UNIVERSITÉ DE MONTRÉAL

A HEURISTIC-BASED APPROACH TO LOCATE CONCEPTS IN EXECUTION TRACES

FATEMEH ASADI

DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES

(GÉNIE INFORMATIQUE))

MAI 2010

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

A HEURISTIC-BASED APPROACH TO LOCATE CONCEPTS IN EXECUTION TRACES

présenté par : ASADI, Fatemeh

en vue de l'obtention du diplôme de : Maîtrise ès Sciences Appliquées

a été dûment accepté par le jury d'examen constitué de :

M. GUIBAULT, Francois, Ph.D., président,

M. ANTONIOL, Giuliano, Ph.D., membre et directeur de recherche.

M. ROBILLARD, Pierre-N., Ph.D., membre.

ACKNOWLEDGEMENT

I would like to express my deepest gratitude to Dr. Giuliano Antoniol for giving me a chance to carry out my research under his supervision, in his research group. Further, I would take this opportunity to thank him for all his support and encouragement.

Also, I express my thanks to Dr. Massimiliano Di Penta and Dr. Yann-Gaël Guéhéneuc for their help and constant support during my research.

Finally, I would like to thank my husband, Mohsen, without whose help, motivation and sacrifice this would not have been possible. Last but not the least, I would thank all my colleagues for the good time we had together in the lab.

CONDENSÉ EN FRANÇAIS

La maintenance de logiciels est la dernière phase du cycle de vie des logiciels. Elle joue un rôle important dans le cycle de vie d'un logiciel car plus de 50% du coût du cycle de vie appartient à la maintenance. Un des défis importants avec la maintenance de logiciels est la compréhension.

La compréhension de logiciels est une partie cruciale de la maintenance et est un facteur majeur pour une maintenance efficace ainsi que pour une évolution réussie d'un logiciel.

Un problème commun à la compréhension de logiciels est souvent le manque d'une documentation adéquate. Quelques années après le déploiement, c'est possible que la documentation n'existe plus ou si elle existe, elle est surement dépassée. Les développeurs qui maintiennent le logiciel, la plupart du temps, sont différents de ceux qui l'ont développé. Par conséquent, les développeurs doivent recourir à la lecture du code source du système, avec rien d'autre que des navigateurs de code, pour comprendre le logiciel et accomplir leurs tâches de maintenance et d'évolution. Dans certains cas, la compréhension du code source est facilitée par différentes techniques d'analyse statique et/ou des visualisations construites sur de l'information statique. Dans d'autres cas, le débogage peut être utilisé pour comprendre le comportement d'un logiciel dans un contexte particulier et/ou pour trouver une faille. Toutefois, la navigation manuelle de code source, l'analyse d'une trace d'exécution ou le débogage de longues séquences d'instructions sont des tâches redoutables qui prennent beaucoup de temps.

L'identification et la localisation de concepts ou de fonctionnalités visent à aider les développeurs à effectuer leurs tâches de maintenance et d'évolution, en identifiant des abstractions (par exemple, caractéristiques) et l'emplacement de l'implémentation de ces abstractions. Autrement dit, ils visent à identifier des fragments de code source, c'est à dire l'ensemble d'appels de méthodes dans des séquences d'instructions et les déclarations de ces méthodes dans le code source, responsables de la mise en œuvre des concepts du domaine du logiciel et des caractéristiques observables par l'utilisateur (Antoniol et Gueheneuc (2006); Biggerstaff *et al*. (1993); Kozaczynski *et al*. (1992); Poshyvanyk *et al*. (2007a); Tonella et Ceccato (2004)). Dans la littérature il existe des

approches d'analyse statique (Anquetil et Lethbridge (1998); Marcus *et al*. (2004)) et dynamique (Wilde et Scully (1995); Tonella et Ceccato (2004)) ; Recherche Documentaire (Information Retrieval - IR) (Poshyvanyk *et al*. (2007a)) et des approches d'analyse hybride (statique et dynamique) (Eaddy *et al*. (2008)).

L'analyse statique utilise des données extraites à partir du code source. Les données peuvent être des données structurelles utilisées pour générer des graphes de dépendance de programmes (Chen et Rajlich (2000)), ou des composants d'un groupe au sein du logiciel (Maletic et Marcus (2001)), ou bien de l'information d'ordre sémantique fourni en utilisant des techniques de recherche d'information (Marcus *et al*. (2004)).

Toutes les approches dynamiques utilisent les séquences d'instructions (traces), obtenue par l'exécution du logiciel, pour identifier les fonctionnalités du code source. Les traces utilisées dans ces approches sont générées par l'exécution des scénarios qui sont conçus à partir des cas de test ou des requis du logiciel. En utilisant l'exécution de scénarios, chaque technique a sa propre méthode pour détecter une fonctionnalité particulière. Certaines techniques identifient une fonctionnalité en comparant deux traces avec et sans la fonctionnalité, d'autres effectuent certaines opérations de classement sur les méthodes d'une trace.

Les deux techniques, statiques et dynamiques, ont leurs propres limites. En général, les techniques dynamiques sont conservatives par nature, car les traces sont souvent très vastes et contiennent beaucoup de bruit. La sélection appropriée des cas de test ou des scénarios à exécuter est un autre problème de ces techniques. D'autre part, l'analyse statique peut rarement déterminer les méthodes contribuant à l'exécution d'un scénario spécifique. En fait, chaque type d'analyse recueille des données qui seraient non disponibles pour l'autre type d'analyse, mais qui peuvent améliorer la performance du processus d'identification de fonctionnalités. Autrement dit, chaque approche peut être considérée comme un complément de l'autre. Des travaux récents ont accentué sur une combinaison des données statiques et dynamiques (Eaddy *et al*. (2008); Poshyvanyk *et al*. (2007a)), où le problème de localisation de fonctionnalités à partir de traces d'exécution multiples est modélisé comme un problème d'IR, qui a l'avantage de simplifier le processus de localisation et, souvent,

d'améliorer la précision (Poshyvanyk *et al*. (2007a)).

Cette thèse propose une nouvelle approche pour identifier des fragments cohésifs et découplés dans des traces d'exécution, qui participent probablement à mettre en œuvre les concepts liés à certaines fonctionnalités. L'approche repose sur des techniques optimisées pour la recherche en utilisant un algorithme de métaheuristiques, analyse textuelle du code source du logiciel en utilisant l'indexation sémantique latente et des techniques de compression de traces.

Le domaine du génie logiciel orienté recherche (SBSE) tente de trouver des solutions aux problèmes du génie logiciel en reformulant ces derniers en problèmes de recherche pour ensuite appliquer des techniques de recherche métaheuristiques. Les techniques de recherche métaheuristique sont utilisées pour résoudre les problèmes où la solution doit être trouvée dans des espaces de recherche très étendues. Ils explorent l'espace de recherche de façon itérative et essayent d'améliorer la solution actuelle en utilisant des fonctions coût. La solution finale obtenue en utilisant une technique de recherche métaheuristique peut ne pas être la solution optimale, mais elle est généralement proche de la solution optimale. Nous avons utilisé trois algorithmes métaheuristiques dans notre approche : escalade, recuit simulé et algorithme génétique.

La science de recherche d'information (IR) est une science d'extraction d'informations à partir de documents. Sa tâche est de trouver des documents pertinents dans une grande base de documents pour une requête spécifique. IR accélère le processus de recherche de documents en les représentant avec des modèles mathématiques. Pour extraire de l'information linguistique à partir du code source, dans notre approche, nous avons utilisé l'indexation sémantique latente (Latent Semantic Indexing - LSI) qui est une version développée du modèle d'espace vectoriel. LSI utilise une technique mathématique appelée décomposition en valeurs singulières (SVD) pour identifier des patrons de la relation entre les termes et concepts dans un recueil de texte.

Un problème typique pour lequel l'approche proposée peut être bénéfique est le suivant. Supposons qu'un échec a été observé lors de l'exécution d'un scénario particulier d'un logiciel, mais malheureusement le risque de reproduire les conditions d'exécution de cet échec est très faible. Les développeurs qui maintiennent le logiciel sont alors confrontés au problème de l'analyse de

la trace d'exécution produite par ce scénario et l'identification des abstractions de haut niveau qui participent probablement à la fonctionnalité qui produit le comportement indésirable.

Pour faire face au problème décrit ci-dessus, l'approche proposée identifie des concepts qui composent un scénario d'exécution en regroupant les méthodes qui sont (i) invoquées ensemble de façon séquentielle/dans l'ordre, et (ii) cohésives et découplées de point de vue conceptuel. L'hypothèse sous-jacente est que, si une fonctionnalité spécifique est exécutée dans un scénario complexe (par exemple, " Ouvrir une page Web à partir d'un navigateur " ou " Enregistrer une image dans une application graphique "), alors l'ensemble des méthodes successivement invoquées est susceptible d'être cohérent sur le plan conceptuel et découplé des autres fonctionnalités. Nous utilisons la cohésion conceptuelle et le couplage de Marcus et al (Marcus *et al*. (2008)) Poshyvanyk et al(Poshyvanyk et Marcus (2006a)).

L'approche fonctionne comme suit. Premièrement, nous indexons le code source de chaque méthode d'un système textuellement. Ensuite, nous instrumentons et exerçons le système de collecte de traces d'exécution pour certains scénarios liés aux différentes fonctionalités et, par conséquent, contenant des ensembles de concepts différents. Nous compressons les traces pour supprimer des fonctions utilitaires et des concepts entrecroisés et pour abstraire les répétitions de mêmes sous séquences de méthodes. Enfin, nous appliquons une technique de recherche basée sur l'optimisation, c'est-à-dire, un algorithme génétique, afin de scinder les traces comprimées en fragments cohésifs et découplés. Nous assurons la performance via la parallélisation de l'algorithme sur plusieurs ordinateurs.

Pour évaluer l'approche proposée, nous avons effectué une étude empirique en appliquant l'approche proposée sur deux logiciels libres, ArgoUML et JHotDraw. Les résultats ont montré que l'approche est stable, et, globalement, localise les concepts avec une précision élevée. La précision a tendance à baisser pour les éléments réalisés en utilisant des séquences de méthodes très similaires, comme c'est parfois le cas dans JHotDraw, où différents types de formes sont dessinées essentiellement de la même façon. Les chevauchements entre un oracle manuellement construits et les segments détectés automatiquement varient en fonction de la cohésion des fonctionnalités

analysées, car l'approche a tendance à diviser des traces liées à de grandes fonctionnalités en plus petits segments liés à des sous concepts cohésifs.

Les futurs travaux suivront différentes directions. Premièrement, nous améliorons l'approche proposée pour accroître sa performance en réglant mieux la recherche basée sur l'optimisation et les techniques d'indexation de texte. D'autres futurs travaux consisteront à améliorer et implémenter notre idée d'assigner automatiquement des étiquettes appropriées pour les concepts localisés. Nous devons aussi travailler sur le problème du passage à l'échelle, trouver une solution appropriée et pratique en mesure d'être appliquée à de très longues traces. Aussi, l'approche devrait être étendue pour être capable de travailler avec des systèmes multiprocessus.

L'Approche d'Identification de Concept

L'approche proposée pour l'identification de concept comprend les étapes suivantes :

- L'analyse textuelle du code source des méthodes
- L'instrumentation du système
- La récupération des traces d'exécution
- La réduction et la compression des traces
- L'identification des concepts à l'aide de techniques d'optimisation basées sur la recherche

Analyse Textuelle du Code Source des Méthodes

Pour déterminer la cohésion conceptuelle des méthodes, notre approche utilise la métrique de cohésion conceptuelle définie par Marcus *et al.* (Marcus *et al*. (2008)). Nous considérons chaque méthode comme un document, donc notre corpus est l'ensemble des méthodes dans le système. Nous extrayons un ensemble de termes de chaque méthode en procédant à l'analyse lexicale du code source et des commentaires qui y sont rattachées. Nous retirons ensuite les caractères spéciaux, les mots clés du langage de programmation ainsi que des mots (anglais) non pertinents tels que les articles et les prépositions. A l'étape qui suit, nous séparons à chaque lettre majuscule les termes composés créés avec la convention Camel-Case. Ensuite, nous retrouvons les racines des

termes obtenus en utilisant l'outil de Porter (Porter (1980)). Par exemple, le mot anglais " visited " sera remplacé par sa racine " visit ". Une fois que les termes appartenant à chaque méthode ont été extraits, nous indexons ces termes en utilisant les mécanismes d'indexation $TF - IDF$ (Baeza-Yates et Ribeiro-Neto (1999)). Nous obtenons ainsi une matrice terme-document, où les documents sont toutes les méthodes de toutes les classes du système étudié et les termes sont tous les termes extraits (et divisés) des commentaires et du code source des méthodes. Enfin, nous appliquons l'indexation sémantique latente (LSI) (Deerwester *et al*. (1990)), en utilisant l'implémentation par décomposition en valeurs singulières (SVD) dans R, afin de réduire la matrice document-terme à une matrice document-concept. Pour chaque paire de méthodes dans le corpus, nous calculons leur similitude en calculant le cosinus entre leurs vecteurs correspondants dans la matrice méthode-terme (Figure 1).

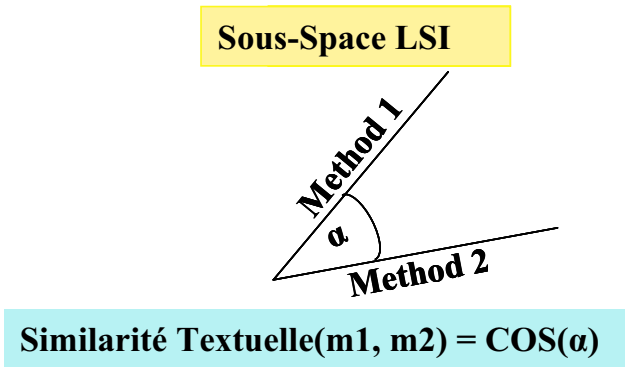

Figure 1 Similarité Textuelle Entre Deux Méthodes.

Instrumentation du Système et Récupération des Traces d'Exécution

Pour générer les traces d'exécution, d'abord nous instrumentons le système logiciel en utilisant l'instrumenteur MoDeC. Ensuite, nous exerçons le système en suivant des scénarios tirés des manuels utilisateurs ou des descriptions de cas d'utilisation et nous extrayons les traces d'exécution.

Réduire et comprimer les traces

Certaines méthodes se produisent trop souvent dans presque tous les scénarios, par exemple, les méthodes liées aux mouvements de la souris. Ces méthodes ne sont reliées à aucun concept particulier et ne sont pas utiles pour l'identification de fonctionnalités. Nous pouvons les considérer comme des méthodes utilitaires gérant des tâches génériques. En analysant la distribution des fréquences d'occurrences des méthodes, nous pouvons procéder à la suppression des méthodes trop fréquentes. Nous retirons les méthodes qui ont une fréquence supérieure à $Q3 + 2 \cdot IQR$, où Q3 est le troisième quartile (percentile 75%) de la distribution et IQR est l'intervalle inter-quartile. Les traces contiennent souvent des répétitions d'appels d'une ou plusieurs méthodes, par exemple m1 () ; m1 () ; m1 () ; ou m1 () ; m2 () ; m1 () ; m2 () ;. Les répétitions ne présentent pas de nouveaux concepts, Ainsi, on comprime les traces en utilisant l'algorithme Run Length Encoding (RLE) pour supprimer les répétitions et garder seulement une occurrence de la répétition. Les deux exemples précédents deviendraient " m1() " et " m1() ; m2() ", respectivement.

Techniques d'Optimisation Basées Sur la Recherche

Nous avons expérimenté des techniques différentes : la descente, le recuit simulé et les algorithmes génétiques (GAs). Finalement, nous avons choisi d'utiliser des GAs puisqu'en raison des caractéristiques de l'espace de recherche, ils ont surpassé les autres techniques. Notre représentation d'un individu est une chaîne de bits aussi longue que la trace d'exécution dans laquelle nous voulons identifier certains concepts (fonctionnalités). Chaque appel de méthode est représenté par un "0", à l'exception de l'appel de la dernière méthode dans un segment, qui est représenté par un "1". Par exemple, la chaîne de bits

$$\underbrace{00010010001}_{11 methodes}$$

signifie que la trace, contenant 11 appels de méthode, est divisé en trois segments décomposés suivant les (i) quatre premiers appels de méthode (ii) les trois suivants, et (iii) les quatre derniers. Un exemple concret de découpage d'un segment est indiqué dans le Tableau 1.

Tableau 1 Example de la representation d'un individu de GA (deuxième colonne)

Method Invocations	Repr.	Segments#
TextTool.deactivate()	0	
TextTool.endEdit()	0	
FloatingTextField.getText()	0	
TextFigure.setText-String()	0	1
TextFigure.willChange()	0	
TextFigure.invalidate()	0	
TextFigure.markDirty()	1	
TextFigure.changed()	0	
TextFigure.invalidate()	0	
TextFigure.updateLocation()	0	2
FloatingTextField.endOverlay()	0	
CreationTool.activate()	1	
JavaDrawApp.setSelectedToolButton()	0	
ToolButton.reset()	0	
ToolButton.select()	0	
ToolButton.mouseClickedMouseEvent()	0	3
ToolButton.updateGraphics()	0	
ToolButton.paintSelectedGraphics()	0	
TextFigure.drawGraphics()	0	
TextFigure.getAttributeString()	1	

L'opérateur de mutation choisit au hasard un bit dans la représentation et le change. Changer un "0" en "1" signifie diviser un segment existant en deux segments, tandis que changer un "1" en "0" signifie la fusion de deux segments consécutifs (Figure 2).

Figure 2 L'opérateur de Mutation

L'opérateur de croisement est le croisement standard bi-points. Partant de deux individus sélectionnés en tant que parents, deux positions aléatoires x, y avec x <y sont choisis dans la chaîne de bits d'un individu et les bits de x à y sont échangés entre les deux individus pour créer de nouveaux individus (fils). Le choix des parents est effectué en utilisant la une roulette biaisée (Figure 3).

La Fonction de Fitness

Comme mentionné précédemment, nous utilisons les principes de cohésion et de couplage dans une conception logicielle. Ces principes ont déjà été utilisés dans le passé pour identifier les mo-

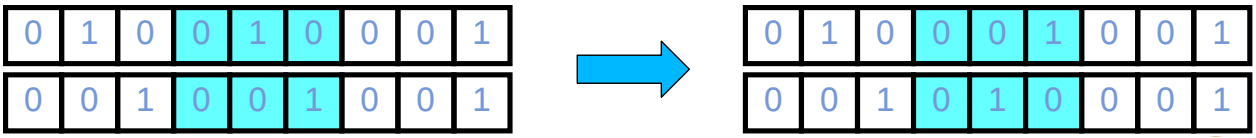

Figure 3 L'opérateur de Croisement

dules dans des systèmes informatiques (Mitchell et Mancoridis (2006)), mais dans notre cas, nous nous appuyons sur des mesures conceptuelles (i.e textuelles), plutôt que sur des mesures de cohésion et de couplage structurelles.

Nous définissons la cohésion d'un segment k comme étant la moyenne de la similarité textuelle entre n'importe quelle paire de méthodes dans un segment k. Nous la calculons en utilisant l'équation 1 où $begin(k)$ est la position (dans la chaîne de bits de chaque individu) du premier appel de méthode du kième segment et $end(k)$ est la position du dernier appel de méthode dans ce segment. La similarité entre deux méthodes est calculée en utilisant la mesure de similarité cosinus sur la matrice LSI extraite à l'étape précédente.

Le couplage d'un segment est défini comme la moyenne de la similarité entre un segment et tous les autres segments de la trace, calculée en utilisant l'équation 2, où L est la longueur de la trace.

Enfin, pour une trace divisée en n segments, la fonction de fitness est représentée par l'équation 3.

$$SegmentCohesion_k = \frac{\sum_{i=begin(k)}^{end(k)-1} \sum_{j=i+1}^{end(k)} similarity(method_i, method_j)}{(end(k) - begin(k) + 1) \cdot (end(k) - begin(k))/2} \quad (1)$$

$$SegmentCoupling_k = \frac{\sum_{i=begin(k)}^{end(k)} \sum_{j=1, j<begin(k) \text{ or } j>end(k)}^{l} similarity(method_i, method_j)}{(l - (end(k) - begin(k) + 1)) \cdot (end(k) - begin(k) + 1)} \quad (2)$$

$$fitness(individual) = \frac{1}{n} \cdot \sum_{k=1}^{n} \frac{SegmentCohesion_k}{SegmentCoupling_k} \quad (3)$$

Tableau 2 Les Statistiques pour les deux Systèmes.

Systems	Number of Classes	Kilo Line of Code	Release Dates
ArgoUML v0.18.1	1,267	203	30/04/05
JHotDraw v5.4b2	413	45	1/02/04

Étude Empirique et les Résultats

Nous avons effectué une étude empirique pour évaluer l'approche que nous proposons pour identifier des concepts. Le but de cette étude est d'analyser notre approche dans le but d'évaluer sa capacité à identifier des concepts pertinents. La perspective est celle de chercheurs souhaitant évaluer comment l'approche proposée peut être utilisée lors de la maintenance et de l'évolution d'un logiciel. Le contexte consiste en une implémentation de notre approche et des données de traces d'exécution extraites de deux systèmes " open source " (JHotDraw et ArgoUML). Le Tableau 2 met en évidence les caractéristiques principales des deux systèmes.

Nous générons des traces en exerçant divers scénarios dans les deux systèmes. Le Tableau 3 résume les scénarios et montre que les traces générées vont de 6.000 à près de 65.000 appels de méthode. Les traces compressées vont de 240 à plus de 750 appels de méthode.

Tableau 3 Les Statistiques pour les Traces.

Systems	Scenarios	Original Size	Cleaned Sizes	Compressed Sizes
ArgoUML	Start, Create note, Stop	34,746	821	588
	Start, Create class, Create note, Stop	64,947	1066	764
	Start, Add text, Draw rectangle, Stop	13,841	753	361
	Start, Draw rectangle, Cut rectangle, Stop	11,215	1206	414
	Start, Spawn window, Draw circle, Stop	16,366	670	433

Nous avons besoin d'un oracle pour évaluer l'exactitude et l'exhaustivité des concepts identifiés. Nous construisons un tel oracle par un marquage manuel des traces d'exécution pour indiquer

le début et la fin de chaque fonctionnalité.

Par cette étude empirique, nous tenterons de répondre aux questions de recherche suivantes :

– RQ1 : Quelle est la stabilité des solutions, obtenues par exécutions multiples de notre approche, lors de l'identification des concepts dans les traces d'exécution ?

– RQ2 : Dans quelle mesure les concepts identifiés correspondent à celles de l'oracle ?

– RQ3 : Quelle est la précision de l'identification de concepts dans les traces d'exécution ?

RQ1 : Stabilité de GA

Les solutions sont considérées stables lorsque les segments identifiés dans une exécution existent dans la solution d'une autre exécution. Nous calculons le chevauchement entre chaque paire de segments pris de deux différentes solutions en utilisant l'indice de chevauchement Jaccard présenté dans l'équation

$$Jaccard(s_{x,i}, s_{y,j}) = \frac{|s_{x,i} \cap s_{y,j}|}{|s_{x,i} \cup s_{y,j}|}$$

Figure 4 montre les indices de chevauchement Jaccard pour deux solutions generées par deux different l'exécutions. Puisqu'un segment de la trace T1 chevauche plusieurs segments de T2, un bon

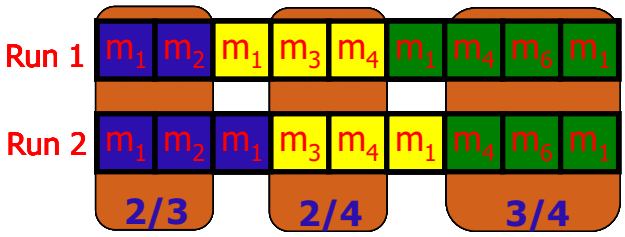

Figure 4 Les Indices de Chevauchement Jaccard pour deux Solutions

appariement est choisi en utilisant l'algorithme Hongrois. Un exemple est présenté à la Figure 5 Nous évaluons la stabilité de l'AG en calculant la similarité moyenne des segments identifiés dans dix exécutions différentes de notre approche. Le Tableau 4 présente les résultats de ces calculs. Dans l'ensemble, les moyennes de similarité pour JHotDraw varient entre 55% et 95%, avec des valeurs médianes comprises entre 70% et 84%. Ces résultats montrent que l'approche est capable

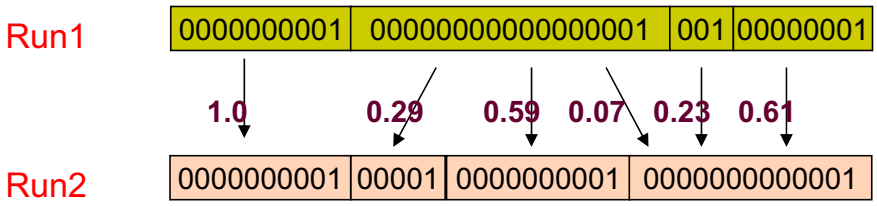

Figure 5 Chevauchement entre Plusieurs Segments

de générer des segments stables à travers de plusieurs exécutions, malgré l'espace de recherche relativement large.

RQ2 : Correspondance avec l'Oracle

Pour répondre à RQ2, nous évaluons la mesure dans laquelle les segments reflètent réellement les fonctionnalités telles qu'elles ont été marquées lors de la construction de l'Oracle. La correspondance entre les solutions obtenues, en appliquant l'approche sur les traces, et l'oracle est calculé en utilisant l'indice de chevauchement Jaccard.

Comme indiqué dans le Tableau 5 Pour certaines fonctions, par exemple, dessiner un rectangle ou un cercle, les moyennes et médianes de l'indice de chevauchement Jaccard sont très élevées, ce qui suggère que les fonctionnalités sont mises en oeuvre à travers des séquences de méthodes très cohésives. Cependant, d'autres fonctionnalités présentent de moins bons indices de chevauchements. Ces bas indices de chevauchements signifient que notre approche a été incapable d'identifier les fonctionnalités. En effet, dans certains cas, par exemple les scénarios " Ajouter le texte " dans JHotDraw et " Créer la note " dans ArgoUML, les fonctionnalités sont réalisées en adaptant la fonctionnalité " édition de texte " en tant que fonctionnalité " dessin de forme " en utilisant le design pattern Adaptateur. L'adaptation de fonctionnalités produit des séquences de méthodes que notre algorithme tend à diviser du fait de leur faible cohésion. Par conséquent, les chevauchements qui en résultent sont, logiquement, bas. Dans d'autres cas, en particulier avec des traces de ArgoUML, les moins bons résultats sont dus au fait que de longs segments sont découpés en segments plus petits et plus cohésifs. Par exemple, " Créer une classe " (ArgoUML) est divisé en cinq segments :

Tableau 4 Les Statistiques Descriptives de Similitude entre les Segments Obtenus des Différent Exécutions.

Systems	Scenarios/Features	Similarity Averages				
		Min.	Max.	Mean	Median	σ
ArgoUML	(1) Add note	0.69	0.95	0.84	0.83	0.07
	(2) Add class, Add note	0.65	0.98	0.80	0.80	0.06
JHotDraw	(1) Draw rectangle	0.55	0.96	0.76	0.76	0.12
	(2) Add text, Draw rectangle	0.54	0.93	0.72	0.70	0.10
	(3) Draw rectangle, Cut rectangle	0.73	0.93	0.85	0.84	0.05
	(4) Spawn window, Draw circle	0.67	0.86	0.76	0.76	0.04

création d'objets, ajout du type de projet, gestion d'espace de noms, paramétrage des propriétés d'objet et gestion de la persistance du diagramme. Les segments détectés par notre approche sont plus cohésifs et suggèrent que notre oracle devrait être amélioré.

Tableau 5 La Similitude (Jaccard chevauchement) entre les Segments Identifiés par l'Approche et les Segments Indiqués par l'Oracle.

Systems	Scenarios	Features	Jaccard				
			Min.	Max.	Mean	Median	σ
ArgoUML	(1)	Add note	0.15	0.39	0.28	0.27	0.08
	(2)	Create class	0.11	0.28	0.22	0.25	0.05
	(2)	Create note	0.22	0.56	0.35	0.31	0.14
JHotDraw	(1)	Draw rectangle	0.63	0.93	0.84	0.89	0.13
	(2)	Add text	0.21	0.31	0.26	0.27	0.05
	(2)	Draw rectangle	0.53	0.70	0.63	0.61	0.06
	(3)	Draw rectangle	0.42	0.76	0.64	0.72	0.14
	(3)	Cut rectangle	0.16	0.23	0.22	0.23	0.02
	(4)	Draw circle	0.54	0.96	0.85	0.88	0.14
	(4)	Spawn window	0.07	0.41	0.20	0.16	0.11

RQ3 : Précision dans l'Identification des Concepts

Nous utilisons Précision, équation pour montrer le degré de précision des résultats obtenus, par rapport à l'oracle. La précision est définie comme suit :

$$Precision(s_{x,i}, s_{y,o}) = \frac{|s_{x,i} \cap s_{y,o}|}{|s_{x,i}|}$$

où $s_(x, i)$ sont des segments obtenus par notre approche et $s_(y, o)$ sont des segments de la trace correspondante de l'oracle. Un exemple de calcul de précision est indiqué à la Figure 6 Le Tableau

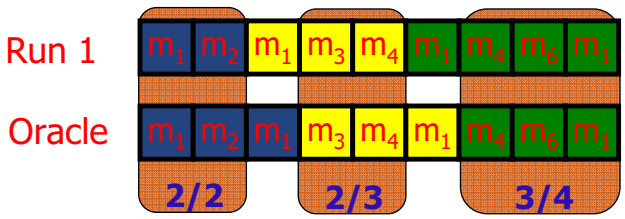

Figure 6 Un Exemple de Calcul de Précision

6 rend compte de la précision des segments identifiés par rapport à l'oracle. La précision est souvent très élevée, avec des valeurs médianes, dans la plupart des cas, supérieures à 85% et très souvent égale à 100%. Dans certains cas, la précision est plus faible mais cela s'explique aisément. Par exemple, dans le scénario (2) de JHotDraw, composé de " Ajouter du texte " et " Dessiner un rectangle ", les deux fonctionnalités sont implémentées en utilisant une séquence très similaire d'appels de méthode, ce qui les rend difficiles à distinguer.

Les résultats de cette étude empirique montrent que notre approche est stable et capable d'identifier les segments avec une haute précision. Pour les fonctions de haut niveau, l'approche divise le grand segment indiqué par l'oracle en plusieurs segments plus petits et plus cohésifs. En l'état, l'approche proposée est limitée en présence de multi-threading et de méthodes utilitaires telles que celles relatives aux événements dans l'interface graphique.

Tableau 6 La Similitude (Précision) entre les Segments Identifiés par l'Approche et les Segments Indiqués par l'Oracle.

Systems	**Scenarios**	**Features**	**Precision**				
			Min.	**Max.**	**Mean**	**Median**	σ
ArgoUML	(1)	Add note	0.91	1.00	0.97	1.00	0.04
	(2)	Create class	1.00	1.00	1.00	1.00	0.00
	(2)	Create note	1.00	1.00	1.00	1.00	0.00
JHotDraw	(1)	Draw rectangle	0.89	1.00	0.96	1.00	0.06
	(2)	Add text	0.27	0.36	0.32	0.34	0.04
	(2)	Draw rectangle	0.61	1.00	0.69	0.66	0.13
	(3)	Draw rectangle	0.73	1.00	0.94	1.00	0.11
	(3)	Cut rectangle	1.00	1.00	1.00	1.00	0.00
	(4)	Draw circle	0.81	1.00	0.91	0.95	0.09
	(4)	Spawn window	1.00	1.00	1.00	1.00	0.00

ABSTRACT

Maintenance is the last phase of software life cycle and plays an important role in the life cycle of a system. More than 50% of the cost of the whole life cycle belongs to the maintenance phase. One of the most challenging problem of software maintenance is program comprehension.

Program comprehension is a crucial part of maintenance and is a major factor in providing effective software maintenance and enabling successful evolution of a software system.

A common problem in understanding software systems is that software systems often lack an adequate documentation. Most of the time, the only available source to understand the program is the source code. Therefore, developers must resort to reading the system source code, without specific tool support but code browsers, to understand the systems and perform their maintenance and evolution tasks.

Concept or feature location and identification aim at helping developers to perform their maintenance and evolution tasks, by identifying abstractions (*i.e.*, features) and the location of the implementation of these abstractions. Specifically, they aim at identifying *code fragments*, *i.e.*, set of method calls in traces and the related method declarations in the source code, responsible for the implementation of domain concepts and user-observable features. The literature reports approaches built upon static and dynamic analyses; Information Retrieval (IR) and hybrid (static and dynamic) techniques.

This thesis proposes a novel approach to identify cohesive and decoupled fragments in execution traces, which likely participate in implementing concepts related to some features. The approach relies on search-based optimization techniques using metaheuristic algorithm, textual analysis of the system source code using latent semantic indexing, and trace compression techniques.

The proposed approach is evaluated to identify features from execution traces of two open source systems from different domains, JHotDraw and ArgoUML. Results show that the approach is stable and is generally able to locate concepts with a high precision.

RESUMÉ

La maintenance de logiciels est la dernière phase du cycle de vie des logiciels. Elle joue un rôle important dans le cycle de vie d'un logiciel car plus de 50% du coût du cycle de vie appartient à la maintenance. Un des défis importants avec la maintenance de logiciels est la compréhension.

La compréhension de logiciels est une partie cruciale de la maintenance et est un facteur majeur pour une maintenance efficace ainsi que pour une évolution réussie d'un logiciel.

Un problème commun à la compréhension de logiciels est souvent le manque d'une documentation adéquate. Quelques années après le déploiement, c'est possible que la documentation n'existe plus ou si elle existe, elle est surement dépassée. Les développeurs qui maintiennent le logiciel, la plupart du temps, sont différents de ceux qui l'ont développé. Par conséquent, les développeurs doivent recourir à la lecture du code source du système, avec rien d'autre que des navigateurs de code, pour comprendre le logiciel et accomplir leurs tâches de maintenance et d'évolution.

L'identification et la localisation de concepts ou de fonctionnalités visent à aider les développeurs à effectuer leurs tâches de maintenance et d'évolution, en identifiant des abstractions et l'emplacement de l'implémentation de ces abstractions. Autrement dit, ils visent à identifier des fragments de code source, c'est à dire l'ensemble d'appels de méthodes dans des séquences d'instructions et les déclarations de ces méthodes dans le code source, responsables de la mise en oeuvre des concepts du domaine du logiciel et des caractéristiques observables par l'utilisateur. Dans la littérature il existe des approches d'analyse statique et dynamiquve ; Recherche Documentaire (Information Retrieval - IR) et des approches d'analyse hybride (statique et dynamique).

Cette thèse propose une nouvelle approche pour identifier des fragments cohésifs et découplés dans des traces d'exécution, qui participent probablement à mettre en oeuvre les concepts liés à certaines fonctionnalités. L'approche repose sur des techniques optimisées pour la recherche en utilisant un algorithme de métaheuristiques, analyse textuelle du code source du logiciel en utilisant l'indexation sémantique latente et des techniques de compression de traces.

Pour évaluer l'approche proposée, nous avons effectué une étude empirique en appliquant l'ap-

proche proposée sur deux logiciels libres, ArgoUML et JHotDraw. Les résultats ont montré que l'approche est stable, et, globalement, localise les concepts avec une précision élevée.

TABLE OF CONTENTS

LISTE OF TABLES

LISTE OF FIGURES

LIST OF APPENDICES

CHAPITRE 1

INTRODUCTION

Maintenance is the last phase of the software life cycle but plays an important role in the life cycle. More than 50% of the cost of a system belongs to the maintenance phase. Maintenance does not only deal with the correction of faults and errors that are found in the system after its delivery. Some modifications are applied to answer new user requirement, some others are performed to make a system compatible with a new environment. There are also changes made to improve maintainability of the system. According to Lientz *et al.* in (Lientz (1980)) there are four types of maintenance:

- Corrective: fixing errors and defects in the system. The defects can result from design errors, logical errors or coding errors.
- Adaptive: adapting system to a new environment such as hardware or operating system.
- Perfective: changes that are done to answer to new user requirements.
- Preventive: activities to make a system more maintainable. These activities include updating documents, adding comments to source codes, modifying some structure in source code.

Some software characteristics affect software maintenance activities. These characteristics are system size, system age, and the structure of the system. Some of the most challenging problems of software maintenance are program comprehension, impact analysis, and regression testing. Program comprehension is a crucial part of the maintenance phase and is a major factor in providing effective software maintenance and enabling successful evolution of a software system.

Program comprehension means having the knowledge of what the software system does, what functionalities it provides, which parts of its code contribute to implementing each functionality, how the system relates to its environment, identifying places of the system that are affected by each particular change and knowing how the modified parts should work.

1.1 Problem Definition

A common problem in understanding software systems is that software systems often lack an adequate documentation. A few years after deployment, documentation may no longer exist or if it exists it is almost surely outdated. In addition the developers who perform maintenance activities often are different from those who implemented the system. Therefore, developers must resort to reading the system source code, without specific tool support but code browsers, to understand the systems and perform their maintenance and evolution. In some cases, code understanding is supported by static analysis and–or visualizations built upon static data. In other case, debugging can be used to understand the behavior of a system in a particular context and–or to locate a fault. However, manually browsing of source code, inspecting an execution trace or debugging long sequences of instructions are time consuming and daunting tasks.

1.1.1 Feature Location

Concept or feature location aims at helping developers to perform maintenance and evolution of the system, by identifying abstractions (*i.e.*, features) and the location of the implementation of these abstractions in the source code. Specifically, they aim at identifying *code fragments*, that are responsible for the implementation of domain concepts and user-observable features (Antoniol et Gueheneuc (2006); Biggerstaff *et al*. (1993); Kozaczynski *et al*. (1992); Poshyvanyk *et al*. (2007a); Tonella et Ceccato (2004)). The literature reports approaches built upon static (Anquetil et Lethbridge (1998); Marcus *et al*. (2004)) and dynamic (Wilde et Scully (1995); Tonella et Ceccato (2004)) analyses; Information Retrieval (IR) (Poshyvanyk *et al*. (2007a)) and hybrid (static and dynamic) (Eaddy *et al*. (2008)) techniques.

Static approaches mostly use *IR* techniques to locate the features in the source codes. They consider classes and methods in the source code as the documents whose terms are identifiers and comments. To locate a particular feature in the source code, the developer should apply a query describing the feature to be found in the corpus. The most similar methods or classes to the query are considered as implementing the feature.

All dynamic approaches use system execution traces to identify features in the source code. The execution traces used in these approaches are generated by executing scenarios that are designed based on system test cases or specifications. Thus, in all of these approaches, the developer is aware of the feature or features executed in the scenario. Thus none of these approaches can be used to extract the concepts in a trace that is generated by an unknown scenario where there is no information about the features that are exercised in it. Using the execution scenario, each approach has its own method to detect a particular feature. Some of them identify a feature by comparing two traces with and without the feature, some others rank the methods in a trace.

Having several traces based on the scenarios requires lots of effort and care to design precise, appropriate scenarios to generate traces that provide the approach method with enough data.

1.2 Objective of the project

The objective of this project is to propose an approach to identify all the concepts existing in a single execution trace and assign meaningful labels to concepts. The label of a located concept should represent the functionality of that concept.

The thesis proposes a novel approach to identify cohesive and decoupled fragments in execution traces, which likely participate in implementing concepts related to some features. A typical problem for which the proposed approach can be beneficial is the following. Suppose that a failure has been observed when executing a particular scenario of a software system; unfortunately the likelihood to reproduce the execution conditions for that failure is very low. Maintainers are then faced with the problem of analyzing the execution trace produced by a particular scenario and identifying high-level abstractions (concepts) that likely participate in the feature producing the unwanted behavior. Thus it can be helpful in finding problem causing concepts.

To deal with the above described scenario, the proposed approach identifies concepts composing an execution scenario by grouping together methods that are (i) sequentially invoked together and (ii) cohesive and decoupled from a conceptual point of view. The underlying assumption is that, if a specific feature is being executed within a complex scenario (*e.g.*, “Open a Web page from

a browser" or "Save an image in a paint application"), then the set of methods being invoked, implementing a specific feature (open the Web page), is likely to be conceptually cohesive, decoupled from those of other features, and sequentially invoked. We use conceptual cohesion and coupling from Marcus *et al.* (Marcus *et al*. (2008)) and Poshyvanyk *et al.*(Poshyvanyk et Marcus (2006a)).

The approach works as follows. First, we index the source code of each method of a system textually. Then, we instrument and exercise the system to collect execution traces for some scenarios related to different features and, therefore, containing sets of different concepts. We compress the traces to remove utility and cross-cutting concepts and to abstract repetitions of the same sub-sequences of methods. Then, we apply a search-based optimization technique, *i.e.*, a genetic algorithm, to split the compressed traces into cohesive and decoupled fragments. We ensure performances by parallelizing computations over multiple computers. Finally, we assign a representing label to each detected concept. The label represents the functionality of a concept.

1.3 The Contributions of This Work

In general, the contributions of this thesis are:

1. A novel approach combining IR techniques, dynamic analysis, and search-based optimization techniques to identify concepts into execution traces;
2. An empirical study that shows the applicability and the performances of the proposed approach in identifying concepts into execution traces of two systems, JHotDraw and ArgoUML. Results indicate that the approach is able to identify concepts (with a precision in most cases greater than 80%), while the overlap with a manually-built oracle varies depending on the cohesiveness of the concepts to be identified.
3. Parallelization of the search-based optimization technique used in this work (a genetic algorithm) by implementing four client-server architectures conceived to improve performance and reduce GA computation times to resolve the concept location problem. We could find an architecture that reduces extremely the execution time.

We published our work at the The European Conference on Software Maintenance and Reengineering (CSMR) 2010. The article entitled *"A Heuristic-based Approach to Identify Concepts in Execution Traces"* by Fatemeh Asadi, Massimiliano di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc (Asadi *et al*. (2010b)).

Our second article that explains the Distributed Architectures for GA parallelization is published at the 2nd International Symposium on Search Based Software Engineering (SSBSE 2010). The article entitled *"Concept Location with Genetic Algorithms: A Comparison of Four Distributed Architectures"* by Fatemeh Asadi, Giuliano Antoniol, Yann-Gaël Guéhéneuc (Asadi *et al*. (2010a)).

The following is a detailed description of our contributions in this thesis:

1. At the beginning, we had a basic problem definition which we improved during our study based on our observations and findings.
2. We have tried several equations to find an appropriate fitness function that guides the search to find a good segmentation with meaningful segments. We have performed several experiments by using each equation and we improved the equation based on the results of these experiments.
3. We applied several meta heuristic search algorithms namely, hill-climbing, simulated annealing and genetic algorithms. For the first two ones we used our own implementations and for the genetic algorithm we used GAlib, a Java genetic algorithm library, and we optimized its computations. We also defined the transformations which were performed by each algorithm to improve the solution.
4. We defined several representation of the problem, and we performed some experiments with each one to find a proper representation that helped to find meaningful segmentations.
5. To verify the stability of the search algorithms we have proposed and implemented two verification algorithms, one used with our local searches results and the other one with the GA results.
6. We implemented and used our own run length encoding algorithm to compress the execution traces,.

7. We speeded up the execution of our approach by parallelizing GA computations. We implemented four different client-server configurations to distribute GA computations among several servers. Having performed some experiments, we could find the configuration that reduces enormously the execution time.

1.4 Plan of the thesis

The remainder of this Thesis is organized as follows: Chapter 2 presents some previous works. Chapter 3 gives a brief explanation of supporting techniques. Chapter 4 describes the approach. Chapter 5 presents an empirical study and reports its results with some discussions. Chapter 6 explains the distributed architectures we used to parallelize the GA computations in order to reduce the execution time. Chapter 7 concludes the thesis and outlines future work.

CHAPITRE 2

LITERATURE REVIEW

Although many concept identification approaches exist, none of these approaches attempts to identify all concepts in a single execution trace *automatically*. This chapter presents some of previous work in locating concepts in source code. Some of them use only static analysis by analyzing data extracted from source code, the others are based on dynamic analysis and try to locate features by extracting execution traces and analyzing them. There are also approaches that use both static and dynamic analyses.

2.1 Static Approaches

Static analysis use the data extracted from source code. The data can be structural data used to generate program dependency graphs (Chen et Rajlich (2000)) or clustering components within a software system (Maletic et Marcus (2001)) or data extracted by using information retrieval techniques (Marcus *et al*. (2004)).

Chen and Rajlich (Chen et Rajlich (2000)) developed an approach to identify features using Abstract System Dependencies Graphs (ASDG). In C, an ASDG models functions and global variables as well as function calls and data flow in a system source code. They used only static data to identify features and a manual process. Their approach is not completely automatic. It is based on the interactions between the computer and the programmer. The computer is responsible for generating the dependency graph of the system and updating the search graph. The search graph is the part of the ASDG that was visited during the search. The programmer is responsible for locating starting points, choosing next component for visit, exploring the source code, dependency graph, and documentation, to understand the component and decide whether it is relevant or irrelevant to the feature and finally checking if all components related to the the feature have been detected.

In (Maletic et Marcus (2001)) Maletic *et al.* proposed an approach to support program comprehension by using semantic and structural information. Semantic information is provided by using Latent Semantic Indexing (LSI) and is used to define a semantic similarity measure between software components. The similarity measure is then used to cluster components in the software system. Structural information is used to assess the semantic cohesion of the clusters. This approach does not aim to locate features in the source code. It only tries to provide the maintainer with some organized systematic data about the system that helps them in understanding a program.

Marcus *et al.* (Marcus *et al*. (2004)) proposed a LSI based approach to locate concepts in the system source code. LSI is used to map the concepts that are expressed in natural language by the programmer, to the related parts of the source code. Using LSI mitigates the problem of polysemy and synonymy that existed in previous approaches like grep based techniques. The other advantage of using LSI is that the method is not dependent on a specific programming language, thus the source code preprocessing is simpler than building the dependency graph.

The corpus is built by considering system methods as the documents and terms that exist in the identifiers and comments as LSI terms. Then the concepts are located by applying queries to the corpus. The method can be used in two distinct ways : one way is based on automatically generated queries. In the other way the user formulates queries and directly queries the system.

The results obtained from a case study that applied this technique to locate concept in NCSA Mosaic showed that the proposed method outperformed the previous methods based on regular expression searches and searches on the program dependency graph.

2.2 Dynamic Approaches

Dynamic feature location is based on collecting and analyzing execution traces, which identify methods that are executed for a specific scenario. In their pioneering work, Wilde and Scully (Wilde et Scully (1995)) presented the first approach to identify features by analyzing execution traces. They used two sets of test cases to build two execution traces, one where a feature is exercised and another where the feature is not. They compared the execution traces to identify the feature in the

system. In their work, they used only dynamic data to identify features and they performed no static analysis of the program.

As they mention in the paper, their proposed approach has some limitations. First, it cannot be used for the functionalities that are always present in the program. Second, choosing the test cases should be done very carefully and they should be classified according to the functionalities they present. Otherwise, the obtained results will be meaningless. Finally, their proposed approach is effective in detecting the parts of source code that uniquely belong to a particular feature but it does not guaranty to find all the components and source code that participate in implementing the feature. In other words, the results obtained by using their method presents only starting points.

Wong *et al.* (Wong *et al*. (2000)) analyzed execution slices of test cases to identify features in source code. They defined an execution slice as a group of basic programming blocks and a feature as "an abstract description of a functionality given in the specifications". To locate parts of source code that implemented a feature, they executed the program by an input that forced to exercise that feature. Finding basic blocks (execution slices) related to a feature, they tried to measure the quantitative interaction between a feature and a program component. In order to do that, they defined three metrics : *disparity*, *concentration* and *dedication*. *Disparity* says how much a feature is related to a component. *Concentration* shows how much of the code in a component is dedicated to implement a feature and *dedication* says how much of the code implementing a feature, exists in a component. By using these metrics, they believed that their method gave a bigger overview of how the implementation of a feature is spread over a system. It helps to find the components implementing a feature compared to its previous work that only found starting implementation points. They used a special tool called $\chi^{Suds^{TM}}$ to calculate these metrics. To show how their metrics are useful in program understanding and how they help programmers in software maintenance activities, they presented a case study that was performed on a software system called *SHARPE*[1]. They chose six features from the regression test suit of *SHARPE* and they found the slices related to each one of these six features. Then they asked some experts who knew the system very well to evaluate

1. Symbolic Hierarchical Automated Reliability and Performance Evaluator

their findings. The feedback of the experts said that they agree using the introduced metrics in the paper were helpful in detecting components that collaborate in implementing a feature. Moreover, the experts believed that using these metrics helped them to capture more precisely the places in the system that are related to each feature than using their own knowledge of the system which provides only a qualitative understanding of where each feature is implemented in the system.

Wilde's original idea was later extended in several works (Antoniol et Gueheneuc (2006); Poshyvanyk *et al*. (2007a); Edwards *et al*. (2004); Eisenberg et Volder (2005)) to improve its accuracy by introducing new criteria on selecting execution scenarios and by analyzing the execution traces differently.

Edwards *et al.*(Edwards *et al*. (2004)) suggested a solution for the feature location problem for distributed systems. Since distributed systems usually run continuously by executing different features, the concept of "time interval" should be used instead of the concept of "test case". Determination of intervals in distributed systems is a complex task and it is difficult to determine the correct order of events. Thus they defined a time interval based on causal ordering of events. Then they compared the execution traces between the determined intervals to locate a particular feature.

Eisenbarth *et al.* in (Eisenberg et Volder (2005)) introduced a feature identification approach by using test cases. First, they partitioned test cases in feature-specific subsets manually and used them to generate traces. They believed that the previous techniques which relied on comparison between two traces, are very sensitive to the quality of inputs. Thus they proposed a heuristical ranking-based technique to decrease the sensitivity to input. The ranks are used to describe the relevancy of a method to a particular feature. They defined three different ranks : *Multiplicity*, *Specialization* and *Depth*. *Multiplicity* of a method in a test set is the ratio of number of execution of the method in the test set to the number of execution of the same method in all the other test sets. *Specialization* shows how a method is special for a test set or how it is common between all the test sets. The traces that are analyzed in this approach are generated by running test cases of the system such that each feature of the system is executed at least with one of test cases. Then by computing the three ranks mentioned above they detected the methods that are relevant to a particular feature.

They developed a tool for the Java programming language and applied this tool to three different systems.

2.3 Hybrid Approaches

Both static and dynamic techniques have their own limitations. In general, dynamic techniques are conservative in nature, as execution traces are often very large and contain a lot of noise. Selection of proper test cases or scenarios to be executed is another problem of these techniques. On the other hand, static analysis can rarely identify methods contributing to a specific execution scenario exactly. In fact each type of analysis collects data that would be unavailable to the other analysis but can improve the performance of the feature location process. In the other words, each approach can be considered as a complement to the other. Recent works focused on a combination of static and dynamic data (Eaddy *et al*. (2008); Poshyvanyk *et al*. (2007a)), in which, essentially, the problem of features location from multiple execution traces is modeled as an IR problem, which has the advantage to simplify the location process and, often, improves accuracy (Poshyvanyk *et al*. (2007a)).

Antoniol *et al.* (Antoniol et Gueheneuc (2005)) proposed a statistical analysis technique to detect features in multi threaded object oriented programs. Their approach used both static and dynamic data collected from a program source code and its executions. Static analysis is used to build an architectural model of a program from its source code. This model is later used to create feature models. In dynamic data collection phase, they used processor emulation to improve the order and the precision of collected data. Since all the events occurred while executing a scenario are not interesting, regarding to a particular feature, they removed some obviously unrelated events from the execution trace by using Knowledge-based Filtering. Then, they used probabilistic ranking technique to classify events as relevant or irrelevant to a feature. At the end, they used the events related to each feature to build micro-architectures that highlighted the parts of the source code that implement that feature. By comparing and highlighting differences among micro-architectures, maintainers can understand and compare behaviors of different functionalities or of

a same functionality with different scenarios.

An extension of this work was presented later in (Antoniol et Gueheneuc (2006)) in which they improved their previous approach by using the concept of epidemiology of diseases in locating the features. Their assumption was that the events that are frequent in the scenarios in which a particular feature is exercised, can be related to that feature with high probability. Thus they used an epidemiological metaphor besides probabilistic ranking technique to classify events as relevant and irrelevant and also to rank the relevant events to associate top ranked events with a feature. The results obtained by using this method were more accurate than their previous approach's results.

In (Poshyvanyk *et al*. (2007b)) Poshyvanyk *et al.* proposed a hybrid method, called *PROMESIR*, by combining the static approach presented in (Marcus *et al*. (2004)) and the dynamic approach presented in (Antoniol et Gueheneuc (2005)) and (Antoniol et Gueheneuc (2006)) to find starting points in impact analysis. *PROMESIR* static approach is an information retrieval-based approach that uses latent semantic indexing to find related methods and classes to a particular feature. The LSI-based approach considers each method and class as a document composed by identifiers and comments. To find methods and classes that implement a particular feature, a developer formulates a query by selecting a set of words that describe the feature. The LSI-based system tries to find related documents (classes and methods) to the features described in the query by computing the similarity between the documents in the corpus and the query. Then the retrieved documents are ranked based on the similarity measures. The dynamic approach is the scenario-based probability approach (SPR) explained in (Antoniol et Gueheneuc (2005),Antoniol et Gueheneuc (2006)). Faunally, they considered SPR ranking and LSI ranking as the judgement of two experts and combined them using the formula presented in (Jacobs (1995)).

Each one of these two approaches collects and analyzes data that would be unavailable to the other approach but can improve the performance of feature location process. Using LSI, developers can query static documents and obtain a ranked list of code fragments related to a feature. Using SPR, developers analyze dynamic traces of the execution of different scenarios and obtain a list of entities (methods and classes), again ranked according to their relevance to a feature of inter-

est. Thus each one can be considered as a complement for the other one and using both together improves the performance and precision of feature location tasks.

To show the superiority of their method, the authors performed a case study for bug location in two large open source programs : Eclipse and Mozilla. The case study showed that LSI and SPR complement each other, and the results obtained with the combined techniques are better than those of any of the techniques used independently.

Eisenbarth *et al.* (Eisenbarth *et al*. (2003)) proposed an approach by combining both static and dynamic data to identify features. First, they performed a dynamic analysis of a system using profiling techniques to identify features, similar to Wilde and Scully's approach. Then, they applied concept analysis techniques to link features together, and to guide a static analysis that narrowed the scope of the identified features. In fact, they used the results obtained from dynamic analysis just to guide the analyst in the static analysis and not as a definite answer. They experimented their approach on two C systems with good results.

The test cases used as scenarios in the proposed approach should be designed carefully and precisely by a domain expert. The domain expert should design several scenarios, all triggering the same feature but with different sets of inputs. To obtain effective and efficient coverage, it is needed to build equivalence classes of relevant inputs. Thus, to understand a system by using this approach, having a vast previous knowledge of the system is necessary.

This approach is a semiautomatic approach where the tasks related to the dynamic analysis are performed automatically but the static analysis that consists of interpretations of concept lattice and static dependency analysis are performed by an analyst.

Salah and Mancoridis (Salah et Mancoridis (2004)) used both static and dynamic data to identify features in Java systems. They went beyond feature identification by creating feature-interaction views, which highlight dependencies among features. They defined four types of views : object-interaction, class-interaction, feature-interaction, and feature-implementation. Each view abstracted preceding views to present only the most relevant data. Feature-interaction and implementation views highlighted relationships among views. Their work was extended to allow feature identifica-

tion and evolution analysis in large-scale systems, *e.g.*, Mozilla (Salah *et al*. (2005)).

Yet, Liu *et al.* (Liu *et al*. (2007)) showed that a single trace suffices to build an IR system and locate useful data. Their proposed method, called *SITIR*, is a semiautomated method in which user and computer interact with each other to find the methods related to a feature. The approach starts by using LSI to capture relations between terms (identifiers and comments) and documents (methods) in the source code. Then the user formulates a scenario that executes the desired feature and collects the execution trace. In the next step the user should formulate a query describing the desired feature. Using terms similar to the terms used in the system is helpful in finding more precise results. Applying the query on the execution trace, all the methods in the trace are ranked based on their textual similarity to the query. Then the user examines the source code starting with the methods at the top of the sorted list. Inspecting the source code, the user can reformulate the scenario or improve the query.

To evaluate the performance of their proposed approach, the authors performed two case studies, one by using JEdit and the other one by using Eclipse. For each of these two, they tried to locate three features that needed to be modified. They defined the effectiveness of a feature location technique by considering the ranking of a first method that participates in implementing the desired feature. Thus a feature location technique is considered better than another one "if it returns at least one method relevant to the feature on a better position in the list of ranked results". The results obtained from these two case studies showed that SITIR outperforms the approach based only on LSI or SPR. The results of SITIR are very close to the ones of PROMESIR. However, SITIR is much simpler than PROMESIR and it uses only one unique trace, instead of multiple traces in PROMESIR, to detect the desired features. The way in which the static and dynamic analysis results are combined, is also more transparent in the SITIR. It is also indicated that defining the scenarios in SITIR is straightforward as they do not have to be very precise and SITIR is less sensitive to poor user queries than LSI alone.

A brief summary of all these approaches is presented in table 2.1.

We share with previous work the use of dynamic data and IR techniques to identify features.

Table 2.1 Summary of feature location approaches

Approach	Static	Static Use	Dynamic	# Of Traces	Dynamic Use
Wilde(Wilde et Scully (1995))	-	-	✓	Two traces	Comparing
Wong(Wong *et al*. (2000))	-	-	✓	Multiple Traces	Ranking
EdwardsEdwards *et al*. (2004)	-	-	✓	Two Traces	Comparing
AntoniolAntoniol et Gueheneuc (2005)	✓	Extract architectural model	✓	Multiple Traces	Probabilistic Ranking
PoshyvanykEisenberg et Volder (2005)	✓	LSI	✓	Multiple Traces	SPR Ranking
ChenChen et Rajlich (2000)	✓	ASD^2 Graphs	-	-	-
EisenbarthEisenbarth *et al*. (2003)	✓	Concept Analysis	✓	Multiple Traces	profiling
EisenbarthEisenberg et Volder (2005)	-	-	✓	Multiple Traces	Ranking
SalahSalah et Mancoridis (2004)	✓	?	✓	Multiple Traces	Interaction Views
LiuLiu *et al*. (2007)	✓	LSI	✓	Single Trace	Query Answering
MarcusMarcus *et al*. (2004)	✓	LSI	-	-	Query Answering

However, instead of querying traces using an IR technique, *e.g.*, similar to Poshyvanyk *et al.* (Poshyvanyk *et al*. (2007a)), we determine cohesive and decoupled fragments likely being relevant to a concept *automatically*. Our approach is based on two conjectures not yet fully investigated : (1) methods helping to implement a concept are likely to share some linguistic information ; (2) methods responsible to implement a feature are likely to be called close to each other in an execution trace. Therefore, the conceptual coupling of methods participating in a concept should be high and these methods should appear relatively close together in the execution trace. The first conjecture is grounded on the findings published in (Poshyvanyk *et al*. (2007a)) and other publications based on IR to locate features and concepts. IR-inspired works assume some form of commonalities between a query and linguistic information of entities. Locality of concept manifestation in traces is more questionable, however we believe unlikely that a user-observable feature or concept, not constituting a crosscutting concern, will be uniformly spread in a trace.

CHAPITRE 3

SUPPORTING TECHNIQUES

3.1 SBSE and Meta-heuristic Algorithms

Search based software engineering (SBSE) seeks solution to software engineering problems by reformulating them as search problems and applying metaheuristic search techniques.

Meta-heuristic search techniques are used to solve problems where solution must be found in large search spaces. They explore the large search space iteratively and try to improve the current solution by using fitness functions. The final solution obtained by using a metaheuristic search technique may not be the optimal solution but it is usually what is "close" to the optimal solution. The reason of using metaheuristic search techniques instead of exact optimization techniques is that applying the exact optimization techniques, such as linear programming to large scale software problems is not practical. Metaheuristic search techniques are used in different software engineering areas such as, software testing ,requirements analysis, software design, software development, and software maintenance.

There are two main categories of metaheuristic search techniques, local search techniques and evolutionary algorithms.

Local search techniques are used to solve the problems

- for which a neighborhood relation can be defined in the search space and
- that can be formulated as finding a (near) optimal solution among a number of candidates in the defined neighborhood

Local search techniques move from one solution to the other in the search space until finding an optimal (maybe local optimal) solution or reaching a stopping criteria. A local search algorithm starts from a candidate solution and then iteratively moves to a neighbor solution. Typically, every candidate solution has more than one neighbor solution. The choice of moving to which neigh-

bor depends on the policy defined by the search algorithm. For instance in some algorithms, only moving to an improving neighbor is accepted. They may check which neighbor improves the solution more or may choose the first detected neighbor that improves the solution without verifying the others. In some others, moving toward a non-improving solution can happen according to a probability function to explore more of the search space.

Evolutionary algorithms (EAs) are search methods that take their inspiration from biological evolution and use operations such as reproduction, mutation, recombination, and selection. EAs involve a search from a population of solutions. They start with an initial population that is randomly generated. Then, in each iteration, they evolve the population into the regions of the space that contain more optimal solutions. The evolution of population, generation of next population, is done in several steps. First a competitive selection is done to get rid of poor solutions. Then the solutions with high fitness are recombined with other solutions by swapping parts of a solution with another. Solutions may also mutate by making a small change to a single element of the solution. To use any metaheuristic search technique, the two following tasks should be performed(Harman *et al*. (2007)) :

1. Choosing an appropriate representation of the problem that can be used by the metaheuristic algorithm. For example if the problem is finding the test data for the Triangle method in algorithm 3.1 and we decide to solve this problem by using a genetic algorithm, we can represent this problem as a population of chromosomes where each chromosome consists of three integer genes.

2. Defining a fitness function. A fitness function is an equation that guides the search to improve the solution. They are generated according to the problem goal. In fact, we translate the problem criteria to the fitness function. The definition of fitness function is not always straightforward. There is no exact approach or standard that shows how a fitness function should be defined for a given problem. We are sometimes obliged to define and try several fitness functions to find an appropriate one.

   When the search algorithm generates a new solution, it evaluates the solution by calculating

the fitness. The algorithm uses the fitness value either to know if the solution has been improved and it can be accepted as the current result (hill climbing) or to rank the solution against the other ones (genetic algorithm).

Algorithm 3.1. Triangle

```
boolean Triangle(int x, int y, int z) {
    boolean isTriangle ;
    if(x + y > z&&x + z > y&&y + z > x)
        isTriangle = true ;
    else
        isTriangle = false ;
    return isTriangle ;
}
```

We implemented our concept location approach, explained in chapter 4, using three metaheuristic algorithms : hill climbing, simulated annealing, and genetic algorithm. In the following sections we explain each of these algorithms.

3.1.1 Hill Climbing

Hill climbing is one of the local search techniques. It relies on the neighborhood of the current solution. The first steps in using hill climbing to solve a problem are defining the solution search space and the goal of the search. The algorithm looks for the optimal solution in the defined search space. To know the value of the solutions with respect to the defined objective, each solution is evaluated by the fitness function. The fitness function $f : S \rightarrow R$ is defined based on the objective of the search and it assigns a real value R to any solution S.

Hill climbing starts with an initial solution that is generated randomly. At each iteration, the algorithm chooses from the current solution's neighborhood a neighbor that improves the fitness value and considers it as the new current solution. There are two choices for choosing a neighbor as the next solution. In the next ascent hill climbing, the algorithm moves to the first neighbor that

improves fitness. In steepest ascent hill climbing, the algorithm checks all the neighbors and moves to the neighbor that improves fitness more than others.

The algorithm continues working until there is no neighbor that improve fitness. This is the stop point for the algorithm and it shows that the algorithm has reached a (possibly local) optimum. A simple schema of hill climbing is shown in algorithm 3.2 :

Algorithm 3.2. Pseudo-code of Hill Climbing

```
currentNode = randomeStartNode() ;
loop do
    improved = false ;
    N = NEIGHBORS(currentNode) ;
    currEval = EVAL(curentNode) ;
    for all n in N
        if (EVAL(n) > currEval)
            currentNode = n ;
            currEval = EVAL(n) ;
            improved = true ;
            break ;
while(improved) ;
```

The problem with the hill climbing approach is that the hill located by the algorithm may be a local optimum that is much poorer than the global optimum in the search space. In fact this algorithm has a high tendency to converge to the local optimum that is close to the initial location in the search space (Figure 3.1).

A possible solution to this problem is to repeatedly restart the algorithm by using new initial solutions, in other words starting the movement from different locations in the search space. This approach is called multiple-restart hill climbing. Using this approach we can hope to find the solution closer to the global optimum but there is still no guarantee of finding the best possible solution, ı.e the global optimum.

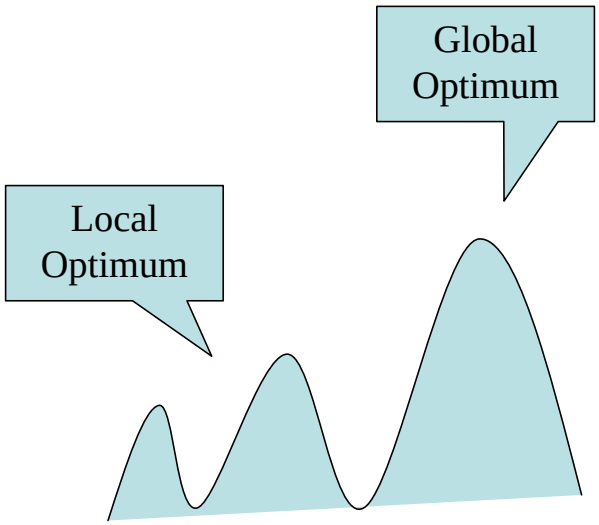

Figure 3.1 Local optimums in hill climbing

3.1.2 Simulated Annealing

Simulated annealing can be considered as an improved version of hill climbing. Simulated annealing is used to avoid the problem of local optimum by allowing the moves towards the neighbors that have poorer fitness. The algorithm simulates the process of annealing a metal. In the annealing process, a highly heated metal cools slowly to become harder. In high temperatures, the atoms can move freely. As the temperature decreases the atoms are limited in their movement and have less freedom.

In a simulated annealing algorithm, the cooling process is simulated by a probability function. In the first iterations, the probability of accepting solutions worse than the current solution is high. By increasing the iteration number, the probability of accepting less fit solutions is reduced, simulating the lower temperature in the annealing process. In the last iterations, the algorithm works as a hill climbing algorithm choosing only the solutions that improve fitness. A simple schema of simulated annealing is shown in Algorithm 3.3.

Algorithm 3.3. Pseudo-code of Simulated Annealing

randomly select currentNode

repeat

$j \leftarrow 0$

repeat

$T \leftarrow T_{max} \cdot e^{-jr}$

select $nextNode$ *in the neighborhood of* $currentNode$

if $fitness(nextNode) > fitness(currentNode)$

$currentNode \leftarrow nextNode$

else if random $[0, 1) < e^{\frac{fitness(nextNode) - fitness(currentNode)}{T}}$

$currentNode \leftarrow nextNode$

until $T < T_{min}$

3.1.3 Genetic Algorithm

The genetic algorithm is one of the evolutionary algorithms and also the most famous one. It is the most applied search technique in SBSE (Harman *et al*. (2007)). It works by using the concepts of population and recombination. Such as other evolutionary algorithms, in a genetic algorithm, each iteration involves the evolution of the whole population, in contrast with the local search techniques where in each iteration, only one individual is evolved.

The genetic algorithm starts with a definite number of individuals that are generated randomly. The number of individuals depends on the nature of the problem. The number of individuals in all the generations will be the same as the initial population.

In each iteration, some of the individuals of the current generation are selected to generate the next generation. Let us refer to them as parents. The parents are chosen by using a fitness-based probability function. Using this function, fitter individuals have a greater chance to be selected as parents. The reason of choosing good quality parents is that good quality parents have more chance to produce good quality descendants. The selected parents then mate using the genetic operators. A child generated by these operators share many characteristic with its parents. It is expected that the average fitness of the new generation increases since the best individuals have been selected to

generate the new population.

There are two genetic operators used in the mating phase to generate the next generation : crossover and mutation.

Crossover

Crossover is a genetic operator that combines two parents to produce two new chromosomes (offspring). The idea is that the new offspring may be fitter than both its parents if it inherits the best characteristic from each of its parents. Crossover occurs while evolution according to crossover probability. There exist several types of crossover operators such as : one-point, two-point, uniform, arithmetic, and heuristic.

One-Point Crossover

In one-point crossover the operator selects randomly a split point within the chromosomes. It splits both chromosomes at this points. The new offsprings are created by exchanging the tails of both chromosomes. An example of one-point crossover is illustrated in Figure 3.2.

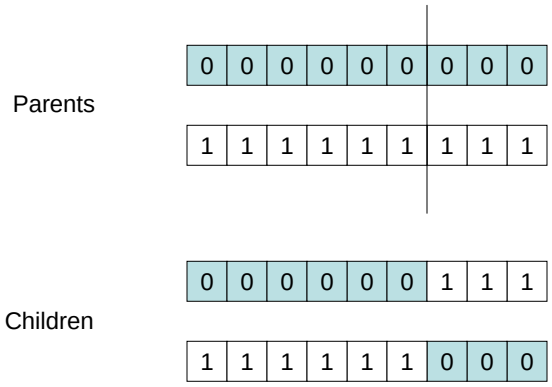

Figure 3.2 One-Point Crossover

Two-Point Crossover

This operator selects randomly two split points within the chromosomes and splits the chromosomes at these points. Then the chromosomes exchange the segment located between these two split points and generates two new offspring. An example of two-point crossover is illustrated in Figure 3.3.

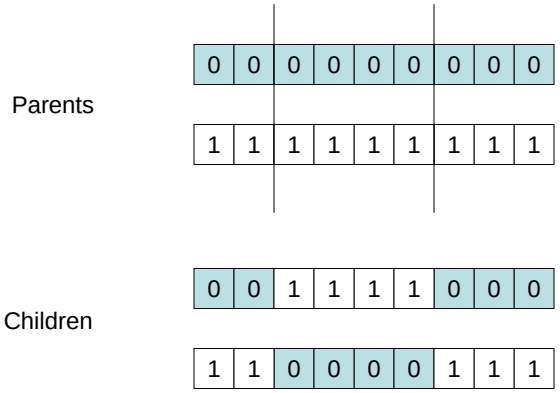

Figure 3.3 Two-Point Crossover

Uniform Crossover

In uniform crossover the parent chromosomes mix with each other at gene level rather than segment level as in the two previous crossover methods. The uniform crossover considers the first child, and for each of its genes, it flips a coin to decide from which one of its parents, the first child should inherits this gene, then the other parent's gene is assigned to the second child. An example of two-point crossover is illustrated in Figure 3.4.

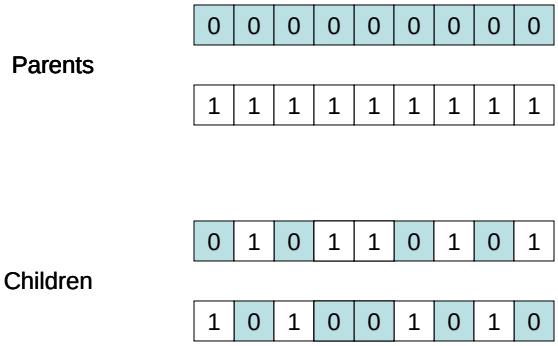

Figure 3.4 Uniform Crossover

Whole Arithmetic Crossover

Whole arithmetic crossover is useful for individuals that are represented as vectors of real values. It produces the children using the following equations :

$$\begin{aligned} child_1.gene(i) &= a \times parent_1.gene(i) + (1-a) \times parent_2.gene(i) \\ child_2.gene(i) &= (1-a) \times parent_1.gene(i) + a \times parent_2.gene(i) \end{aligned}$$

Where a is a weighting factor selected randomly in the interval [0, 1]. An example of arithmetic crossover is illustrated in Figure 3.5

Mutation

Mutation is another genetic operator that alters one or some gene values in a chromosome. This operator can result in generating a new chromosome. Mutation is important because it helps to prevent the population from converging to a local optimum. Mutation probability should be low, lower than or equal to 0.05, because a high value probability drives the search close to a random search. An example of applying a mutation operator is illustrated in Figure 3.6.

There are different types of mutations such as flip-bit for binary genes ; boundary, uniform, and non-uniform for integer and float genes ; and character mutation for character genes. In flip-bit

Figure 3.5 Arithmetic Crossover, $a = 0.7$

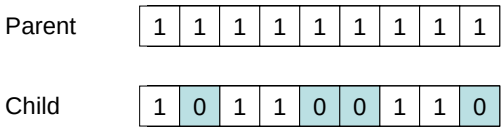

Figure 3.6 Mutation

mutation, the mutation operator simply inverts the value of the selected gene. In other words it replaces a 1 with a 0 and vice versa. In boundary mutation the value of the selected gene is replaced by the lower bound or upper bound defined for that gene. The choice of replacing with the upper or lower bound is done randomly. In uniform mutation, a selected gene's value is replaced with a value that is selected randomly between the user defined lower and upper bound. Non-uniform mutation draws a random number with a Gaussian distribution and then modifies it a bit for each gene separately. The mutation operator for character genes selects two genes randomly and exchanges their values.

Elitism

Elitism is the act of copying the best chromosome or some of the best chromosomes in the population of the next generation. Elitism guarantees the survival of the best solutions from one generation to the next one. Copying the best chromosomes into the next population is done before

performing any crossover operation. The best chromosomes are not removed from the population so they have a chance to be selected as crossover operands. During the final control to generate the next generation, the mutation is not applied to the fittest chromosome. Thus, we are sure that the best chromosome is transferred to the next generation.

Fitness Evaluations

The fitness evaluation is performed for all the individuals of the population. The evaluation of the fitness depends directly on the problem definition and goal of the search. In some cases it is needed to execute a program that receives the individuals as the input, do some processing using the data in the individual, and return some kind of result. The results are collected by the GA then evaluation is performed according to the collected results. An example of evaluating the fitness in this way, is using GA to automatically generate test cases for white box testing. In other cases the fitness evaluation is performed according to the data that are extracted from chromosome.

Depending on the search strategy, some individuals are transferred to the next generation without being affected by any of the genetic operators. Since fitness evaluation is a time consuming process, it would be the waste of time to reevaluate the fitness for these non-modified individuals. Thus, a mechanism may be needed to indicate if the current individual has been modified since the last generation and if its fitness should be reevaluated or not.

Stopping criteria

The generation process continues until reaching a terminating condition. While evolving the generation, it is possible that the population converges to a small area in the search space. In this situation, the fitness values evaluated for all the individuals are very close together and the diversity of the solutions is very low. This situation can be considered as a stopping criteria.

In some cases the number of iterations is defined by the user and the evolution process stops after a certain number of iterations.

A terminating condition can also be defined as finding a solution that satisfies the goal of the

search. However, it is possible that the population converge to an area where there is no solution to satisfy all the defined constrains. To avoid being in a loop, a maximum number of iterations should be defined. Defining a maximum number of iterations prevents the algorithm from running indefinitely. A simple schema of genetic algorithm is shown in algorithm 3.4.

Algorithm 3.4. ***Pseudo-code of Genetic Algorithm*** *4.2*

begin GA

$g := 0$ *{ generation counter }*

Initialize population P(g)

Evaluate population P(g) { i.e., compute fitness values }

while not done do

g :=g+1

Select P(g) from P($g - 1$*)*

Crossover P(g)

Mutate P(g)

Evaluate P(g)

end while

end GA

3.2 Information Retrieval and Latent Semantic Indexing

Information retrieval (IR) is the technique of extracting information from documents. It is the task of finding relevant documents in a large document database to a given query. It accelerates the process of searching documents by representing them with mathematical models. The different models used in the information retrieval process to represent the documents can be classified into the following categories(Frakes et Baeza-Yates (1992); Singhal (2001)) :

– Set-theoretic models : they represent documents as sets of words or phrases. Similarities are calculated based on set-theoretic operations on the representative sets. Common models in this category are the Standard Boolean model, the Extended Boolean model and the Fuzzy

retrieval model.

- Algebraic models : they represent documents and queries as vectors. The similarity between a query and a document is the cosine value of the angle between their vectors. Common models in this category are the Vector Space Model, the Generalized vector space model, the (Enhanced) Topic-based Vector Space Model, the Extended Boolean model, and the Latent Semantic Indexing.
- Probabilistic models : they consider the information retrieval process as a probabilistic inference. Similarities are defined as probabilities that a document is relevant for a given query by using probabilistic theorems like Bayes' theorem. This category includes the following models : Binary Independence Model, Uncertain inference, Language models, Divergence-from-randomness model and, Latent Dirichlet allocation.

Using any representation model, an information retrieval process begins when a user sends a query into the system. A query can be any statement of information-needs. A well known example of a system using IR techniques to perform the search between documents is the *Google* web search engine.

In information retrieval, a query does not identify some objects as its perfect matches. In other words, the retrieved documents may not match completely with the query but they are relevant to it by a relevancy degree. Thus, most IR systems compute a score that shows how well each object matches the query and objects are ranked according to this computed value. Then the top ranking objects are shown to the user as the query results.

3.2.1 A Simple Example of the Vector Space Model

Before describing the vector space model, let us explain basic information retrieval with a simple example. Suppose that we have the following documents :

- Document (1) I love fish.
- Document (2) I eat fish and I hate cats.
- Document (3) I love cats and dogs love cats.

- Document (4) I do not eat fish but the cats eat fish and the dogs do not hate fish.
- Document (5) The cats do not love the dogs, but the dogs love the fish.
- Document (6) The dogs hate the cats.
- Document (7) The cats hate the fish.
- Document (8) Not fish, not cats, I love dogs.

We want to create the vector-space model corresponding to these documents. By removing the articles propositions, propositions and conjunctions, we have the following terms in the vector-space dictionary : "I", "love", "fish", "eat", "hate", "cats", "dogs", "do" and "not". After extracting the terms from the documents, a document-term matrix is generated. The generated document–term matrix is a matrix that contains the weight of each term in each document. The weight of a term in a document can be defined in different ways. It may be presented only by 0 or 1 to show the presence or absence of a term in a document. The weight can be also defined as the frequency of the presence of a term in a document. There are some other weights such as $TF - IDF$ (Lalmas *et al.* (2006))that indicates how important a term is to a document in comparison with other documents in the set of documents (section 3.2.4). In the document–term matrix of this example, we use the occurrence frequency of a term in a document as the weight of that term in the document. The document–term matrix of the example is shown in 3.1

Table 3.1 Document-Term Matrix of the Given Example

	I	**Love**	**fish**	**eat**	**hate**	**cats**	**dogs**	**do**	**not**
Document (1)	1	1	1	0	0	0	0	0	0
Document (2)	2	0	1	1	1	1	0	0	0
Document (3)	1	2	0	0	0	2	1	0	0
Document (4)	1	0	3	2	1	1	1	2	2
Document (5)	0	1	1	1	0	1	2	1	1
Document (6)	0	0	0	0	1	1	1	0	0
Document (7)	0	0	1	0	1	1	0	0	0
Document (8)	1	1	1	0	0	1	1	0	2

Each row in the term–document matrix is the vector assigned to the corresponding document and contains the weight of the terms in that document.

Suppose that a user wants to find the relevant documents to this query : “I do not love fish". To apply this query to the generated corpus, we consider it as a document and we generate its corresponding vector. The corresponding vector of this query is as follows : To find the relative

Query)	1	1	1	0	0	0	0	1	1

document to this query the similarity between the query and each one of the documents in the corpus should be computed. In vector-space model the similarity between two documents is defined as the cosine of the angle between the vectors of the two documents. The results of calculating similarity between the query and the documents in the corpus is presented in table 3.2

Table 3.2 Similarity Values Between the Documents and the Query

	Similarity Value
Document (1)	0.7746
Document (2)	0.4743
Document (3)	0.4243
Document (4)	0.7155
Document (5)	0.5657
Document (6)	0
Document (7)	0.2582
Document (8)	0.7454

After computing the similarity values, the documents can be sorted according to their relevance to the given query.

Table 3.3 Relevancy Sorted List

Sorted List
Document (1)
Document (8)
Document (4)
Document (5)
Document (2)
Document (3)
Document (7)
Document (6)

As it is shown in the sorted list *Document 1)* is the most relevant document to the given query while the *Document 6)* is the least relevant document.

3.2.2 Measuring Performance

The two most important measures used to evaluate the performance of an information retrieval system are precision and recall. Precision indicates the exactness of the retrieval process. It is defined as the number of relevant documents retrieved by a search divided by the total number of documents retrieved by that search. Precision value shows how much the retrieved documents are relevant to the applied query :

$$Precision = \frac{\{\textit{relevant documents}\} \bigcap \{\textit{retrieved documents}\}}{\{\textit{retrieved documents}\}} \tag{3.1}$$

Recall indicates the completeness of retrieval process. It is defined as the number of relevant documents retrieved by a search divided by the total number of existing relevant documents and shows the percent of relevant documents that are retrieved :

$$Recall = \frac{\{\textit{relevant documents}\} \bigcap \{\textit{retrieved documents}\}}{\{\textit{relevant documents}\}} \tag{3.2}$$

3.2.3 Vector Space Model

Vector space model (or term vector model) is one of the IR algebraic models. It has been widely used in the traditional IR field. Most search engines also use similarity measures based on this model to rank Web documents.

A vector space model creates a space in which both documents and queries are represented as vectors of terms :

$d_i = \{w_{1,i}, w_{2,i} \cdots w_{n,i}\}$, $q = \{w_{1,q}, w_{2,q} \cdots w_{n,q}\}$ where "d_i" is a document in the corpus, "w"s are weights of terms in the document and "q" is a query applied to the corpus. Each dimension corresponds to a term. If a term occurs in the document the value of its corresponding dimension should be different from zero. There are several ways to compute this value that can be considered as the weight of a term in a document. One way is using the frequency of the term occurrence in the document as the weight of that term. By computing the weights in this way, all terms are considered

equally important because it shows how frequent a term is in a document but it cannot indicate if the term is a common term that occurs frequently in other documents too, or if it is a special term that belongs to this document more than others. One of the best schemes to compute this weight is *tf-idf* that is explained in the following section.

The terms are defined differently according to the applications. Typically, the terms are the words in the set of documents on which the queries are applied (corpus). Distinct words in the corpus form a dictionary. The dimensionality of the vectors representing the documents, is the number of terms in the dictionary. The query is considered as a document and is represented by the same kind of vector as the documents.

To rank documents against a query, we need to calculate the similarity between the query and each document. The similarity between a document and a query can be defined based on deviation of angles between their vectors (θ). The more narrow the angle, the higher similarity. In practice the cosine of the angle between the document's vector and the query's vector is used as their similarity.

$$Similarity(d_i, q) = \cos\theta = \frac{d_i \cdot q}{\|d_i\|\|q\|} \tag{3.3}$$

3.2.4 TF–IDF

The *TF–IDF* weight (term frequency-inverse document frequency) is a weight often used in information retrieval and text mining. It is a statistical measure used to evaluate how important a word is to a document in a corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus.

Term frequency (*TF*) value is the number of times a given term appears in the document normalized by dividing to occurrences of all terms in the document. The normalization is done to prevent a bias towards longer documents since a long document may have a high frequency number of a term regardless of the real importance of that term in the document. Thus the *TF* value for term i in

document j is defined as follows :

$$TF_(i,j) = \frac{n_(i,j)}{\sum_k n_(i,k)} \tag{3.4}$$

where $n_(i,j)$ is the number of occurrence of term t_i in document d_j.

Inverse document frequency (*IDF*) value indicates the general importance of a term in the corpus. It is defined as the logarithm of the result of dividing the number of all documents in the corpus by the number of documents that contains the given term. The equation used to calculate (*IDF*) value is as follows :

$$IDF_i = \log \frac{\mid D \mid}{\mid \{d : t_i \in d\} \mid} \tag{3.5}$$

where $\mid D \mid$ is the number of all documents in the corpus, t_i, represents the given term and $\mid \{d : t_i \in d\}$ is the number of documents that contain t_i. Low (*IDF*) value means that the given term is a common term that can be found in most documents while high (*IDF*) value indicates that the term is an important term for the containing document.

Finally the $TF - IDF$ weight of term i in document j is calculated using the following equation :

$$(TF - IDF)_(i,j) = TF_(i,j) \times IDF_i \tag{3.6}$$

where $TF_(i,j)$ is the term-frequency of term i in document j ,and IDF_i is the inverse document frequency of term i.

A high TF–IDF weight is obtained when the frequency of a given term's occurrence is high in a given document but the term appears rarely in other documents in the corpus.

3.2.5 Latent Semantic Indexing

Latent Semantic Indexing (*LSI*) is a developed version of vector space model. *LSI* uses a mathematical technique called Singular Value Decomposition (*SVD*) to identify patterns of relations between the terms and concepts in a collection of texts. In fact *LSI* works with the concepts rather

then the exact terms. This characteristic of *LSI* provides the following benefits :

- *LSI* does not require an exact match to return useful results ;
- it overcomes the problem of synonymy (several words with the same meaning) and polysemy (a word with more than one meaning) ;
- it is useful for document categorization and document clustering ;
- *LSI* is very tolerant to noise i.e., misspelled words, typographical errors, unreadable characters, etc ;
- *LSI* is not restricted to sentences in texts. It can work with different types of texts such as lists, free-form notes, emails, Web-based content, etc.

The *LSI* process starts by constructing a weighted term–document matrix *A*. *A* is a $(m \times n)$ matrix where *m* is the number of terms and *n* is the number of documents. Then the term-document matrix is decomposed into three matrices : **U**,**S** ,and **V** where *U* is a term–concept vector matrix, *S* diagonal matrix of weights ordered by weight value, *V* is a concept-document vector matrix and they satisfy the following relations :

$$A = USV^T \tag{3.7}$$

The next step is truncating the singular value matrix S to size **k** where *k* is typically selected in the range of 100 to 300. To conform with the truncated *S* the other matrices should also be modified as follows :

$$A \approx A_k = U_k S_k V_k^T \tag{3.8}$$

The effect of using this reduction is preserving the most important semantic information in the text while reducing noise and other undesirable artifacts of the original space of A.

The transformation of the query vectors in to the new space is performed according to the following equation :

$$q_k = q^T U_k S_k^{-1} \tag{3.9}$$

Now we have in the new space, both term–document matrix, V_k, and query vector, q_k . The similarity between a query and a document is the cosine of the angle between their vectors.

CHAPITRE 4

CONCEPT IDENTIFICATION APPROACH

Concept or feature location identifies abstractions (i.e.,concepts or features) and the location of the implementation of these abstractions. By location we mean highlighting the classes and the methods that implement a concept. Our concept location approach identifies concepts composing an execution scenario by grouping together methods that are (i) sequentially invoked together/in sequence, and (ii) cohesive and decoupled from a conceptual point of view. The underlying assumption is that, if a specific feature is being executed within a complex scenario (e.g., "Open a Web page from a browser" or "Save an image in a paint application"), then the set of methods being invoked, implementing a sub-concept or a concept, is likely to be conceptually cohesive, decoupled from those of other features and sequentially invoked. We adapted the definition of conceptual cohesion and coupling proposed by (Marcus *et al*. (2008)) and (Poshyvanyk et Marcus (2006b)).

The approach works as follows : first, we index the source code of each method of a system by means of IR approaches. Then, we instrument and exercise the system to collect execution traces for some scenarios related to different features and, therefore, containing sets of different concepts.We compress and filter the traces to remove utility and cross-cutting concerns and to remove repetitions of the same sub-sequences of methods. Finally, we apply a search-based optimization technique, i.e., a genetic algorithm, to split the compressed traces into conceptually cohesive and decoupled fragments. We overcome performance issues by parallelizing the generic algorithm over multiple machines.

4.1 Textual Analysis of Method Source Code

To determine the conceptual cohesion of methods, our approach uses the Conceptual Cohesion metric defined by Marcus *et al.* (Marcus *et al*. (2008)). We consider each method as a document, so

our corpus is the collection of methods in the system. We extract a set of terms from each method by tokenizing the method source code and comments, pruning out special characters, programming language keywords, and terms belonging to a stop-word list of the English language such as the articles and the propositions. We assume that comments appearing on top of the method declaration belong to the following method and thus they are associated to it.

In the next step, we split compound terms created via the Camel Case naming convention at each capitalized letter. Following Camel Case convention, the terms constructing a compound term are joined together without any space. The first letter of each term is upper case and the other letters are lower case except the first term that should start with the lower case. The term *getBook* is an example of a compound term created via the Camel Case naming convention. In this step, it is split into *get* and *book*.

Then, we stem the obtained simple terms by using a Porter stemmer (Porter (1980)) to reduce inflected (or derived) words to their stem or root word. For example the term *visited* is replaced by its stem *visit* after stemming step.

We generate a dictionary from the extracted terms. The dictionary is used in both the concept identifying and label assignment phases. In label assignment phase, we assign to each detected feature a descriptive label that indicate the functionality of the feature.

Once terms belonging to each method have been extracted, we index these terms using the TF-IDF indexing mechanisms (Baeza-Yates et Ribeiro-Neto (1999)). We obtain a term-document matrix, where documents are all methods of all classes belonging to the system under study and where terms are all the terms extracted (and split) from comments and method source code. Finally, we apply Latent Semantic Indexing (LSI) (Deerwester *et al*. (1990)), by using the implementation of singular value decomposition (SVD) in R , to reduce the term-document matrix into a concept-document matrix. The meaning of "concept" in LSI is different from that of "concept" in concept location. In LSI, a "concept" is one of the orthonormal dimensions of the LSI space.

Then we compute the conceptual cohesion of methods in a class in the LSI subspace to deal with synonymy, polysemy, and term dependency. Conceptual cohesion of each two methods is the

cosine between their corresponding vectors in method-term matrix.

4.2 System Instrumentation and Trace Collection

The traces, from which the concepts are identified, are sequences of method calls during the execution of specific scenarios. To generate such traces the system should be instrumented first. We instrument the software system using the instrumentor of MoDeC. MoDeC is a tool to extract and model sequence diagrams from Java systems (Ng *et al*. (2009)). The MoDeC instrumentor is a dedicated Java bytecode modification tool implemented on top of the Apache BCEL bytecode transformation library1. It inserts appropriate and dedicated method invocations in the system to trace method/constructor entries/exits, taking care of exceptions and system exits (System.exit(int)). It also allows the user to add to the traces tags containing meta-information e.g., delimiting and labeling sequences of method calls related to some specific features being exercised.

An execution scenario is composed of a sequence of cohesive steps. For example, exercising a Web browser could consist in a sequence of the following steps (i) open the browser ; (ii) insert a URL and access a Web page ; and (iii) save the Web page into a local HTML file. We exercise the instrumented system to collect execution traces by following scenarios taken from user manuals or use case descriptions. Resulting traces are text files listing method calls and including the class of the object caller, the unique ID of the caller, the class of the receiver, its unique ID, and the complete signature of the method.

4.3 Pruning and Compressing Traces

There are some methods that occurs to much in almost all scenarios, e.g., methods related to mouse movements. Even in a single execution trace of an application with a graphical user interface, mouse tracking methods will largely exceed all other method invocations. It is likely that such methods are not related to any particular concept and they are not really useful for feature identification, e.g., they are utility methods. These methods do not provide useful information for developers when locating a concept, because they are common in many concepts. These methods are similar

to low-discriminating terms occurring in many documents when applying an IR technique. Such terms are penalized by indexing measures like TF-IDF [21].

Similarly, we built the distributions of the frequencies of method occurrences to remove too-frequent methods. We then prune out the methods having a frequency greater than $Q3 + 2 \cdot IQR$, where Q3 is the third quartile (75% percentile) of the distribution and IQR is the inter-quartile range. An alternative approach to deal with these methods is aspect mining (Tonella et Ceccato (2004)), (Marin *et al*. (2007)). We do not use aspect mining because we are interested in pruning these methods to locate concepts, not to crosscutting concerns.

Traces often contain repetitions of one or more method invocations, for example m1() ; m1() ; m1() ; or m1() ; m2() ; m1() ; m2() ;. Repetitions do not introduce new concepts, they are dedicated to do the same task again and again in order to have for example a figure in a larger scale in a graphical system. Thus we compress traces using the Run Length Encoding (RLE) algorithm to remove repetitions and keep one occurrence of any repetition only. The two previous examples would become m1() and m1() ;m2(), respectively. Compression is performed for any sub-sequences of method invocations having an arbitrary length.

4.4 Search-based Concept Location

In this step we segment execution traces into conceptually cohesive segments related to the feature being exercised (and thus to a specific concept). Determining a (near) optimal splitting of a trace into segments is NP-hard [25]. Therefore we decided to find a (near) optimal solution by using search-based techniques.

We experimented different techniques : hill climbing, simulated annealing, and genetic algorithms (GA). The representation of the problem and the possible movements in each iteration defined in local search techniques, hill climbing and simulated annealing, was different from those in GA. Comparing the obtained results of these three algorithm we chose to use GA since, due to the characteristics of the search space, it outperformed other techniques.

A short explanation of our implementation of hill climbing and simulated annealing comes in

the following section then we explain the use of GA in identifying and locating the concepts in an execution trace more in details.

4.4.1 Local Search Implementation

The first mental model of the problem was based on the presence of segments and gaps. We assumed that a trace is a set of segments and gaps. A segment is a sequence of method invocations implementing a concept (a feature) and its length, i.e. the number of its methods, should be greater than one. A segment is supposed to have high conceptual cohesion. A gap contains the method invocations that do not collaborate in implementing any special concept (utility methods) like the methods related to the mouse movements. They do not share much common linguistic information with other methods so their similarity with other methods should be very low. The length of a gap, i.e. the number of its methods, can be equal to 1. A gap is supposed to have very low conceptual cohesion.

According to this definition a trace should be split to pieces representing segments and gaps such that each two segments are separated from each other by a gap.

Solution Representation

Our representation of the hill climbing solution is a bit-string as long as the trace in which we want to identify the concepts. Each segment is represented as a sequence of "1"s and each gap is represented as a sequence of "0"s. For example, the bit-string

$$\underbrace{111100011111}_{12 methods}$$

means that the trace, containing 12 method invocations, is split into two segments and a gap decomposed into the first four method invocations (first segment), the next three (gap), and the last five (second segment).

Initial Solution

The initial solution is generated randomly. By randomly we mean that the number of pieces, gaps and segments, and the length of each piece is selected randomly in a defined range. The following constraints should be considered while defining the range of possible piece numbers and their lengths :

1. The length of a segment should be equal to or greater than two.
2. The length of a gap should be equal to or greater than one.
3. Each two segments are separated from each other by a gap.
4. We decided that the first solution starts and ends with the gaps.

The constraints (3) and (4) result in

$$gapNo = segmentNo + 1 \tag{4.1}$$

where segmentNo is the total number of segments in a trace and gapNo is the total number of gaps in the same trace. So if we have $segmentNo = n$ then we have $gapNo = n + 1$. considering the constraints (1) and (2) we have

$$|segment| \geq 2 \tag{4.2}$$

$$|gap| \geq 1 \tag{4.3}$$

If we show the length of the trace with *L* then we have

$$\begin{aligned} L &= n|segment| + (n+1)|gap| \\ &\geq (2n) + (n+1) \\ &\geq 3n + 1 \end{aligned}$$

Now we can calculate the upper limit for *n*.

$$n \leq \frac{L-1}{3} \tag{4.4}$$

By randomly selecting *n* in the range $[1, \frac{L-1}{3}]$ we have *n* segments and n+1 gaps. Assigning the length to the generated pieces is done in two steps. First, the length of each segment is defined equal to 2 and each gap equal to 1. Then we use a loop in which we randomly select a piece and a number between 1 and the length of the trace minus the current length of all the pieces. The random selected number is added to the length of the random chosen piece.

Hill Climbing Transformation

We defined the following transformations for exploring the neighborhood via hill climbing :

– Boundary movement : each segment has two boundaries, begin and end. Each boundary can be moved towards two directions, left and right. So we have four possible boundary movements for each segment.

  Each boundary movement add/remove one method to/from a segment.

– Merge : this transformation merges two segments by getting rid of the gap between them.

– Split : this transformation splits the segment into two segments with a gap between them. If it applies to a gap it creates a segment in the gap.

At each iteration, the algorithm tries to improve the solution by applying one of these transformations to the current solution. Solution improvement starts by applying boundary movement on the segments. If this transformation can improve, the solution the current iteration is done and the algorithm goes to the next iteration. If by applying all possible movements the solution is not improved, then the algorithm applies merge and split to find a better solution.

The iteration cycle comes to an end when the solution cannot be improved any more. It happens when the fitness value of current solution is higher than the fitness value of all its neighbors, so any movement cannot increase the fitness value.

Hill Climbing Fitness Function

We tried our implementation of hill climbing with several fitness functions. Each fitness equation was obtained by improving the last equation based on manual validation of the concept location algorithm in splitting traces into segments and gaps. In this section we are going to present some of the fitness functions we used in our hill climbing implementation.

First fitness equation was defined using the following elements :

1. Number of concepts - positively contributes to fitness
2. Number of gaps - negatively contributes
3. Conceptual cohesion of segments - positively contributes to fitness
4. Conceptual cohesion of gaps - negatively contributes to fitness

Conceptual cohesion of a piece, gap or segment, with *n* methods is defined as the average of textual similarity between each pair of methods in that segment. The cohesion value is calculated as follows :

$$conceptualCohesion(P_k) = \frac{2}{n(n-1)} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} S(m_i, m_j) \tag{4.5}$$

Where P_k is the segment or gap in the k*th* position in the trace and $S(m_i, m_j)$ is the textual similarity between these two methods obtained from the similarity matrix generated by using latent semantic indexing. The fitness function that is defined using the mentioned elements is as follows :

$$fitness = w_1 \cdot segNo + w_2 \cdot segCohAVG - (w_3 \cdot gapNo + w_4 \cdot gapCohAVG) \tag{4.6}$$

where w_i is a positive weight, *segNo* and *gapNo* are the number of segments and gaps in the related solution, and segCohAVG and *gapCohAVG* are the average of conceptual cohesion of segments and gaps. The results we obtained by using the equation 4.6 were not promising. Thus we decided to improve the fitness equation by adding the conceptual coupling to it. We defined the conceptual coupling of a segment as the average of textual similarity between the methods of that segment with the methods of its two neighbors. The equation used to calculate the coupling for a piece k

(gap or segment) is as follows :

$$conceptualCoupling(k) = \frac{\sum_{i=begin(k)}^{end(k)} \sum_{j=begin(k-1)}^{end(k-1)} S(m_i, m_j)}{(begin(k) - end(k) + 1)(begin(k-1) - end(k-1) + 1)} + \frac{\sum_{i=begin(k)}^{end(k)} \sum_{j=begin(k+1)}^{end(k+1)} S(m_i, m_j)}{(begin(k) - end(k) + 1)(begin(k+1) - end(k+1) + 1)} \quad (4.7)$$

where $begin(k)$ is the position (in the trace) of first method invocation of k^{th} piece and $end(k)$ the position of last method invocation of that piece.

The new fitness equation obtained by adding the coupling to the fitness function elements, is as follows :

$$fitness = \frac{segCohAVG}{(1 + gapCohAVG)(1 + cohAVG)(1 + gapNo)(1 + segNo)} \quad (4.8)$$

where *cohAVG* is the average of cohesion of all the trace. By verifying the results obtained by doing some experiments using the equation 4.6 we decided that the number of pieces, segments or gaps, has a negative effect on solution improvement. This is to say that in the new fitness function the number of segments contributes negatively in the fitness.

Simulated Annealing (SA)

The results we obtained from hill climbing were not stable enough, it means that different starting points generated very different segmentations. It seemed that each solution converged to a local optimum and stopped there. Therefor, we decided to use simulated annealing. SA accepts non improving movements in the first iterations with a probability that depends on the difference between the corresponding fitness values and on a global parameter T (the temperature), that is gradually decreased during the process. The dependency is such that the current solution changes almost randomly when T is large (high temperature), but increasingly "uphill" as T goes to zero. The allowance for "downhill" moves saves the solution from converging to a local optimum and becoming stuck at it, which is the defect of the hill climbing algorithm.

In the implementation of SA, for a determined number of times , when the algorithm reaches its stopping criteria, the temperature is reset to its initial value and the process starts again. Using the concept of temperature besides several restarts, we hoped to overcome the problem of convergence to a local optimum and have stable solutions (in different runs).

The representation of the solution and the transformations are the same as what we explained in the implementation of hill climbing. The pseudo-code of implemented SA is as follows :

Algorithm 4.1. pseudo-code of implemented SA

$tries \leftarrow 0$

$randomlyselectsol_c$

repeat

$j \leftarrow 0$

repeat

$T \leftarrow T_{max} \cdot e^{-jr}$

select sol_n *in the neighborhood of* sol_c

if $fitness(sol_n) > fitness(sol_c)$

$sol_c \leftarrow sol_n$

else if random $[0, 1) < e^{\frac{fitness(sol_n) - fitness(sol_c)}{T}}$

$sol_c \leftarrow sol_n$

until $T < T_{min}$

tries $\leftarrow tries + 1$

until tries = MAXTRIES

In our implementation we defined *Tmin = 0.01, Tmax = 0.30* and *r = 0.001*.

4.4.2 Local Searches Stability

When using any search algorithm, it is important to have consistent solutions in different runs. Solution consistency shows that the algorithm is stable and all search processes in different runs

have been converted to the same or very similar optimum, i.e. the fitness function always leads the search to the same area. Thus the search algorithm and its result(s) are reliable.

Here the solutions are consistent when an identified segment or gap in one run exists in the solution of the other run. A detected segment or gap may exactly be repeated in the other solution or may partially overlap with some others in the second solution. Figure 4.1 gives a demonstration of this explanation. The arrows indicate the positions where two solutions are different. Stability

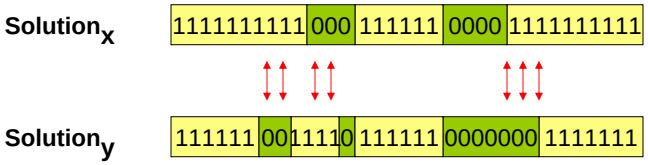

Figure 4.1 Two Local Search Solutions

value is defined as $(1 - distance)$ when the $distance$ is the normalized value of the number of positions in which two solutions are different. Distance value is calculated by using $X - OR$ operator as follows :

$$distance = \frac{|solution_x \bigoplus solution_y|}{|solution_x|}$$

when $|solution_x| = |solution_y|$. In the example given in Figure 4.1, the number of different positions in two solutions is 7 and solutions length is 33, so distance is $\frac{7}{33} \simeq 0.21$ and stability value is $1 - 0.21 = 0.79$.

Stability values calculated for the local search solutions are not high, the average stability value is around 0.5. One reason is that the search space of our problem is very large with many local optima. Thus different running processes converge to different local optima and it produces low consistency between final solutions.

4.4.3 Genetic Algorithm (GA)

By using local search techniques we could not reach consistent solutions. The main reason was the very large search space. Thus we decided to apply a genetic algorithm because it is able to

Table 4.1 Example of GA individual representation (second column).

Method Invocations	Repr.	Segments#
TextTool.deactivate()	0	
TextTool.endEdit()	0	
FloatingTextField.getText()	0	
TextFigure.setText-String()	0	1
TextFigure.willChange()	0	
TextFigure.invalidate()	0	
TextFigure.markDirty()	1	
TextFigure.changed()	0	
TextFigure.invalidate()	0	
TextFigure.updateLocation()	0	2
FloatingTextField.endOverlay()	0	
CreationTool.activate()	1	
JavaDrawApp.setSelectedToolButton()	0	
ToolButton.reset()	0	
ToolButton.select()	0	
ToolButton.mouseClickedMouseEvent()	0	3
ToolButton.updateGraphics()	0	
ToolButton.paintSelectedGraphics()	0	
TextFigure.drawGraphics()	0	
TextFigure.getAttributeString()	1	

outperform the two previous techniques in large search spaces with many local optimum.

At this point we realized that we can improve our problem definition by removing the concept of the gaps. According to our definition, a gap is a segment with low conceptual cohesion that does not participate in implementing any special concept, such as mouse tracking methods. This definition is not completely correct, because some of these methods that are not related to any special concept, have high conceptual cohesion with each other. Thus we can consider them as utility methods that implement a generic, common concept such as mouse move.

A New Individual Representation

Our representation of an individual is a bit-string as long as the execution trace in which we want to identify some feature-related concepts. Each method invocation is represented as a "0", except the last method invocation in a segment, which is represented as a "1". For example, the bit-string

$$\underbrace{00010010001}_{11 methods}$$

means that the trace, containing 11 method invocations, is split into three segments decomposed into the first four method invocations, the next three, and the last four. A real example of segment splitting [1] is shown in Table 4.1.

Initial Population

The only constraint is that the length of a segment should not be less than two.

$$|segment| \geq 2 \tag{4.9}$$

If we define *n* as the number of segments in the trace and *L* as the length of the trace, then considering equation 4.9 we have

$$\begin{aligned} L &= n \cdot |segment| \\ L &\geq 2n \end{aligned}$$

Now for the upper limit for n we have

$$n \leq \frac{L}{2}$$

The initial solution is generated in two random steps. In the first step the number of segments is randomly selected in the rang $[1, \frac{L}{2}]$. Then the length of each segment is defined with a randomly selected number in the range $[2, \frac{L}{n}]$.

Genetic Algorithm Operators and Parameters

The mutation operator randomly chooses one bit in the representation and flips it over. Flipping a "0" into a "1" means splitting an existing segment into two segments, while flipping a "1" into a "0" means merging two consecutive segments.

The crossover operator is the standard two-point crossover. Selecting two individuals as parents,

1. The segment splitting shown in Table 4.1 has been obtained randomly and does not correspond to an actual "good" solution.

two random positions x, y with $x < y$ are chosen in one individual's bit-string and the bits from x to y are swapped between the two individuals to create the new offsprings. The selection of the parents is performed by using the roulette-wheel selection. Roulette-wheel selection uses a probability function to define the chance of each individual for being selected as a parent. Based on this probability function, the individuals with higher fitness value have a greater chance as a candidate for generating the members of the next generation. In fact, it simulates a casino roulette wheel, where each proportion of the wheel is assigned to an individual based on its fitness.

We do not have elitism in our GA, i.e. it does not guarantee to retain best individuals across subsequent generations. This is done to prevent GA to converge too fast to a local optimum.

Fitness Function

As we mentioned before, in our fitness function, we use the software design principles of cohesion and coupling, already adopted in the past to identify modules in software systems (Mitchell et Mancoridis (2006)), although we use conceptual (i.e. textual) cohesion and coupling measures, rather than structural cohesion and coupling measures.

Similar to hill climbing, segment cohesion is the average (textual) similarity between any pair of methods in a segment k and is computed using the formulas in Equation 4.10 where $begin(k)$ is the position (in the individual's bit-string) of the first method invocation of the k^{th} segment and $end(k)$ the position of the last method invocation in that segment. The similarity between two methods is computed using the cosine similarity measure over the LSI matrix extracted in the previous step.

Our definition of coupling in GA fitness is different from the coupling in hill climbing. Here segment coupling is defined as the average similarity between a segment and all other segments in the trace, computed using Equation 4.11, where l is the trace length. As our conjecture is that a concept should be implemented by method calls locally close to each other, on the one hand the algorithm favors the merging of consecutive segments containing methods with high average conceptual similarity. On the other hand, the algorithm penalizes solutions where consecutive segments are highly coupled together. The segment coupling represents, for a given segment, the average similarity bet-

ween methods in that segment and those in different ones. Finally, for a trace split into n segments, the fitness function is shown in Equation 4.12.

$$SegmentCohesion_k = \frac{\sum_{i=begin(k)}^{end(k)-1} \sum_{j=i+1}^{end(k)} similarity(method_i, method_j)}{(end(k) - begin(k) + 1) \cdot (end(k) - begin(k))/2} \tag{4.10}$$

$$SegmentCoupling_k = \frac{\sum_{i=begin(k)}^{end(k)} \sum_{j=1, j<begin(k) \text{ or } j>end(k)}^{l} similarity(method_i, method_j)}{(l - (end(k) - begin(k) + 1)) \cdot (end(k) - begin(k) + 1)} \tag{4.11}$$

$$fitness(individual) = \frac{1}{n} \cdot \sum_{k=1}^{n} \frac{SegmentCohesion_k}{SegmentCoupling_k} \tag{4.12}$$

4.4.4 GA Stability

As it is mentioned before, when using a search algorithm, it is important to have consistent solutions in different runs.

In the case of our GA implementation, solutions are constant when an identified segment in one run exists in the solution of the other run. A detected segment is either repeated in the other solution or has a correspondent with high intersection in that solution. Figure 4.2 reports an example of similarity and stability computation. It shows two solutions $solution_x$ and $solution_y$ of a trace that consists of 28 method invocations. The first detected segment in both solutions is exactly the same. The second segment of $solution_x$ has overlap with three segments of the $solution_y$. In such cases the segments with the biggest overlap are matched together and the similarity is calculated between them. Thus in figure 4.2 the third segment of the $solution_y$ is chosen as the corresponding segment with the second segment of $solution_x$. The arrows indicate the matching segments of two

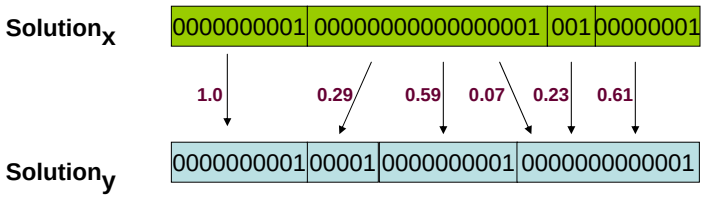

Figure 4.2 An example of matching segments in two GA solutions

solutions. Beside each arrow there is a number. This number is the calculated similarity between two matched segments.

The similarity between two segments does not depend only on their intersection but also on their union. We calculate similarity between two segments by using *Jaccard index*[2] (free encyclopedia (2010)) that is presented in equation 4.13.

$$similarity(S_{x,i}, S_{y,j}) = \frac{|S_{x,i} \cap S_{y,j}|}{|S_{x,i} \cup S_{y,j}|} \tag{4.13}$$

where $S_{x,i}$ and $S_{y,j}$ are respectively the $i'th$ segment in solution x and the $j'th$ segment in solution y. The similarity between two solutions is defined as the average of the similarity between matched

Table 4.2 Match Matrix of the Solutions in Figure 4.2

Segments	$S_{x,1}$	$S_{x,2}$	$S_{x,3}$	$S_{x,4}$
$S_{y,1}$	1.0	0.0	0.0	0.0
$S_{y,2}$	0.0	0.29	0.0	0.0
$S_{y,3}$	0.0	0.59	0.0	0.0
$S_{y,4}$	0.0	0.07	0.23	0.61

pairs, it means that the segments from two solutions should be matched together such that the sum of their similarity is maximum. Thus the first step in calculating similarity between two segments is finding appropriate pairs that maximize the summation. It is a kind of assignment problem[3] in which we should try to assign to each segment in a solution a segment in the other solution such that the sum of similarities between all the pairs be the maximum possible value. To solve this problem we use the *Hungarian algorithm*[4]. The Hungarian algorithm finds the optimal path in a graph by choosing the appropriate assigned pairs, which may not always be the pair with the highest similarity. The algorithm works on the matrix that represents the graph. In our implementation, The

2. The Jaccard index, also known as the Jaccard similarity coefficient is a statistic used for comparing the similarity and diversity of sample sets.

3. "The assignment problem is one of the fundamental combinatorial optimization problems in the branch of optimization or operations research in mathematics. It consists of finding a maximum weight matching in a weighted bipartite graph." (free encyclopedia (2010))

4. "The Hungarian method is a combinatorial optimization algorithm which solves the assignment problem in polynomial time and which anticipated later primal-dual methods. "(free encyclopedia (2010))

matrix consists of the similarity between each two segments from two different solutions. Table 4.2 presents match matrix of the two solutions in figure 4.2

Then the algorithm receives the matched matrix as the input and calculates the optimal solution matching. At the end, the value calculated by the algorithm is normalized in the range of [0,1] by being divided to the number of matches.

High similarity value indicates that the results are consistent while low value is a sign of instability. Calculated similarity for the GA solutions, shows that the results have high consistency and the algorithm is stable. We discuss the stability of the GA in the next chapter 4.2 with more details.

4.5 Segment Labeling

To identify the concepts (segments) in the execution trace, it is needed to assign them meaningful labels that introduce their functionalities. An introducing label can be a term or a bunch of terms textually relevant to the located concept. As it is explained in chapter 4.2 the relevance of a term to a method is defined by the TF-IDF weight. TF-IDF indicates how important a word is to a document in a corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus. We use the same idea to compute the relevance of a term to a segment.

The weight of a term t in a segment s that consists of n methods is defined as the average of the TF-IDF weight of t in each method of s. It is computed by using the formulas in equation 4.14.

The relevance of a term t to a segment s_k is defined as the $weight(t, s_k)$ minus the average of the weight of t in other segments of the trace. It is computed by using the formulas in equation 4.15 where $|S|$ is the number of identified segments in the trace.

$$weight(t, s) = \frac{1}{n} \sum_{i=1} nTF - IDF(t, m_i) \tag{4.14}$$

$$weight(t, s_k) = weight(t, s_k) - \frac{1}{|S| - 1} \sum_{i=1, i \neq k} |s| weight(t, s_i) \tag{4.15}$$

However, several problems were encountered preventing to obtain meaningful, good labeling. Most noticeably, high number of *JDK* and various *JAR* keywords and method names were polluting the linguistic information.

4.6 Conclusion

In this chapter, we introduced our proposed approach to identify concepts in an execution trace. The proposed approach is a hybrid approach that uses textual information extracted from source code (static analysis) to detect the concepts that exist in an execution trace (dynamic analysis). To identify the methods contributing in implementation of a concept, the approach splits a trace into textually cohesive and decoupled segments and tries to find a (near) optimal segmentation by using the genetic algorithm.

We have made the following assumptions :

1. methods helping to implement a concept are likely to share some linguistic information and,
2. methods responsible to implement a feature are likely to be called close to each other in an execution trace.

CHAPITRE 5

EMPIRICAL STUDY AND RESULTS

We performed an empirical study to evaluate the proposed concept location approach. The *goal* of this study is to analyze the novel concept location approach based on dynamic data, with the *purpose* of evaluating its capability of identifying meaningful concepts. The *quality focus* is the accuracy and completeness of the identified concepts. The *perspective* is that of researchers who want to evaluate how the proposed approach can be used during maintenance and evolution. The *context* consists of an implementation of our approach and of the execution traces extracted from two open source systems, JHotDraw and ArgoUML.

The setting of genetic algorithm for this empirical study was as follows : the population size was set to 200 individuals and a number of generations of 2,000 for shorter traces (those of JHotDraw) and 3,000 for longer ones (those of ArgoUML) was used. The crossover probability was set to 70% and the mutation probability to 5%, which are widely used values in many GA applications (McGill (July 01, 2010)).

5.1 Context

Table 5.1 Statistics for the two systems.

Systems	Number of Classes	Kilo Line of Code	Release Dates
ArgoUML v0.18.1	1,267	203	30/04/05
JHotDraw v5.4b2	413	45	1/02/04

The context of our study are execution traces from ArgoUML and JHotDraw. Table 5.1 high-

lights the main characteristics of the two systems. *ArgoUML*[1] is an open source UML modeling tool with advanced software design features, such as reverse engineering and code generation. The ArgoUML project started in September 2000 and is still active. We analyzed release 0.19.8. *JHotDraw*[2] is a Java framework for drawing 2D graphics. JHotDraw started in October 2000 with the main purpose of illustrating the use of design patterns in a real context. We analyzed release 5.1.

Table 5.2 Statistics for the collected traces.

Systems	Scenarios	Original Size	Cleaned Sizes	Compressed Sizes
ArgoUML	Start, Create note, Stop	34,746	821	588
	Start, Create class, Create note, Stop	64,947	1066	764
	Start, Add text, Draw rectangle, Stop	13,841	753	361
	Start, Draw rectangle, Cut rectangle, Stop	11,215	1206	414
	Start, Spawn window, Draw circle, Stop	16,366	670	433

We generate traces by exercising various scenarios in the two systems. Table 5.2 summarizes the scenarios and shows that the generated traces include from 6,000 up to almost 65,000 method invocations. The compressed traces include from 240 up to more than 750 method invocations.

By exercising these scenarios, we do not want to identify concepts related to the systems' startup, *Start*, and closing, *Stop*. Therefore, in the following, when naming the scenarios and their associated features, we no longer include the Start and Stop concepts.

The GA was implemented using the *Java GA Lib*[3] library.

5.2 Building the Oracle

We need an oracle to assess the accuracy and completeness of the identified concepts. We build such an oracle by manually tagging the execution traces. Two "Start" and "Stop" tags enclose the method invocations related to a particular concept. While executing the instrumented system, before and after a step in the execution scenario (*e.g.*, Draw rectangle), the user, through a command

1. http ://argouml.tigris.org
2. http ://www.jhotdraw.org
3. http ://sourceforge.net/projects/javagalib/

in the instrumentor interface, inserts the appropriate tags in the execution trace and then continues to exercise the instrumented system with the next steps of the scenario. Consequently, the collected traces are composed of a sequence of method invocations interleaved with tags separating the invocations belonging to different steps.

5.3 Research Questions

In the empirical study we addressed the three following research questions :

- **RQ1 :** *How stable is the GA, through multiple runs, when identifying concepts into execution traces ?* Approaches based on GAs could suffer from the randomness of the search : the initial individuals are randomly generated and the crossover, mutation, and selection operators are influenced by random choices. However, it is desirable that the representation, operators, fitness, and other settings (*e.g.*, population size and stopping criteria) be chosen so that multiple runs of the GA yield similar solutions.
- **RQ2 :** *To what extent the identified concepts match the ones in the oracle ?* We are interested to evaluate the extent to which the identified segments overlap with the ones in our oracle, obtained by manually tagging the traces.
- **RQ3 :** *How accurate is the identification of concepts in execution traces ?* Finally, we are interested to evaluate the extent to which the identified segments are accurate, *i.e.*, how many of the method invocations that they contain are also in the oracle and how many are not.

5.4 Study Settings and Analysis Method

To answer **RQ1**, we evaluate the extent to which the segments identified in multiple runs of the GA, and occurring in the same position of the trace, overlap each other. Let us consider a compressed trace composed of N method invocations $T \equiv m_1, \ldots m_N$ and partitioned at run i of the GA in k_i segments $s_{1,i} \ldots s_{k,i}$. For each segment $s_{x,i}$ obtained at run i, and for all the segmentations obtained at run $j \neq i$, we compute the maximum overlap between $s_{x,i}$ and the

segments obtained at run j as follows :

$$max(Jaccard(s_{x,i}, s_{y,j})), y = 1 \dots k_j$$

where :

$$Jaccard(s_{x,i}, s_{y,j}) = \frac{|s_{x,i} \cap s_{y,j}|}{|s_{x,i} \cup s_{y,j}|}$$

and where union and intersection are computed considering method invocations occurring at a given position in the trace. Stability is evaluated by means of descriptive statistics computed across the above obtained overlap values.

RQ2 is answered similarly to **RQ1** but, in this question, we compare the overlap between manually-tagged segments in the execution traces with segments identified by our approach. Specifically, given the segments determined by the tags in the trace (our oracle) and given the segments obtained by an execution of the system, we compute the overlap between each manually-tagged segment in the trace and the most similar automatically-identified segment.

Finally, **RQ3** is addressed like **RQ2**, with the only difference that we use precision instead of the Jaccard score, because we are interested in evaluating the accuracy of our approach. Precision is defined as follows :

$$Precision(s_{x,i}, s_{y,o}) = \frac{|s_{x,i} \cap s_{y,o}|}{|s_{x,i}|}$$

where $s_{x,i}$ are segments obtained by our approach and $s_{y,o}$ are segments in the corresponding trace in the oracle.

5.5 RESULTS AND DISCUSSION

This subsection reports the results of our experimental evaluation : the collected data and their analysis to address the previous research questions.

Table 5.3 Descriptive statistics of similarity among segments obtained in ten different runs.

Systems	Scenarios/Features	Similarity Averages				
		Min.	Max.	Mean	Median	σ
ArgoUML	(1) Add note	0.69	0.95	0.84	0.83	0.07
	(2) Add class, Add note	0.65	0.98	0.80	0.80	0.06
JHotDraw	(1) Draw rectangle	0.55	0.96	0.76	0.76	0.12
	(2) Add text, Draw rectangle	0.54	0.93	0.72	0.70	0.10
	(3) Draw rectangle, Cut rectangle	0.73	0.93	0.85	0.84	0.05
	(4) Spawn window, Draw circle	0.67	0.86	0.76	0.76	0.04

5.5.1 RQ1 : How Stable is the GA across Multiple Runs ?

We assess the stability of the GA by computing the average similarity of the segments identified in ten different runs of the approach. Table 5.3 shows the similarity results. Overall, the similarity averages for JHotDraw range between 55% and 95%, with median values ranging between 70% and 84% . They are slightly higher for ArgoUML, between 80% and 83%. Thus, we conclude that, despite the potentially large size of the search space, our approach is able to generate stable segments across multiple runs. In addition, increasing the number of generations and the population size would potentially further increase the approach stability.

5.5.2 RQ2 : To What Extent the Identified Concepts Match the Ones in the Oracle ?

Table 5.4 Similarity (Jaccard overlap) between segments identified by the approach and features tagged in the trace.

Systems	Scenarios	Features	Jaccard				
			Min.	Max.	Mean	Median	σ
ArgoUML	(1)	Add note	0.15	0.39	0.28	0.27	0.08
	(2)	Create class	0.11	0.28	0.22	0.25	0.05
	(2)	Create note	0.22	0.56	0.35	0.31	0.14
JHotDraw	(1)	Draw rectangle	0.63	0.93	0.84	0.89	0.13
	(2)	Add text	0.21	0.31	0.26	0.27	0.05
	(2)	Draw rectangle	0.53	0.70	0.63	0.61	0.06
	(3)	Draw rectangle	0.42	0.76	0.64	0.72	0.14
	(3)	Cut rectangle	0.16	0.23	0.22	0.23	0.02
	(4)	Draw circle	0.54	0.96	0.85	0.88	0.14
	(4)	Spawn window	0.07	0.41	0.20	0.16	0.11

To address **RQ2**, we evaluate the extent to which the segments actually reflect features as they were manually tagged when executing the instrumented system to generate the execution traces.

Table 5.5 Similarity (precision) between segments identified by the approach and features tagged in the trace.

Systems	Scenarios	Features	Precision				
			Min.	Max.	Mean	Median	σ
ArgoUML	(1)	Add note	0.91	1.00	0.97	1.00	0.04
	(2)	Create class	1.00	1.00	1.00	1.00	0.00
	(2)	Create note	1.00	1.00	1.00	1.00	0.00
JHotDraw	(1)	Draw rectangle	0.89	1.00	0.96	1.00	0.06
	(2)	Add text	0.27	0.36	0.32	0.34	0.04
	(2)	Draw rectangle	0.61	1.00	0.69	0.66	0.13
	(3)	Draw rectangle	0.73	1.00	0.94	1.00	0.11
	(3)	Cut rectangle	1.00	1.00	1.00	1.00	0.00
	(4)	Draw circle	0.81	1.00	0.91	0.95	0.09
	(4)	Spawn window	1.00	1.00	1.00	1.00	0.00

For some features, *e.g.*, drawing a rectangle or a circle, the average (and median) Jaccard overlap is very high, suggesting that the features are implemented through sequences of very cohesive methods. Yet, other features exhibit lower overlaps. These lower overlaps do not mean that our approach was unable to successfully identify the features. Indeed, in some cases, for example the scenarios *Add text* in JHotDraw and *Create note* in ArgoUML, the features are realized by adapting a textual-editing feature as a shape-drawing feature, using the Adapter design pattern. The feature adaptation produces sequences of methods with a low cohesion, which our algorithm tend to split. As a consequence, the resulting overlaps are appropriately low.

In other cases, in particular with traces from ArgoUML, a large trace segment corresponding to a feature is split into two or more segments by our approach. Thus, the overlap between the (larger) manually-tagged segment and the corresponding automatically-identified segment is low. A manual study of such cases revealed that the manually-tagged segment is indeed composed of several smaller and cohesive sub-concepts that our algorithm tends to split, as illustrated and discussed in the following subsection.

5.5.3 RQ3 : How Accurate is the Identification of Concepts in Execution Traces ?

Table 5.5 reports the precision of the identified segments with respect to the manually-tagged ones. Precision is often very high, with median values in most cases above 85% and very often equal to 100%.

Lower precision values sometimes occur with explainable reasons. For example, in the scenario (2) of JHotDraw, composed of *Add text* and *Draw rectangle*, the two features are implemented using a very similar sequence of method invocations, making them hard to distinguish. Because these features are executed one after the other, our search-based optimization technique is unable to split the trace into segments similar to the ones from the oracle. Consequently, the precision of *Add text* drops to a median value of 34% and that of *Draw rectangle*, usually very high in other scenarios, is only 66%.

5.5.4 Discussion

We analyze in detail some results to understand how the approach split the traces into segments. We focus on cases where the Jaccard score is low. In other cases, we know that the segments are meaningful because they are consistent with the oracle. Tables 5.6, 5.7 and 5.8 show excerpts of three segments.

The *Add class* feature of ArgoUML was matched with a very low Jaccard score. The manual tags in the trace delimited a sequence of 199 method invocations. The approach split this sequence into 5 segments comprising a total of 172 method invocations, out of which 16 invocations occurred before the tag and, thus, do not belong to the oracle. The remaining $199-172+16 = 43$ invocations were grouped in small segments mainly related to GUI-event handling. In details, the five segments are related to (1) creation of the objects responsible for handling the class diagram through an instance of the Factory design pattern ; (2) adding the class to the project ; (3) adding the class to the current name-space ; (4) setting properties of the class through a Façade design pattern ; and, (5) handling the persistence of the diagram in the XMI file representing the UML diagram.

For the *Create note* feature of ArgoUML, the tagged segment is composed of 88 method invocations while the best matching segment identified by our approach is composed of 50 methods. The identified segment deals with the creation of a note, *i.e.*, creation of the object through a Factory, addition to the project, setting of its property. When compared to the *Add class* feature, only one segment was identified instead of five because the segment for creating a note is shorter than that of

Table 5.6 Excerpt of segments identified by the approach.

Create note (ArgoUML)
FacadeMDRImpl.isSingleton(...)
FacadeMDRImpl.isUtility(...)
CoreFactory.getCoreFactory()
CoreFactoryMDRImpl.buildComment(...)
CoreFactoryMDRImpl.createComment()
CoreFactoryMDRImpl.initialize(...)
ModelEventPumpMDRImpl.flushModelEvents()
UndoCoreHelperDecorator.addAnnotatedElement(...)
ModelEventPumpMDRImpl.flushModelEvents()
ClassDiagramGraphModel.addNode(...)
ClassDiagramGraphModel.canAddNode(...)
FacadeMDRImpl.isAInterface(...)
FacadeMDRImpl.isASubsystem(...)
Project.getRoot()
ModelManagementFactory.getModelManagementFactory()
ModelManagementFactoryMDRImpl.getRootModel()
CoreHelperMDRImpl.isValidNamespace(...)
FacadeMDRImpl.getModel(...)
FacadeMDRImpl.isAModel(...)
FacadeMDRImpl.isAFeature(...)
...

adding a class (50 invocations vs. 172) and because this smaller number of method invocations has a higher cohesion than that of the *Add class* feature. In addition, 32 of the remaining $88 - 50 = 38$ methods belong to the end of the trace and were not put in the same segment, while the sequence of these a methods continued after the tag with 24 other invocations. The continuation of the sequence *after* the tags means that the oracle is not precise enough. We explain this lack of precision by the extensive use of multi-threading in ArgoUML.

All methods related to setting properties through the Façade design pattern were not grouped in a same segment by our approach because these methods were invoked in a loop, in which each iteration of the loop contained a slightly different sequence of invocations. Consequently, (1) the RLE compression algorithm was not able to group together the various loop iterations and (2) the various iterations were not cohesive and thus the trace was split into several segments.

The *Cut rectangle* feature of JHotDraw has been tagged as a sequence of 172 method invocations. However, in the best case shown in Table 5.4, only 39 of these methods were grouped together by our approach, *i.e.*, the methods belonging to the last part of the tagged segment. We inspected this sequence and discovered that it is related to (1) add the rectangle content to the clip-

Table 5.7 Excerpt of segments identified by the approach.

Spawn window (JHotDraw)
JavaDrawApp.createTools(...)
MySelectionTool.MySelectionTool(...)
TextFigure.TextFigure()
TextFigure.setAttribute(...)
FigureAttributes.FigureAttributes()
FigureAttributes.set(...)
TextFigure.changed()
TextFigure.invalidate()
TextFigure.updateLocation()
TextTool.TextTool(...)
TextTool.TextTool(...)
TextFigure.TextFigure()
TextFigure.setAttribute(...)
FigureAttributes.FigureAttributes()
FigureAttributes.set(...)
TextFigure.changed()
TextFigure.invalidate()
TextFigure.updateLocation()
ConnectedTextTool.ConnectedTextTool(...)
ConnectedTextTool.ConnectedTextTool(...)
...

board, (2) modify the properties of the drawn rectangle so that it appears as "cut" in the painter, and (3) update the menu commands (*e.g.*, the command "Paste" is now enabled). The preceding sequence of $172 - 39 = 133$ methods was split in many small segments in which GUI events and actions performed by clicking the mouse button are interleaved, resulting in a sequence of loosely cohesive invocations.

The *Spawn window* feature of JHotDraw includes, in the manually-tagged segment, 197 method invocations ; the segment with the highest overlap only included, however, 72 of these invocations. This sequence of 72 method invocations is actually related to re-sizing and re-adjusting figures in the panel while spawning the window. The remaining invocations (at the end of the trace) kept out by our approach are mainly related to restoring and setting-up the status of the menu commands.

Finally, as previously explained for the *Add text* feature of JHotDraw, the low Jaccard score and low precision are due to the high similarity between the sequences of methods of the *Add text* and *Draw rectangle* features, which leads our approach to put together both features in a segment of 168 method invocations.

On the one hand, the previous discussion highlights the capability of our approach to split

Table 5.8 Excerpt of segments identified by the approach.

Cut rectangle (JHotDraw)
StorableOutput.close()
Clipboard.Clipboard()
Clipboard.getClipboard()
Clipboard.setContents(...)
CutCommand.deleteSelection()
BouncingDrawing.removeAll(...)
BouncingDrawing.figureRequestRemove(...)
AnimationDecorator.removeFromContainer(...)
AnimationDecorator.invalidate()
AnimationDecorator.removeFigureChangeListener(...)
AnimationDecorator.changed()
AnimationDecorator.invalidate()
AnimationDecorator.release()
RectangleFigure.removeFromContainer(...)
RectangleFigure.removeFigureChangeListener(...)
RectangleFigure.changed()
RectangleFigure.release()
RectangleFigure.removeFromContainer(...)
RectangleFigure.removeFigureChangeListener(...)
RectangleFigure.changed()
...

execution traces into conceptually cohesive segments, despite the low Jaccard overlap with respect to manually-tagged segments. On the other hand, it shows some difficulties in identifying concepts in execution traces, due to :

– design patterns and, in general, object-orientation mechanisms (*e.g.*, polymorphism, dynamic binding), which make traces for different features almost identical (*e.g.*, *Add text* and *Draw rectangle* in JHotDraw) ;

– imprecision when generating and tagging traces due to multi-threading ;

– the compression algorithm that is unable to group loop iterations consisting of slightly different sequences of method invocations.

In particular, despite the good results obtained by our approach when analyzing traces from JHotDraw (both with the Jaccard score and in precision), the extensive use of inheritance and design patterns in JHotDraw explain the lower results when compared to those obtained with ArgoUML. Inheritance and design patterns lead to the generation of many method invocations not directly related to a feature, but *supporting* and–or *enabling* the implementation of this feature. Consequently, these method invocations can appear in many different segments related to different

features and thus can be a confounding factor for our approach.

Another difficulty of trace-based concept location approaches is to deal with method invocations related to GUI and system events. For example, hundreds of method invocations in both ArgoUML and JHotDraw execution traces correspond to GUI events, such as `mousePressed(...)`. These methods are not feature-specific and can appear almost anywhere in a trace and could lead to different segmentation across different runs. We deal with these methods by compressing the traces, removing sub-sequences of such methods, and using conceptual cohesion and coupling measures, which lead to the creation of small segments containing only such method invocations.

5.5.5 Threats to validity

We now discuss the threats to validity that can have affected our empirical study.

Construct validity threats concern the relation between theory and observation. In this study, they are mainly due to measurement errors. The traces are automatically produced by executing the instrumented systems against some scenarios. Thus, the information contained in the traces is reliable. However, multi-threading could change the ordering of method calls in different traces exercising the same sub-scenarios. The performances of the proposed approach are evaluated by using the Jaccard overlap, already used in the past to evaluate concept location approaches and by using the standard IR precision measure, because we are also interested to split the trace into segments that only contain methods related to the feature of interest.

We only performed a preliminary assessment of the meaning of the identified concepts, by manually analyzing sequences of method invocations belonging to different segments. In future work, we plan to use automated techniques to label segments and thus better help the maintainer by assigning meanings to segments automatically.

Threats to *internal validity* concern confounding factors that could affect our results. The manually-tagged traces that we use as oracle pose such a threat. Indeed, it is possible that tags would appear in slightly different positions in the traces obtained by exercising the same scenarios in different runs. The slightly different positions result from multi-threading, as well as from me-

thod invocations related to mouse and other GUI events. In particular, extra method calls related to GUI events or other uncontrollable system events could be interleaved in the traces.

Methods declared in class libraries could also introduce "noise" in our approach. For example, calls to methods from the Java class libraries frequently occur in the traces obtained in our experiments. They do not occur frequently enough to be discarded as "utility" method calls yet are not related to interesting concepts. Therefore, in future work, we will consider adding these methods in our list of stop-words.

A last threat to internal validity relates to the intrinsic randomness of GAs. However, in **RQ1**, we showed that, overall, results are quite stable across different GA runs.

Reliability validity threats concern the possibility of replicating this study. We attempted to provide all the necessary details to replicate our study.

Threats to *external validity* concern the possibility to generalize our results. We studied two systems having different size and belonging to different domains. However, we are aware that this is a first study aimed at validating the proposed approach and that we only split traces on a small sample of scenarios for the two software systems. Other traces could possibly lead to different results. Also, further validation on a larger set of different systems is desirable. Yet, within its limits, our results confirm the stability and precision of our approach for concept location.

A final remark concerns the complexity of our approach and computation times. Although this is a proof of concept, we are aware that excessive computation times or complexity may prevent further studies and practical application. On average, identifying concepts in a compressed trace of about 400 methods on a single high end PC (*i.e.*, with at least 4GB RAM) took about one day ; when GA was mapped on multiple serves as described, the time went down to 20 minutes. Clearly, to make the approach appealing, we need to improve scalability both in time and space as well as in the possibility to handle longer traces.

CHAPITRE 6

GA AND DISTRIBUTED ARCHITECTURE

We started our experiments with a basic GA implementation running on a single computer. We found that computations were overly time consuming, impairing the possibility to actually obtain results in a reasonable amount of time. As an example, running an experiment with a compressed trace from JHotDraw v5.4b2, and the scenario *Start-DrawRectangle-Quit*, that contains 240 method calls, with a number of iterations equal to 2,000, took about 12 hours.

Table 6.1 Example of individual coding and segment redundancy

Id1	0001 0001 0000001 0001 0001
Id2	0001 0001 001 0001 0001 0001
Id3	001 00001 0000000001 0001
Id4	00001 0001 0000001 000001
Id5	0001 0001 0000001 0001 001 01

To reduce computation time, we decided to resort on the client–server architectural style, customized into more specific architectures detailed in the following. The rationale behind the different architectures comes from the illustrative population shown in Table 6.1 : several individuals share some segments. For example, the first two segments of individuals Id1, Id2, and Id5 are identical (*i.e.*, beginning and end are the same) ; Id1 and Id5 are almost identical but for their last segments. Thus, once Id1's fitness value is calculated, if segment cohesion and coupling were stored, they could be reused to compute the fitness values of Id2 and Id5.

6.1 Different Architectures

Using client–server architectural style, the computations are distributed among several machines. In fact our distributed GA implementations are based on the GA global parallelization model of Stender *et al.* Stender (1993) in which a computer acting as the master applies the ge-

netic operators on the individuals' genomes and distributes the individuals among slave computers (servers), which compute the fitness values of the individuals.

In the following, we minimally define that a client computer (master in Stender's work (Stender (1993))), performing mutation, crossover, and population evolution, distributes fitness computation to multiple servers, which compute the received individual's fitness value and return it back to the client.

6.1.1 A Simple Client Server Architecture

The simplest distributed client–server architecture is shown in Figure 6.1. The servers have no local memory, do not communicate among themselves or store data locally or on a global and shared device. The client sends the individuals' encodings to the servers and waits for the fitness values to be returned. Each server has only its own local LSI matrix and computes fitness values based on the equations presented in the previous section.

6.1.2 A Database Client Server Architecture

Figure 6.2 shows the architecture of a client–server in which a database server stores global shared storage device. When a segment cohesion or coupling value is required, a server first queries the database before computing it if missing.

The database holds two tables : a cohesion table and a coupling table, each with three columns. Each record in these tables keeps a similarity/coupling value for one segment. The first column, called beginning, keeps the index of the first method invocation in a segment and the second column keeps the index of the last method invocation in the same segment. The third column contains the cohesion/coupling value of that segment.

Whenever the fitness value for a new individual must be computed, the responsible server checks first the database. If it can find the needed values (already calculated in the last iterations or by other servers for other individuals), it uses these to compute the fitness value using a simple division. Else, it computes cohesion and coupling for the new segment and stores the values in the

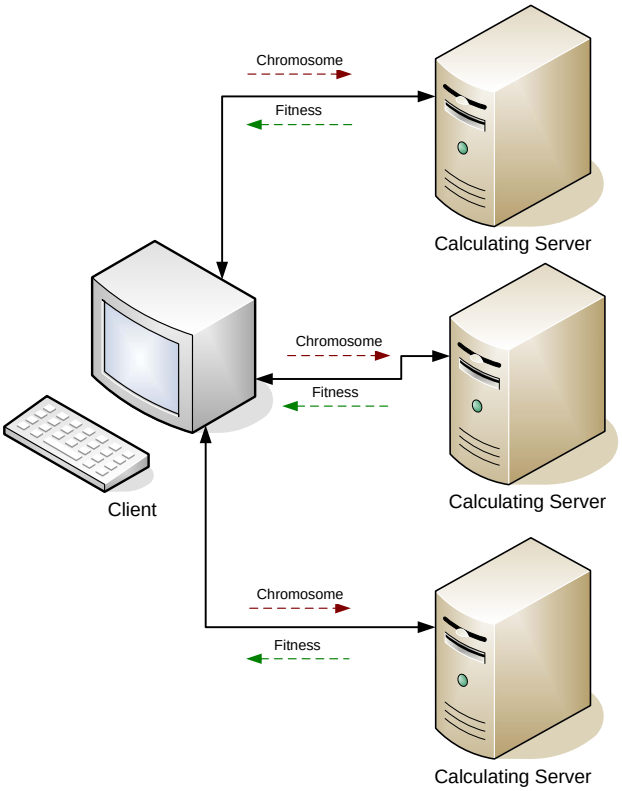

Figure 6.1 Baseline Client Server Configuration.

database. Thus, computation is performed if and only if the values can not be retrieved from the database : as much data as possible is shared between servers to reduce computation times.

There is an extra cost due to database queries and network communication. A central database implies that all servers write in and read from the same database. Yet, we would expected that using a database reduces the computation times by caching already-calculated values. However, sending data over the network, acquiring and releasing locks, and performing queries are also time consuming operations.

6.1.3 A Hash-database Client Server Architecture

To limit the possible communication between servers and the database, the architecture shown in Figure 6.3 was devised. The goal of this architecture is to further reduce computation time by decreasing the number of accesses to the central database using a local cache on each server,

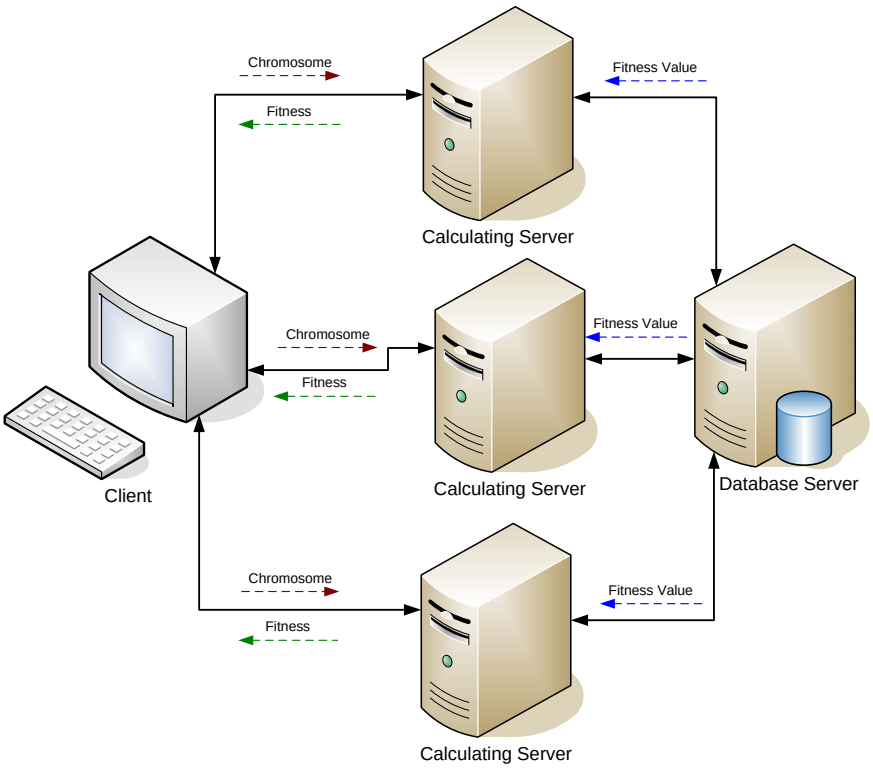

Figure 6.2 DBMS Client Server Configuration.

implemented with a hash table.

The architecture works as follows : whenever a server wants to compute the fitness value of a segment, it searches its hash table. If the required data does not exist in its local hash table, then the server queries the central database. If the server finds the required data in the database, it uses it to compute the fitness value and and stores it in its hash table, else it computes the required data and stores the results in both its hash table for its future use and in the central database for the other servers use. Figure 6.4 reports the flowchart of the process of this architecture.

6.1.4 A Hash Client Server Architecture

This last architecture is a compromise between the two previous ones : only local data is stored in the local hash table of servers. No data is shared among servers. As shown in Figure 6.5, servers only communicate with the client and no global data is kept and available.

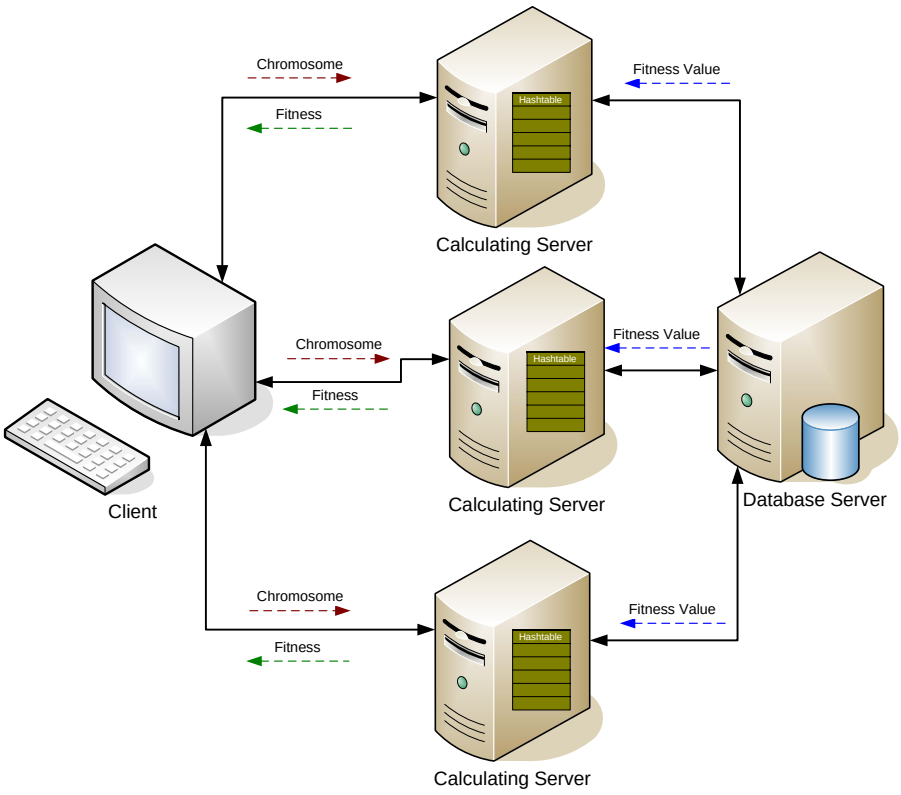

Figure 6.3 Database-Hash table Configuration for GA-Concept Identifier

Each server has two hash tables : one for similarity cohesion and the other for coupling values for each segment. The key of the hash tables is a combination of the indexes of the first and last method invocations of a segment. Each server uses its own hash tables and thus cannot benefit from the computation results of others. However, because all the data is stored locally and there is no access policy using locking algorithms, the access to the already-calculated data as well as their storage is efficient.

6.2 Results and Discussions

This section, reports the typical timing obtained with the different architectures on two compressed traces from JHotDraw.

The traces were collected by instrumenting JHotDraw and executing the scenarios *Start-DrawRectangle-Quit* and *Start-Spawn-Window-Draw-Circle-Stop*. These scenarios generated respectively traces of

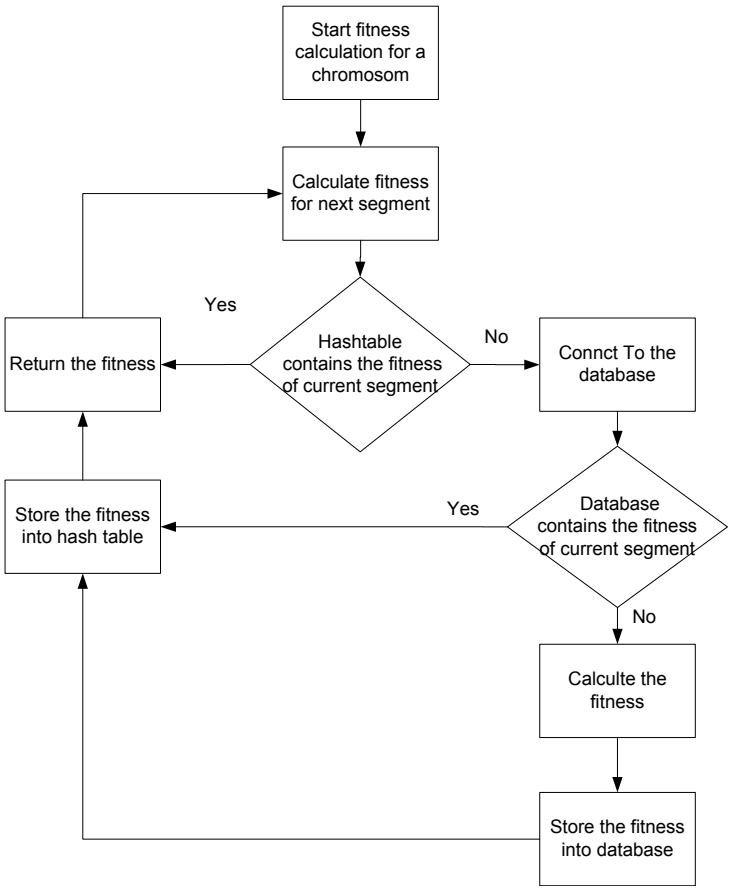

Figure 6.4 Flow chart of Database-Hash table configuration process

6,668 and 16,366 method calls ; once utility methods were removed their sizes are reduced to 447 and 670 calls. Finally RLE [1] compression brought down the numbers of distinct calls to 240 and 432.

In our experiments, we distributed computations over a sub-network of 14 workstations. Five high-end workstations, the most powerful ones, are connected in a Gigabit Ethernet LAN ; low-end workstations are connected to a LAN segment at 100 MBit/s and talk among themselves at 100 Mbit/s. Each experience was run on a subset of ten computers : nine servers and one client.

Workstations run CentOS v5 64 bits ; memory varies between four to 16 Gbytes. Workstations are based on Athlon X2 Dual Core Processor 4400 ; the five high-end workstations are either single or dual Opteron. Workstations run basic Unix services (*e.g.*, network file system, SAMBA,

1. Run Length Encoding

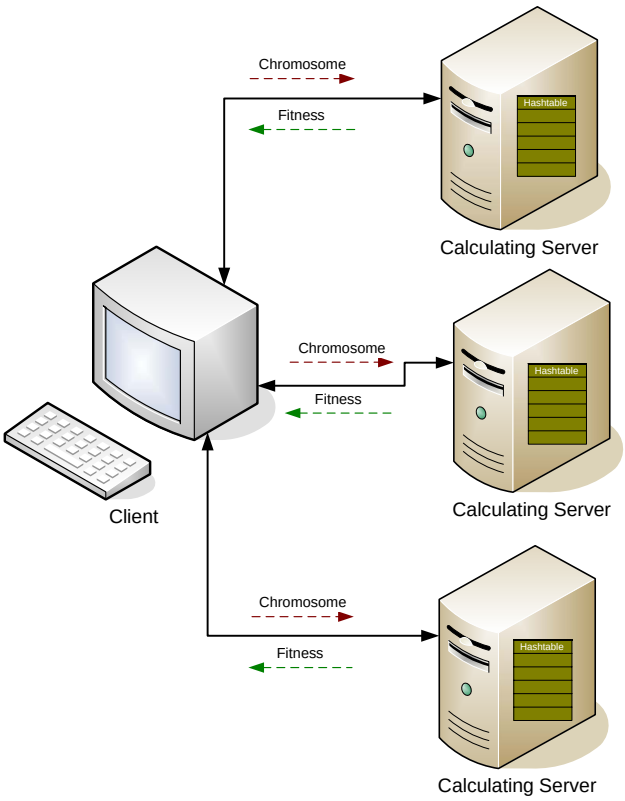

Figure 6.5 Hash Table Client Server Configuration.

mySQL) and user processes. User processes are typically editing and compilation of programs, e-mail clients, Web browsers, and so on. No special care was taken to ensure a specific network condition (*e.g.*, priorities were not altered) and thus times and ratios between times can be considered typical of a industrial or research environment. However, the sizes of the GA processes never exceeded the physical memory of the workstations to avoid paging ; workstations were managed to ensure that each computationally-intensive user processes had a dedicated CPU.

The client computer was also responsible to measure execution times and to verify the liveness of connections ; connections to servers as well as connections to the database were implemented on top of TCP/IP (AF_INET) sockets. All components have been implemented in Java 1.5 64bits. The database server, shown in Figures 6.2 and 6.3, was MySQL server v5.0.77.

Table 6.2 reports computation times for the different architectures. The times reported for the single-computer architecture come from an optimized implementation of our approach. In our first

Table 6.2 Computation times for desktop solution and the different architectures of Figures 6.1, 6.2, and 6.5 with the *Start-DrawRectangle-Quit* scenario – Compressed trace length of 240 methods

Time Measurement			
Architectures	Runs #	Measures	Average
Desktop	1	12 :09 h	12 :07 h
	2	11 :39 h	
	3	12 :21 h	
	4	11 :50 h	
	5	12 :38 h	
Client–server	1	1 :44 h	2 :01 h
	2	2 :36 h	
	3	1 :53 h	
	4	1 :40 h	
	5	2 :13 h	
Database	1	16 :36 h	13 :50 h
	2	15 :3 h	
	3	9 :52 h	
Hash Table	1	5 :13 m	5 :17 m
	2	5 :19 m	
	3	5 :20 m	
	4	5 :27 m	
	5	5 :10 m	

Table 6.3 Computation times for desktop solution and the architecture of Figure 6.5 with the *Start-Spawn-Window-Draw-Circle-Stop* scenario – Compressed trace length of 432 methods

Time Measurement			
Architectures	Runs #	Measures	Average
Desktop	1	45 :38 h	44 :07
	2	41 :28 h	
	3	45 :07h	
Hash Table	1	7 :21 m	7 :24 m
	2	7 :21 m	
	3	7 :32 m	

implementation, we reused the Java GALib library, which is freely available from SourceForge and implements a simple GA. GALib makes no assumptions on crossover and mutation operators and assumes that the fitness of an individual must be recomputed even if it was passed unchanged from the old generation to the new one. This recomputation resulted in about 30% of computation-time increase because between 20% and 30% of individuals are not subject to mutation or crossover between generations. Thus, to reduce computation time, we modified GALib to compute only the fitness values of individuals that have changed between the last generation and the current one.

Distributing the computation, shown in Figure 6.1, clearly results in an important reduction of computation time ; as shown in the second row of Table 6.2. Computation time went from 12

hours to about two hours ; however, the gain in terms of time reduction is considerably lower than expected as we had nine computers available (excluding the client) and, thus, expected computation times close to one hour.

We felt that there was still room for improvement and Amdahl's law Amdahl (1967) was only partially the reason for the reduced gain. We observed that the nature of our problem was such that crossover and mutation preserve a large fraction of segments unchanged and that for those segments, previous cohesion and coupling values could be reused.

Thus, we tested the two architectures in Figures 6.2 and 6.3. Table 6.2 in its third row reports results for such database client–server architecture : to our surprise, sharing data among servers via a central database increased computation times.

Finally, Table 6.2, in its last row, reports the computation times for the architecture in Figure 6.5, which is the fastest architectures. The gain in computation times obtained is of about 140 times. The implementation of this GA parallelization is moreover relatively simple.

We obtained similar gains with other traces. For example, the trace generated by the scenario *Start-Spawn-Window-Draw-Circle-Stop*, with the desktop architecture, was split in about 44 hours while, with the fastest architecture, the client–server with the hash table, computation time is of about 7 minutes. Table 6.3 reports the results of splitting the trace with two architectures.

6.2.1 Discussion

We conjecture that poor performance of the database architecture, in Figure 6.2, is mainly due to the database accesses (reading, writing, and locking) for the computation of each coupling and cohesion values. These frequent accesses are responsible for the increase in computation times. To limit the number of database accesses, we introduced the hybrid architecture in Figure 6.3. Results have not been reported in Table 6.2 because they are not substantially different (better) then those of the database. We are investigating the reason of this unexpected behavior to locate the bottleneck cause.

Indeed, in our current implementation, accesses to the local hash table and the database are

managed serially. Performance could improve by parallelizing writing in the database and access to the hash table and by loading the hash table only once at the beginning of each computation. Unfortunately, given the size of the search space and the huge number of possible segments, the probability that in two consecutive runs a relevant number of the segments will be exactly the same is very low. This fact makes the architecture in Figure 6.3 interesting from a theoretical point of view but not practical.

Despite the decrease in computation time, the very definition of the concept location problem makes it hard to obtain acceptable computation times for traces longer than few thousands of methods even with the fastest architecture, unless a higher number of servers is available. The definition of this problem is tied to the size of the search space, Equations 4.10 and 4.11, and the bit-string representation. Indeed, the longer the trace, the higher the number of methods contributing to the segment coupling. However, we believe that if concepts are indeed implemented in cohesive and decoupled segments, then computing coupling with Equation 4.11 is overly conservative and redefining the problem could substantially reduce computation time. We are currently working to restate the concept location problem in using audio digital signal processing and time windowing.

We have reported data of two traces of one software system, namely JHotDraw, therefore we cannot generalize to other traces though the performance issue is likely to be non-specific to JHotDraw or the used traces. Indeed, we experienced similar results with traces of different lengths of ArgoUML. Much in the same way, we cannot generalize to other search-based software engineering problems. However, we observed that the trade-off between the complexity of the fitness function and the local and global knowledge representations ; similar trade-offs are known to be general and common to many application of optimization techniques to software engineering.

6.3 Conclusion

In this chapter we presented and discussed four client–server architectures conceived to improve performance and reduce GA computation times to resolve the concept location problem. To our surprise, we discovered that on a standard TCP/IP network, the overhead of database accesses,

communication, and latency may impair a dedicated solutions. Indeed, in our experiments, the fastest solution was an architecture where each server kept track only of its computations without exchanging data with other servers. This simple architecture reduced GA computation by about 140 times when compared to a simple implementation, in which all GA operations are performed on a single machine.

CHAPITRE 7

CONCLUSION

Maintenance is the most expensive phase in a software life cycle. As time passes and products age, keeping the system updated with new user requirements becomes more and more difficult. Sometimes, it is required to modify a system in order to make it adaptive to new environment. Maintenance costs developers time, effort, and money. This requires that the maintenance phase be as efficient as possible (Erdil *et al*. (2006)).

There are several steps in the software maintenance phase. The first is to try to understand the program that already exists.

The problem with understanding a program is that usually the documentation is not complete and up-to-date. In many cases the only available source to understand a program is its source codes. Studying the source code is a time consuming, tedious task. There is no accepted standard that explains the process of program understanding. There are some studies that discuss direction of program understanding as well as its necessary depth. Some believe that top-down approaches is more effective in program understanding while others claim that bottom-up can result in better comprehension. There are also some discussion about using a combination of both to benefit from the advantages of each one of them. Observing the programmers activities during the program comprehension phase shows that some programmers tend to use systematic strategies while others prefer as-needed strategies.

Concept or feature location and identification is a technique to identify the source code constructs that are activated when exercising one or some of the features of a program. Although, many feature and concept identification approaches exist, none of these approaches attempt at identifying concepts in a system trace automatically.

In this thesis, we presented an approach to locate automatically concepts in execution traces by splitting traces into cohesive segments representing concepts related to software system features.

We combined IR techniques, dynamic analysis, and search-based optimization techniques to identify concepts into execution traces. The approach relies on definitions of conceptual cohesion and coupling from the literature (Marcus *et al*. (2008); Poshyvanyk et Marcus (2006a)). We have tried several optimization algorithms and we have chosen the genetic algorithm since it is more efficient and provides more accurate results.

We have tried many fitness equations and several representations of the problem to find the appropriate representation and fitness equation.

The approach has been applied and evaluated on two open source systems, ArgoUML and JHotDraw. Results showed that the approach is stable, and, overall, locates concepts with a high precision. Precision tends to drop for features realized using very similar sequences of methods, as sometimes happens in JHotDraw, where different kinds of shapes are drawn essentially in the same way. The overlaps between a manually-built oracle and the automatically-located segments vary depending on the cohesion of the features being analyzed, as the approach tends to split traces related to large features into smaller segments related to cohesive sub-concepts.

Contrary to other dynamic approaches that use traces generated according to a well designed scenario, our approach is not dependent to any designed scenario and can be applied to any traces generated during program execution. It is not limited to locate one feature at a time and is able to identify all the features executed in a trace.

To reduce the execution time, we have distributed GA computations among several servers. We have implemented four different client server configurations and we found one that speed up enormously the execution of the approach. As an example in the case of Start-Draw-Rectangle-Stop trace this configuration reduced GA computation by about 140 times when compared to a simple implementation, in which all GA operations are performed on a single machine.

The approach proposed in this thesis has the following advantages :

– It combines both static and dynamic analysis together and uses them as complements to have more effective results.
– It is fully automatic and does not need any interaction with the developer.

- Because of using LSI, the approach is independent from the programming language.
- It benefits from two software design principles, cohesion and coupling, already adopted in the past to identify modules in software systems.
- It uses a single execution trace that may or may not be generated based on a designed scenario.
- The developer does not need to know about the features executed in the trace.
- It is not limited to locate one feature at a time and can detect any number of features existing in the trace.
- The approach tries to identify all the methods contributing in implementing a concept not only the start points
- It can work with completely unfamiliar systems when the source code is available.
- It uses a distributed configuration to speed up the computations.

Considering the mentioned advantages, using this approach is easier and more effective than previous approaches proposed in literature.

Unfortunately the results we have obtained for the labeling part have not been promising so far. One reason may be that the terms we have extracted from the source codes are too general and they are not limited to developer-defined terms only.

7.1 Future Works

Future work will follow different directions. First, we are improving the proposed approach to increase its performance by better tuning the search-based optimization and the text indexing techniques.

As it was mentioned before we have not obtained the promised results for the automatic label assignment part. Thus one of our future works is trying to improve our idea and implementation to assign automatically appropriate, meaningful labels to the located concepts.

We are working on scalability issue to find an appropriate, practical solution to be able to apply this approach to very long traces too. The approach should also be extended to be able to work with

multi threaded systems.

We should compare the performance of our approach in detecting a desired concept, with some pervious approaches.

Setting the number of GA iterations based on the size of the execution traces needs to be investigated further. Also, a study to find the relation between the number of the servers, which are used in distributing the computations, and the execution time should be performed.

REFERENCES

AMDAHL, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS '67 : Proceedings of the spring joint computer conference*. ACM, New York, NY, USA, 483–485.

ANQUETIL, N. et LETHBRIDGE, T. (1998). Extracting concepts from file names : A new file clustering criterion. K. Futatsugi et R. Kemmerer, éditeurs, *Proceedings of the* 20^{th} *International Conference on Software Engineering*. IEEE Computer Society Press, 84–93.

ANTONIOL, G. et GUEHENEUC, Y. (2005). Feature identification : a novel approach and a case study. *Proceedings of IEEE International Conference on Software Maintenance*. IEEE Press, Budapest Hungary, 357–366.

ANTONIOL, G. et GUEHENEUC, Y.-G. (2006). Feature identification : An epidemiological metaphor. *Transactions on Software Engineering (TSE)*, 32, 627–641.

ASADI, F., ANTONIOL, G. et GUÉHÉNEUC, Y.-G. (2010a). Concept location with genetic algorithms : A comparison of four distributed architectures. *2nd International Symposium on Search Based Software Engineering (SSBSE 2010)*.

ASADI, F., PENTA, M. D., ANTONIOL, G. et GUEHENEUC, Y.-G. (2010b). A heuristic-based approach to identify concepts in execution traces. *14th European Conference on Software Maintenance and Reengineering (CSMR 2010)*, 31–40.

BAEZA-YATES, R. et RIBEIRO-NETO, B. (1999). *Modern Information Retrieval*. Addison-Wesley.

BIGGERSTAFF, T. J., MITBANDER, B. G. et WEBSTER, D. E. (1993). The concept assignment problem in program understanding. IEEE Computer Society Press / ACM Press, 482–498.

CHEN, K. et RAJLICH, V. (2000). Case study of feature location using dependence graph. A. von Mayrhauser et H. Gall, éditeurs, *Proceedings of the* 8^{th} *International Workshop on Program Comprehension*. IEEE Computer Society Press, 241–252.

DEERWESTER, S., DUMAIS, S. T., FURNAS, G. W., LANDAUER, T. K. et HARSHMAN, R. (1990). Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41, 391–407.

EADDY, M., AHO, A. V., ANTONIOL, G. et GUÉHÉNEUC, Y.-G. (2008). CERBERUS : Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. R. Krikhaar et R. Lämmel, éditeurs, *Proceedings of the* 16^{th} *International Conference on Program Comprehension (ICPC)*. IEEE Computer Society Press. 10 pages.

EDWARDS, D., SIMMONS, S. et WILDE, N. (2004). An approach to feature location in distributed systems. Rapport technique, Software Engineering Research Center.

EISENBARTH, T., KOSCHKE, R. et SIMON, D. (2003). Locating features in source code. *IEEE Transactions on Software Engineering*, 29, 210–224.

EISENBERG, A. D. et VOLDER, K. D. (2005). Dynamic feature traces : Finding features in unfamiliar code. *proceedings of the* 21^st *International Conference on Software Maintenance*. IEEE Press, 337–346.

ERDIL, K., FINN, E., KEATING, K., MEATTLE, J., PARK, S. et YOON, D. (2006). Software maintenance as part of the software life cycle. *Oracle Magazine, Technology articles for developers and DBAs and more.*

FRAKES, W. B. et BAEZA-YATES, R. (1992). *Information Retrieval Data Structures & Algorithms*. Prentice Hall.

FREE ENCYCLOPEDIA (2010). Wikipedia web-site. Http ://www.wikipedia.org/.

HARMAN, M., HASSOUN, Y., LAKHOTIA, K., MCMINN, P. et WEGENER, J. (2007). The impact of input domain reduction on search-based test data generation. *ESEC-FSE '07 : Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, New York NY USA, 155–164.

JACOBS, R. A. (1995). Methods for combining experts' probability assessments. *Neural Computation*, 7, 867–888.

KOZACZYNSKI, V., NING, J. Q. et ENGBERTS, A. (1992). Program concept recognition and transformation. *IEEE Transactions on Software Engineering*, 18, 1065–1075.

LALMAS, M., MACFARLANE, A., RUGER, S., TOMBROS, A. et TSIKRIKA, T.AND YAVLINSKY, A. (2006). *Advances in Information Retrieval*. 28th European Conference on IR Research, ECIR 2006.

LIENTZ, B. P. (1980). *Software Maintenance Management : A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley.

LIU, D., MARCUS, A., POSHYVANYK, D. et RAJLICH, V. (2007). Feature location via information retrieval based filtering of a single scenario execution trace. *ASE '07 : Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, New York NY USA, 234–243.

MALETIC, J. I. et MARCUS, A. (2001). Supporting program comprehension using semantic and structural information. *Proceedings of the 23rd International Conference on Software Engineering*, 103 – 112.

MARCUS, A., POSHYVANYK, D. et FERENC, R. (2008). Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34, 287–300.

MARCUS, A., SERGEYEV, A., RAJLICH, V., JONATHAN et MALETIC (2004). An information retrieval approach to concept location in source code. *Proceedings of the 11th Working Conference on Reverse Engineering*, 214 – 223.

MARIN, M., VAN DEURSEN, A. et MOONEN, L. (2007). Identifying crosscutting concerns using fan-in analysis. *ACM Trans. Softw. Eng. Methodol.*, 17.

MCGILL (July 01, 2010). Genetic algorithms survey.
http ://cgm.cs.mcgill.ca/eden/PrimitiveGenetics/Overview.htm.

MITCHELL, B. S. et MANCORIDIS, S. (2006). On the automatic modularization of software systems using the Bunch Tool. *IEEE Trans. Software Eng.*, 32, 193–208.

NG, J. K.-Y., GUÉHÉNEUC, Y.-G. et ANTONIOL, G. (2009). Identification of behavioral and creational design motifs through dynamic analysis. *Journal of Software Maintenance and Evolution : Research and Practice (JSME)*.

PORTER, M. F. (1980). An algorithm for suffix stripping. *Program*, 14, 130–137.

POSHYVANYK, D., GUÉHÉNEUC, Y.-G., MARCUS, A., ANTONIOL, G. et RAJLICH, V. (2007a). Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *Transactions on Software Engineering (TSE)*, 33, 420–432. 14 pages.

POSHYVANYK, D., GUEHNEUC, Y.-G., MARCUS, A., ANTONIOL, G. et RAJLICH, V. (2007b). Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33, 420–432.

POSHYVANYK, D. et MARCUS, A. (2006a). The conceptual coupling metrics for object-oriented systems. *Proceedings of 22nd IEEE International Conference on Software Maintenance*. IEEE CS Press, Philadelphia, Pennsylvania, USA, 469 – 478.

POSHYVANYK, D. et MARCUS, A. (2006b). The conceptual coupling metrics for object-oriented systems. *ICSM '06 : Proceedings of the 22nd IEEE International Conference on Software Maintenance*. IEEE Computer Society, Washington DC USA, 469–478.

SALAH, M., MANCORDIS, S., ANTONIOL, G. et DI PENTA, M. (2005). Towards employing use-cases and dynamic analysis to comprehend Mozilla. *proceedings of the* 21^{st} *International Conference on Software Maintenance*. IEEE Press, 639–642.

SALAH, M. et MANCORIDIS, S. (2004). A hierarchy of dynamic software views : From object-interactions to feature-interactions. M. Harman et B. Korel, éditeurs, *Proceedings of the* 20^{th} *International Conference on Software Maintenance*. IEEE Computer Society Press, 72–81.

SINGHAL, A. (2001). Modern information retrieval : a brief overview. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*.

STENDER, J. (1993). *Parallel Genetic Algorithm : Theory and Applications*, vol. 14 Frontiers in Artificial Intelligence and Applications.

TONELLA, P. et CECCATO, M. (2004). Aspect mining through the formal concept analysis of execution traces. *Proceedings of IEEE Working Conference on Reverse Engineering*. 112–121.

WILDE, N. et SCULLY, M. C. (1995). Software reconnaissance : Mapping program features to code. K. H. Bennett et N. Chapin, éditeurs, *Journal of Software Maintenance : Research and Practice*. John Wiley & Sons, 49–62.

WONG, W. E., GOKHALE, S. S. et HORGAN, J. R. (2000). Quantifying the closeness between program components and features. *Journal of Systems and Software*, 54, 87 – 98. Special Issue on Software Maintenance.

APPENDIX A

Published Article in CSMR 2010

The article is published at the The European Conference on Software Maintenance and Reengineering (CSMR) 2010. The article entitled *"A Heuristic-based Approach to Identify Concepts in Execution Traces"* by Fatemeh Asadi, Massimiliano di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc (Asadi *et al*. (2010b)).

A Heuristic-based Approach to Identify Concepts in Execution Traces

Fatemeh Asadi*, Massimiliano Di Penta**, Giuliano Antoniol*, and Yann-Gaël Guéhéneuc***
fatemeh.asadi@polymtl.ca dipenta@unisannio.it antoniol@ieee.org yann-gael.gueheneuc@polymtl.ca

* SOCCER Lab. – DGIGL, École Polytechnique de Montréal, Québec, Canada
** Department of Engineering, University of Sannio, Benevento, Italy
*** Ptidej Team – DGIGL, École Polytechnique de Montréal, Québec, Canada

***Abstract*—Concept or feature identification, *i.e.*, the identification of the source code fragments implementing a particular feature, is a crucial task during software understanding and maintenance. This paper proposes an approach to identify concepts in execution traces by finding cohesive and decoupled fragments of the traces. The approach relies on search-based optimization techniques, textual analysis of the system source code using latent semantic indexing, and trace compression techniques. It is evaluated to identify features from execution traces of two open source systems from different domains, JHotDraw and ArgoUML. Results show that the approach is always able to identify trace segments implementing concepts with a high precision and, for highly cohesive concepts, with a high overlap with the manually-built oracle.**

***Keywords*—Concept location, dynamic analysis, information retrieval.**

I. Introduction

Software systems often lack an adequate and up-to-date documentation. Therefore, developers must resort to reading the system source code, without specific tool support but code browsers, to understand the systems and perform their maintenance and evolution tasks. In some cases, code understanding is supported by static analysis and–or visualizations built upon static information. In other case, debugging can be used to understand the behavior of a system in a particular context and–or to locate a fault. However, manually browsing of source code, inspecting an execution trace or debugging long sequences of instructions are time consuming and daunting tasks.

Concept or feature location and identification aim at helping developers to perform their maintenance and evolution tasks, by identifying abstractions (*i.e.*, features) and the location of the implementation of these abstractions. Specifically, they aim at identifying *code fragments*, *i.e.*, set of method calls in traces and the related method declarations in the source code, responsible for the implementation of domain concepts and–or user-observable features [1], [2], [3], [4], [5]. The literature reports approaches built upon static [6] and dynamic [7], [5] analyses; Information Retrieval (IR) [4] and hybrid (static and dynamic) [8] techniques.

This paper proposes a novel approach to identify cohesive and decoupled fragments in execution traces, which likely participate in implementing concepts related to some features. A typical problem for which the proposed approach can be beneficial is the following. Suppose a failure has been observed when executing a particular scenario of a software system; unfortunately the likelihood to reproduce the execution conditions for that failure are very low. Maintainers are then faced with the problem of analyzing the execution trace produced by that scenario and identifying high level abstractions that likely participate in the feature producing the unwanted behavior.

To deal with the above described problem, the proposed approach identifies concepts composing an execution scenario by grouping together methods that are (i) sequentially invoked together/in sequence and (ii) cohesive and decoupled from a conceptual point of view. The underlying assumption is that, if a specific feature is being executed within a complex scenario (*e.g.*, "Open a Web page from a browser" or "Save an image in a paint application"), then the set of methods being invoked is likely to be conceptually cohesive, decoupled from those of other features, and sequentially invoked. We use conceptual cohesion and coupling from Marcus *et al.* [9] and Poshyvanyk *et al.* [10].

The approach works as follows. First, we index the source code of each method of a system textually. Then, we instrument and exercise the system to collect execution traces for some scenarios related to different features and, therefore, containing sets of different concepts. We compress the traces to remove utility and cross-cutting concepts and to abstract repetitions of the same sub-sequences of methods. Finally, we apply a search-based optimization technique, *i.e.*, a genetic algorithm, to split the compressed traces into cohesive and decoupled fragments. We ensure performances by parallelizing the algorithm over multiple computers.

Overall, the contributions of this paper are:

1) A novel approach combining IR techniques, dynamic analysis, and search-based optimization techniques to identify concepts into execution traces;
2) An empirical study that shows the applicability and the performances of the proposed approach in identifying concepts into execution traces of two systems, JHotDraw and ArgoUML. Results indicate that the

approach is able to identify concepts (with a precision in most cases greater than 80%), while the overlap with a manually-built oracle varies depending on the cohesiveness of the concepts to be identified.

The remainder of the paper is organized as follows. Section II presents related work. Section III describes the approach. Section IV presents an empirical study and Section V report its results and some discussions. Section VI concludes the paper and outlines future work.

II. Related Work

Although, many feature and concept identification approaches exist, none of these approaches attempts to identify concepts in a system trace *automatically*.

In their pioneering work, Wilde and Scully [7] presented the first approach to identify features by analyzing execution traces. They used two sets of test cases to build two execution traces, one where a feature is exercised and another where the feature is not. They compared the execution traces to identify the feature in the system. Similarly, Wong *et al.* [11] analyzed execution slices of test cases to identify features in source code. Wilde's original idea was later extended in several works [1], [4], [12], [13] to improve its accuracy by introducing new criteria on selecting execution scenarios and by analyzing the execution traces differently.

Chen and Rajlich [14] developed an approach to identify features using Abstract System Dependencies Graphs (ASDG). In C, an ASDG models functions and global variables as well as function calls and data flow in a system source code. Chen and Rajlich identified features using the ASDG following a precise manual process. In contrast to Wilde and Scully's work, Chen and Rajlich used only static data to identify features and a manual process.

Eisenbarth *et al.* [15] combined previous approaches by using both static and dynamic data to identify features. In a following work, Eisenbarth *et al.* [13] introduced an approach to feature identification using test cases.

Salah and Mancoridis [16] used both static and dynamic data to identify features in Java systems. They went beyond feature identification by creating feature-interaction views, which highlight dependencies among features. Their work was extended to allow feature identification and evolution analysis in large-scale systems, *e.g.*, Mozilla [17].

More recent pieces of work focused on a combination of static and dynamic data [8], [4], in which, essentially, the problem of features location from multiple execution traces is modeled as an IR problem, which has the advantage to simplify the location process and, often, improves accuracy [4]. Yet, Liu *et al.* [18] showed that a single trace suffices to build an IR system and locate useful data. Execution traces were also used to mine aspects by Tonella and Ceccato [5].

We share with previous work the use of dynamic data and IR techniques to identify features. However, instead of querying traces using an IR technique, *e.g.*, similar to Poshyvanyk *et al.* [4], we determine cohesive and decoupled fragments likely being relevant to a concept *automatically*. Our approach is based on two conjectures not yet fully investigated: (1) methods helping to implement a concept are likely to share some linguistic information; (2) methods responsible to implement a feature are likely to be called close each other in an execution trace. Therefore, the conceptual coupling of methods participating in a concept should be high and these methods should appear relatively close together in the execution trace. The first conjecture is grounded on the findings published in [4] and other publications based on IR to locate features and concepts. IR-inspired works assume some form of commonalities between a query and linguistic information of entities. Locality of concept manifestation in traces is more questionable, however we believe unlikely that a user-observable feature or concept, not constituting a crosscutting concern, will be uniformly spread in a trace.

III. The Approach

This section describes the proposed approach to identify concepts by analyzing execution traces. The approach consists in five steps. First, the system is instrumented. Second, the system is exercised to collect execution traces. Third, the collected traces are compressed to reduce the search space that must be explored to identify concepts. Fourth, each method of the system is represented by means of the text that it contains. Fifth, a search-based optimization technique is used to identify, within execution traces, sequences of method invocations that are related to a concept.

A. Steps 1 and 2 – System Instrumentation and Trace Collection

First, the software system is instrumented using the *instrumentor* of MoDeC. MoDeC is a tool to extract and model sequence diagrams from Java systems [19]. The MoDeC instrumentor is a dedicated Java bytecode modification tool implemented on top of the Apache BCEL bytecode transformation library[1]. It inserts appropriate and dedicated method invocations in the system to trace method/constructor entries/exits, taking care of exceptions and system exits (`System.exit(int)`). It also allows the user to add to the traces tags containing meta-information *e.g.*, delimiting and labeling sequences of method calls related to some specific features being exercised.

Following the user manual (or use-case documents, if available) of the system to be analyzed, an execution scenario is composed of a sequence of cohesive steps. For example, exercising a Web browser could consist in a sequence of the following steps (i) open the browser; (ii) insert a URL and access a Web page; and (iii) save the Web page into a local HTML file. We exercise the instrumented

[1] http://jakarta.apache.org/bcel/

system to collect execution traces by following execution scenarios. Resulting traces are text files listing method calls and including the class of the object caller, the unique ID of the caller, the class of the receiver, its unique ID, and the complete signature of the method.

B. Step 3 – Pruning and Compressing Traces

Usually, execution traces contain methods invoked in most scenarios, *e.g.*, methods related to logging. Even in a single execution trace of an application with graphical user interface, mouse tracking methods will largely exceed all other method invocations. It is likely that such methods are not related to any particular concept, *i.e.*, they are utility methods. These methods do not provide useful information for developers when locating a concept, because they are common in many concepts. They are similar to low-discriminating terms occurring in many documents when applying a IR technique. Such terms are penalized by indexing measures like *tf-idf* [20].

Similarly, we built the distributions of the frequencies of method occurrences to remove too-frequent methods. We then prune out the methods having a frequency greater than $Q3 + 2 \cdot IQR$, where $Q3$ is the third quartile (75% percentile) of the distribution and IQR is the inter-quartile range. An alternative approach to deal with these methods is aspect mining [5], [21], because such methods can constitute crosscutting concerns. We do not use aspect mining because we are interested in pruning these methods to identify concepts, not in crosscutting concerns.

Traces often contain repetitions of one or more method invocations, for example `m1(); m1(); m1();` or `m1(); m2(); m1(); m2();`. A repetition does not introduce a new concept, thus we compress traces using the Run Length Encoding (RLE) algorithm to remove repetitions and keep one occurrence of any repetition only. The two previous examples would become `m1()` and `m1(); m2()`, respectively. Compression is performed for any sub-sequences of method invocations having an arbitrary length. Other encoding schema such as suffix trees or LZH algorithm are likely to produce even better results in a future work.

C. Step 4 – Textual Analysis of Method Source Code

To determine the conceptual cohesion of methods, our approach uses the Conceptual Cohesion metric defined by Marcus *et al.* [9]. We extract a set of terms from each method by tokenizing the method source code and comments, pruning out special characters, programming language keywords, and terms belonging to a stop-word list for the English language. (We assume that comments appearing on top of the method declaration belong to the following method.)

We split compound terms following the Camel Case naming convention at each capitalized letter, *e.g.*, `getBook` is split into `get` and `book`. Then, we stem the obtained simple terms using a Porter stemmer [22].

Once terms belonging to each method have been extracted, we index these terms using the *tf-idf* indexing mechanisms [20]. We obtain a term–document matrix, where documents are all methods of all classes belonging to the system under study and where terms are all the terms extracted (and split) from the method source code.

Finally, we apply Latent Semantic Indexing (LSI) [23] to reduce the term–document matrix into a concept–document matrix. The meaning of "concept" in LSI is different from that of "concept" in concept location. In LSI, a concept is one of the orthonormal dimensions of the LSI space. Following Marcus *et al.* [9], we compute the conceptual cohesion of methods in a class in the LSI subspace to deal with synonymy, polysemy, and term dependency. The chosen size of the LSI subspace is 50.

D. Step 5 – Search-based Concept Location

We now segment execution traces into conceptually-cohesive segments related to the feature being exercised (and thus to a specific concept). Determining a (near) optimal splitting of a trace into segments is NP-hard. Therefore, we use a genetic algorithm to perform the splitting and parallelize its computations.

1) Choice of the Optimization Technique: We experimented different techniques: hill climbing, simulated annealing, and genetic algorithms (GAs). We chose to use GAs because they outperformed other techniques due to the characteristics of the search space.

A GA may be defined as an iterative procedure that searches for the best solution to a given problem among a constant-size population [24]. The search starts from an initial population of individuals, represented by finite strings of symbols (the *genome*), often randomly generated. At each evolution step, individuals are evaluated using a *fitness function* and selected using a *selection mechanism*. High-fitness individuals have the highest reproduction probability. The evolution (*i.e.*, the generation of a new population) is affected by two genetic operators: the *crossover operator* and the *mutation operator*. The crossover operator takes two individuals (the *parents*) of one generation and exchanges parts of their genomes, producing one or more new individuals (the *offspring*) in the new generation. The mutation operator prevents the convergence to local optima: it randomly modifies an individual's genome (*e.g.*, by flipping some of its symbols).

2) Use of the Optimization Technique: Our representation of an individual is a bit-string as long as the execution trace in which we want to identify some feature-related concepts. Each method invocation is represented as a "0", except the last method invocation in a segment, which is represented as a "1". For example, the bit-string

$$\underbrace{00010010001}_{11}$$

$$SegmentCohesion_k = \frac{\sum_{i=begin(k)}^{end(k)-1} \sum_{j=i+1}^{end(k)} similarity(method_i, method_j)}{(end(k) - begin(k) + 1) \cdot (end(k) - begin(k))/2} \quad (1)$$

$$SegmentCoupling_k = \frac{\sum_{i=begin(k)}^{end(k)} \sum_{j=1, j<begin(k) \text{ or } j>end(k)}^{l} similarity(method_i, method_j)}{(l - (end(k) - begin(k) + 1)) \cdot (end(k) - begin(k) + 1)} \quad (2)$$

$$fitness(individual) = \frac{1}{n} \cdot \sum_{k=1}^{n} \frac{SegmentCohesion_k}{SegmentCoupling_k} \quad (3)$$

Table I
EXAMPLE OF GA INDIVIDUAL REPRESENTATION (SECOND COLUMN).

Method Invocations	Repr.	Segments
TextTool.deactivate()	0	
TextTool.endEdit()	0	
FloatingTextField.getText()	0	
TextFigure.setText-String()	0	1
TextFigure.willChange()	0	
TextFigure.invalidate()	0	
TextFigure.markDirty()	1	
TextFigure.changed()	0	
TextFigure.invalidate()	0	
TextFigure.updateLocation()	0	2
FloatingTextField.endOverlay()	0	
CreationTool.activate()	1	
JavaDrawApp.setSelectedToolButton()	0	
ToolButton.reset()	0	
ToolButton.select()	0	
ToolButton.mouseClickedMouseEvent()	0	3
ToolButton.updateGraphics()	0	
ToolButton.paintSelectedGraphics()	0	
TextFigure.drawGraphics()	0	
TextFigure.getAttributeString()	1	

means that the trace, containing 11 method invocations, is split into three segments decomposed into the first four method invocations, the next three, and the last four. An real example of segment splitting[2] is shown in Table I.

The mutation operator randomly chooses one bit in the representation and flips it over. Flipping a "0" into a "1" means splitting an existing segment into two segments, while flipping a "1" into a "0" means merging two consecutive segments. The crossover operator is the standard 2-points crossover. Given two individuals, two random positions x, y with $x < y$ are chosen in one individual's bit-string and the bits from x to y are swapped between the two individuals to create a new offspring. The selection operator is the roulette-wheel selection. We use a simple GA with no elitism, *i.e.*, it does not guarantee to retain best individuals across subsequent generations. We set the population size to 200 individuals and a number of generations of 2,000 for shorter traces (those of JHotDraw) and 3,000 for longer ones (those of ArgoUML). The crossover probability was set to 70% and the mutation to 5%, which are widely used values in many GA applications.

A fitness function drives the GA to produce individuals that represent (near) optimal splittings of a trace into segments likely to relate to some concepts. In our fitness function, we use the software design principles of cohesion and coupling, already adopted in the past to identify modules in software systems [25], although we use conceptual (*i.e.*, textual) cohesion and coupling measures [9], [10], rather than structural cohesion and coupling measures.

Segment cohesion is the average (textual) similarity between any pair of methods in a segment k and is computed using the formulas in Equation 1 where $begin(k)$ is the position (in the individual's bit-string) of the first method invocation of the k^{th} segment and $end(k)$ the position of the last method invocation in that segment. The similarity between two methods is computed using the cosine similarity measure over the LSI matrix extracted in the previous step. Thus, it is the average of the similarity defined by [9], [10] to all pairs of methods in a given segment.

Segment coupling is the average similarity between a segment and all other segments in the trace, computed using Equation 2, where l is the trace length. As our conjecture is that a concept should be implemented by method calls locally close each other, on the one hand the algorithm favors the merging of consecutive segments containing methods with high average conceptual similarity. On the other hand, the algorithm penalizes solutions where consecutive segments are highly coupled together. The segment coupling represents, for a given segment, the average similarity between methods in that segment and those in different ones.

Finally, for a trace split into n segments, the fitness function is shown in Equation 3.

3) Parallelization of the Optimization Technique: One of the main advantages of GAs with respect to other optimization techniques is the possibility of parallelizing their computations, *e.g.*, the evaluations of the fitness of different individuals. In our approach, we use parallelization to reduce computation time. (However, a detailed study of the performances is out of scope of this paper and will be treated in a future work).

In our experiments, we distributed computations over a network of five servers and nine workstations. Servers are connected in a Gigabit Ethernet LAN while workstations are connected to a LAN segment at 100 MBit/s and talk to servers at 100 Mbit/s. Servers and workstations run CentOS

[2]The segment splitting shown in Table I has been obtained randomly and does not correspond to an actual "good" solution.

Table II
STATISTICS FOR THE TWO SYSTEMS.

Systems	NOC	KLOC	Release Dates
ArgoUML v0.18.1	1,267	203	30/04/05
JHotDraw v5.4b2	413	45	1/02/04

5, 64 bits; memory varies between four and 16 Gbytes. Workstations are based on Athlon X2 Dual Core Processor 4400; the five servers are either single or dual Opteron. The distributed architecture comprises one workstation taking charge of distributing the GA individuals to other computers by means of socket connections. On the slave computers, a server receives the computation requests and the individuals, computes the value of the fitness function and returns the value back to the central computer. Only the fitness of new individuals, with respect to previous generations, are computed. The fitness values of individuals already evaluated are not recomputed but retrieved from a hash table storing the previous values.

Distribution over several computers is crucial to ensure acceptable computation time. With the described configuration, a single run takes about one hour. Scalability on very large traces will require different computation architectures (*e.g.*, sharing information between slaves) and possibly dividing a large trace into chunks with approaches inspired by overlapping time windows as in digital signal processing. This possibility will be studied in a future work.

IV. EMPIRICAL STUDY DESCRIPTION

We report an empirical study evaluating the proposed concept location approach. The *goal* of this study is to analyze the novel concept location approach based on dynamic data, with the *purpose* of evaluating its capability of identifying meaningful concepts. The *quality focus* is the accuracy and completeness of the identified concepts. The *perspective* is that of researchers who want to evaluate how the proposed approach can be used during maintenance and evolution. The *context* consists of an implementation of our approach and of the execution traces extracted from two open source systems, JHotDraw and ArgoUML.

A. Context

The context of our study are execution traces from ArgoUML and JHotDraw. Figure IV highlights main characteristics of the two systems. *ArgoUML*[3] is an open source UML modeling tool with advanced software design features, such as reverse engineering and code generation. The ArgoUML project started in September 2000 and is still active. We analyzed release 0.19.8. *JHotDraw*[4] is a Java framework for drawing 2D graphics. JHotDraw started in October 2000 with the main purpose of illustrating the use of design patterns in a real context. We analyzed release 5.1.

[3]http://argouml.tigris.org
[4]http://www.jhotdraw.org

Table III
STATISTICS FOR THE COLLECTED TRACES.

Systems	Scenarios	Original Size	Cleaned Sizes	Compressed Sizes
ArgoUML	Start, Create note, Stop	34,746	821	588
	Start, Create class, Create note, Stop	64,947	1066	764
JHotDraw	Start, Draw rectangle, Stop	6,668	447	240
	Start, Add text, Draw rectangle, Stop	13,841	753	361
	Start, Draw rectangle, Cut rectangle, Stop	11,215	1206	414
	Start, Spawn window, Draw circle, Stop	16,366	670	433

We generate traces by exercising various scenarios in the two systems. Table IV-A summarizes the scenarios and shows that the generated traces include from 6,000 up to almost 65,000 method invocations. The compressed traces include from 240 up to more than 750 method invocations. By exercising these scenarios, we do not want to identify concepts related to the systems' startup, *Start*, and closing, *Stop*. Therefore, in the following, when naming the scenarios and their associated features, we no longer include the Start and Stop concepts.

The GA was implemented using the *Java GA Lib*[5] library.

B. Building the Oracle

We need an oracle to assess the accuracy and completeness of the identified concepts. We build such an oracle by manually tagging the execution traces. Two "Start" and "Stop" tags enclose the method invocations related to a particular concept. While executing the instrumented system, before and after a step in the execution scenario (*e.g.*, Draw rectangle), the user, through a command in the instrumentor interface, inserts the appropriate tags in the execution trace and then continues to exercise the instrumented system with the next steps of the scenario. Consequently, the collected traces are composed of a sequence of method invocations interleaved with tags separating the invocations belonging to different steps.

C. Research Questions

This study aims at answering the three following research questions:

- **RQ1:** *How stable is the GA, through multiple runs, when identifying concepts into execution traces?* Approaches based on GAs could suffer from the randomness of the search: the initial individuals are randomly generated and the crossover, mutation, and selection operators are influenced by random choices. However, it is desirable that the representation, operators, fitness,

[5]http://sourceforge.net/projects/javagalib/

and other settings (*e.g.*, population size and stopping criteria) be chosen so that multiple runs of the GA yields to similar solutions.

- **RQ2:** *To what extent the identified concepts match the ones in the oracle?* We are interested to evaluate the extent to which the identified segments overlaps with the ones in our oracle, obtained by manually tagging the traces.
- **RQ3:** *How accurate is the identification of concepts in execution traces?* Finally, we are interested to evaluate the extent to which the identified segments are accurate, *i.e.*, how many of the method invocations that they contain are also in the oracle and how many are not.

D. Study Settings and Analysis Method

To answer **RQ1**, we evaluate the extent to which the segments identified in multiple runs of the GA, and occurring in the same position of the trace, overlap each other. Let us consider a compressed trace composed of N method invocations $T \equiv m_1, \ldots m_N$ and partitioned at run i of the GA in k_i segments $s_{1,i} \ldots s_{k,i}$. For each segment $s_{x,i}$ obtained at run i, and for all the segmentations obtained at run $j \neq i$, we compute the maximum overlap between $s_{x,i}$ and the segments obtained at run j as follows:

$$max(Jaccard(s_{x,i}, s_{y,j})), y = 1 \ldots k_j$$

where:

$$Jaccard(s_{x,i}, s_{y,j}) = \frac{|s_{x,i} \cap s_{y,j}|}{|s_{x,i} \cup s_{y,j}|}$$

and where union and intersection are computed considering method invocations occurring at a given position in the trace. Stability is evaluated by means of descriptive statistics computed across the above obtained overlap values.

RQ2 is answered similarly to **RQ1** but, in this question, we compare the overlap between manually-tagged segments in the execution traces with segments identified by our approach. Specifically, given the segments determined by the tags in the trace (our oracle) and given the segments obtained by an execution of the system, we compute the overlap between each manually-tagged segment in the trace and the most similar automatically-identified segment.

Finally, **RQ3** is addressed like **RQ2**, with the only difference that we use precision instead of the Jaccard score, because we are interested in evaluating the accuracy of our approach. Precision is defined as follows:

$$Precision(s_{x,i}, s_{y,o}) = \frac{|s_{x,i} \cap s_{y,o}|}{|s_{y,o}|}$$

where $s_{x,i}$ are segments obtained by our approach and $s_{y,o}$ are segments in the corresponding trace in the oracle.

V. Results and discussion

This section reports the results of our experimental evaluation: the collected data and their analyses to address the previous research questions.

A. RQ1: How Stable is the GA across Multiple Runs?

We assess the stability of the GA by computing the average similarity of the segments identified in ten different runs of the approach. Table IV shows the similarity results. Overall, the similarity averages for JHotDraw range between 55% and 95%, with median values ranging between 70% and 84% . They are slightly higher for ArgoUML, between 80% and 83%. Thus, we conclude that, despite the potentially large size of the search space, our approach is able to generate stable segments across multiple runs. In addition, increasing the number of generations and the population size would potentially further increase the approach stability.

B. RQ2: To What Extent the Identified Concepts Match the Ones in the Oracle?

To address **RQ2**, we evaluate the extent to which the segments actually reflect features as they were manually tagged when executing the instrumented system to generate the execution traces.

For some features, *e.g.*, drawing a rectangle or a circle, the average (and median) Jaccard overlap is very high, suggesting that the features are implemented through sequences of very cohesive methods. Yet, other features exhibit lower overlaps. These lower overlaps do not mean that our approach was unable to successfully identify the features. Indeed, in some cases, for example the scenarios *Add text* in JHotDraw and *Create note* in ArgoUML, the features are realized by adapting a textual-editing feature as a shape-drawing feature, using the Adapter design pattern. The feature adaptation produces sequences of methods with a low cohesion, which our algorithm tend to split. As a consequence, the resulting overlaps are appropriately low.

In other cases, in particular with traces from ArgoUML, a large trace segment corresponding to a feature is split into two or more segments by our approach. Thus, the overlap between the (larger) manually-tagged segment and the corresponding automatically-identified segment is low. A manual study of such cases revealed that the manually-tagged segment is indeed composed of several smaller and cohesive sub-concepts that our algorithm tend to split, as illustrated and discussed in the following subsection.

C. RQ3: How Accurate is the Identification of Concepts in Execution Traces?

The right side of Table V reports the precision of the identified segments with respect to the manually-tagged ones. Precision is often very high, with median values in most cases above 85% and very often equal to 100%.

Lower precision values sometimes occur with explainable reasons. For example, in the scenario (2) of JHotDraw, composed of *Add text* and *Draw rectangle*, the two features are implemented using a very similar sequence of method invocations, making them hard to distinguish. Because these features are executed one after the other, our search-based

Table IV
DESCRIPTIVE STATISTICS OF SIMILARITY AMONG SEGMENTS OBTAINED IN TEN DIFFERENT RUNS.

Systems	Scenarios/Features	Similarity Averages				
		Min.	Max.	Mean	Median	σ
ArgoUML	(1) Add note	0.69	0.95	0.84	0.83	0.07
	(2) Add class, Add note	0.65	0.98	0.80	0.80	0.06
JHotDraw	(1) Draw rectangle	0.55	0.96	0.76	0.76	0.12
	(2) Add text, Draw rectangle	0.54	0.93	0.72	0.70	0.10
	(3) Draw rectangle, Cut rectangle	0.73	0.93	0.85	0.84	0.05
	(4) Spawn window, Draw circle	0.67	0.86	0.76	0.76	0.04

Table V
SIMILARITY (JACCARD OVERLAP AND PRECISION) BETWEEN SEGMENTS IDENTIFIED BY THE APPROACH AND FEATURES TAGGED IN THE TRACE.

Systems	Scenarios	Features	Jaccard					Precision				
			Min.	Max.	Mean	Median	σ	Min.	Max.	Mean	Median	σ
ArgoUML	(1)	Add note	0.15	0.39	0.28	0.27	0.08	0.91	1.00	0.97	1.00	0.04
	(2)	Create class	0.11	0.28	0.22	0.25	0.05	1.00	1.00	1.00	1.00	0.00
	(2)	Create note	0.22	0.56	0.35	0.31	0.14	1.00	1.00	1.00	1.00	0.00
JHotDraw	(1)	Draw rectangle	0.63	0.93	0.84	0.89	0.13	0.89	1.00	0.96	1.00	0.06
	(2)	Add text	0.21	0.31	0.26	0.27	0.05	0.27	0.36	0.32	0.34	0.04
	(2)	Draw rectangle	0.53	0.70	0.63	0.61	0.06	0.61	1.00	0.69	0.66	0.13
	(3)	Draw rectangle	0.42	0.76	0.64	0.72	0.14	0.73	1.00	0.94	1.00	0.11
	(3)	Cut rectangle	0.16	0.23	0.22	0.23	0.02	1.00	1.00	1.00	1.00	0.00
	(4)	Draw circle	0.54	0.96	0.85	0.88	0.14	0.81	1.00	0.91	0.95	0.09
	(4)	Spawn window	0.07	0.41	0.20	0.16	0.11	1.00	1.00	1.00	1.00	0.00

optimization technique is unable to split the trace into segment similar to the ones from the oracle. Consequently, the precision of *Add text* drops to a median value of 34% and that of *Draw rectangle*, usually very high in other scenarios, is only 66%.

D. Discussion

We analyze in detail some results to understand how the approach split the traces into segments. We focus on cases where the Jaccard score is low. In other cases, we know that the segments are meaningful because they are consistent with the oracle. Table VI shows excerpts of three segments. (Due to lack of space, we cannot report complete segments.)

The *Add class* feature of ArgoUML was matched with a very low Jaccard score. The manual tags in the trace delimited a sequence of 199 method invocations. The approach split this sequence into 5 segments comprising in a total of 172 method invocations, out of which 16 invocations occurred before the tag and, thus, do not belong to the oracle. The remaining $199 - 172 + 16 = 43$ invocations were grouped in small segments mainly related to GUI-event handling. In details, the five segments are related to (1) creation of the objects responsible for handling the class diagram through an instance of the Factory design pattern; (2) adding the class to the project; (3) adding the class to the current name-space; (4) setting properties of the class through a Façade design pattern; and, (5) handling the persistence of the diagram in the XMI file representing the UML diagram.

For the *Create note* feature of ArgoUML, the tagged segment is composed of 88 method invocations while the best matching segment identified by our approach is composed of 50 methods. The identified segment deals with the creation of a note, *i.e.*, creation of the object through a Factory, addition to the project, setting of its property. When compared to the *Add class* feature, only one segment was identified instead of five because the segment for creating a note is shorter than that of adding a class (50 invocations vs. 172) and because this smaller number of method invocations has a higher cohesion than that of the *Add class* feature. In addition, 32 of the remaining $88 - 50 = 38$ methods belong to the end of the trace and were not put in the same segment, while the sequence of these a methods continued after the tag with 24 other invocations. The continuation of the sequence *after* the tags means that the oracle is not precise enough. We explain this lack of precision by the extensive use of multi-threading in ArgoUML.

All methods related to setting properties through the Façade design pattern were not put in a same segment by our approach because these methods were invoked in a loop, in which each iteration of the loop contained a slightly different sequence of invocations. Consequently, (1) the RLE compression algorithm was not able to group together the various loop iterations and (2) the various iterations were not cohesive and thus the trace was split in several segments. We will explore in future work more complex compression techniques to deal with such cases.

The *Cut rectangle* feature of JHotDraw has been tagged as a sequence of 172 method invocations. However, in the best case shown in Table V, only 39 of these methods were grouped together by our approach, *i.e.*, the methods belonging to the last part of the tagged segment. We inspected this sequence and discovered that it is related to (1) add the rectangle content to the clipboard, (2) modify

Table VI
EXCERPT OF SEGMENTS IDENTIFIED BY THE APPROACH.

Create note (ArgoUML)	**Spawn window** (JHotDraw)	**Cut rectangle** (JHotDraw)
FacadeMDRImpl.isSingleton(...)	JavaDrawApp.createTools(...)	StorableOutput.close()
FacadeMDRImpl.isUtility(...)	MySelectionTool.MySelectionTool(...)	Clipboard.Clipboard()
CoreFactory.getCoreFactory()	TextFigure.TextFigure()	Clipboard.getClipboard()
CoreFactoryMDRImpl.buildComment(...)	TextFigure.setAttribute(...)	Clipboard.setContents(...)
CoreFactoryMDRImpl.createComment()	FigureAttributes.FigureAttributes()	CutCommand.deleteSelection()
CoreFactoryMDRImpl.initialize(...)	FigureAttributes.set(...)	BouncingDrawing.removeAll(...)
ModelEventPumpMDRImpl.flushModelEvents()	TextFigure.changed()	BouncingDrawing.figureRequestRemove(...)
UndoCoreHelperDecorator.addAnnotatedElement(...)	TextFigure.invalidate()	AnimationDecorator.removeFromContainer(...)
ModelEventPumpMDRImpl.flushModelEvents()	TextFigure.updateLocation()	AnimationDecorator.invalidate()
ClassDiagramGraphModel.addNode(...)	TextTool.TextTool(...)	AnimationDecorator.removeFigureChangeListener(...)
ClassDiagramGraphModel.canAddNode(...)	TextTool.TextTool(...)	AnimationDecorator.changed()
FacadeMDRImpl.isAInterface(...)	TextFigure.TextFigure()	AnimationDecorator.invalidate()
FacadeMDRImpl.isASubsystem(...)	TextFigure.setAttribute(...)	AnimationDecorator.release()
Project.getRoot()	FigureAttributes.FigureAttributes()	RectangleFigure.removeFromContainer(...)
ModelManagementFactory.getModelManagementFactory()	FigureAttributes.set(...)	RectangleFigure.removeFigureChangeListener(...)
ModelManagementFactoryMDRImpl.getRootModel()	TextFigure.changed()	RectangleFigure.changed()
CoreHelperMDRImpl.isValidNamespace(...)	TextFigure.invalidate()	RectangleFigure.release()
FacadeMDRImpl.getModel(...)	TextFigure.updateLocation()	RectangleFigure.removeFromContainer(...)
FacadeMDRImpl.isAModel(...)	ConnectedTextTool.ConnectedTextTool(...)	RectangleFigure.removeFigureChangeListener(...)
FacadeMDRImpl.isAFeature(...)	ConnectedTextTool.ConnectedTextTool(...)	RectangleFigure.changed()
...	...	...

the properties of the drawn rectangle so that it appears as "cut" in the painter, and (3) update the menu commands (*e.g.*, the command "Paste" is now enabled). The preceding sequence of $172 - 39 = 133$ methods was split in many small segments in which GUI events and actions performed by clicking the mouse button are interleaved, resulting in a sequence of loosely cohesive invocations.

The *Spawn window* feature of JHotDraw includes, in the manually-tagged segment, 197 method invocations; the segment with the highest overlap only included, however, 72 of these invocations. This sequence of 72 method invocations is actually related to re-sizing and re-adjusting figures in the panel while spawning the window. The remaining invocations (at the end of the trace) keep out by our approach are mainly related to restoring and setting-up the status of the menu commands.

Finally, as previously explained for the *Add text* feature of JHotDraw, the low Jaccard score and low precision are due to the high similarity between the sequences of methods of the *Add text* and *Draw rectangle* features, which leads our approach to put together both features in a segment of 168 method invocations.

On the one hand, the previous discussion highlights the capability of our approach to split execution traces into conceptually cohesive segments, despite the low Jaccard overlap with respect to manually-tagged segments. On the other hand, it shows some difficulties in identifying concepts in execution traces, due to:

- design patterns and, in general, object-orientation mechanisms (*e.g.*, polymorphism, dynamic binding), which make traces for different features almost identical (*e.g.*, *Add text* and *Draw rectangle* in JHotDraw);
- imprecision when generating and tagging traces due to multi-threading;
- the compression algorithm that is unable to group loop iterations consisting of slightly different sequences of method invocations.

In particular, despite the good results obtained by our approach when analyzing traces from JHotDraw (both with the Jaccard score and in precision), the extensive use of inheritance and design patterns in JHotDraw explain the lower results when compared to those obtained with ArgoUML. Inheritance and design patterns lead to the generation of many method invocations not directly related to a feature, but *supporting* and–or *enabling* the implementation of this feature. Consequently, these method invocations can appear in many different segments related to different features and thus can be a confounding factor for our approach.

Another difficulty of trace-based concept location approaches is to deal with method invocations related to GUI and system events. For example, hundreds of method invocations in both ArgoUML and JHotDraw execution traces correspond to GUI events, such as `mousePressed(...)`. These methods are not feature-specific and can appear almost anywhere in a trace and could lead to different segmentation across different runs. We deal with these methods by compressing the traces, removing sub-sequences of such methods, and using conceptual cohesion and coupling measures [9], [10], which lead to the creation of small segments containing only such method invocations.

E. Threats to validity

We now discuss the threats to validity that can have affected our empirical study.

Construct validity threats concern the relation between theory and observation. In this study, they are mainly due to measurement errors. The traces are automatically produced by executing the instrumented systems against some scenarios. Thus, the information contained in the traces is

reliable. However, multi-threading could change the ordering of method calls in different traces exercising the same sub-scenarios. The performances of the proposed approach are evaluated by using the Jaccard overlap, already used in the past to evaluate concept location approaches [17] and by using the standard IR precision measure, because we are also interested to split the trace into segments that only contain methods related to the feature of interest.

We only performed a preliminary assessment of the meaning of the identified concepts, by manually analyzing sequences of method invocations belonging to different segments. In future work, we plan to use automated techniques to label segments and thus better help the maintainer by assigning meanings to segments automatically.

Threats to *internal validity* concern confounding factors that could affect our results. The manually-tagged traces that we use as oracle pose such a threat. Indeed, it is possible that tags would appear in slightly different positions in the traces obtained by exercising the same scenarios in different runs. The slight different positions result from multi-threading, as well as from method invocations related to mouse and other GUI events. In particular, extra method calls related to GUI events or other uncontrollable system events could be interleaved in the traces.

Methods declared in class libraries could also introduce "noise" in our approach. For example, calls to methods from the Java class libraries frequently occur in the traces obtained in our experiments. They do not occur frequently enough to be discarded as "utility" method calls yet are not related to interesting concepts. Therefore, in future work, we will consider adding these methods in our list of stop-words.

A last threat to internal validity relates to the intrinsic randomness of GAs. However, in **RQ1**, we showed that, overall, results are quite stable across different GA runs.

Reliability validity threats concern the possibility of replicating this study. We attempted to provide all the necessary details to replicate our study.

Threats to *external validity* concern the possibility to generalize our results. We studied two systems having different size and belonging to different domains. However, we are aware that this is a first study aimed at validating the proposed approach and that we only split traces on a small sample of scenarios for the two software systems. Other traces could possibly lead to different results. Also, further validation on a larger set of different systems is desirable. Yet, within its limits, our results confirm the stability and precision of our approach for concept location.

A final remark concerns the complexity of our approach and computation times. Although this is a proof of concept, we are aware that excessive computation times or complexity may prevent further studies and practical application. On average, identifying concepts in a compressed trace of about 400 methods on a single high end PC (*i.e.*, with at least 4GB RAM) took about one day; when GA was mapped on multiple serves as described, the time went down to 20 minutes. Clearly, to make the approach appealing, we need to improve scalability both in time and space as well as in the possibility to handle longer traces.

VI. Conclusions and future work

This paper presented an approach to locate automatically concepts in execution traces by splitting traces into cohesive segments representing concepts related to a software system features. The approach relies on definitions of conceptual cohesion and coupling from the literature [9], [10] and on a search-based optimization technique, based on a genetic algorithm, to find (near) optimal splittings of traces into segments.

The approach has been applied and evaluated on two open source systems, ArgoUML and JHotDraw. Results showed that the approach is stable, and, overall, locates concepts with a high precision. Precision tend to drop for features realized using very similar sequences of methods, as sometimes happens in JHotDraw, where different kinds of shapes are drawn essentially in a same way. The overlaps between a manually-built oracle and the automatically-located segments vary depending on the cohesion of the features being analyzed, as the approach tends to split traces related to large features into smaller segments related to cohesive sub-concepts.

Future work will follow different directions. First, we are improving the proposed approach to increase its performance by better tuning the search-based optimization and the text indexing techniques. Also, we want to assign automatically meaningful labels to trace segments identified by the approach to help maintainers understand their meanings. Finally, we will carry other empirical studies to evaluate the approach on traces obtained from different systems.

VII. Acknowledgements

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada (Research Chair in Software Evolution and in Software Patterns and Patterns of Software) and by G. Antoniol's Individual Discovery Grant.

References

[1] G. Antoniol and Y.-G. Guéhéneuc, "Feature identification: An epidemiological metaphor," *Transactions on Software Engineering (TSE)*, vol. 32, no. 9, pp. 627–641, September 2006.

[2] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster, "The concept assignment problem in program understanding," in *Proceedings of the* 15^{th} *International Conference on Software Engineering*, IEEE Computer Society Press / ACM Press, May 1993, pp. 482–498.

[3] V. Kozaczynski, J. Q. Ning, and A. Engberts, "Program concept recognition and transformation," *IEEE Transactions on Software Engineering*, vol. 18, no. 12, pp. 1065–1075, Dec 1992.

[4] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *Transactions on Software Engineering (TSE)*, vol. 33, no. 6, pp. 420–432, June 2007.

[5] P. Tonella and M. Ceccato, "Aspect mining through the formal concept analysis of execution traces," in *Proceedings of IEEE Working Conference on Reverse Engineering*, 2004, pp. 112–121.

[6] N. Anquetil and T. Lethbridge, "Extracting concepts from file names: A new file clustering criterion," in *Proceedings of the* 20^{th} *International Conference on Software Engineering*, IEEE Computer Society Press, May 1998, pp. 84–93.

[7] N. Wilde and M. C. Scully, "Software reconnaissance: Mapping program features to code," in *Journal of Software Maintenance: Research and Practice*, John Wiley & Sons, January-February 1995, pp. 49–62.

[8] M. Eaddy, A. V. Aho, G. Antoniol, and Y.-G. Guéhéneuc, "CERBERUS: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis," in *Proceedings of the* 16^{th} *International Conference on Program Comprehension (ICPC)*, IEEE Computer Society Press, June 2008.

[9] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 287–300, 2008.

[10] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *Proceedings of 22nd IEEE International Conference on Software Maintenance*. Philadelphia, Pennsylvania, USA: IEEE CS Press, 2006, pp. 469 – 478.

[11] W. E. Wong, S. S. Gokhale, and J. R. Horgan, "Quantifying the closeness between program components and features," *Journal of Systems and Software*, vol. 54, no. 2, pp. 87 – 98, 2000, special Issue on Software Maintenance.

[12] D. Edwards, S. Simmons, and N. Wilde, "An approach to feature location in distributed systems," Software Engineering Research Center, Tech. Rep., 2004.

[13] A. D. Eisenberg and K. D. Volder, "Dynamic feature traces: Finding features in unfamiliar code," in *proceedings of the* 21^{st} *International Conference on Software Maintenance*. IEEE Press, September 2005, pp. 337–346.

[14] K. Chen and V. Rajlich, "Case study of feature location using dependence graph," in *Proceedings of the* 8^{th} *International Workshop on Program Comprehension*, IEEE Computer Society Press, June 2000, pp. 241–252.

[15] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 210–224, March 2003.

[16] M. Salah and S. Mancoridis, "A hierarchy of dynamic software views: From object-interactions to feature-interactions," in *Proceedings of the* 20^{th} *International Conference on Software Maintenance*, IEEE Computer Society Press, September 2004, pp. 72–81.

[17] M. Salah, S. Mancordis, G. Antoniol, and M. Di Penta, "Towards employing use-cases and dynamic analysis to comprehend Mozilla," in *proceedings of the* 21^{st} *International Conference on Software Maintenance*. IEEE Press, September 2005, pp. 639–642.

[18] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. New York NY USA: ACM, 2007, pp. 234–243.

[19] J. Ka-Yee Ng, Y.-G. Guéhéneuc, and G. Antoniol, "Identification of behavioral and creational design motifs through dynamic analysis," *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, December 2009.

[20] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.

[21] M. Marin, A. van Deursen, and L. Moonen, "Identifying crosscutting concerns using fan-in analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 1, 2007.

[22] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[23] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, 1990.

[24] D. E. Goldberg, *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley Pub Co, Jan 1989.

[25] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the Bunch Tool." *IEEE Trans. Software Eng.*, vol. 32, no. 3, pp. 193–208, 2006.

APPENDIX B

Published Article in SSBSE 2010

The article explains the Distributed Architectures for GA parallelization. It is published at the 2nd International Symposium on Search Based Software Engineering (SSBSE 2010). The article entitled *"Concept Location with Genetic Algorithms : A Comparison of Four Distributed Architectures"* by Fatemeh Asadi, Giuliano Antoniol, Yann-Gaël Guéhéneuc (Asadi *et al*. (2010a)).

Concept Location with Genetic Algorithms: A Comparison of Four Distributed Architectures

Fatemeh Asadi*, Giuliano Antoniol*, Yann-Gaël Guéhéneuc†
*SOCCER Lab. – DGIGL, École Polytechnique de Montréal, Québec, Canada
†Ptidej Team – DGIGL, École Polytechnique de Montréal, Québec, Canada
{fatemeh.asadi,yann-gael.gueheneuc}@polymtl.ca
antoniol@ieee.org

***Abstract*—Genetic algorithms are attractive to solve many search-based software engineering problems because they allow the easy parallelization of computations, which improves scalability and reduces computation time. In this paper, we present our experience in applying different distributed architectures to parallelize a genetic algorithm used to solve the concept identification problem. We developed an approach to identify concepts in execution traces by finding cohesive and decoupled fragments of the traces. The approach relies on a genetic algorithm, on a textual analysis of source code using latent semantic indexing, and on trace compression techniques. The fitness function in our approach has a polynomial evaluation cost and is highly computationally intensive. A run of our approach on a trace of thousand methods may require several hours of computation on a standard PC. Consequently, we reduced computation time by parallelizing the genetic algorithm at the core of our approach over a standard TCP/IP network. We developed four distributed architectures and compared their performances: we observed a decrease of computation time up to 140 times. Although presented in the context of concept location, our findings could be applied to many other search-based software engineering problems.**

***Keywords*-Concept location; dynamic analysis; information retrieval; distributed architectures.**

I. Introduction

Genetic algorithms (GAs) are an effective technique to solve complex optimization problems. GAs are effective in finding approximate solutions when the search space is large or complex, when mathematical analysis or traditional methods are not available, and—in general—when the problem to be solved is NP-complete or NP-hard [1]. Informally, a GA may be defined as an iterative procedure that searches for the best solution to a given problem among a constant-size population, represented by a finite string of symbols, the *genome*. The search starts from an initial population of individuals, often randomly generated. At each evolutionary step, individuals are evaluated using a *fitness function*. Highly fit individuals have the highest probability to reproduce in the next generation. GAs have been applied to many software engineering problems; from library miniaturization [2], to project staffing [3], to test input data generation [4], to software refactorings [5].

One of the attractive feature of GAs is that the evaluation of the fitness function is often performed on each individual in isolation: to assign its fitness value to an individual, the GA only needs its genome representation because there are no interactions with other individuals in the population. Such an isolation in the evaluation of the fitness function leads naturally to parallelize computations of the fitness function to reduce computation time [6], [7], [8].

In this paper, we report our experience in distributing the computation of a fitness function to parallelize a GA to solve the concept location problem. To the best of our knowledge, this is the first time that GA parallelization via the distribution of the fitness function computation is applied to solve the concept location problem.

Concept location approaches help developers perform their maintenance and evolution tasks by identifying abstractions (*i.e.*, concepts or features) and the location of the implementation of these abstractions in source code. They aim at identifying *code fragments*, *i.e.*, set of method calls in traces and the related method declarations in the source code, responsible for the implementation of domain concepts and–or user-observable features [9], [10], [11], [12], [13].

In [14], we presented an approach to identify cohesive and decoupled fragments in execution traces, which likely participate in implementing concepts related to some features. The approach builds upon previous concept location approaches [15], [16], [13], [12], [17] and uses a GA to automatically locate cohesive and decoupled fragments. Although promising, our approach is computationally intensive and suffers from scalability issues.

To resolve the scalability issues of our approach, we developed, tested, and compared four different architectures where a client (master) distributes the computation of the fitness function among servers (slaves) over a TCP/IP network. To our surprise, the most effective architecture to reduce computation time defines servers that only use local data and do not share data and–or results with other servers.

Consequently, the contribution of this paper is an application of GA parallelization to a software engineering problem and the comparison and discussion of our findings for four different architectures. Although presented in the context

of concept location, our findings could be applied to other search-based software engineering problems.

The remainder of the paper is organized as follows: Section II presents related work followed by Section III where the concept location problem is summarized. Section IV describes the approach to speed up computation. Section V reports the results and some discussions. Section VI concludes the paper and outlines some future work.

II. Related Work

This paper focuses on the parallelization of a GA using a distributed architecture to reduce the computation time of an approach to solve the concept location problem. Therefore, we focus in the following on previous work related to the concept location problem (*i.e.*, feature identification), to the distribution of optimizations in software engineering, and to the parallelization of GAs in other domains.

A. Feature Identification

In their pioneering work, Wilde and Scully [16] presented the first approach to identify features by analyzing execution traces. They used two sets of test cases to build two execution traces, one where a feature is exercised and another where the feature is not. They compared the execution traces to identify the feature in the system. Similarly, Wong *et al.* [18] analyzed execution slices of test cases to identify features in source code. Wilde's original idea was later extended in several works [9], [12], [19], [20] to improve its accuracy by introducing new criteria on selecting execution scenarios and by analyzing the execution traces differently. Search based techniques have been used by Gold *et al.* [21] for concept binding, the work extends a previous contribution [22] and uses hill climbing and GA to locate (possibly overlapping) concepts in the source code.

More recent works focused on a combination of static and dynamic data [17], [12], in which, essentially, the problem of features identification from multiple execution traces is modelled as an information-retrieval (IR) problem, which has the advantage to simplify the identification process and, often, improves its accuracy [12]. Yet, Liu *et al.* [23] showed that a single trace suffices to build an IR system and identify useful features. Execution traces were also used to mine aspects by Tonella and Ceccato [13].

We share with this previous work the use of dynamic data and IR techniques to identify features. In our approach, we determine, in an execution trace the cohesive and decoupled fragments likely to be relevant to a feature using the values of the conceptual cohesion and coupling [24], [25] metrics of the methods participating in each fragment. The computational costs of conceptual cohesion and coupling together with the size of the execution traces are at the root of the scalability issues of our approach.

B. GA Parallelization in Software Engineering

A limited number of works in software engineering addressed complex optimization problems by distributing computations among several servers. Mitchal *et al.* [26] proposed an approach to remodularize large systems by grouping together related components by means of clustering techniques. They used different search strategies based on hill-climbing and GAs. To improve the performance of their approach, they distributed the hill-climbing computations.

More recently, Mahdavi *et al.* [27] used a distributed hill-climbing for software module clustering. The fitness function clusters together modules that are cohesive and decoupled from the other clusters. The algorithm was parallelized on 23 processing units running Linux.

C. GA Parallelization in Other Domains

The literature on the parallel implementation of GAs reports that parallelization does not influence the quality of results but makes GA execution much faster.

Parallel GAs have been to solve problems in different domains. For example, parallel GAs were used for shortest-path routing [28], multi-objective optimization [29], finding roots of complex functional equation [30], image restoration [31], service restoration in electric power distribution [32], and rule discovery in large databases [33].

The scalability of a parallel system refers to its ability to use an increasing number of processors (and–or computers) in an effective way. Rivera [7] discussed the scalability of parallel GAs based on their *iso-efficiency*, which is defined according to the problem size, number of processors, and the execution time of the parallel algorithm. A parallel system is scalable iff it uses an iso-efficient fitness function.

Stender *et al.* [6] classified parallel GAs into three categories, each one using a different parallelization strategy. In the category of global parallelization, only the evaluation of the individuals' fitness is parallelized: a computer acting as master applies the genetic operators on the individuals' genomes and distributes the individuals among slave computers, which compute the fitness values of the individuals.

In the category of coarse-grained parallelization (island model), a computer divides a population into sub-populations and assigns each sub-population to another computer. A GA is executed on each sub-population. When it is needed, the computers exchange data related to the sub-populations using a migration process. This model inspired Zorman *et al.* [34]: they used a Java service-oriented architecture to implement the island model using a migration process to solve the knapsack problem.

In the category of fine-grained parallelization, each individual is assigned to a computer and all the GA operations are performed in parallel. Our approach to GA parallelization of the concept location problem falls under this category: it is essentially a global parallelization where servers are in charge of computing fitness values. Moreover,

our work is the first to presents four distributed architectures and their related trade-offs for the computation of the fitness function to parallelize the concept location problem.

III. Background

This section summarizes our approach [14] to locate concepts by analyzing execution traces. We provide details of our approach for the sake of completeness and because they are necessary to understand the rationale behind the four different architectures that we implemented.

Our concept location approach consists of five steps. First, the system under analysis is instrumented. Second, it is exercised to collect execution traces. Third, the collected traces are compressed to reduce the search space that must be explored to identify concepts. Fourth, each method of the system is represented by means of the text that it contains. Fifth, a GA-based technique is used to identify, within execution traces, sequences of method invocations that are related to a concept.

A. Steps 1 and 2 – System Instrumentation and Trace Collection

First, the software system is instrumented using the *instrumentor* of MoDeC. MoDeC is a tool to extract and model sequence diagrams from Java systems [35]. MoDeC instrumentor is a dedicated Java bytecode modification tool implemented on top of the Apache BCEL bytecode transformation library[1]. It inserts appropriate and dedicated method invocations in the system to trace method/constructor entries/exits, taking care of exceptions and system exits. It also allows the user to add tags containing meta-information to the traces, *e.g.*, tags delimiting and labelling sequences of method calls related to some specific features being exercised. Resulting traces are text files listing method invocations and including the class of the object caller, the unique ID of the caller, the class of the receiver, the unique ID of the callee, and the complete signature of the method.

B. Step 3 – Pruning and Compressing Traces

Usually, execution traces contain methods invoked in most scenarios, *e.g.*, methods related to logging or start-up and shut-down. In the execution trace of a system with a graphical user interface, mouse tracking methods will largely exceed all other method invocations. Yet, it is likely that such methods are not related to any particular concept, *i.e.*, they are utility methods. We filter out these utility methods using the distributions of the frequencies of their occurrences.

Moreover, traces often contain repetitions of one or more method invocations, for example `m1(); m1(); m1();` or `m1(); m2(); m1(); m2();`. A repetition does not introduce a new concept and makes a trace longer that necessary to locate concepts. Consequently, we compress traces using the Run Length Encoding (RLE) algorithm to remove repetitions and keep one occurrence of any repetition only. The previous examples would become `m1()` and `m1(); m2()`, respectively. We compression any sub-sequences of method invocations having an arbitrary length.

[1] http://jakarta.apache.org/bcel/

Table I
Example of GA individual representation (second column).

Method Invocations	Repr.	Segments
TextTool.deactivate()	0	
TextTool.endEdit()	0	
FloatingTextField.getText()	0	
TextFigure.setText-String()	0	1
TextFigure.willChange()	0	
TextFigure.invalidate()	0	
TextFigure.markDirty()	1	
TextFigure.changed()	0	
TextFigure.invalidate()	0	
TextFigure.updateLocation()	0	2
FloatingTextField.endOverlay()	0	
CreationTool.activate()	1	
JavaDrawApp.setSelectedToolButton()	0	
ToolButton.reset()	0	
ToolButton.select()	0	
ToolButton.mouseClickedMouseEvent()	0	
ToolButton.updateGraphics()	0	3
ToolButton.paintSelectedGraphics()	0	
TextFigure.drawGraphics()	0	
TextFigure.getAttributeString()	1	

C. Step 4 – Textual Analysis of Method Source Code

To determine the conceptual cohesion and coupling of invoked methods, our approach uses the metrics defined by Marcus *et al.* [24], [25]. We extract a set of terms from each method by tokenizing the method source code and comments, pruning out special characters, programming language keywords, and terms belonging to a stop-word list for the English language. (We assume that comments appearing on top of the method declaration belong to the following method.)

We then split compound terms based on the Camel Case naming convention at each capitalized letter, *e.g.*, `getBook` is split into `get` and `book`. Then, we stem the obtained simple terms using a Porter stemmer [36].

Once terms belonging to each method extracted, we index these terms using the *tf-idf* indexing mechanisms [37]. We thus obtain a term–document matrix, where documents are all methods of all classes belonging to the system under study and where terms are all the terms extracted (and split) from the method source code. Finally, we apply Latent Semantic Indexing (LSI) [38] to reduce the term–document matrix into a concept–document matrix.

We follow previous process and suggestion [24], [25] when computing the conceptual cohesion and coupling of methods in a class in the LSI subspace to deal with synonymy, polysemy, and term dependency. We choose a size of 50 for the LSI subspace.

$$SegmentCohesion_k = \frac{\sum_{i=begin(k)}^{end(k)-1}\sum_{j=i+1}^{end(k)} similarity(method_i, method_j)}{(end(k)-begin(k)+1)\cdot(end(k)-begin(k))/2} \quad (1)$$

$$SegmentCoupling_k = \frac{\sum_{i=begin(k)}^{end(k)}\sum_{j=1,j<begin(k)\ or\ j>end(k)}^{l} similarity(method_i, method_j)}{(l-(end(k)-begin(k)+1))\cdot(end(k)-begin(k)+1)} \quad (2)$$

$$fitness(individual) = \frac{1}{n}\cdot\sum_{k=1}^{n}\frac{SegmentCohesion_k}{SegmentCoupling_k} \quad (3)$$

D. Step 5 – Search-based Concept Location

We now have all the data to segment execution traces into conceptually-cohesive and -decoupled segments related to a feature being exercised and, thus, to a specific concept.

1) Problem Definition: Suppose that the collected trace contains N methods; determining a (near) optimal solution (splitting a trace into segments) means exploring a search space of all possible binary strings, of length N, that do not contain two consecutive 1. In other words, the order of the problem search space is 2^N and, therefore, we use a GA to perform the splitting.

At each step of the GA, individuals are evaluated using a *fitness function* and selected using a *selection mechanism*. Highly fit individuals have the highest reproduction probability. The evolution (*i.e.*, the generation of a new population) is affected by the *crossover operator* and the *mutation operator*.

2) Problem Representation: Our representation of an individual is a bit-string of the length of the compressed execution trace in which we want to identify some feature-related concepts. Each method invocation is represented as a "0", except the last method invocation in a segment, which is represented as a "1". For example, the bit-string

$$\underbrace{00010010001}_{11}$$

means that the trace, containing 11 method invocations, is split into three segments (*i.e.*, concepts) composed by the first four method invocations, the next three, and the last four. Table I shows an example of a real segment splitting[2].

Other representations could be more compact, for example, a book keeping of segments beginnings and ends. The disadvantage of such representation is that mutation and crossover would be more complex and costly in time. Among different representations, we found that the bit-string representation is suitable to large traces: even for a trace of one million method calls and hundreds of individuals, memory requirement is still manageable on a standard PC. Moreover, the bit-string representation allows to easily understand the size of the search space, which is roughly related to the number of bit strings.

[2]The segment splitting shown in Table I has been obtained randomly and does not correspond to actual concepts.

3) Mutation: The mutation operator prevents the convergence to a local optimum: it randomly modifies an individual's genome (*e.g.*, by flipping some of its symbols). The mutation operator randomly chooses one bit in the representation and flips it over. Flipping a "0" into a "1" means splitting an existing segment into two segments, while flipping a "1" into a "0" means merging two consecutive segments. Mutation operator is thus implemented with constant time complexity.

4) Crosssover: The crossover operator takes two individuals (the *parents*) of one generation and exchanges parts of their genomes, producing one or more new individuals (the *offspring*) in the new generation. The crossover operator is the standard 2-point crossover. Given two individuals, two random positions x, y with $x < y$ are chosen in one individual's bit-string and the bits from x to y are swapped between the two individuals to create two new offsprings. Crossover operator is thus implemented with linear time complexity in the length of the bit-string individual representation.

5) Fitness Function: A fitness function drives the GA to produce individuals that represent a splitting of the trace into segments that are related to some concepts. We use the software design principles of cohesion and coupling, already adopted in the past to identify modules in systems [39].

However, instead of structural cohesion and coupling measures, we use conceptual (*i.e.*, textual) cohesion and coupling measures [24], [25]. Segment cohesion is the average (textual) similarity between any pair of methods in a segment k and is computed using the formulas in Equation 1 where $begin(k)$ is the position (in the individual's bit-string) of the first method invocation of the k^{th} segment and $end(k)$ the position of the last method invocation in that segment. The similarity between two methods is computed using the cosine similarity measure over the LSI matrix extracted in the previous step. Thus, it is the average of the similarity [24], [25] to all pairs of methods in a given segment.

Segment coupling is the average similarity between a segment and all other segments in the trace, computed using Equation 2, where l is the trace length. Segment coupling represents, for a given segment, the average similarity between methods in that segment and those in different ones.

Cohesion and coupling have quadratic costs in the trace

length, plus each similarity computation between a pair of methods involves a scalar product in the LSI subspace, with a cost proportional to d, the number of retained LSI dimensions. Thus, when compared with the bit operations required to perform mutation (constant time) and crossover (linear time), it is evident that the main source of complexity and computation costs come from Equations 1 and 2 that have polynomial time complexity in the bit-string individual representation, *i.e.*, number of methods in the trace. For a trace split into n segments, the fitness function is shown in Equation 3.

6) GA Parameters: We use a simple GA with no elitism, *i.e.*, it does not guarantee to retain best individuals across subsequent generations; the selection operator is the roulette-wheel selection. We set the population size to 200 individuals and a number of generations of 2,000. Crossover and mutation are respectively performed on each individual of the population with probability *pcross* and *pmut* respectively, where $pmut \ll pcross$. The crossover probability was set to 70% and the mutation to 5%, which are values widely used in many GA applications.

IV. GA and Distributed Architecture

We started our experiments with a basic GA implementation running on a single computer. We found that computations were overly time consuming, impairing the possibility to actually obtain results in a reasonable amount of time. As an example, running an experiment with a compressed trace from JHotDraw v5.4b2, and the scenario *Start-DrawRectangle-Quit*, that contains 240 method calls, with a number of iterations equal to 2,000, took about 12 hours.

We could expect a substantial improvement by parallelizing computations on several computers. However, according to Amdahl's law [40], the performance increase is not linear with the number of computers due to the sequential code, *e.g.*, mutation and crossover. In addition, network latency, available bandwidth between computers and, in general, available resources complicate performance prediction and could lessen time reduction. A detailed study of performance in function of network latency, number of computers, and speed-up is out of scope of this paper and will be treated in a future work. Yet, we report the user-experienced speed-up obtained with different architectures.

Table II
Example of individual coding and segment redundancy

Id1	0001 0001 0000001 0001 0001
Id2	0001 0001 001 0001 0001 0001
Id3	001 00001 0000000001 0001
Id4	00001 0001 0000001 000001
Id5	0001 0001 0000001 0001 001 01

To reduce computation time, we decided to resort on the client–server architectural style [41], customized into more

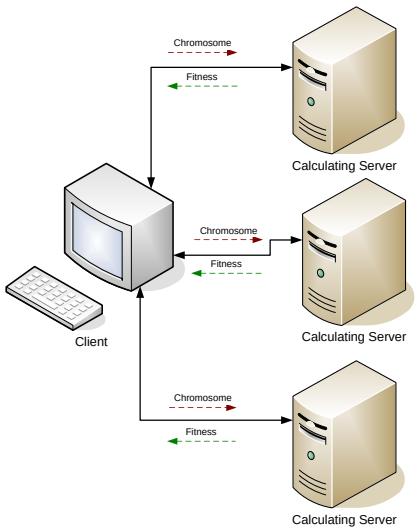

Figure 1. Baseline Client Server Configuration.

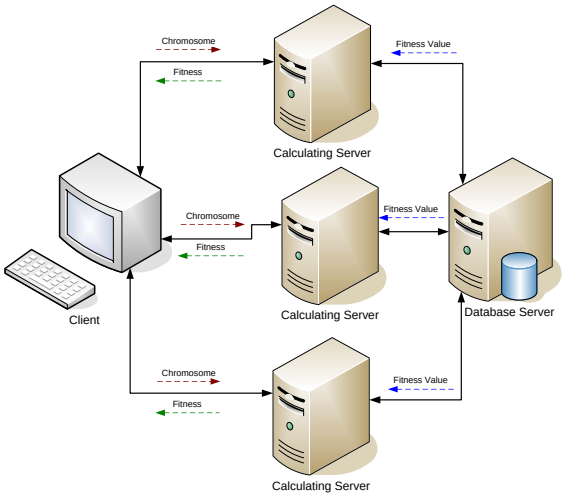

Figure 2. DBMS Client Server Configuration.

specific architectures detailed in the following. The rationale behind the different architectures comes from the illustrative population shown in Table II: several individuals share some segments. For example, the first two segments of individuals Id1, Id2, and Id5 are identical (*i.e.*, beginning and end are the same); Id1 and Id5 are almost identical but for their last segments. Thus, once Id1's fitness value is calculated, if segment cohesion and coupling were stored, they could be reused to compute the fitness values of Id2 and Id5.

In the following, we minimally define that a client computer (master in Stender's work [6]), performing mutation, crossover, and population evolution, distributes fitness computation to multiple servers, which compute the received individual's fitness value and return it to the client.

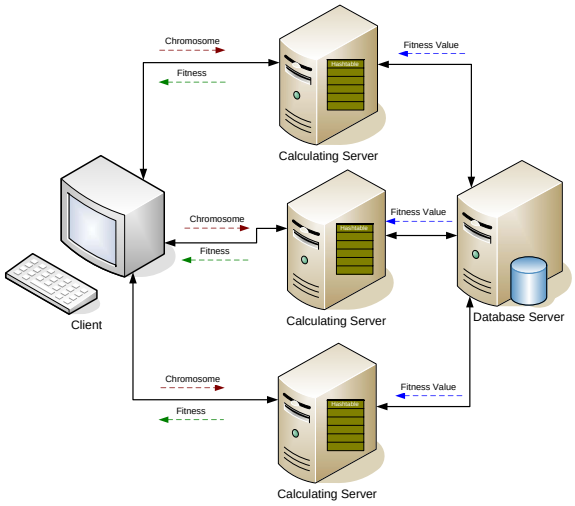

Figure 3. Database-Hash table Configuration for GA-Concept Identifier

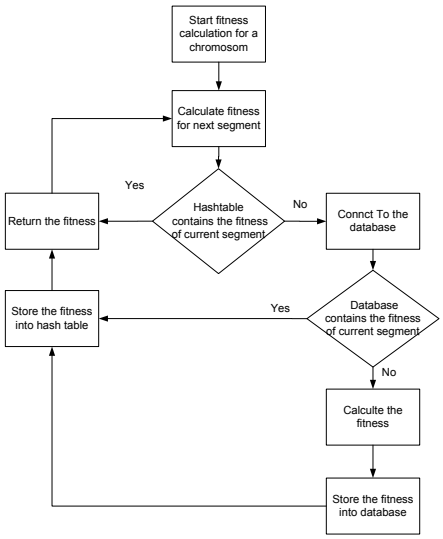

Figure 4. Flow chart of Database-Hash table configuration process

A. A Simple Client Server Architecture

The simplest distributed client–server architecture is shown in Figure 1. The servers have no local memory, do not communicate among themselves or store data locally or on a global and shared device. The client sends the individuals' encodings to the servers and waits for the fitness values to be returned. Each server has only its own local LSI matrix and computes fitness values based on the equations presented in the previous section.

B. A Database Client Server Architecture

Figure 2 shows the architecture of a client–server in which a database server stores global shared storage device. When a segment cohesion or coupling value is required, a server

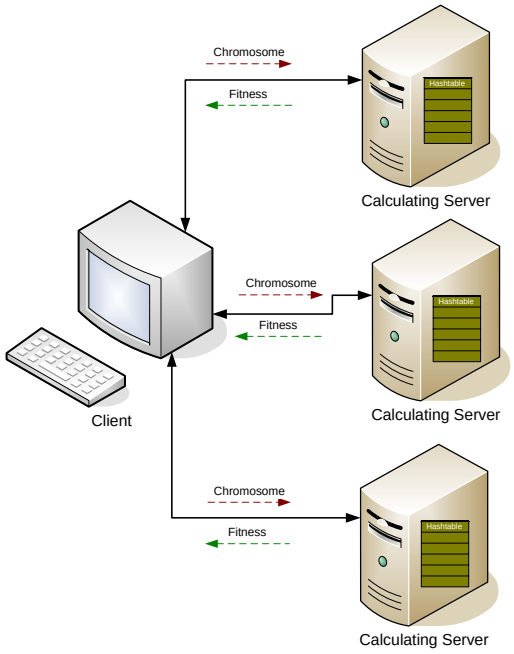

Figure 5. Hash Table Client Server Configuration.

first queries the database before computing it if missing.

The database holds two tables: a cohesion table and a coupling table, each with three columns. Each record in these tables keeps a similarity/coupling value for one segment. The first column, called beginning, keeps the index of the first method invocation in a segment and the second column keeps the index of the last method invocation in the same segment. The third column contains the cohesion/coupling value of that segment.

Whenever the fitness value for a new individual must be computed, the responsible server checks first the database. If it can find the needed values (already calculated in the last iterations or by other servers for other individuals), it uses these to compute the fitness value using a simple division. Else, it computes cohesion and coupling for the new segment and stores the values in the database. Thus, computation is performed if and only if the values can not be retrieved from the database: as much data as possible is shared between servers to reduce computation times.

There is an extra cost due to database queries and network communication. A central database implies that all servers write in and read from the same database. Yet, we would expected that using a database reduces the computation times by caching already-calculated values. However, sending data over the network, acquiring and releasing locks, and performing queries are also time consuming operations.

C. A Hash-database Client Server Architecture

To limit the possible communication between servers and the database, the architecture shown in Figure 3 was devised.

The goal of this architecture is to further reduce computation time by decreasing the number of accesses to the central database using a local cache on each server, implemented with a hash table.

The architecture works as follows: whenever a server wants to compute the fitness value of a segment, it searches its hash table. If the required data does not exist in its local hash table, then the server queries the central database. If the server finds the required data in the database, it uses it to compute the fitness value and and stores it in its hash table, else it computes the required data and stores the results in both its hash table for its future use and in the central database for the other servers use. Figure 4 reports the flowchart of the process of this architecture.

D. A Hash Client Server Architecture

This last architecture is a compromise between the two previous ones: only local data is stored in the local hash table of servers. No data is shared among servers. As shown in Figure 5, servers only communicate with the client and no global data is kept and available.

Each server has two hash tables: one for similarity cohesion and the other for coupling values for each segment. The key of the hash tables is a combination of the indexes of the first and last method invocations of a segment. Each server uses its own hash tables and thus cannot benefit from the computation results of others. However, because all the data is stored locally and there is no access policy using locking algorithms, the access to the already-calculated data as well as their storage is efficient.

V. Results and Discussion

We now report the typical timing obtained with the different architectures on two compressed traces from JHotDraw.

JHotDraw[3] is a Java framework for drawing 2D graphics. JHotDraw started in October 2000 with the main purpose of illustrating the use of design patterns in a real context. Version v5.4b2 used in our previous work [14] has a size of about 413 KLOCs.

The traces were collected by instrumenting JHotDraw and executing the scenarios *Start-DrawRectangle-Quit* and *Start-Spawn-Window-Draw-Circle-Stop*. These scenarios generated respectively traces of 6,668 and 16,366 method calls; once utility methods were removed their sizes are reduced to 447 and 670 calls. Finally RLE compression brought down the numbers of distinct calls to 240 and 432.

In our experiments, we distributed computations over a sub-network of 14 workstations. Five high-end workstations, the most powerful ones, are connected in a Gigabit Ethernet LAN; low-end workstations are connected to a LAN segment at 100 MBit/s and talk among themselves at 100 Mbit/s. Each experience was run on a subset of ten computers: nine servers and one client.

[3]http://www.jhotdraw.org

Workstations run CentOS v5 64 bits; memory varies between four to 16 Gbytes. Workstations are based on Athlon X2 Dual Core Processor 4400; the five high-end workstations are either single or dual Opteron. Workstations run basic Unix services (*e.g.*, network file system, SAMBA, mySQL) and user processes. User processes are typically editing and compilation of programs, e-mail clients, Web browsers, and so on. No special care was taken to ensure a specific network condition (*e.g.*, priorities were not altered) and thus times and ratios between times can be considered typical of a industrial or research environment. However, the sizes of the GA processes never exceeded the physical memory of the workstations to avoid paging; workstations were managed to ensure that each computationally-intensive user processes had a dedicated CPU.

The client computer was also responsible to measure execution times and to verify the liveness of connections; connections to servers as well as connections to the database were implemented on top of TCP/IP (AF_INET) sockets. All components have been implemented in Java 1.5 64bits. The database server, shown in Figures 2 and 3, was MySQL server v5.0.77.

Table III
Computation times for desktop solution and the different architectures of Figures 1, 2, and 5 with the *Start-DrawRectangle-Quit* scenario – Compressed trace length of 240 methods

Time Measurement			
Architectures	Runs #	Measures	Average
Desktop	1	12:09 h	12:07 h
	2	11:39 h	
	3	12:21 h	
	4	11:50 h	
	5	12:38 h	
Client–server	1	1:44 h	2:01 h
	2	2:36 h	
	3	1:53 h	
	4	1:40 h	
	5	2:13 h	
Database	1	16:36 h	13:50 h
	2	15:3 h	
	3	9:52 h	
Hash Table	1	5:13 m	5:17 m
	2	5:19 m	
	3	5:20 m	
	4	5:27 m	
	5	5:10 m	

Table III reports computation times for the different architectures. The times reported for the single-computer architecture come from an optimized implementation of our approach. In our first implementation, we reused the Java GALib library, which is freely available from SourceForge and implements a simple GA. GALib makes no assumptions on crossover and mutation operators and assumes that the fitness of an individual must be recomputed even if it was passed unchanged from the old generation to the new one. This recomputation resulted in about 30% of computation-time increase because between 20% and 30% of individuals

Table IV
COMPUTATION TIMES FOR DESKTOP SOLUTION AND THE ARCHITECTURE OF FIGURE 5 WITH THE *Start-Spawn-Window-Draw-Circle-Stop* SCENARIO – COMPRESSED TRACE LENGTH OF 432 METHODS

Time Measurement			
Architectures	Runs #	Measures	Average
Desktop	1	45:38 h	44:07
	2	41:28 h	
	3	45:07h	
Hash Table	1	7:21 m	7:24 m
	2	7:21 m	
	3	7:32 m	

are not subject to mutation or crossover between generations. Thus, to reduce computation time, we modified GALib to compute only the fitness values of individuals that have changed between the last generation and the current one.

Distributing the computation, shown in Figure 1, clearly results in an important reduction of computation time; as shown in the second row of Table III. Computation time went from 12 hours to about two hours; however, the gain in terms of time reduction is considerably lower than expected as we had nine computers available (excluding the client) and, thus, expected computation times close to one hour.

We felt that there was still room for improvement and Amdahl's law [40] was only partially the reason for the reduced gain. We observed that the nature of our problem was such that crossover and mutation preserve a large fraction of segments unchanged and that for those segments, previous cohesion and coupling values could be reused.

Thus, we tested the two architectures in Figures 2 and 3. Table III in its third row reports results for such database client–server architecture: to our surprise, sharing data among servers via a central database increased computation times.

Finally, Table III, in its last row, reports the computation times for the architecture in Figure 5, which is the fastest architectures. The gain in computation times obtained is of about 140 times. The implementation of this GA parallelization is moreover relatively simple.

We obtained similar gains with other traces. For example, the trace generated by the scenario *Start-Spawn-Window-Draw-Circle-Stop*, with the desktop architecture, was split in about 44 hours while, with the fastest architecture, the client–server with the hash table, computation time is of about 7 minutes. Table IV reports the results of splitting the trace with two architectures.

A. Discussion

We conjecture that poor performance of the database architecture, in Figure 2, is mainly due to the database accesses (reading, writing, and locking) for the computation of each coupling and cohesion values. These frequent accesses are responsible for the increase in computation times. To limit the number of database accesses, we introduced the hybrid architecture in Figure 3. Results have not been reported in Table III because they are not substantially different (better) then those of the database. We are investigating the reason of this unexpected behavior to locate the bottleneck cause.

Indeed, in our current implementation, accesses to the local hash table and the database are managed serially. Performance could improve by parallelizing writing in the database and access to the hash table and by loading the hash table only once at the beginning of each computation. Unfortunately, given the size of the search space and the huge number of possible segments, the probability that in two consecutive runs a relevant number of the segments will be exactly the same is very low. This fact makes the architecture in Figure 3 interesting from a theoretical point of view but not practical.

Despite the decrease in computation time, the very definition of the concept location problem makes it hard to obtain acceptable computation times for traces longer than few thousands of methods even with the fastest architecture, unless a higher number of servers is available. The definition of this problem is tied to the size of the search space, Equations 1 and 2, and the bit-string representation. Indeed, the longer the trace, the higher the number of methods contributing to the segment coupling. However, we believe that if concepts are indeed implemented in cohesive and decoupled segments, then computing coupling with Equation 2 is overly conservative and redefining the problem could substantially reduce computation time. We are currently working to restate the concept location problem in using audio digital signal processing and time windowing.

We have reported data of two traces of one software system, namely JHotDraw, therefore we cannot generalize to other traces though the performance issue is likely to be non-specific to JHotDraw or the used traces. Indeed, we experienced similar results with traces of different lengths of ArgoUML. Much in the same way, we cannot generalize to other search-based software engineering problems. However, we observed that the trade-off between the complexity of the fitness function and the local and global knowledge representations; similar trade-offs are known to be general and common to many application of optimization techniques to software engineering.

VI. CONCLUSION AND FUTURE WORK

GAs have been successfully applied to many complex software engineering problems. To the best of our knowledge, no previous work distributed fitness computation on several servers to exploit the intrinsic parallel nature of GAs to reduce computation times for concept location.

This paper presented and discussed four client–server architectures conceived to improve performance and reduce GA computation times to resolve the concept location problem. To our surprise, we discovered that on a standard

TCP/IP network, the overhead of database accesses, communication, and latency may impair a dedicated solutions. Indeed, in our experiments, the fastest solution was an architecture where each server kept track only of its computations without exchanging data with other servers. This simple architecture reduced GA computation by about 140 times when compared to a simple implementation, in which all GA operations are performed on a single machine.

Future work will follow different directions. First, we are working on reformulating the concept location problem. Also, we want to experiment different communication protocols (*e.g.*, UDP) and synchronization strategies. We will carry out other empirical studies to evaluate the approach on more traces, obtained from different systems, to verify the generality of our findings. Finally, we will reformulate other search-based software engineering problems to exploit parallel computation to verify further our findings.

VII. Acknowledgements

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada (Research Chair in Software Evolution and in Software Patterns and Patterns of Software) and by G. Antoniol's Individual Discovery Grant.

References

[1] M. Garey and D. Johnson, *Computers and Intractability: a Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[2] G. Antoniol and M. D. Penta, "Library miniaturization using static and dynamic information," in *Proceedings of IEEE International Conference on Software Maintenance*. Amsterdam The Netherlands: IEEE Press, Sep 22-26 2003, pp. 235–244.

[3] G. Antoniol, A. Cimitile, A. D. Lucca, and M. D. Penta, "Assessing staffing needs for a software maintenance project through queuing simulation," *IEEE Transactions on Software Engineering*, vol. 30, no. 1, pp. 43–58, Jan 2004.

[4] P. McMinn and M. Holcombe, "Evolutionary testing using an extended chaining approach," *Evol. Comput.*, vol. 14, no. 1, pp. 41–64, 2006.

[5] M. O'Keeffe and M. O. Cinnéide, "Search-based refactoring: an empirical study," *J. Softw. Maint. Evol.*, vol. 20, no. 5, pp. 345–364, 2008.

[6] J. Stender, *Parallel Genetic Algorithm: Theory and Applications*, 1993, vol. 14 Frontiers in Artificial Intelligence and Applications.

[7] W. Rivera, "Scalable parallel genetic algorithms," *Artif. Intell. Rev.*, vol. 16, no. 2, pp. 153–168, 2001.

[8] E. Alba, *Parallel Metaheuristics*. John Wiley and Sons Inc, 2005.

[9] G. Antoniol and Y.-G. Guéhéneuc, "Feature identification: An epidemiological metaphor," *Transactions on Software Engineering (TSE)*, vol. 32, no. 9, pp. 627–641, September 2006, 15 pages.

[10] T. Biggerstaff, B. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in *Proceedings of the International Conference on Software Engineering*, IEEE Computer Society Press, May 1993, pp. 482–498.

[11] V. Kozaczynski, J. Q. Ning, and A. Engberts, "Program concept recognition and transformation," *IEEE Transactions on Software Engineering*, vol. 18, no. 12, pp. 1065–1075, Dec 1992.

[12] Denys Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *Transactions on Software Engineering (TSE)*, vol. 33, no. 6, pp. 420–432, June 2007, 14 pages.

[13] P. Tonella and M. Ceccato, "Aspect mining through the formal concept analysis of execution traces," in *Proceedings of IEEE Working Conference on Reverse Engineering*, 2004, pp. 112–121.

[14] F. Asadi, M. D. Penta, G. Antoniol, and Y.-G. Gueheneuc, "A heuristic-based approach to identify concepts in execution traces," in *European Conference on Software Maintenance and Reengineering*, Madrid (Spain), Mar 15-18 2010, pp. 31–40.

[15] A. Nicolas and L. Timothy, "Extracting concepts from file names; a new file clustering criterion," in *Proceedings of the International Conference on Software Engineering*, April 1998, pp. 84–93.

[16] N. Wilde and M. Scully, "Software reconnaissance: Mapping program features to code," *Journal of Software Maintenance - Research and Practice*, vol. 7, no. 1, pp. 49–62, Jan 1995.

[17] M. Eaddy, A. Aho, G. Antoniol, and Y.-G. Gueheneuc, "Cerberus: Tracing requirements to source code using information retrieval dynamic analysis and program analysis," in *ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*. Washington DC USA: IEEE Computer Society, 2008, pp. 53–62.

[18] W. E. Wong, S. S. Gokhale, and J. R. Horgan, "Quantifying the closeness between program components and features," *Journal of Systems and Software*, vol. 54, no. 2, pp. 87 – 98, 2000, special Issue on Software Maintenance.

[19] D. Edwards, S. Simmons, and N. Wilde, "An approach to feature location in distributed systems," Software Engineering Research Center, Tech. Rep., 2004.

[20] A. D. Eisenberg and K. D. Volder, "Dynamic feature traces: Finding features in unfamiliar code," in *proceedings of the* 21^st *International Conference on Software Maintenance*. IEEE Press, September 2005, pp. 337–346.

[21] N. Gold, M. Harman, Z. Li, and K. Mahdavi, "Allowing overlapping boundaries in source code using a search based approach to concept binding," *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, pp. 310–319, 2006.

[22] N. Gold, "Hypothesis-based concept assignment to support software maintenance," *17th IEEE International Conference on Software Maintenance (ICSM'01)*, pp. 545–548, 2001.

[23] L. Dapeng, M. Andrian, P. Denys, and R. Vaclav, "Feature location via information retrieval based filtering of a single scenario execution trace," in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. New York NY USA: ACM, 2007, pp. 234–243.

[24] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 287–300, 2008.

[25] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *Proceedings of 22nd IEEE International Conference on Software Maintenance*. Philadelphia Pennsylvania USA: IEEE CS Press, 2006, pp. 469 – 478.

[26] B. S. Mitchell, M. Traverso, and S. Mancoridis, "An architecture for distributing the computation of software clustering algorithms," *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA '01)*, pp. 181–190, 2001.

[27] K. Mahdavi, M. Harman, and R. M. Hierons, "A multiple hill climbing approach to software module clustering," *Proceedings of the International Conference on Software Maintenance (ICSM '03)*, pp. 315–324, 2003.

[28] S. Yussof, R. A. Razali, and O. H. See, "A parallel genetic algorithm for shortest path routing problem," *Proceedings of the 2009 International Conference on Future Computer and Communication*, pp. 268–273, 2009.

[29] W. Zhi-xin and J. Gang, "A parallel genetic algorithm in multi-objective optimization," *Control and Decision Conference, 2009. CCDC '09. Chinese*, pp. 3497–3501, 2009.

[30] F. Liu, X. Chen, and Z. Huang, "Parallel genetic algorithm for finding roots of complex functional equation," *2nd International Conference on Pervasive Computing and Applications, 2007. ICPCA*, pp. 542–545, 2007.

[31] Y.-W. Chen, Z. Nakao, and X. Fang, "A parallel genetic algorithm based on the island model for image restoration," *Proceedings of the 1996 IEEE Signal Processing Society Workshop on Neural Networks for Signal Processing [1996] VI.*, pp. 109 – 118, 1996.

[32] Y. Fukuyama and H.-D. Chiang, "A parallel genetic algorithm for service restoration in electric power distribution systems," *In Proceedings of the 1995 IEEE International Conference on Fuzzy Systems*, pp. 275–282, 1995.

[33] D. L. A. de Araujo, H. S. Lopes, and A. A. Freitas, "A parallel genetic algorithm for rule discovery in large databases," *1999 IEEE International Conference on Systems, Man, and Cybernetics, 1999 (IEEE SMC99)*, vol. 3, pp. 940–945, 1999.

[34] B. Zorman, G. M. Kapfhammer, and R. Roos, "Creation and analysis of a javaspace-based distributed genetic algorithm," *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, vol. 3, pp. 1107 – 1112, 2002.

[35] Janice Ka-Yee Ng, Y.-G. Guéhéneuc, and G. Antoniol, "Identification of behavioral and creational design motifs through dynamic analysis," *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, December 2009, under publication.

[36] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[37] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.

[38] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, 1990.

[39] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool." *IEEE Trans. Software Eng.*, vol. 32, no. 3, pp. 193–208, 2006.

[40] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS '67: Proceedings of the spring joint computer conference*. New York, NY, USA: ACM, 1967, pp. 483–485.

[41] D. Garlan and M. Shaw, "An introduction to software architecture," in *Advances in Software Engineering and Knowledge Engineering*, vol. 1. New York: World Scientific, 1993.