'un arbre de la base, la correspondance est remontée à la racine et associée à une sur-racine virtuelle : celle-ci ne peut plus être propagée dans l'arbre de la base.

11.2.5 Algorithme général

Nous présentons maintenant le squelette de l'algorithme général de consolidation à travers les arbres de syntaxes (algorithme 7). Initialement les correspondances élémentaires sont trouvées et associées au parent de leur composante sur la base. Chaque parent de la base ayant au moins une correspondance pour enfant est ajouté dans une file de priorité permettant d'obtenir le plus petit sous-arbre de la base (en terme de nombre de nœuds) dont les correspondances enfants n'ont pas encore fait l'objet de procédé de fusion puis de propagation. À chaque itération, un sous-arbre de la base est ainsi sélectionné. Nous appliquons le processus de fusion sur ses correspondances enfant. Chaque correspondance enfant créée ou subsistant après la fusion est considérée pour son éventuelle propagation d'un niveau dans l'arbre de la base. Si la condition de propagation est remplie, la correspondance transformée est associée à son grand-parent de l'arbre de la base qui est ajouté, s'il n'était pas encore présent, dans la file de sélection des sous-arbres de la base. L'algorithme s'achève lorsque la file des sous-arbres de la base est soit vide, soit constituée uniquement de sur-racines virtuelles.

Les correspondances conservées sont les correspondances racines dans l'arbre des correspondances, i.e. les correspondances n'ayant pas été utilisées comme base à une opération de fusion ou de propagation.

11.2.6 Exemple

Nous traitons ici d'un petit exemple de consolidation de correspondances à travers une fonction en langage Java ayant pour fonction d'obtenir les caractères d'un fichier texte dans leur ordre inverse. Une version obfusquée est réalisée en changeant la structure de la fonction. Le bloc try-finally est supprimé avec l'entourage de l'instruction du finally par une structure conditionnelle complétée d'un bloc *sinon* inutile, tandis que, à un niveau supérieur, le code de lecture de fichier est entouré par une boucle inutile à itération unique. La version originale du code, la version obfusquée ainsi que les squelettes des arbres de syntaxe correspondants sont spécifiés en figure 11.1. À des fins de simplification, les déclarations initiales ainsi que l'instruction de retour sont ignorées. D'autre part, toujours pour simplifier, toute correspondance préfusionnée est automatiquement fusionnée définitivement : $t_m = t_M$.

Les correspondances élémentaires sont d'abord cherchées pour chaque sous-arbre de la fonction f de la base : nous trouvons (A', A) (lecture du fichier), (B', B) (fermeture du fichier) et (C', C) (inversion de la chaîne). Pour simplifier, nous convenons des volumes suivants pour les sous-arbres manipulés : $\mathcal{V}(A) = \mathcal{V}(A') = 2$, $\mathcal{V}(B) = \mathcal{V}(B') = 1$, $\mathcal{V}(C) = \mathcal{V}(C') = 2$ et $\mathcal{V}(\text{else}) = 2$. Nous examinons le plus petit sous-arbre de f (base) dont les enfants participent

Données : Correspondances élémentaires (U_i, V_i) associées à leur sous-arbre parent de la base $\mathcal{P}(V_i)$
Résultat : Correspondances consolidées racines R

début
  $F \longleftarrow \{\cdots \mathcal{P}(V_i) \cdots\}$;
  tant que $F \neq \emptyset$ **faire**
    $\beta \longleftarrow$ plus petit arbre de F ;
    $F \longleftarrow F - \{\beta\}$;
    $\text{corr}(\beta) = \{(U_1, V_1), (U_2, V_2), \cdots, (U_k, V_k)\} \longleftarrow$ correspondances telles que V_i soit un ensemble de sous-arbres enfants de β ;
    $\text{corr'}(\beta) = \{(U'_1, V'_1), (U'_2, V'_2), \cdots, (U'_{k'}, V'_{k'})\} \longleftarrow \text{fusion}(\text{corr}(\beta))$ $k' \leq k$;
    pour $(U'_i, V'_i) \in \text{corr'}(\beta)$ **faire**
      si V'_i *est propageable* $(\frac{\mathcal{V}(V'_i)}{\mathcal{V}(\mathcal{P}(V'_i))} \geq t_p)$ **alors**
        $(U'_i, V'_i) \longleftarrow (U'_i, \beta)$;
        si β *n'est pas racine d'un arbre* **alors**
          Association de (U'_i, β) avec $\mathcal{P}(\beta)$;
          $F \longleftarrow F \cup \{\mathcal{P}(\beta)\}$;
          $\lambda \longleftarrow$ faux ;
        sinon
          $\lambda \longleftarrow$ vrai ;
      sinon
        $\lambda \longleftarrow$ vrai ;
      si λ **alors**
        $R \longleftarrow R \cup \{(U'_i, V'_i)\}$;
  retourne R
fin

Algorithme 7 : Consolidation de correspondances par arbres de syntaxe

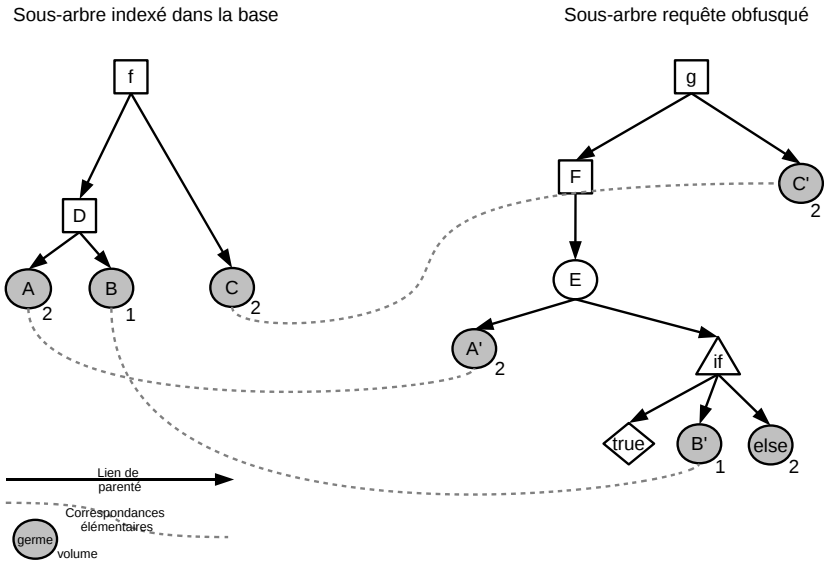
```
1  String reverseFile(String filename) throws
       IOException
   {
    StringBuilder sb = null;
    Reader r = null;
    try {
6     r = new FileReader(filename);
     sb = new StringBuilder();
     while (int read = r.read() != -1)
      sb.appendCodePoint(read);
    } finally
11  {
     r.close();
     System.err.println("Reading the file was
          successful");
    }
    for (int k=0; k < sb.length(); k++)
16  {
     // Swap the characters k and len-1-k
     char tmp = sb.charAt(k);
     sb.setCharAt(k, sb.charAt(sb.length()-1-k));
     sb.setCharAt(sb.length()-1-k, tmp);
21  }
    return sb.toString();
   }
```

(a) Fonction originale indexée dans la base

```
1  String reverseFileContent(String f) throws
       IOException
2  {
    StringBuilder sb = null;
    Reader reader = null;
    for (int i=0; i % 2 == 0; i+=1)
    {
7    reader = new FileReader(f);
     sb = new StringBuilder();
     while (int read = reader.read() != -1)
      sb.appendCodePoint(read);

12   if (i % 3 <= 1)
     {
      r.close();
      System.err.println("Success of the operation"
           );
     } else
17   {
      for (int j=0; j < sb.length(); j++)
       for (int k=0; k < j; k++)
       {
        int s = 0;
22      while (s == 0)
         s += 1;
        sb.setCharAt(k, '\0');
       }
     }
27  }
    {
     for (int j=0; j < sb.length(); k++)
     {
      char tmp = sb.charAt(j);
32    sb.setCharAt(j, sb.charAt(sb.length()-1-j));
      sb.setCharAt(sb.length()-1-j, tmp);
     }
    }
    return sb.toString();
37 }
```

(b) Fonction obfusquée requête



(c) Arbres de syntaxes

FIG. 11.1 – Un exemple de deux fonctions avec leurs squelettes d'arbres de syntaxe formant une correspondance approchée déterminée par consolidation

à au moins une correspondance : il s'agit de D. Pouvons nous fusionner les correspondances (A', A) et (B', B) ? Le LCA de A' et B' est E dans l'arbre requête. En supposant les volumes liés au nœud *if* et à la condition invariablement vrai nuls, $\rho = \frac{\mathcal{V}(A')+\mathcal{V}(B')}{\mathcal{V}(D)=\mathcal{V}(A')+\mathcal{V}(B')+\mathcal{V}(\text{else})} = \frac{3}{5}$. Selon la valeur du seuil de fusion, deux situations sont rencontrées :

1. Si $\rho \geq t_m$, la fusion est réalisée pour obtenir la correspondance $(E, \{A, B\})$. Toutefois comme $\frac{\mathcal{V}(E)}{\mathcal{V}(\mathcal{P}(E)=F)} = 1$, nous choisissons le parent du LCA E, soit F, d'où la correspondance $(F, \{A, B\})$. Celle-ci est ensuite propagée à $\mathcal{P}(\{A, B\})$, i.e. D. La correspondance devient (F, D). f possède donc deux correspondances sur ces enfants : la correspondance consolidée (F, D) et la correspondance élémentaire (C', C). Celles-ci sont fusionnées en $(lca(C', F) = g, \{D, C\})$ car $\rho = 1$: par propagation nous obtenons la correspondance (g, f).
2. Si $\rho < t_m$, la fusion n'est pas réalisée. (A, A') ou (B, B') peuvent néanmoins se voir propagés dans le sous-arbre f de la base si $\frac{\mathcal{V}(A)}{\mathcal{V}(D)} \geq t_p$ et $\frac{\mathcal{V}(B)}{\mathcal{V}(D)}$ respectivement. Si $\frac{1}{3} < t_p \leq \frac{2}{3}$, seule (A, A') est propagée pour devenir (D, A'). (D, A') et (C, C') ne peuvent ensuite pas être fusionnés car $\rho = \frac{\mathcal{V}(A')+\mathcal{V}(C')}{\mathcal{V}(lca(A',C')=g)} = \frac{4}{7} < \frac{3}{5} < t_m$.

Nous noterons que dans tous les cas, si $\mathcal{V}(\text{else})$ tend vers l'infini, aucune fusion sur A' et B' ne peut être réalisée : inonder le code source par du code inutile limite les possibilités de consolidation de correspondances. Toutefois, adopter un seuil de fusion t_m très bas peut conduire, dans certains conditions, à des fusions non désirées. Trouver un compromis sur les seuils t_m et t_p est donc délicat et nécessite des tests expérimentaux.

Il est possible d'introduire une externalisation du code d'inversion de la chaîne dans la version obfusquée de la fonction : ce code serait présent dans une fonction externe appelée depuis la fonction principale. Les graphes d'appels de la fonction originale et obfusquée sont alors nécessaires afin de consolider le code de la fonction principale avec celui externalisé (cf 11.3).

11.2.7 Complexité

La complexité spatiale du processus de consolidation est linéaire en nombre de nœuds de l'arbre requête (pour le stocker et conserver la table de détermination de *lca*) ainsi qu'en nombre de correspondances élémentaires. La complexité temporelle dépend directement du nombre de paires de correspondances candidates pour chaque processus de fusion, du nombre d'opérations de fusion réalisées ainsi que du nombre d'opérations de propagation.

Dans le pire des cas ($t_p = 0$), toutes les correspondances sont propagées à leur parent dans l'arbre de la base : une correspondance entre un sous-arbre de la base et un sous-arbre de l'arbre requête sera étendue à la racine de l'arbre concerné de la base. Le coût temporel est en $O(h)$ où h est la hauteur maximale d'un arbre de la base.

Concernant les opérations de fusion, nous devons considérer le nombre de correspondances $\gamma = |\text{corr}(B)|$ associée à un sous-arbre B de la base. Dans le pire des cas, les correspondances sont fusionnées dans l'ordre inverse de celui de leur présence dans la pile. Cela nécessite $O(\gamma^2)$ opérations de calcul de LCA et de ratio ρ de couverture pour chacune des paires testées

pour chaque préfusion réussie, soit globalement $O(\gamma^3)$. Dans le pire des cas, en supposant le nombre de duplications internes de chaque sous-arbre dans l'arbre de requête borné par d, jusqu'à dk correspondances peuvent être associées à un sous-arbre B d'arité k de la base (soit une complexité pour l'étape de fusion de $O((dk)^3)$). Le coût peut ainsi devenir important pour, par exemple, un bloc d'instructions de la base contenant de nombreuses instructions de squelette typique aussi présentes en grand nombre dans l'arbre de requête (e.g. instructions de déclaration, d'incrémentation...). Il peut donc être intéressant d'ignorer l'existence de tels sous-arbres fréquents pour le procédé de fusion ainsi que d'extension : les correspondances les concernant sont ignorées et la détermination de la métrique de volume est révisée en conséquence.

11.3 Extension à travers les graphes d'appels

L'extension des correspondances à travers les graphes d'appels afin de gérer les opérations de factorisation et/ou développement de fonctions est assez similaire à la procédure d'extension par les arbres de syntaxe. Il s'agit, à partir de correspondances infra-fonctionnelles, de réaliser des opérations de fusion de composantes sur le graphe d'appels requête et de propagation sur le graphe d'appels de la base.

Dans un premier temps, nous clarifions la notion de correspondances infra-fonctionnelles. La connaissance des graphes d'appels étant par nature indispensable pour la procédure d'extension, nous nous intéressons à leur obtention ainsi que leur conservation pour les projets de la base.

11.3.1 Correspondances infra-fonctionnelles

Après la consolidation des correspondances élémentaires par les arbres de syntaxe, nous obtenons un jeu de correspondances consolidées racines ne pouvant être ni fusionnées ni propagées. Ces correspondances peuvent mettre en relation, aussi bien pour la composante sur l'arbre de requête que celle sur la base, des sous-arbres infra-fonctionnels (appartenant à un arbre représentant une fonction) ou ultra-fonctionnels (représentant une structure contenant des fonctions telle qu'une unité de compilation ou une classe). Nous supposerons, dans un souci de simplification, qu'il n'existe pas d'implantations de fonctions imbriquées[3]. Si des fonctions imbriquées sont présentes, celles-ci peuvent être désencapsulées lors de l'étape de normalisation de l'arbre de syntaxe au prix de la communication de paramètres complémentaires pour la communication de variables de contexte local.

Lors du processus de consolidation via les arbres de syntaxe décrit dans la section précédente, nous conservons les correspondances portant sur des paires d'arbres infra-fonctionnels lorsqu'une fusion entraîne l'obtention d'une composante ultra-fonctionnelle (survenant par exemple pour deux unités de compilation similaires) ou lorsque ceci est le fait d'une propagation dans un arbre de la base (par exemple si l'arbre de la base est une unité de compilation

[3]Ceci est vrai pour certains langages procéduraux (C) et orientés objet (Java, C++, C#...) dans une certaine mesure (si l'on exclut par exemple les classes anonymes en Java). D'autres langages tels que les langages fonctionnels (Haskell, Caml...), des langages procéduraux (Fortran, Pascal...) et des langages orientés objet (Ada, Python, JavaScript...) supportent les fonctions imbriquées (*closures*).

ne contenant qu'une fonction similaire avec une fonction de l'arbre requête). Les correspondances infra-fonctionnelles sont, pour la consolidation par graphes d'appels, homologues aux correspondances élémentaires pour la consolidation par arbre de syntaxe.

Les opérations d'obfuscation par développement ou factorisation de fonctions sont susceptibles d'être réalisées sur plusieurs unités de compilation représentées par des arbres de syntaxe distincts. Les correspondances infra-fonctionnelles doivent donc préalablement être recherchées pour chacun des arbres de syntaxe du projet requête.

11.3.2 DAG d'appels

Comme explicité en section 3.4, l'obtention du graphe d'appels pour un projet nécessite la résolution des liens d'appels. Nous notons que cette résolution n'est pas toujours réalisable statiquement pour certains langages permettant l'usage de pointeurs de fonction ou de fonctions homonymes.

Nous calculons à partir du graphe d'appels le DAG d'appels (cf sous-section 3.4.4), graphe acyclique reliant les composantes fortement connexes du graphe d'appel. Chaque composante fortement connexe est associée à un volume de couverture, mesurant la quantité de code potentiellement accessible depuis une fonction de la composante fortement connexe.

11.3.3 Fusion et propagation des correspondances

Les processus de fusion et de propagation de correspondances décrits en 11.2.3 et en 11.2.4 pour les arbres de syntaxe sont appliqués à partir des correspondances infra-fonctionnelles avec l'aide des DAG d'appels du projet requête et des projets de la base. Quelques remarques doivent néanmoins être réalisées.

Fusion

Considérons deux correspondances (U_1, V_1) et (U_2, V_2) pour illustrer la fusion. Chacune des composantes, du projet requête et d'un des projets de la base, est représentée par un ensemble de structures infra-fonctionnelles et/ou de fonctions complètes. Fusionner les deux composantes du projet requête U_1 et U_2 est réalisé par le calcul du LCA des composantes U_1 et U_2 du projet requête ainsi que du ratio de couverture ρ.

Si les deux composantes U_1 et U_2 représentent des structures infra-fonctionnelles de la même fonction, leur LCA est calculé par l'utilisation de l'arbre de syntaxe de l'unité concernée. Dans le cas contraire, parmi U_1 et U_2, il existe au moins une fonction complète. En supposant U_1 une fonction complète, la métrique de volume utilisée correspond à son volume de couverture tel que défini 3.4.4, i.e. la somme des volumes propres des fonctions de sa clôture transitive. En revanche, si U_2 est une composante infra-fonctionnelle, son volume est défini par son volume propre, ne prenant donc pas en compte les fonctions appelées en son sein. Les LCA de U_1 et U_2 sont alors des fonctions complètes déterminées par le DAG d'appel. Il est utile de rappeler que le volume de couverture d'un ensemble de fonctions n'est pas égal à la somme des volumes de couverture de chacune des fonctions : il est donc nécessaire de conserver, pour chaque fonction, l'ensemble de sa clôture transitive afin de pouvoir calculer la couverture d'un ensemble de fonctions par la somme des volumes propres de l'union des clôtures transitives.

LCA dans un graphe acyclique

Il existe un unique LCA pour deux éléments a et b quelconques d'un arbre : il s'agit de l'ancêtre commun à a et b n'admettant pas parmi ses descendants un ancêtre commun à a et b. Pour un DAG, chaque paire d'éléments (a, b) peut admettre aucun, un ou plusieurs LCA. Calculer les LCA de (a, b), peut nécessiter, dans le pire des cas, de parcourir l'ensemble des prédécesseurs de a et de b en suivant jusqu'à $O(m)$ arêtes pour un DAG à n nœuds et m arêtes. Kowaluk et Lingas [13] proposent une méthode de détermination des LCA de toutes les paires de nœuds de complexité temporelle $\Theta(mn)$ et nécessitant $\Theta(n^2)$ en espace. Elle consiste à calculer, pour chaque nœud du DAG son ensemble de descendants puis l'ensemble des nœuds avec lequel il partage un ancêtre commun (l'union des ensemble des descendants de ses prédécesseurs directs) avec un pointeur vers cet ancêtre. Nous pouvons ensuite déduire toutes les paires de nœuds admettant au moins un LCA avec les pointeurs vers leurs LCA. Si le DAG considéré est dense, ce qui est rarement le cas pour les DAG d'appel, des méthodes plus rapides permettent le calcul d'un des LCA de chaque paire de nœuds (s'il existe). On peut ainsi rechercher [13] les témoins maximaux du produit des matrices booléennes en $O(n^{2.575})$ (complexité liée au produit matriciel) de la clôture transitive du graphe et de sa contraposée, les nœuds étant ordonnés selon un tri topologique.

La préfusion de composantes donne lieu à l'obtention d'autant de correspondances préfusionnées que de LCA, les correspondances de ratio de couverture inférieur au seuil de préfusion t_m étant ignorées.

Pour chaque LCA $\mathcal{L}$ sélectionné sur les composantes de la base pour la préfusion, nous vérifions si le choix d'un des parents du LCA ne modifie pas notoirement ρ : pour l'ensemble des parents $\mathcal{P}(\mathcal{L})$, nous sélectionnons le parent p tel que $\rho' = \frac{\mathcal{V}^+(\mathcal{L})}{\mathcal{V}^+(p)}$ soit maximal avec $\rho' > 1-\epsilon$.

Propagation

Lorsque toutes les correspondances associées à leur composante parent dans la base ont été soumises à l'heuristique de préfusion et de validation de fusion, il est nécessaire de déterminer si celles-ci seront propagées à leur parent. Si la composante de la base est infra-fonctionnelle, celle-ci est testée pour sa propagation à son unique parent. Dans le cas où la composante de la base est une fonction, plusieurs parents, qui sont les composantes connexes appelant cette fonction dans le DAG, peuvent coexister. La condition de propagation est donc testée pour chacun des parents.

La condition de propagation est validée si le ratio de couverture des composantes enfants de la base par rapport à la composante parent considérée est au moins égal au seuil t_p.

Algorithme général

L'algorithme général de consolidation à travers le DAG suit le même schéma directeur que celui de consolidation à travers les arbres de syntaxe individuels. Il s'agit de traiter tout d'abord les fonctions de projets de la base dont le volume de couverture de la composante connexe est le plus faible : nous débutons ainsi le traitement par les composantes fortement connexes feuilles du DAG. Pour chaque composante fortement connexe du DAG d'un projet de

la base, nous tentons de fusionner les correspondances impliquant une des composantes enfant (composante fortement connexe ou composante infra-fonctionnelle) d'une des fonctions de la composante. La fusion réalisée, nous tentons de propager les composantes vers les composantes connexes parents.

11.3.4 Exemple

Nous nous proposons d'appliquer l'heuristique de consolidation sur un petit exemple impliquant une opération d'obfuscation par développement de code. Nous considérons l'ensemble des fonctions originales suivantes indexées dans la base ($C_1, \cdots, C_5$ étant des morceaux de code sans sites d'appel, de volume 1 chacun) :

$$\begin{aligned} f_1 = &\ C_1@f_2C_2 \\ f_2 = &\ C_3@f_1C_4 \\ f_3 = &\ C_5@f_1 \end{aligned}$$

Ces fonctions constituent le DAG d'appels suivant contenant deux composantes fortement connexes :

$$\mathrm{DAG}\left(\{f_1, f_2, f_3\}\right) = \{f_3\} \rightarrow \{f_1, f_2\}$$

Nous avons $\mathcal{V}^+(\{f_1, f_2\}) = \mathcal{V}(C_1)+\mathcal{V}(C_2)+\mathcal{V}(C_3)+\mathcal{V}(C_4) = 4$ et $\mathcal{V}^+\left(\{f_3\}\right) = \mathcal{V}^+(\{f_1, f_2\})+\mathcal{V}(C_5) = 5$. Ces trois fonctions sont réécrites en une unique fonction f_q émulant manuellement une pile d'appels (code C_7 de volume 1) pour gérer la récursivité mutuelle de f_1 et f_2, les morceaux $C'_1, \cdots, C'_5$ étant déjà identifiés en tant que clone de $C_1, \cdots, C_5$:

$$f_q = C'_5 \ \text{ while } \left(C_7 \ \text{ switch}(C'_1, C'_2, C'_3, C'_4)\right)$$

Concernant les composantes fortement connexes de la base, $\{f_1, f_2\}$ dispose des correspondances enfants (C'_1, C_1), (C'_2, C_2), (C'_3, C_3), (C'_4, C_4) et $\{f_3\}$ de (C'_5, C_5).

La composante fortement connexe $\{f_1, f_2\}$, de plus faible volume couverture, est traitée en priorité. La préfusion des composantes C'_1 et C'_2 conduit à l'obtention de la fonction LCA f_q : celle-ci est réalisée si $\rho = \frac{\mathcal{V}(C'_1)+\mathcal{V}(C'_2)}{\mathcal{V}(f_q)} = \frac{2}{6} > t_m$. La préfusion peut continuer avec l'ajout de (C'_3, C_3) et (C'_4, C_4) pour parvenir à un ratio de couverture sur f_q de $\frac{4}{7}$. La fusion est ensuite validée si $t_M \leq \frac{4}{7}$.

La correspondance obtenue par fusion $(f_q, \{C'_1, C'_2, C'_3, C'_4\})$ peut-elle être propagée à la composante connexe $\{f_3\}$ parente de $\{f_1, f_2\}$? Oui car $\rho'' = \frac{\mathcal{V}(C'_1)+\mathcal{V}(C'_2)+\mathcal{V}(C'_3)+\mathcal{V}(C'_4)}{\mathcal{V}^+(\{f_1,f_2\})} = 1 \geq t_p$. Nous obtenons ainsi la correspondance $(f_q, \{f_1, f_2\})$ associée à $\{f_3\}$. La fusion de (C'_5, C_5) avec celle-ci permet d'obtenir $(f_q, \{f_3\})$: les fonctions f_q et f_3 sont donc considérées comme similaires après consolidation.

11.4 Métriques d'exactitude

11.4.1 Définition

Afin de quantifier l'importance relative des correspondances élémentaires, issues d'une similarité de sous-arbres par valeur de hachage pour un certain profil, par rapport aux correspondances consolidées obtenues, nous introduisons une métrique d'exactitude. Pour une

correspondance consolidée $C = (c_1, c_2)$ issue des correspondances élémentaires (germes) non-chevauchantes $D_1 = (d_{11}, d_{12}), D_2 = (d_{21}, d_{22}), \cdots, D_l = (d_{l1}, d_{l2})$, nous définissons deux métriques d'exactitude $\mathcal{E}_1$ et $\mathcal{E}_2$:

$$\mathcal{E}_i = \frac{\mathcal{V}(c_i)}{\sum_{k\in[1..l]} \mathcal{V}(d_{ki})}$$

La métrique d'exactitude $\mathcal{E}_1$ quantifie la participation des germes du projet requête sur l'entité structurelle requête consolidée par fusion. $\mathcal{E}_2$, quant à elle, représente la participation des germes de la base sur l'entité structurelle consolidée de la base obtenue par propagations successives. Il est possible également d'intégrer dans les volumes des composantes des germes un facteur d'exactitude tel que défini en 10.4.5.

11.4.2 Utilité

Les métriques d'exactitude ne sauraient être utilisées pour quantifier la pertinence précise dans une optique de réorganisation du code ou de détection de plagiat d'une correspondance consolidée. Il est possible qu'une correspondance intéressante soit associée à une valeur d'exactitude faible. Ainsi, un bloc de code camouflé par un niveau important d'imbrication dans des multiples structures conditionnelles *si* avec clause *sinon* inutile voit sa valeur d'exactitude $\mathcal{E}_1$ tendre vers 0. La correspondance demeure néanmoins intéressante. Inversement, une correspondance peu pertinente peut être associée à une valeur d'exactitude forte ; cela peut survenir lorsque malgré la proximité des correspondances élémentaires, leur fusion ou propagation n'était pas judicieuse ou simplement lorsque le code support est idiomatique.

Les métriques d'exactitude peuvent être comparées afin d'en déduire d'éventuels scénarios d'édition pour une correspondance (A, B) :

1. Si $\mathcal{E}_1((A,B)) << \mathcal{E}_2((A,B))$, il est plausible que A soit une copie de B avec ajout de code inutile. Malgré cet ajout de code, lors des fusions le LCA englobant les germes sur le projet requête est choisi avec un ratio global de couverture des germes par rapport au LCA faible ($\mathcal{E}_1((A,B))$) : ce ratio peut être potentiellement inférieur à t_M avec les fusions successives.
2. Si $\mathcal{E}_1((A,B)) >> \mathcal{E}_2((A,B))$, une déduction inverse peut être réalisée. Il est néanmoins nécessaire de rappeler que pour des correspondances cousines sur un arbre de la base, seule une propagation des correspondances jusqu'à leur LCA pourrait permettre de les réunir : les correspondances frères doivent à ce titre posséder un ratio de couverture dépassant t_p pour être propagées.
3. Si $\mathcal{E}_1((A,B)) \sim \mathcal{E}_2((A,B))$, les composantes sur le projet requête et la base sont de volume proche. Toutefois, les opérations de fusion étant réalisées sans considération d'ordre sur les sous-arbres, il est possible que les sous-arbres germes soient organisés différemment dans A et B. Cela nous permet notamment de relever des cas de copie avec transposition de code mais peut également générer des cas de fausse positivité lorsque les germes représentent des sous-arbres d'occurrences trop fréquentes (ce qui est généralement le cas pour les petits sous-arbres idiomatiques).

11.5 Étude expérimentale de l'extension de correspondances sur arbres de syntaxe

À titre d'évaluation empirique de l'heuristique d'extension de correspondances proposée, nous nous proposons d'étudier les correspondances étendues relevées sur deux jeux de codes sources à caractéristiques différentes. Dans un premier temps, nous évaluons le projet en C Weltab comportant un fort taux de duplication et déjà très étudié dans des publications traitant de réingénierie de code. Ensuite, nous nous intéressons au paquetage Ant d'Eclipse comportant une densité de duplication plus faible.

11.5.1 Étude de Weltab

Nous évaluons tout d'abord l'heuristique d'extension de correspondances sur le projet Weltab (environ 11K lignes de code). Ce projet est un système de gestion de résultats d'élections porté de Fortran vers C au début des années 1980 et utilisé dans l'État du Michigan. Il présente un style de programmation très linéaire peu structuré (utilisation d'étiquettes de branchement et de sauts) avec la présence de nombreux clones originels. Nous étudions la version utilisée par le comparatif de Bellon et al [86] qui comporte en outre de nouveaux clones artificiellement ajoutés. Les paramètres de l'heuristique d'extension sont fixés aux seuils de $\frac{1}{4}$ pour la pré-fusion, $\frac{1}{2}$ pour la fusion et $\frac{1}{2}$ pour la propagation. Préalablement à l'étape d'extension, nous identifions l'ensemble des paires de morceaux de code dont les arbres de syntaxe respectifs sont exactement similaires et comportent au moins 30 nœuds (volume assimilé au nombre de nœuds) avec une technique d'indexation sur fenêtre[4]. Nous relevons 1282 classes d'équivalence totalisant 30921 paires de clones (ceux-ci pouvant s'intersecter en raison de l'indexation sur fenêtre) qui serviront de germes lors de l'extension. Celle-ci est réalisée en ordonnant arbitrairement les unités de compilation $u_1, u_2, \cdots, u_n$ puis en recherchant pour chaque unité requête u_k des similarités sur la base des unités $u_1, u_2, \cdots, u_{k-1}$. Le processus d'extension étant asymétrique, le choix de l'ordre des unités est susceptible de modifier le résultat obtenu.

Après l'extension, on relève un nombre important de correspondances consolidées racines (1680). Celles dont le volume sur l'arbre requête ou l'arbre de la base est le plus important mettent en relation de larges fonctions ou des unités complètes. Nous constatons en particulier que les unités de compilation `r01tmp.c`, `r11tmp.c`, `r26tmp.c`, `r51tmp.c`, `r101tmp.c` et `rsum.c` forment un groupe d'unités quasiment similaires dont chacune des paires est liée à une correspondance de racine d'arbre de syntaxe. Ils ne diffèrent que par l'initialisation et la condition de continuation d'une boucle itérative. Les unités `rsumxx.c`, `lans.c` et `ejcn88.c` complètent ce groupe d'unités similaires : celles-ci présentent cependant plus d'opérations d'édition. Par exemple, si nous comparons `rsum.c` et `ejcn88.c` par alignement global sur les lignes brutes, outre les renommages de variables ou modifications de littéraux (modifications insensibles par la manipulation de lexèmes abstraits), une quinzaine de lignes d'instructions ainsi que la fonction `setdatetime` sont insérées dans `ejcn88.c` (`rsum.c` comptabilise 387 lignes et `ejcn88.c` 415). Selon la méthode exposée en section 12.3.1, ces deux unités présentent une similarité normalisée par rapport à `ejcn88.c` (ou leur union) de $0,942$ et de 1 par rapport à `rsum.c` (entièrement contenu dans `ejcn88.c`).

[4] L'indexation sur fenêtre est ici particulièrement nécessaire en raison de la présence de longs blocs d'instructions élémentaires.

Les autres paires d'unités présentent une similarité normalisée sur l'union inférieure à $\frac{9}{10}$ (avec 9 paires de similarité comprise entre $\frac{5}{10}$ et $\frac{9}{10}$).

Clones ajoutés Nous examinons la détection des clones injectés au projet Weltab. Outre les clones exacts et variant par des modifications d'identificateurs ou types (de type 1 et 2 selon la taxonomie de Bellon) naturellement relevés, 4 clones de type 3 inter-unités sont présents. Si des structures dupliquées comportant 6 à 7 champs (avec variantes avec ajout et suppression de champs) présentées en figure 11.2 ne sont pas détectées en raison du seuil de volume d'indexation trop faible (leur pertinence en tant que clone pouvant être questionnée), les clones comprenant un germe assez volumineux le sont, notamment pour le fichier `linecoun.c` comprenant une boucle légèrement modifiée de comptage de lignes d'une fonction de `spol.c`

```
24 typedef struct {
25   int vect;
     char* ptr;
     void* next;
     char flags[16];
     unsigned int err;
30   char* ptrs[4];
     void* parent;
   } cnv1_t;
```

(a) cnv1.c

```
23 typedef struct {
     int vect;
25   float val;
     char* ptr;
     void* next;
     char flags[16];
     unsigned int err;
30 } canv1_t;
   typedef struct {
     int type;
     float amount;
     char* ptr;
35   void* next;
     char passwd[16];
     unsigned int flags;
   } canv2_t;
```

(b) canv.c

FIG. 11.2 – Copies de structures

Correspondances d'exactitudes faibles Afin de repérer l'éventuelle existence d'opérations de consolidation peu pertinentes, nous examinons maintenant les correspondances présentant les exactitudes les plus faibles sur arbre requête ou sur la base. Parmi celles-ci certaines présentent une exactitude sur arbre requête forte avec une exactitude sur arbre de la base modeste. À titre d'exemple nous présentons en figure 11.3 une séquence de quatre instructions de `wellib.c` (arbre requête) de volume de 37 nœuds en correspondance avec deux copies dans `vfix.c` (arbre de la base) : l'exemplaire le plus profond de l'arbre de la base est propagé quatre fois (à travers une conditionnelle et une boucle) avec un ratio minimal de $0,66$ jusqu'à atteindre le bloc principal où une fusion a lieu avec l'autre exemplaire. On notera que si les rôles d'arbre requête et d'arbre de la base avaient été inversés, la consolidation n'aurait pu avoir lieu qu'en une unique étape de fusion entre les deux exemplaires de `vfix.c` ; leur ratio de volume cumulé par rapport au volume du LCA (bloc entier) aurait été insuffisant (ici $t_M = \frac{1}{2}$). Une rapide analyse des correspondances présentant un ratio minimal de fusion (minimum : 0,536) ou de propagation (minimum : 0,502) confirme la pertinence des extensions réalisées.

```
        trimlen = itrim(80,report);
        /* printf("trimlen = %d\n",trimlen); */
        report[trimlen] = '\0';
        fprintf(fileid,"%s\n",report);
        blkbuf(80,report);
```

(a) wellib.c

```
x525: sclear();
x526: gtoffi(j,report,0);
            len = itrim(50,report);
            report[len] = '\0';
            fprintf(stderr,"%s\n\n",report);
            blkbuf(80,report);
            askchange("this office",&change,&quit,TRUE,FALSE);
            /* askchange("this office",&change,&quit,TRUE,TRUE); */
            if (quit == 999) goto x535;
            if (quit) goto x600;
            if (! change) continue;
            /* office okay for precinct, proceed with query */
x530: fprintf(stderr,"\nExisting results are:\n");
            totoff = 0L;
            if (offinf[CANDCOUNT] > 0) goto x535;
            /* if (offinf[WRITEINCOUNT] > 0) goto x535; */
            fprintf(stderr,"*** Office has no candidates\n");
            goto x570;
            /* election has ballot candidates */
x535: istart = offinf[CANDSTART];
            iend = istart + offinf[CANDCOUNT] - 1;
            iscat = 0;
            /* iscat = iend + 1; */
            /* if (offinf[WRITEINCOUNT] < 1) iscat = 0; */
            /* if (offinf[WRITEINCOUNT] >= 1)
                  iend += offinf[WRITEINCOUNT]; */
            if (istart <= 0 || iend < istart) failure(512);
            if (quit == 999) goto x238;
            for (k=istart;k<(iend+1);k++) {
                  if (((k-istart+1) > 1) &&
                        (((k-istart+1) % 10) == 1) &&
                      (k < (iend - 1))) {
                        pause();
                        sclear();
                        gtoffi(j,report,0);
                        len = itrim(50,report);
                        report[len] = '\0';
                        fprintf(stderr,"%s (continued)\n\n",report);
                        blkbuf(80,report);
                        };
                  gtcand(k,report,2);
                  cvicl(vote[k],report,28,10);
```

(b) vfix.c

FIG. 11.3 – Utilisation multiple d'un germe de l'arbre requête ($\mathcal{E}_1 = 0,411$, $\mathcal{E}_2 = 1$)

11.5.2 Étude d'Eclipse Ant

Apache Ant est un logiciel en langage Java utilisé pour la construction de projets à partir de la spécification de règles de dépendances entre leurs composantes. La version étudiée est le paquetage livré avec l'IDE Eclipse et utilisé pour le comparatif de Bellon : celle-ci comporte 178 unités de compilation avec environ 19K lignes de code utiles. Nous réalisons tout d'abord une recherche de classes de portions de code similaires d'au moins 20 nœuds utilisant une abstraction ignorant les sous-arbres comportant 4 nœuds ou moins : 1917 classes d'équivalence sont trouvées. Parmi ces classes, nous écartons celles comprenant un nombre de membres trop élevés (plus de 10 membres) car représentant des similarités potentiellement idiomatiques (successions de déclaration de membres de classes ou de constantes, d'affectations simples, de définition de *getters* ou *setters* basiques...). Au sein des classes sélectionnées nous relevons 1087 paires en correspondance. Sont dérivées de ces paires germes, par extension, 481 correspondances consolidées racines.

Classement des correspondances racines Parmi les correspondances racines (non utilisées comme composante pour la fusion ou la propagation), nous distinguons trois ensembles de correspondances :

1. Les correspondances germes non-consolidées (272 paires) dites *solitaires*. Il s'agit de correspondances n'ayant pu être regroupées avec d'autres correspondances germes proches. Elles représentent environ 57% de la totalité des germes.
2. Les correspondances consolidées dont les exactitudes sur l'arbre requête et la base sont de 1 (173 correspondances). Elles sont le résultat du regroupement de correspondances frères consécutives aussi bien sur l'arbre requête que l'arbre de la base. On notera que ces correspondances auraient pu être obtenues également par l'usage d'un farmax sur les chaînes de nœuds consécutifs appartenant à une correspondance comme expliqué en section 10.4.4. L'ensemble des correspondances obtenues ici est néanmoins plus étendu en raison de la méthode de fusion utilisée qui est agnostique sur l'ordre des composantes frères regroupées.
3. Les autres correspondances consolidées dites *approchées* (au nombre de 36). Cette catégorie mérite d'être étudiée plus en détails afin d'évaluer la pertinence des consolidations réalisées.

Correspondances approchées Parmi les correspondances approchée obtenues, deux sous-catégories sont distinguées : la première concerne des correspondances composée d'un unique germe (19 correspondances) ayant subi une ou plusieurs opérations de propagation dans leur arbre de la base. Les correspondances consolidées sont de volume moyen compris entre 31 et 80. Les opérations de propagation sont généralement réalisées à l'échelle locale lorsqu'un germe occupe la majorité du volume d'un bloc d'une boucle ou d'une structure conditionnelle ; elles se limitent dans la quasi-totalité des cas à deux propagations successives. Un seul cas impliquant plus de 2 opérations de propagation (6) a été relevé ; il se base sur un germe non pertinent qui est propagé ainsi inutilement à une fonction entière. Il est exposé en figure 11.4.

Nous choisissons de nous intéresser à la seconde sous-catégorie des correspondances comprenant plusieurs germes (au nombre de 17). Elles comprennent des similarités à un niveau local voire fonctionnel avec des instructions insérées, supprimées ou des expressions réécrites.

```
201     public synchronized static IntrospectionHelper getHelper(Class c) {
            IntrospectionHelper ih = (IntrospectionHelper) helpers.get(c);
            if (ih == null) {
                ih = new IntrospectionHelper(c);
205             helpers.put(c, ih); }
            return ih; }
```

(a) ant.IntrospectionHelper

```
465     private InputStream getResourceStream(File file, String resourceName) {
            try {
                if (!file.exists()) {
                    return null; }
                if (file.isDirectory()) {
470                 File resource = new File(file, resourceName);
                    if (resource.exists()) {
                        return new FileInputStream(resource); } }
                else {
                    // is the zip file in the cache
475                 ZipFile zipFile = (ZipFile)zipFiles.get(file);
                    if (zipFile == null) {
                        zipFile = new ZipFile(file);
                        zipFiles.put(file, zipFile); }
                    ZipEntry entry = zipFile.getEntry(resourceName);
480                 if (entry != null) {
                        return zipFile.getInputStream(entry); } } }
            catch (Exception e) {
                log("Ignoring␣Exception␣" + e.getClass().getName() + ":␣" + e.getMessage() +
                    "␣reading␣resource␣" + resourceName + "␣from␣" + file, Project.MSG_VERBOSE); }
485         return null; }
```

(b) ant.AntClassLoader

FIG. 11.4 – Propagation d'un germe de pertinence faible

Quelques cas présentent une similarité à l'échelle d'une classe complète de faible volume comprenant quelques fonctions.

Parmi les correspondances consolidées approchées multi-germes examinées, une peut être considérée non-pertinentes à cause de ses germes qui ne le sont pas par le choix d'une abstraction effaçant les petits sous-arbres. Une autre est étendue inutilement par un germe inadéquat. Les correspondances restantes apparaissent pertinentes et pourraient faire l'objet d'une factorisation ; certains commentaires des développeurs associés à des duplications prévoient cette perspective. L'utilité d'employer une méthode de regroupement de 2-correspondances similaires est soulignée par la présence des classes KaffeRmic, Javac12 et SunRmic regroupables en une 3-correspondance. On notera également l'imbrication sur ZipLong de deux correspondances.

En conclusion, nous pouvons constater sur cet exemple que la consolidation permet d'augmenter le volume des correspondances par le regroupement de plusieurs germes (ici 52 germes ont été regroupés pour former 17 correspondances). Lorsque les germes sont pertinents, la consolidation apparaît indiquée. La principale difficulté réside dans le choix de la méthode d'obtention de germes. Si le choix d'une abstraction forte permet un meilleur rappel de germes, elle introduit de nombreuses correspondances non-pertinentes. Il est possible de s'en prémunir en ignorant comme ici les classes d'équivalence de forte cardinalité. Toutefois, cela pourrait être préjudiciable lorsque des germes issues de classes peuplées et d'autres de classes de faible cardinalité se cotoient dans un arbre de syntaxe requête : les premiers sont ignorés et peuvent compromettre la fusion des seconds. Une solution serait de considérer l'ensemble des classes d'équivalence et d'introduire, à des fins de filtrage à l'issue de l'extension, une métrique de singularité. Lors de la détermination des germes, nous utilisons plusieurs familles d'abstraction et associons à chaque germe une métrique de singularité. Celle-ci serait basée sur la cardinalité de la classe d'équivalence la plus spécialisée contenant les deux exemplaires du germe. Pour toute correspondance consolidée, nous associons la singularité du germe le plus singulier. Cette métrique permettrait d'estimer l'idiomaticité de la correspondance consolidée et de prioritiser l'examen des correspondances pour un évaluateur humain.

11.6 Quelques améliorations envisageables

Outre l'introduction d'une métrique de singularité associée au germes et correspondances consolidées évoquée précédemment, quelques pistes peuvent être explorées afin d'améliorer l'heuristique de consolidation de correspondances sur arbre de syntaxe et DAG d'appel présenté.

11.6.1 Favoritisme des fusions et propagations sur composantes d'ordonnancement concordant

En premier lieu, nous notons que la méthode de fusion est ensembliste : le rassemblement de correspondances frères sur la base et la fusion, avec choix de LCA commun, de correspondances proches sur le projet requête, sont réalisés sans aucune considération d'ordre. Ainsi, par exemple, ce procédé permet d'obtenir une correspondance consolidée par fusion $(T_q, \{A_2, A_1\})$ pour les deux arbres $T_q = a(A'_1, A'_2, A'_3)$ et $T_{\mathcal{B}} = a(A_2, A_1, A_4)$. Ceci permet de gérer les cas de transposition de code mais est susceptible d'introduire des cas faux-positifs. En particulier, il serait souhaitable qu'à ratio de couverture égal légèrement inférieur aux seuils ($\rho + \epsilon = t_m$,

Correspondance	Volume requête (exactitude)	Volume base (exactitude)	Germes	Niveau	Observations	Pertinence
Move(106-113), Copy(275-284)	60 (1)	60 (0,98)	4	Local	Réécriture d'expression	Oui
Replace(69-84), RuntimeConfigurable(59-79)	85 (0,75)	65 (1)	2	Multi-fonctionnel	Réécriture d'expression	Non
Rmic[115-168], Javadoc[91-445]	134 (1)	302 (0,83)	8	Multi-fonctionnel	Transposition, duplication	Partiellement
Touch(153-157), Copy(196-202)	48 (1)	69 (0,70)	2	Local	Insertion d'instruction	Oui
War, Ear	180 (1)	224 (0,80)	7	Classe	Insertion d'instruction	Oui
DefaultCompilerAdapter(310-326), Jikes(92-107) (11.6)	121 (0,65)	105 (0,74)	3	Local	Réécriture d'expression, insertion d'instruction	Oui
Kjc(104-112), DefaultCompilerAdapter(210-222)	45 (1)	46 (0,98)	2	Local	Similarité exacte propagée au bloc parent	Oui
DefaultRmicAdapter(117-162), DefaultCompilerAdapter(137-175)	226 (0,81)	177 (0,81)	4	Local	Blocs conditionnels avec ajout d'un cas *else if*	Oui
KaffeRmic, Javac12 (2.3)	125 (0,70)	182 (0,46)	3	Classe	Réécriture d'expression, insertion d'instruction	Oui
SunRmic, KaffeRmic	133 (0,67)	151 (0,58)	3	Classe	Réécriture d'expression, insertion d'instruction	Oui
SunRmic, Kjc	133 (0,64)	113 (0,69)	2	Classe	Insertion d'instruction	Oui
CommandLine(229-233), Javadoc(471-477)	41 (1)	51 (0,82)	2	Fonctionnel	Réécriture d'expression	Oui
Jdk14RegexMatcher(94-98), JakartaRegexMatcher(85-97)	38 (1)	49 (0,78)	2	Fonctionnel	Réécriture d'expression	Oui
Jdk14RegexMatcher(1-99), JakartaOroMatcher(66-98)	92 (0,53)	122 (0,71)	3	Classe	Réécriture d'expression	Oui
ZipEntry(160-177), MailMessage(246-267)	106 (1)	130 (0,815)	5	Fonctionnel	Insertion d'instruction, clone ajouté	Oui
ZipOutputStream(532-536), ZipLong(88-99)	63 (1)	70 (0,9)	3	Local, fonctionnel	Propagation fonctionnelle de similarité exacte	Oui
ZipShort(86-111), ZipLong(88-115) (11.7)	79 (0,91)	108 (0,68)	3	Multi-fonctionnel	Insertion d'instruction	Oui

FIG. 11.5 – Correspondances consolidées approchées à plusieurs germes sur Eclipse-Ant

```
310             if (Commandline.toString(args).length() > 4096) {
                    PrintWriter out = null;
                    try {
                        tmpFile = new File("jikes"+(new Random(System.currentTimeMillis())).nextLong());
                        out = new PrintWriter(new FileWriter(tmpFile));
315                     for (int i = firstFileName; i < args.length; i++) {
                            out.println(args[i]); }
                        out.flush();
                        commandArray = new String[firstFileName+1];
                        System.arraycopy(args, 0, commandArray, 0, firstFileName);
320                     commandArray[firstFileName] = "@" + tmpFile.getAbsolutePath();
                    } catch (IOException e) {
                        throw new BuildException("Error creating temporary file", e, location);
                    } finally {
                        if (out != null) {
325                         try {out.close();} catch (Throwable t) {} } }
                } else {
```

(a) ant.taskdefs.compilers.DefaultCompilerAdapter

```
 91             if (myos.toLowerCase().indexOf("windows") >= 0
                    && args.length > 250) {
                    PrintWriter out = null;
                    try {
 95                     tmpFile = new File("jikes"+(new Random(System.currentTimeMillis())).nextLong());
                        out = new PrintWriter(new FileWriter(tmpFile));
                        for (int i = 0; i < args.length; i++) {
                            out.println(args[i]); }
                        out.flush();
100                     commandArray = new String[] { command,
                                                      "@" + tmpFile.getAbsolutePath()};
                    } catch (IOException e) {
                        throw new BuildException("Error creating temporary file", e);
                    } finally {
105                     if (out != null) {
                            try {out.close();} catch (Throwable t) {} } }
                } else {
```

(b) ant.taskdefs.Jikes

FIG. 11.6 – Correspondance consolidée approchée pertinente d'exactitude homogène sur Eclipse-Ant ($\mathcal{E}_0$ 0,65, $\mathcal{E}_1$ 0,74)

```
532         byte[] result = new byte[4];
            result[0] = (byte) ((value & 0xFF));
            result[1] = (byte) ((value & 0xFF00) >> 8);
535         result[2] = (byte) ((value & 0xFF0000) >> 16);
            result[3] = (byte) ((value & 0xFF000000l) >> 24);
```

(a) zip.ZipOutputStream

```
88      /**
         * Get value as two bytes in big endian byte order.
90       *
         * @since 1.1
         */
        public byte[] getBytes() {
            byte[] result = new byte[4];
95          result[0] = (byte) ((value & 0xFF));
            result[1] = (byte) ((value & 0xFF00) >> 8);
            result[2] = (byte) ((value & 0xFF0000) >> 16);
            result[3] = (byte) ((value & 0xFF000000l) >> 24);
            return result; }
100     /**
         * Get value as Java int.
         *
         * @since 1.1
         */
105     public long getValue() {
            return value; }
        /**
         * Override to make two instances with same value equal.
         *
110      * @since 1.1
         */
        public boolean equals(Object o) {
            if (o == null || !(o instanceof ZipLong)) {
                return false; }
115         return value == ((ZipLong) o).getValue(); }
```

(b) zip.ZipLong

```
86      /**
         * Get value as two bytes in big endian byte order.
         *
         * @since 1.1
90       */
        public byte[] getBytes() {
            byte[] result = new byte[2];
            result[0] = (byte) (value & 0xFF);
            result[1] = (byte) ((value & 0xFF00) >> 8);
95          return result; }
        /**
         * Get value as Java int.
         *
         * @since 1.1
100      */
        public int getValue() {
            return value; }
        /**
         * Override to make two instances with same value equal.
         *
         * @since 1.1
         */
        public boolean equals(Object o) {
            if (o == null || !(o instanceof ZipShort)) {
110             return false; }
            return value == ((ZipShort) o).getValue(); }
```

(c) zip.ZipShort

FIG. 11.7 – Correspondances imbriquées utilisant une conversion d'entier en octets sur Eclipse-Ant

$\rho' + \epsilon = t_M$, ou $\rho'' + \epsilon = t_p$), que ce soit par rapport à un LCA (fusion) ou à un parent (propagation), des ensembles de composantes, dont l'ordonnancement concorde, soit acceptées pour la fusion ou la propagation, alors que des mêmes composantes, avec un ordonnancement non concordant soit refusées. Ainsi, si $t_M = \frac{1}{2} + \epsilon$, avec $\mathcal{V}(A_i) = \mathcal{V}(A'_i) = 1$, la fusion de A'_1 et A'_2 ne serait pas possible, alors que pour $T_r = a(A'_2, A'_1, A'_3)$, celle-ci aurait été possible, car d'ordre concordant avec la fratrie enfant de $T_{\mathcal{B}}$.

Afin de privilégier les fusions et propagations sur composantes d'ordonnancement concordant, une piste envisagée serait l'utilisation d'une métrique de volume corrigée sur le numérateur pour calculer les ratios de couverture. Celle-ci pourrait être une fonction hyperlinéaire du volume, tel que $x \longrightarrow x^{1+\alpha}$ (α étant un petit réel). Nous obtiendrions ainsi le volume corrigé $\mathcal{V}' = \mathcal{V}^{1+\alpha}$.

11.6.2 Considération sémantique

La méthode présentée utilise des seuils de préfusion, fusion et propagation fixes sans prise en compte de la nature des sous-arbres fusionnés ou propagés. Il pourrait être intéressant, à la lumière des résultats expérimentaux, de proposer des seuils personnalisés en fonction des nœuds considérés afin d'améliorer le rappel et la précision.