UQAC
Université du Québec
à Chicoutimi

MÉMOIRE

PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

MOHAMED ILYES RAHIM

COMPRÉHENSION D'UN PROGRAMME À TRAVERS LA SEGMENTATION ET L'ANALYSE DES TRACES

MAI 2019

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

REMERCIEMENTS

Je tiens à remercier tous ceux ont participé de prés ou de loin à la réalisation de ce mémoire, particulièrement :

Mon directeur de recherche, Prof. Raphaël Khoury pour avoir accepter de diriger mes recherches, pour ses conseils et ses corrections.

Mon co-directeur de recherche, Prof. Sylvain Hallé pour m'avoir pris en charge dans son laboratoire.

Mon père et ma mère pour leur incessant soutien.

Finalement, je voudrais remercier mes amis et les personnes avec qui j'ai travaillé et qui ont rendu cette aventure possible.

RÉSUMÉ

L'analyse des traces permet aux ingénieurs logiciels de mieux comprendre le comportement des systèmes qu'ils gèrent et constituent donc un outil essentiel pour la réalisation de nombreuses tâches nécessitant une compréhension des systèmes complexes, notamment l'analyse de sécurité, le débogage et la maintenance. Cependant, la taille considérable des traces d'exécutions peut nuire à l'efficacité de l'analyse des traces. Pour résoudre ce problème, nous proposons dans ce mémoire une approche de segmentation de la trace à trois facteurs qui offre aux utilisateurs la possibilité de diviser une trace d'exécution volumineuse en un nombre réduit de segments, chacun correspondant à une phase d'exécution distincte. Nous démontrons expérimentalement que cette méthode est plus efficace pour diviser une trace de manière concordante avec le comportement sous-jacent du programme que les algorithmes existants. Nous examinons également la question de l'extraction d'éléments-clés à partir d'une trace et démontrons une nouvelle fois expérimentalement que les traces segmentées en utilisant notre approche à trois facteurs peuvent être plus facilement soumises à cette analyse.

CHAPITRE 1

INTRODUCTION

1.1 PROBLÈME ET MOTIVATIONS

Pour faire les tâches de maintenance d'un grand système logiciel, les ingénieurs logiciels doivent comprendre comment ce système est construit. La compréhension de ce système permet aux ingénieurs logiciels d'effectuer des activités comme le débogage, l'évolution ou l'amélioration des performances du système. La difficulté qui fait face aux mainteneurs lorsqu'ils essaient de comprendre le comportement d'un système est que souvent la documentation ne reflète pas l'implémentation du système. Ceci peut être dû au cout élevé de la modification de la documentation, à une documentation initiale de mauvaise qualité (ou même inexistante), ou au fait que le concepteur original du système a passé à une autre projets ou d'autre entreprise, etc. Plusieurs approches sont proposées pour aider les ingénieurs logiciels à comprendre le comportement des systèmes. L'objectif de ces approches est de réduire le temps consacré aux tâches de maintenance.

Les ingénieurs logiciels ont généralement recours à l'analyse du code source pour comprendre le comportement des logiciels. Une solution potentielle consiste à utiliser les techniques de fouille de texte mises au point à l'origine pour faciliter l'analyse de grands corpus de

textes écrits en langues naturelles telles que l'anglais. Ces techniques peuvent être adaptées et appliquées à des problèmes d'ingénierie logicielle tels que l'analyse de traces.

Quelle que soit la méthode utilisée pour analyser le code source, il existe deux catégories de techniques d'analyse du code source, soit l'analyse statique et l'analyse dynamique. On utilise ces deux techniques selon que l'analyse nécessite l'exécution du système ou non (Corbi, 1989).

L'analyse statique est basée sur l'analyse de tout le code source du programme sans l'exécuter. Elle donne une vue complète du système ; cependant, lorsque les ingénieurs logiciels ont besoin de comprendre une partie du système, elle devient un inconvénient, de plus, les concepts de programmation avancés tels que la ligature dynamique, le polymorphisme et le parallélisme rendent difficile l'application de l'analyse statique (Shafiee, 2013). L'analyse dynamique se concentre sur une partie du système par l'exécution d'un scénario choisi par le mainteneur. L'exécution d'un scénario permet aux mainteneurs de se concentrer sur les parties du système qui les intéressent. L'analyse dynamique est souhaitable aussi lorsque le mainteneur veut comprendre le comportement d'un système cible en lui donnant des entrées variées. Dans ce mémoire, on va plutôt se concentrer sur l'analyse dynamique.

Les donnes collectées au cours de l'exécution du scénario sont généralement stockées dans un fichier appelé trace d'exécution (Pirzadeh, 2012). Nous pouvons tracer n'importe quel aspect d'un système et le type de la trace dépend de ce que nous voulons savoir sur lui. Une trace peut être une trace d'appels de méthodes, une trace d'instructions, ou une trace de communication entre processus, etc. Les traces générées par l'exécution d'un scénario ont une taille importante, ils peuvent atteindre plusieurs milliers de lignes par exécution, ce qui rend difficile leur analyse manuelle. Il est donc nécessaire de trouver des solutions pour abstraire les traces de grande taille afin de faciliter leur compréhension. Plusieurs approches

ont été proposées pour abstraire les traces de grande taille (voir Cornelissen et al. (2009) pour une étude sur ces approches). Récemment, la plupart de ces approches reposent sur le concept de la segmentation des traces. Le but est de segmenter une trace de grande taille en un ensemble de phases qui reflètent les phases d'exécution d'un programme (voir Pirzadeh et Hamou-Lhadj (2011a), Medini et al. (2012), Medini et al. (2014), Benomar et al. (2014), Pirzadeh et Hamou-Lhadj (2011b), Pirzadeh et al. (2011)).

Dans ce mémoire nous proposons une approche de segmentation des traces basées sur les lois de Gestalt (similarité et continuité). Ces lois sont introduites en psychologie et indiquent que nos perceptions obéissent à un certain nombre de lois qui nous aident à reconnaître les formes. Ils expliquent comment notre système de perception se fonctionne afin de regrouper visuellement les objets perçus en fonction de leur contexte pour construire des formes qu'on peut les reconnaître. (Koffka, 1935). Nous étendons une approche existante (Pirzadeh et al., 2011) en incorporant aussi un rapproche des éléments de la trace selon la similarité dans les paramètres et valeurs de retours des méthodes : si deux méthodes ont un paramètre ou valeur de retour similaire on va rapprocher ces deux méthodes l'une vers l'autre. Cette approche résume ensuite chaque phase résultant de la segmentation pour faciliter la compréhension de ces phases par les mainteneurs.

1.2 MÉTHODOLOGIE

Dans cette recherche nous suivons une méthodologie à plusieurs étapes.

La première étape consistait à étudier les différentes approches de résumé des traces d'appels de méthodes et à spécifier la problématique de résumé des traces.

Dans la deuxième étape, on propose une méthode de segmentation des traces qui se base sur

une méthode existante de Pirzadeh et Hamou-Lhadj (2011a).

La troisième étape consistait à segmenter les traces avec notre méthode de segmentation et la méthode de Pirzadeh et Hamou-Lhadj (2011a).

La dernière étape consistait à résumer les segments résultant de notre méthode de segmentation et de la méthode de Pirzadeh et al. en utilisant deux algorithmes de segmentation (TF-IDF et Helmholtz).

1.3 CONTRIBUTIONS

Les principales contributions à la recherche de ce mémoire sont les suivantes :

1. L'utilisation des paramètres des méthodes pour segmenter les traces de grande taille.

2. La comparaison entre deux méthodes de résumé (Helmholtz et TF-IDF) pour résumer les phases d'exécutions.

3. La validation de l'approche par l'application sur des traces de plusieurs systèmes.

1.4 ORGANISATION

Le reste de mémoire est structuré comme suit.

Le chapitre 2 fait un survol des domaines qui sont en lien avec notre recherche, tel que la compréhension des programmes et la maintenance des logiciels. Nous présenterons aussi les approches existantes dans la littérature en matière de segmentation et de résumé des traces.

Le chapitre 3 présente notre approche de segmentation basée sur des travaux antérieurs et discute la principale contribution de ce travail qui est l'ajout des paramètres de méthodes dans

le processus de segmentation des traces ainsi que la présentation des concepts de similarité et continuité utilisés pour segmenter les traces. Nous présentons aussi les algorithmes, tirés du domaine du *text mining*, qui sont utilisés pour résumer les phases.

Le chapitre 4 porte sur une évaluation empirique de l'approche de segmentation des traces développée au chapitre 3, et contrastant son efficacité avec celle d'approches existantes. Nous comparons aussi l'efficacité de deux méthodes utilisées pour produire un résumé des segments, à savoir TF-IDF et Helmholtz.

Enfin, nous concluons le mémoire par un résumé des contributions, et un survol des travaux futurs possibles.

CHAPITRE 2

ÉTAT DE L'ART

2.1 LA MAINTENANCE LOGICIELLE ET LA COMPRÉHENSION DES PROGRAMMES

La maintenance logicielle est définie dans la norme IEEE 1219 comme : «la modification d'un logiciel après son entrée en production, afin de corriger ses erreurs, d'améliorer ses performances ou autres attributs, ou pour adapter le produit à un environnement modifié.» (IEEE-1219, 1998).

La compréhension des systèmes accélère considérablement les tâches de maintenance (Khoury et al., 2016). Cette compréhension nous permet de faire des taches de maintenance comme le débogage (Jerding et Stasko, 1998), l'amélioration des fonctionnalités (Systa, 2000), ou l'analyse de performance (Becker et al., 2007).

La compréhension des systèmes est définie comme l'activité consistant à comprendre comment fonctionne un système logiciel ou une partie de celui-ci (Maalej et al., 2014). Les ingénieurs dans cette discipline passent entre 50 % et 60 % de leur temps à comprendre un système (Basili, 1996). Il existe plusieurs modèles pour comprendre le fonctionnement des programmes (Storey, 2006; Hamou-Lhadj, 2005). Nous les décrivons brièvement ci-dessous.

2.1.1 LE MODÈLE DESCENDANT

Ce modèle est utilisé généralement lorsque le mainteneur est familiarisé avec le système. Celui-ci débute son travail par des suppositions sur le dit système. Par la suite, il vérifie ses suppositions initiales en les projetant sur un code source (Brooks, 1977). On peut appliquer un tel modèle à l'analyse des traces en effectuant une comparaison entre les hypothèses de départ sur la trace et le code source du système. Ainsi, un premier travail d'investigation est alors complété (Hamou-Lhadj, 2006).

2.1.2 LE MODÈLE ASCENDANT

Dans ce modèle, l'ingénieur logiciel commence par la lecture du code source plusieurs fois pour construire une idée sur le code (Letovsky, 1986). Ce modèle peut être appliqué à l'analyse des traces par l'exploration des lignes de la trace, et en essayant d'attribuer des résumés aux différents patrons de lignes.

2.1.3 LE MODÈLE INTÉGRÉ

Le troisième modèle, appelé le modèle intégré, est fréquemment employé par les ingénieurs logiciels. En théorie, il consiste à combiner l'utilisation des deux modèles expliqués ci-haut, soit descendant et ascendant. En pratique, les mainteneurs s'accordent à n'utiliser que l'un ou l'autre modèle selon leurs besoins (Pirzadeh, 2012).

2.2 LE CONCEPT DE TRACE D'EXÉCUTIONS

Une trace d'exécution est l'ensemble des évènements collectés au cours de l'exécution d'un scénario sur un programme donné. On retrouve différents types d'événement collectés. À ce sujet, le mainteneur peut tracer les appels des méthodes (nom, valeurs de retour, paramètres, niveau d'imbrication des méthodes), ou encore la communication entre les processus ou autres aspects jugés utiles pour la bonne comprehension du système (Shafiee, 2013). Dans ce mémoire, nous nous intéressons aux traces d'appels de méthodes. Lorsque nous évoquons ce dernier concept, nous faisons référence aux fonctions, routines et procédures.

Un exemple de trace d'appel de méthodes est illustré à la Figure 2.1. Dans cet exemple, la trace est représentée sous la forme d'arbre dans laquelle le routine R1 appelle R2, R5, R6, et R7, et le routine R2 appel R3 et R4.

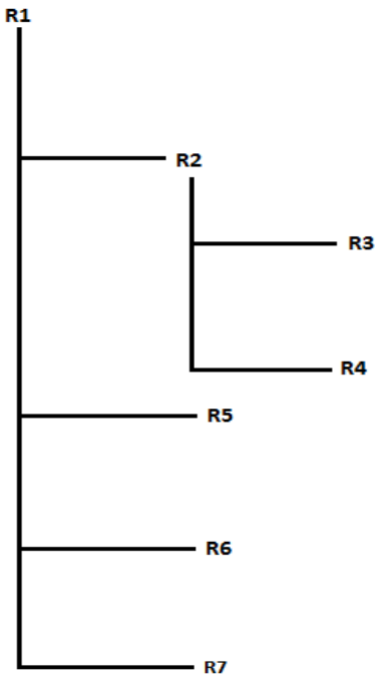

Figure 2.1 – Exemple de trace d'appels de routine (Shafiee, 2013)

Une trace peut être générée par instrumentation du code source. Celle-ci consiste à ajouter des instructions au code source original. On peut citer ici comme exemple l'ajout des instructions d'impression. On peut également instrumenter le bytecode du programme ou encore l'environnement d'exécution (système d'exploitation ou machine virtuelle) (Hamou-Lhadj, 2005).

Une autre façon de générer une trace d'exécution d'un programme consiste à utiliser la programmation orientée aspect (POA). Cette méthode permet en effet aux ingénieurs d'insérer des bouts de code dans le programme sans altérer le code source. On peut aussi utiliser le débogueur pour générer une trace ; dans ce cas l'ingénieur place des points d'arrêt dans le programme (Shafiee, 2013). Dans notre travail, nous allons utiliser la programmation orientée aspect pour générer les traces. Ce type de programmation permet de recueillir toutes les informations des méthodes sans pour autant modifier le code source. Ces informations renferment des paramètres de méthodes, des valeurs de retour, les noms des méthodes et autres informations pertinentes sur les méthodes.

2.3 LE CONCEPT DE PHASE D'EXÉCUTION

Une approche utilisée pour comprendre facilement le contenu des traces d'exécution consiste à segmenter le processus d'ensemble en courtes phases. Reiss et al. ont défini une phase comme suit : « une phase de programme peut être définie comme une partie de l'exécution du programme qui partage un ensemble de propriétés ou de traits communs. Du point de vue du concepteur du programme, ces propriétés se rapportent à ce que le programmeur fait à un niveau élevé, par exemple, lire une entrée, traiter une commande, accéder à une base de données, attendre une connexion ou calculer un ensemble de valeurs.» (Reiss, 2005). Segmenter une longue trace d'exécution (générée à partir de l'exécution d'un scénario) en un

ensemble de phases peut aider les mainteneurs à comprendre plus facilement le contenu de la trace. Ainsi, les phases d'exécution peuvent aider le mainteneur à localiser les erreurs dans le code, à améliorer la performance des programmes et à permettre d'avoir une idée plus globale sur le programme en question (Pirzadeh et Hamou-Lhadj, 2011a).

En revanche, les traces d'exécution sont la plupart du temps de grande taille. À cet égard, un simple scénario peut fournir une grande trace d'exécution. Par conséquent, il devient difficile pour le mainteneur d'identifier visuellement et avec justesse le début et la fin de la phase. Pour remédier à cette situation, plusieurs approches ont été proposées (Khoury et al., 2016; Pirzadeh et Hamou-Lhadj, 2011a; Reiss, 2005) .

D'un point de vue plus théorique, on peut dire qu'une trace d'exécution peut être vue comme étant un ensemble de trois grandes phases d'exécution distinctes, renfermant chacune un ensemble de sous-phases. Ainsi, une trace contient la phase d'initialisation, la phase de calcul ainsi que la phase finalisation (Pirzadeh et Hamou-Lhadj, 2011c).

2.4 L'ANALYSE DES TRACES ET LA FOUILLE DE TEXTE

Dans le domaine de la fouille de texte il existe plusieurs algorithmes qui peuvent être appliqués aux problèmes de l'analyse des traces. Plusieurs chercheurs ont effectué des travaux d'envergure sur cette question (Khoury et al., 2016; Pirzadeh et al., 2011) . Pirzadeh et al. (2011) ont présenté une analogie pertinente entre l'analyse des traces et la fouille de texte. Ils ont considéré une trace comme un corpus, une phase comme un document, et un élément comme un terme. De cette façon, ils peuvent bénéficier d'algorithmes de fouille de texte dans le cadre d'analyses portant sur des traces. La Figure 2.2 montre d'ailleurs cette analogie entre les deux domaines : un document est l'ensemble de termes et le corpus est l'ensemble de documents. À l'opposé, nous pouvons traiter la trace comme un corpus, une phase comme un

document et un élément dans la trace comme un terme.

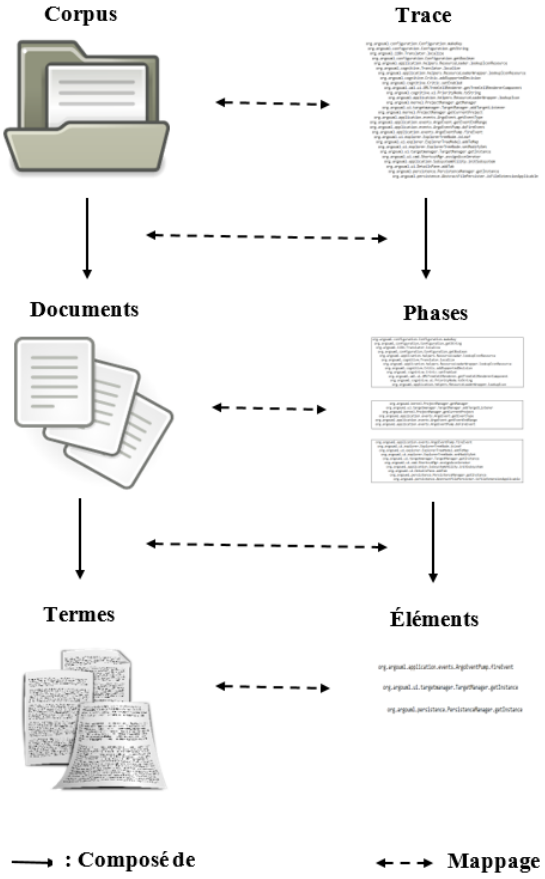

Figure 2.2 – La relation entre la fouille de textes et l'analyse des traces (Pirzadeh et al., 2011)

Dans les deux sous sections suivantes, nous allons presenter deux techniques qui peuvent être utilisées dans le but de résumer une trace ou une phase d'exécution. Les deux techniques tirent leur origine dans le traitement du langage naturel et la psychologie de la forme (ou gestaltisme). En outre, elles peuvent être adaptées au problème de l'analyse des traces.

2.4.1 TF-IDF

Le TF-IDF (de l'anglais *term frequency-inverse document frequency*) (Joachims, 1998) est une technique qui tire son origine dans le traitement de texte en langage naturel. Il fournit entre autres une mesure de l'importance d'un mot dans un document donné lorsque ce document est placé est contextualisé dans un un corpus de textes renfermant plusieurs documents.

Alors que plusieurs variantes de cette algorithme ont été proposées, l'idée principale est d'attribuer un score à chaque mot d'un document. Ce score augmente au fur et à mesure que le nombre d'occurrences de ce mot augmente dans le document. En revanche, il diminue si le mot apparaît fréquemment dans les autres textes à l'intérieur. Par conséquent, les mots qui apparaissent fréquemment tels que «le» ou «et» présentent un faible score, puisqu'ils sont communs à l'ensemble du corpus. De même, les mots qui apparaissant fréquemment dans un texte donné, mais qui sont rares dans le reste du corpus, présenteraient un score TF-IDF élevé. Ces mots sont de nature à être les plus significatifs dans le document à l'étude (Joachims, 1998).

Pour leur part, Pirzadeh et al. (2011) ont montré que cette technique peut être utilisée efficacement en vue d'extraire des éléments clés d'une trace d'exécution segmentée. en vue d'extraire la méthode traite chaque appel de méthode comme un terme et va traiter chaque segment comme un document. Ainsi, TF-IDF filtrera les noms de méthodes qui apparaissent fréquemment tout au long de la trace d'exécution et assignera des poids élevés aux appels de méthodes fréquentes dans un segment donné, mais qui demeurent rares dans le reste de la trace.

2.4.2 PRINCIPE DE HELMHOLTZ

Le principe de Helmholtz est un principe en psychologie qui stipule que les êtres humains semblent détecter plus facilement les patrons lorsque ceux-ci s'écartent du hasard (Desolneux et al., 2007). Pour illustrer notre propos, dans la Figure 2.3, les mêmes cinq structures de point existent dans les deux images. Dans l'image de droite, il est possible de percevoir les cinq groupes de points, vu qu'il y a un grand écart par rapport au hasard. En revanche, dans l'image de gauche, nous ne pouvons pas percevoir les cinq structures de point étant donné qu'ils sont entourés d'un grand nombre de points aléatoires disparates (Dadachev et al., 2012).

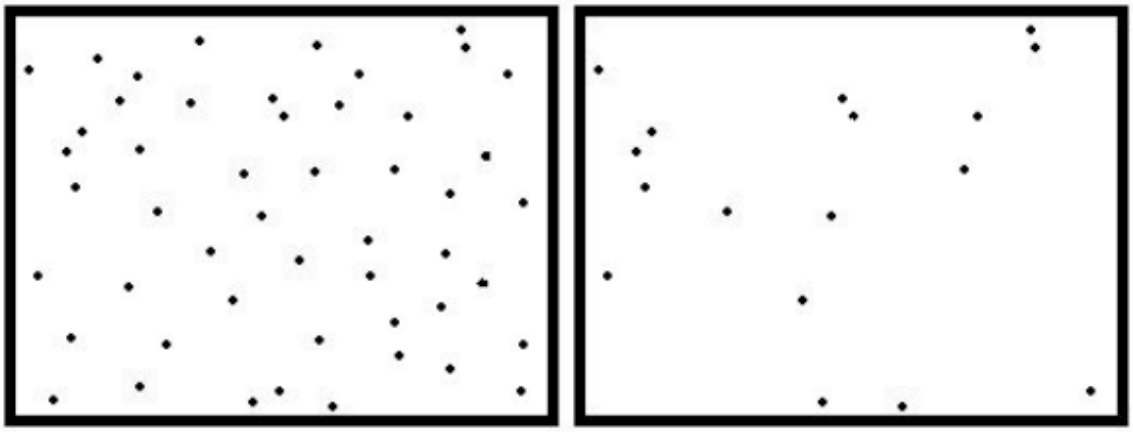

Figure 2.3 – Le principe de Helmholtz dans la perception humaine (Dadachev et al., 2012)

Pour leur part, Dadachev et al. (2012) ont proposé un autre algorithme d'extraction d'éléments clés d'une trace (résumé). Leur méthode s'inspire largement du principe de Helmholtz (Lowe, 2012), et Khoury et al. (2016) et présente une procédure algorithmique fine pour appliquer cette méthode au problème d'extraction des éléments clés des traces d'exécution.

En ce qui concerne l'analyse des appels de fonctions dans une trace, on peut également s'inspirer du principe de Helmholtz pour extraire les éléments de la trace les plus significatifs en

traitant une trace comme un document. Nous désignons alors un élément comme significatif si son taux d'apparition dans la phase représente un écart majeur du hasard, et ce, en comparaison à son taux d'occurrence dans toute la trace (Khoury et al., 2016).

2.5 LA PSYCHOLOGIE DE LA FORME ET LA SEGMENTATION DES DONNÉES

Selon Koffka : «la psychologie de la forme cherche à comprendre les lois qui régissent notre capacité à acquérir et maintenir des perceptions significatives dans un monde qui apparaît chaotique.» (Koffka, 1935). La psychologie de la forme se penche sur les processus de la perception et de la représentation mentale et regroupe les objets les objets que l'on perçoit pour mieux comprendre la scène dans son intégralité (Koffka, 1935).

La psychologie de la forme s'exprime aussi par un ensemble de lois (les lois de la Gestalt) que le système de perception applique pour structurer les objets qu'il perçoit. Ces lois peuvent à leur tour être appliquées à la segmentation de données en générale (Pirzadeh et Hamou-Lhadj, 2011a). Nous en décrivons quelques-unes dans ce qui suit.

2.5.1 LA LOI DE SIMILITUDE

La Loi de similitude explique la façon dont le cerveau essaie d'associer les formes qui sont similaires. Koffka (1935). Par exemple, dans la Figure 2.4 ci-dessous, nous pouvons associer les carrés bleus de la ligne du haut entre eux, tandis que les autres formes ne connaissent aucun lien à la ligne du bas. Dans ce cas-ci, il n'y a pas d'association de forme et donc la loi de la similitude ne s'applique pas.

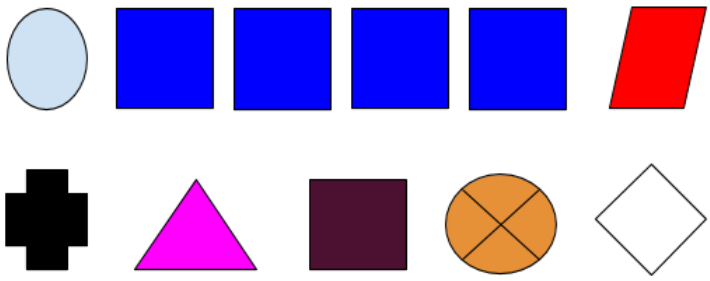

Figure 2.4 – La loi de similitude

2.5.2 LA LOI DE BONNE CONTINUITÉ

Un ensemble de points proches l'un vers l'autre perçus comme une seule forme continue Koffka (1935). La Figure 2.5 montre un ensemble de points noirs qui sont proches les uns des autres et il est possible de distinguer une ligne courbe.

Figure 2.5 – La loi de continuité

2.5.3 LA LOI DE LA PROXIMITÉ

Nous dressons un ensemble d'éléments les plus proches en un seule forme Koffka (1935). La Figure 2.6 montre trois groupes de points distincts formant trois colonnes de points noirs.

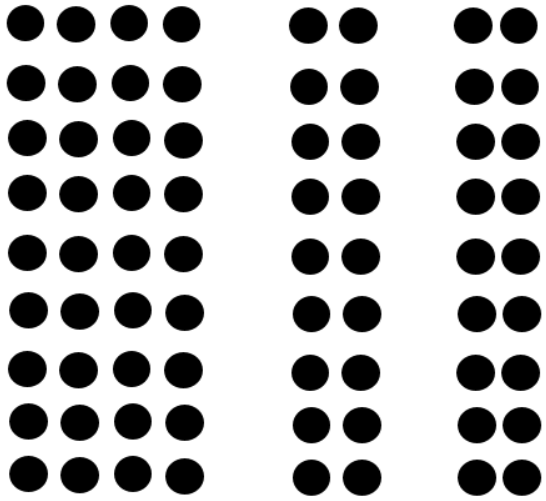

Figure 2.6 – La loi de proximité

2.6 LE RÉSUMÉ DES TRACES

Selon (Radev et al., 2002) un résumé peut être défini comme « un texte produit à partir d'un ou de plusieurs textes, transmettant des informations importantes dans le texte original, et ne représentant pas plus de la moitié de la taille du texte original, et généralement beaucoup moins que cela.». Hamou-Lhadj et Lethbridge. ont defini un resumé d'une trace comme «une représentation abstraite de la trace qui résulte de la suppression des détails inutiles en utilisant à la fois la sélection et la généralisation» (Hamou-Lhadj et Lethbridge, 2006).

2.6.1 APPROCHES DE RÉSUMÉ

Plusieurs approches ont été proposées pour réduire la taille des traces dans le but de faciliter leur compréhension par les mainteneurs. Le principal défi de ces approches est de réduire la taille des traces sans perdre les informations pertinentes qui aideront à comprendre leurs

contenus (Hamou-Lhadj et al., 2005).

Une première approche consiste à filtrer les éléments de la trace selon certains critères. (Kuhn et Greevy, 2006) proposent en effet une méthode de résumé qui s'inspire du domaine du traitement de single. Leur méthode nous permet de bénéficier de la puissance des algorithmes de traitement du signal et de traitement des séries temporelles pour analyser les traces. Un exemple de la représentation d'une trace d'appel de méthodes comme un signal est illustré dans la Figure 2.7, l'axe x représente les séquences d'exécutions et l'axe y représente le niveau d'imbrication. Ils ont appliqué leur approche sur l'application web SmallWiki qui est une application de gestion des pages web et ils ont pu représenter la trace de SmallWiki sur un seul écran.

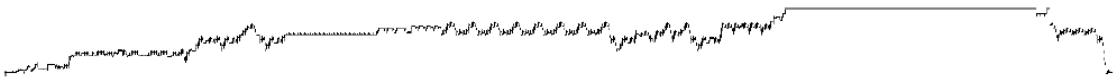

Figure 2.7 – Représentation d'une trace sous forme un signale (Kuhn et Greevy, 2006)

Les auteurs de cette étude commencent par tracer les niveaux d'imbrication des méthodes sur l'axe y et des points dans le temps sur l'axe x. Ensuite, ils coupent le signal entre deux événements consécutifs où le niveau d'imbrication n'augmente pas. Enfin, ils compressent chaque séquence en un seul événement. Ils obtiennent alors un résumé plus court que la trace originale. La Figure 2.8 Illustre le processus de résumé utilisé dans leur approche. De haut en bas 1) représente la trace dans son état initial, 2) les lignes verticales discontinues montrent les points où le niveau d'imbrication n'augmente pas et 3) la compression des événements pour sauver l'espace.

Cette méthode de résumé peut réduire la taille d'une grande trace jusqu'à 90 % et facilite la compréhension des traces par la visualisation de plusieurs traces dans un seul écran, étant donné leurs tailles réduites. L'inconvénient est que cette méthode recherche les fonctionnalités

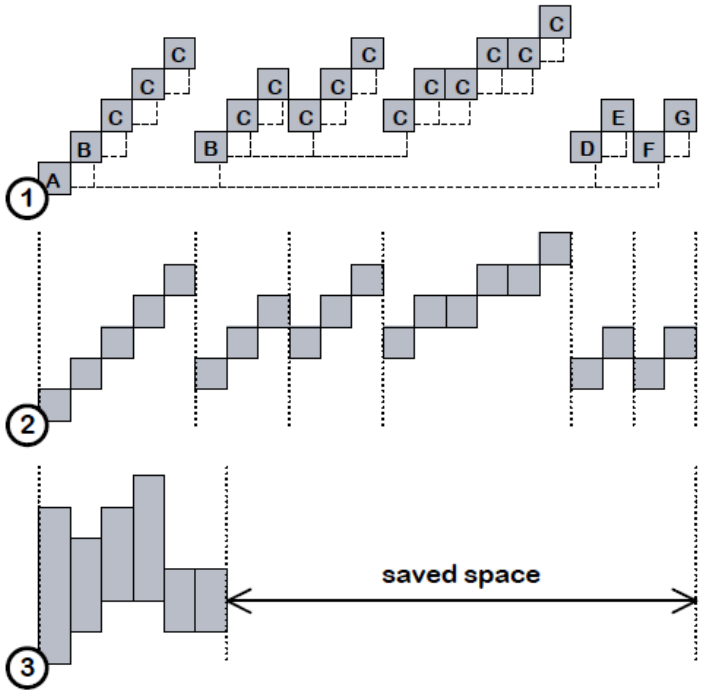

Figure 2.8 – Processus de résumé (Kuhn et Greevy, 2006)

en n'utilisant que le niveau d'imbrication. Elle ne garantit pas la distinction des différentes fonctionnalités inclues dans la trace. Par contre, en plus du niveau d'imbrication, l'utilisation d'autres informations comme les noms des méthodes ou les paramètres des méthodes dans le processus de recherche des fonctionnalités peuvent aider le mainteneur à mieux les distinguer.

.

D'autres approches suppriment les utilitaires qui apparaissent fréquemment dans toutes les parties de la trace. Hamou-Lhadj et Lethbridge. ont défini une utilitaire comme : «Tout élément d'un programme conçu pour la commodité du concepteur et de l'implémenteur et destiné à être accessible à partir de plusieurs endroits dans une certaine portée du programme.» (Hamou-Lhadj et Lethbridge, 2006).

La présence des utilitaires dans la trace représente un bruit indésirable dans un processus

visant à comprendre le contenu d'une trace d'exécution. Pirzadeh et al. (2009a) montrent que l'analyse du système est souvent encombrée lorsque l'ensemble de données analysé contient trop d'utilitaires, comme des méthodes avec un petit corps qui fournissent un service générique à plusieurs clients. Les mêmes méthodes utilitaires peuvent être appelées tout au long de l'exécution d'un programme et les calculs qu'elles effectuent ont tendance à demeurer abstraits. À terme, il est difficile de traiter les informations sur le comportement du programme lorsque celles-ci sont présentes dans un résumé de trace.

Hamou-Lhadj et Lethbridge (2003) proposent une approche qui supprime les données inutiles (méthodes utilitaires) d'une trace. Ils laissent ainsi les méthodes qui sont susceptibles d'aider les mainteneurs dans le processus de compréhension. Leur approche utilise trois techniques de filtrage qui suppriment les méthodes utilitaires : la première technique consiste à supprimer les séquences d'appels de méthodes contiguës qui résultent de l'exécution des boucles ; la deuxième technique consiste à supprimer les méthodes constructrices et destructrices des objets, ces méthodes d'accès aux variables, et à supprimer automatiquement les classes utilitaires à l'aide du graphe de dépendance des classes. Les classes (sommets) qui ont un grand nombre d'arcs entrants et un petit nombre d'arcs sortants sont dites classes utilitaires ; la dernière technique consiste à utiliser le polymorphisme pour occulter les détails d'implémentation. Le polymorphisme fournit une vue assez abstraite du code à travers les interfaces. Ceci permet la suppression des détails de l'implémentation et, par conséquent, la réduction de la taille de la trace.

Les auteurs ont testé leur approche résumé sur des traces générées à partir de l'exécution de 12 scénarios différents de l'application Weka. Initialement, la trace contenait 193121 appels de méthode et après l'application du premier filtre la taille de la trace a été réduite à 6015 appels de méthode. Ensuite, ils appliquent les deux autres filtres sur la trace résultante de l'application du premier filtre. La suppression des constructeurs a donné une trace de 5305

appels de méthodes et la suppression des méthodes d'accès aux variables a donné 6009 appels de méthodes et la suppression des méthodes utilitaires a donné une trace de 6008 appels de méthodes. Enfin, la suppression des détails du polymorphisme a donné une trace de 289 appels de méthodes. Cette procédure a donc permise une réduction significative d'appels de méthode.

D'autres approches lient les éléments de la trace selon des patrons. Systa (2000) a créé un outil qui s'appelle Shimba. Cet outil combine l'analyse statique et dynamique pour mieux comprendre le comportement des systèmes logiciels écrits en langage Java. L'analyse statique est faite par Shimba par l'extraction des classes et méthodes et les relations entre les classes. Ensuite, ces informations sont visualisées à l'aide d'un outil de reverse engineering qui s'appelle Rigi (Müller et al., 1993).

L'analyse dynamique se fait automatiquement au cours de l'exécution du programme à l'aide d'un débogueur personnalisé. Le débogueur génère une trace d'évènements qui est sauvegardés sous forme de deux diagrammes, le diagramme de scénario qui est une dérivation du diagramme de séquence, et le diagramme d'état. Ces deux diagrammes sont ensuite visualisés à l'aide de l'outil SCED (Koskimies et al., 1998).

Ces informations collectées statiquement et dynamiquement sont utilisées par Shimba pour analyser les traces. Cependant, les traces collectées ont une grande taille. Ainsi Shimba recherche les motifs comportementaux (séquence d'éléments du diagramme de scénario répétés au moins une fois) répétés dans une trace en utilisant l'algorithme de recherche de sous-chaînes de Boyer-Moore (Boyer et Moore, 1977). Les motifs trouvés augmentent le niveau d'abstraction des diagrammes de scénario et diminuent leur taille, ce qui aide les mainteneurs à comprendre le comportement des programmes.

D'autres chercheurs ont crée des outils de visualisation des traces pour faciliter l'exploration et l'analyse de ces dernières. Reiss (2005) a proposé un outil de visualisation de traces qui

s'appelle JIVE. La Figure 2.9 montre un exemple de visualisation d'une phase d'exécution par le programme JIVE. Cet outil fournit un résumé en temps réel du comportement des systèmes pour chaque intervalle de temps défini par l'utilisateur. JIVE collecte les informations au niveau classes comme le nombre d'appels de méthodes d'une classe, le nombre d'allocations d'objets de la classe etc.. Il collecte aussi les informations sur les interactions entre les threads comme l'état de chaque thread et le temps passé dans chaque état etc. Ces informations collectées sont ensuite compressées et affichées pour fournir au mainteneur une vue abstraite de ce qui se passe dans un système Java.

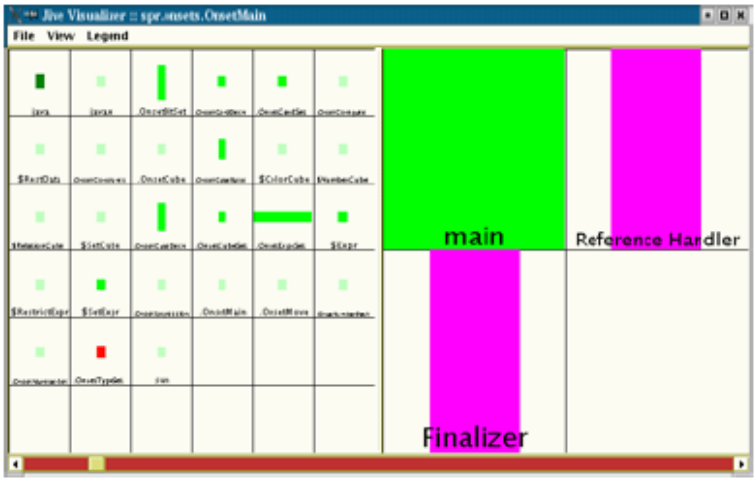

Figure 2.9 – Exemple de visualisation d'une phase d'exécution (Reiss, 2005)

Enfin Hamou-Lhadj et al. (2004) ont créé un outil de visualisation de traces qui s'appelle SEAT, Figure 2.10. Cet outil permet au mainteneur de compresser une trace de grande taille à un niveau qui leur permet de comprendre son contenu. Ils utilisent plusieurs algorithmes de compression pour cacher les détails de l'implémentation et conservent seulement les comportements importants du système.

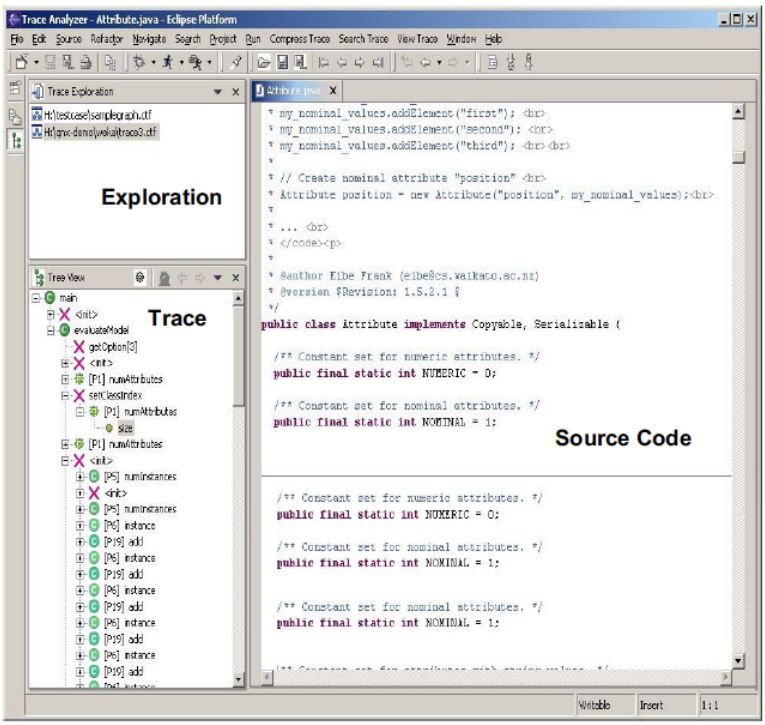

Figure 2.10 – Interface graphique de SEAT (Hamou-Lhadj et al., 2004)

2.7 SEGMENTATION DES TRACES

La segmentation des traces est le processus de division d'une grande trace en petit nombre de «segments». De cette façon, il est possible de révéler les actions faites par l'utilisateur (Pirzadeh et al., 2011). Par exemple, l'exécution d'un programme qui commence par une phase de démarrage au cours de laquelle les composants du dit programme sont chargés et initialisés. Chaque action ultérieure posée par l'utilisateur, comme l'ouverture d'un fichier ou encore l'activation d'une fonctionnalité du programme consiste en une autre phase distincte. Chaque segment peut comporter plusieurs milliers de lignes dans la trace d'exécution, mais pour les besoins de la compréhension du programme, il est utile de le traiter comme un événement atomique unique.

La segmentation des traces est un processus consistant en l'abstraction d'une grande séquence d'événements de bas niveau en un petit nombre d'événements de haut niveau. De cette façon, il est possible de révéler les actions faites par l'utilisateur

2.7.1 APPROCHES DE SEGMENTATION

Lorsque nous abordons les approaches de segmentation, Pirzadeh et Hamou-Lhadj (2011a) nous proposent une approche de segmentation des traces basée sur la loi Gestalt (continuité et similarité (Koffka, 1935)). Leur processus de segmentation est décomposé en deux étapes. La première étape consiste à rapprocher les éléments de la trace selon le nom de méthode. Le rapprochement se fait par la division de la distance entre deux méthodes qui ont le même nom par un facteur défini par l'utilisateur (schéma de similarité). Ensuite, ils rapprochent les éléments de la trace selon le niveau d'imbrication (schéma de continuité). À cet égard, si une méthode fait appel à une autre méthode. les auteurs divisent la distance entre ces deux méthodes par un facteur fixé après la première étape. On obtient alors une trace avec des groupes d'éléments denses. Pour sa part, la deuxième étape segmente cette trace en phases d'exécution par l'utilisation de l'algorithme K-means. Les auteurs ont évalué l'efficacité de leur méthode de segmentation par l'application sur deux systèmes logiciels implémentés en Java (JhotDraw et ArgoUML). Le scénario retenu dans le cas d'ArgoUML était le démarrage d'ArgoUML, ensuite le dessin d'un diagramme de classes et finalement la fermeture de l'application. Le résultat a compté 5 phases d'exécution, soit : la phase d'initialisation, la phase de chargement des modules auxiliaires, la phase de dessin du diagramme de classes, la phase de rafraîchissements et de mise à jour des modules enfin la phase d'arrêt de ArgoUML.

Dans ce mémoire, nous allons segmenter la trace selon trois facteurs au lieu de deux. En plus des deux schémas de similarité et continuité, on va rapprocher les éléments de la trace à partir

des paramètres d'entrée et des valeurs de retour des méthodes.

Medini et al. (2012, 2014) proposent SCAN, une approche qui divise les traces d'exécution en un ensemble de segments permettant une meilleure lecture des programmes. Leur approche est basée sur l'analyse statique du code source. Après avoir collecté l'ensemble des méthodes au cours de l'exécution d'un scénario, SCAN commence par l'analyse des appels entre méthodes pour filtrer celles qui sont des méthodes utilitaires. À titre d'exemple, SCAN supprime les méthodes liées aux évènements du clic. Celles-ci renferment un grand nombre d'appels entrants et un petit nombre d'appels sortants. Ensuite, il compresse la trace par un algorithme d'encodage pour supprimer les appels de méthodes répétés. Par la suite, SCAN construit une structure d'arbre où chaque noeud contient le nom de la méthode (à partir de la trace filtrée et compressée), les paramètres et le corps de la méthode (à partir du code source). Cet arbre est ensuite parcouru pour extraire les termes qui construisent chaque noeud. Ces termes sont ensuite pondérés par l'algorithme TF-IDF. On obtient donc une matrice de termes. Enfin, SCAN utilise cette matrice pour segmenter la trace en utilisant la programmation dynamique.

Pour leur part, Watanabe et al. (2008) ont proposé une technique automatique de détection des phases dans les traces d'exécution de grande taille. Les auteurs ont utilisé l'algorithme moins récemment utilisé (en anglais *least recently used* (LRU)). Cette approche permet l'observation d'objets créés et détruits au cours de l'exécution du programme. Une mise à jour significative du cache LRU montre l'apparition d'une nouvelle phase. Le problème avec cette approche est la scalabilité, étant donné le nombre important d'objets créés et détruits au cours d'une exécution. En outre, cette méthode de segmentation utilise différents paramètres pour fonctionner, obligeant l'utilisateur à tester plusieurs combinaisons de paramètres pour arriver au bon partitionnement de la trace.

Ils appliquent leur méthode de segmentation sur des traces de deux applications. Tool Mana-

gement System qui est une application multithread de gestion des serveurs et des requêtes HTTP et Book management System qui est une aussi une application multithread de gestion de bases de données. La segmentation est par ailleurs appliquée sur les traces avec différents paramètres. Ainsi, le paramètre taille du cache représente le nombre d'objets dans le cache LRU. Le paramètre taille de fenêtre représente la fréquence de mise à jour du cache LRU. Lorsque la fréquence atteint une valeur définie par l'utilisateur cela marque une transition vers une autre phase. Ces deux paramètres affectent la granularité des phases détectées. La Figure 2.11 montre le résultat de la segmentation sur une trace de l'application Tool Management System.

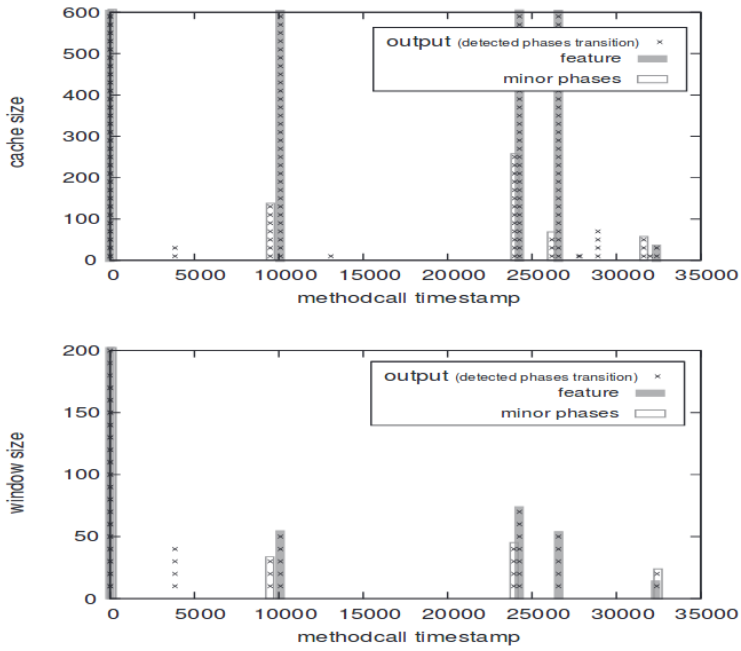

Figure 2.11 – Phases detecté dans une trace de l'application Tool Management System (Watanabe et al., 2008)

La figure si dessus montre le résultat de la segmentation par plusieurs valeurs de paramètre taille du cache et une valeur fixe de paramètre taille de fenêtre égale à 50. Par contre, la figure en bas montre le résultat de la segmentation par plusieurs valeurs de paramètre taille de fenêtre

et une valeur fixe de paramètre taille du cache égal à 300. La marque x indique que la x ème méthode dans la trace marque la transition vers une autre phase, une barre grise signifie une phase et un rectangle gris signifie une sous-phase.

Enfin, Kuhn et Greevy (2006) ont fait une analogie entre l'analyse dynamique et le traitement de signal. Ces auteurs représentent la trace sous forme d'un signal. Ainsi, ils tracent le niveau d'imbrication des méthodes contre des points dans le temps. Par la suite, ils coupent le signal à chaque deux événements consécutifs où le niveau d'imbrication n'augmente pas. En revanche, leur approche supprime beaucoup d'informations utiles qui peuvent nuire à une lecture efficace de la trace.

CHAPITRE 3

PRÉSENTATION DE L'APPROCHE

La segmentation et l'extraction des éléments clés des traces (résumé) peuvent fonctionner ensemble pour fournir une méthode efficace qui permet de déduire le comportement d'un programme à partir d'une trace. Même lorsqu'il est appliqué sur une trace qui capture une séquence d'actions de programme qui n'a jamais été enregistrée auparavant, notre approche [1] peut déduire le comportement sous-jacent du processus en quatre étapes Figure 3.1.

1. Tout d'abord, la trace est divisée en un ensemble de segments en utilisant un algorithme de segmentation des traces dans lequel chaque segment correspond à une phase d'exécution. Dans cette étape nous expliquons notre contribution principale qui est l'utilisation des paramètres de méthodes pour regrouper les méthodes cohésives.
2. Deuxièmement, nous appliquons un algorithme d'extraction d'éléments clés à chaque segment. Cela se traduit par la création d'un résumé de chaque segment de trace contenant ses appels de méthodes les plus représentatifs.
3. Dans la troisième phase, les résumés sont nettoyés des méthodes utilitaires en utilisant un processus algorithmique. Ces méthodes sont des méthodes génériques courtes dont la présence dans un résumé n'est pas informative du point de vue des mainteneurs du

1. https ://drive.google.com/open ?id=1mh6ilhlD2Xz3X_tc1HC4B65WfKiBWuHv

système, qui cherchent à comprendre le comportement sous-jacent de la trace.

4. Enfin, la taille du résumé est réduite en ne prenant que les dix premiers noms de méthodes (classés selon l'algorithme d'extraction d'éléments clés) et en supprimant tous les autres appels de méthodes du résumé.

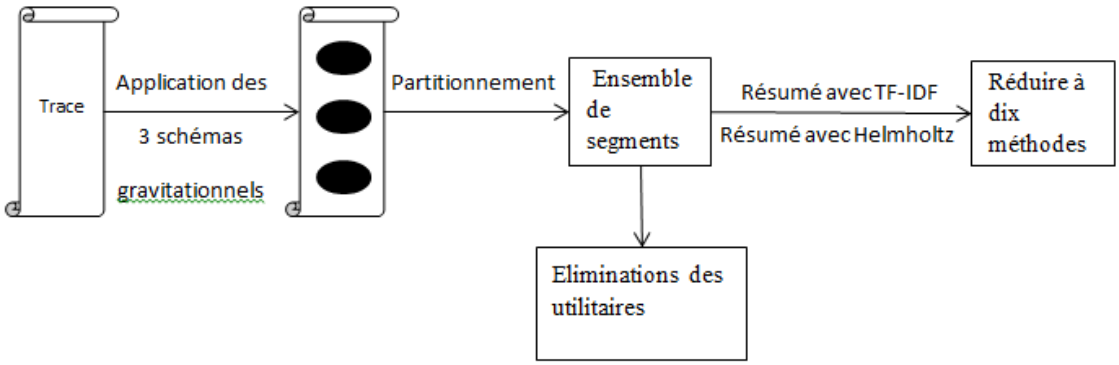

Figure 3.1 – Vue générale de l'approche.

À la fin de ce processus, une trace extrêmement grande sera réduite à une série de courts résumés, chacun comprenant au maximum dix noms de méthodes. Ainsi, comme il sera démontré dans le chapitre suivant il sera facile pour un mainteneur du système de consulter la documentation de dix méthodes et de comprendre son comportement lors de chaque phase d'exécution.

3.1 SEGMENTATION À TROIS FACTEURS

Dans une recherche antérieure, Pirzadeh et al. (Pirzadeh et Hamou-Lhadj, 2011a) ont proposé une approche d'analyse des traces qui fait face aux difficultés inhérentes rencontrées lors de l'extraction des éléments significatifs à partir de traces de grande taille. Cette approche fonctionne en deux étapes.

La première étape s'inspire de la théorie Gestalt et est appelée segmentation de trace. Elle traite de la division automatique de la trace en phases (ou segments) composées d'éléments de trace groupés et partitionnés. Dans la seconde étape, appelée extraction des éléments clés de la trace, un «résumé» est extrait de chacun des segments générés à l'étape 1, constitué des éléments de la trace les plus représentatifs et significatifs selon leur poids. Pirzadeh et al. font ensuite le résumé des traces en utilisant l'algorithme TF-IDF.

En s'appuyant sur la recherche de Pirzadeh et al. (Pirzadeh et Hamou-Lhadj, 2011a), nous détectons les phases candidates en utilisant un schéma gravitationnel. Notre contribution principale est l'ajout des paramètres de méthodes pour regrouper les méthodes cohésives. Notre processus de segmentation de traces fonctionne en deux étapes, à savoir le regroupement des éléments de la trace et le partitionnement de la trace en phases.

3.1.1 REGROUPEMENT DES ÉLÉMENTS DE LA TRACE

Dans cette étape, les éléments de la trace sont regroupés en phases candidates en utilisant des techniques basées sur les lois de la Gestalt à savoir la loi de similitude(similarité dans les noms des méthodes et dans les paramètres et valeur de retour) et la loi de continuité.

Nous attribuons à chaque élément de la trace une position sur un axe. Initialement, les positions sont attribuées de manière séquentielle, chaque élément étant distant de son prédécesseur et de son successeur, de sorte que la position de chaque élément corresponde à son index dans la trace (Figure 3.2). Les éléments jugés «similaires» sur certains critères sont alors glissés le long de cet axe, comme s'ils étaient tirés par des forces gravitationnelles entraînant la formation de groupes denses d'éléments similaires.

Les éléments sont repositionnés sur la base du schéma de similarité, où les éléments syntaxi-

Figure 3.2 – Vue schématique d'une trace avant le début du processus de segmentation.

quement identiques (c'est-à-dire les appels multiples à la même méthode) sont rapprochés l'un vers l'autre (Figure 3.3). Ensuite, un repositionnement basé sur le schéma de continuité est appliqué au vecteur résultant de ce repositionnement initial (Figure 3.4). Le schéma de continuité réduit la distance entre deux appels de méthode en fonction de leur niveau d'imbrication : les appels avec un niveau d'imbrication plus élevé sont repositionnés plus près des méthodes qui les ont appelés.

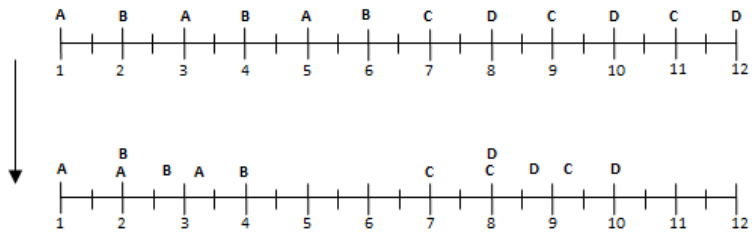

Figure 3.3 – Un exemple d'application du schéma de similarité à une trace (Pirzadeh et Hamou-Lhadj, 2011a).

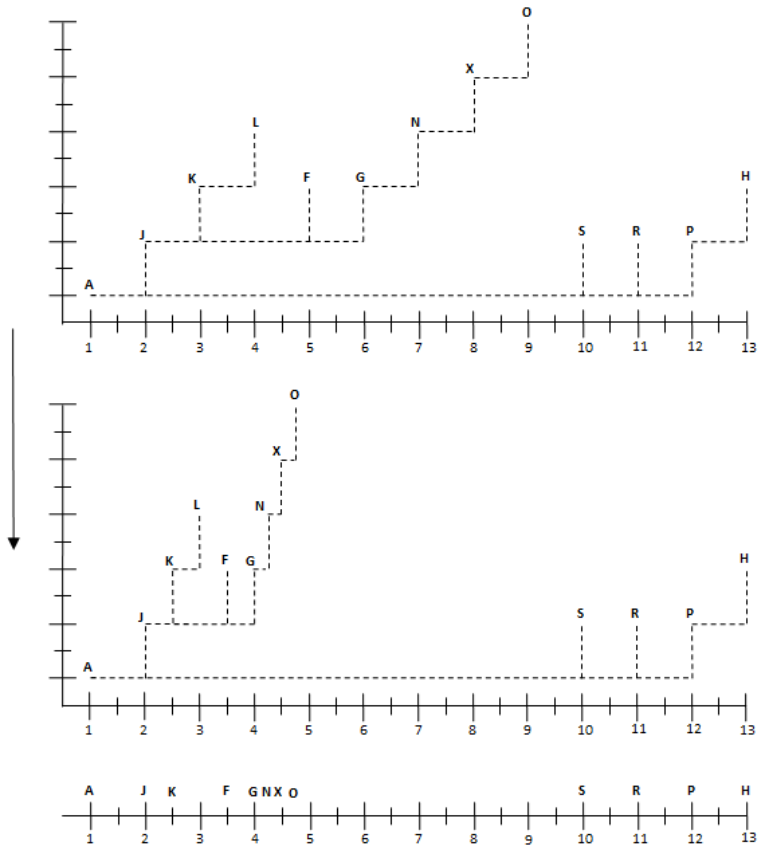

Figure 3.4 – Un exemple d'application du schéma de continuité à une trace (Pirzadeh et Hamou-Lhadj, 2011a).

Outre les schémas de similarité et de continuité proposé dans l'approche de Pirzadeh et al, nous segmentons également les traces selon un schéma de paramètres (schématisé à la Figure 3.5). Deux appels de méthode sont considérés comme similaires et, par conséquent, repositionnées dans des positions plus proches s'ils partagent les mêmes paramètres ou valeurs de retour, ou si la valeur de retour d'une méthode est utilisée comme paramètre pour l'autre. Ceci est basé sur l'intuition que si deux méthodes manipulent le même objet ou si une méthode prend en entrée un objet ou une valeur renvoyée par une autre méthode, alors ces deux méthodes feront probablement partie de la même phase d'exécution. Comme on le verra dans les expériences qui seront présentées au chapitre 4, nous avons constaté que l'utilisation de

paramètres améliore considérablement l'efficacité de l'algorithme de segmentation.

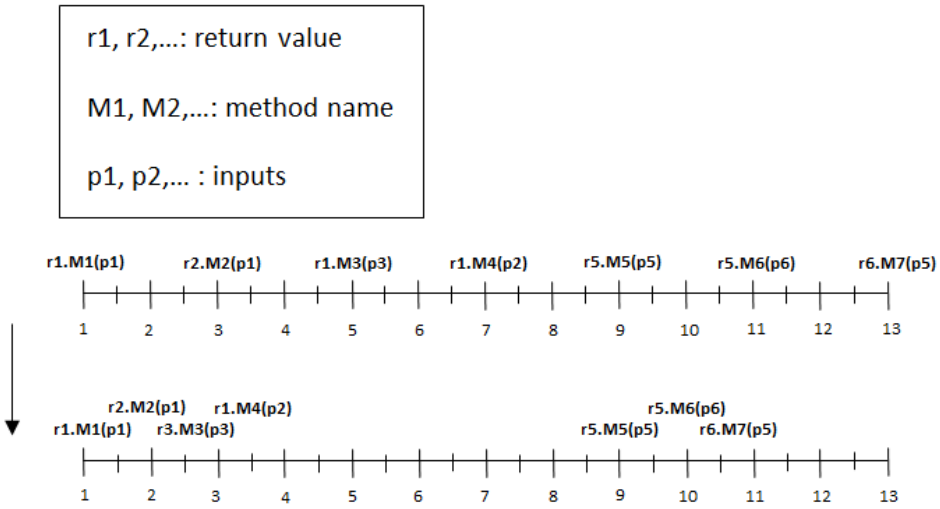

Figure 3.5 – Un exemple de schéma de paramètres

La deuxième étape consiste à partitionner la trace en phases à l'aide de l'algorithme de classification k-means(MacQueen, 1967).

3.1.2 PARTITIONNEMENT DE LA TRACE EN PHASES

Une fois le schéma gravitationnel appliqué, la position de chaque méthode est modifiée pour augmenter sa proximité par rapport aux autres méthodes partageant le même nom, les mêmes paramètres et celles appartenant à la même hiérarchie d'imbrication. Comme ces facteurs semblent corrélés avec les phases d'exécution, nous appliquons ensuite, l'algorithme de partitionnement de données k-moyennes (en anglais *k-means*) à ce vecteur pour identifier les emplacements les plus appropriés pour le début et la fin des phases. k-moyennes (MacQueen, 1967) est un algorithme de regroupement largement utilisé qui permet de regrouper des observations en k groupes, de manière à minimiser la variance à l'intérieur de chaque groupe. Dans notre cas, les observations sont les appels de méthodes positionnés dans un espace

vectoriel et les groupes correspondront aux segments.

Bien que k-moyennes a prouvé son efficacité dans cette tâche lors de recherches précédentes(Khoury et al., 2016; Pirzadeh et al., 2011; Pirzadeh et Hamou-Lhadj, 2011a), elle a un inconvénient : le nombre de groupes attendu doit être spécifié manuellement par l'utilisateur. Plusieurs méthodes algorithmiques ont été proposées pour sélectionner le nombre correct de groupes de manière automatisée. Pirzadeh et al. (Pirzadeh et al., 2011) étudient toutes ces alternatives et suggèrent leur propre solution, à savoir générer plusieurs partitions alternatives avec différentes valeurs de k, et les comparer en utilisant le critère d'information bayésien (en anglais *bayesian information criterion* ; en abrégé BIC) (Pelleg et Moore, 2000) pour déterminer quelle valeur est la plus susceptible d'être exacte. Dans notre approche, on va choisir le nombre de groupes manuellement.

3.2 EXTRACTION D'ÉLÉMENTS CLÉS

L'extraction d'éléments clés est le processus d'identification des principales fonctionnalités du programme qui sont présentés dans une trace ou dans chaque segment d'une trace (Pirzadeh et Hamou-Lhadj, 2011a). En d'autres termes, le but de l'extraction d'éléments clés est de générer un «résumé» qui met en évidence les aspects les plus significatifs de la trace. Cela est souvent nécessaire car même après que la trace ait été segmentée en phases, chaque segment est encore trop long pour être analysé et compris par un programmeur. Dans cette section, nous présentons les deux techniques(TF-IDF et Helmholtz) utilisées pour accomplir cette tâche. Les deux techniques ont leur origine dans le traitement automatique du langage naturel (TALN), et elles sont adaptées au problème de l'analyse des traces.

3.2.1 RÉSUMÉ AVEC TF-IDF

Nous appliquons cet algorithme à une trace d'exécution de méthodes pour trouver les méthodes les plus représentatives dans une trace. Plusieurs variantes de cet algorithme sont utilisées dans la pratique. Notre fonction de pondération est la suivante :

$$P_{i,k} = tf_{i,k} * \log\left(\frac{N}{1+idf_i}\right) \tag{3.1}$$

Où $tf_{i,k}$ est la fréquence de la méthode i dans la phase k, idf_i est la fréquence de la méthode dans la trace, et N est le nombre de phases dans la trace. Nous allons choisisir par la suite un pourcentage (choisi par l'utilisateur) des méthodes avec le poids le plus élevé.

3.2.2 RÉSUMÉ AVEC HELMHOLTZ

Le deuxième algorithme est inspiré du principe de Helmholtz, il est utilisé dans notre recherche pour l'extraction des éléments clé des phases. Ce principe indique que nous percevons une structure dans une image lorsqu'un écart majeur du hasard se produit.

Comme mentionner plus haut dans la section 2.4.2 un élément d'une phase est significatif si son taux d'apparition dans la phase présente un écart majeur du hasard en comparaison à son taux d'occurrence dans toute la trace.

Dadachev et al. (2012) proposent une méthode pour calculer le degré d'écart pour extraire les mots clés dans un document à travers le nombre de fausses alertes (en anglais Number of false alarms). Ce nombre représente la probabilité du nombre d'occurrences d'un événement. L'expression pour mesurer la signification d'un événement (NFA) est la suivante :

$$NFA(n,K,P) = \binom{K}{n} * \frac{1}{P^{(n-1)}}$$

Le nombre K est défini comme la somme des occurrences de l'élément dans la trace, n est la somme des occurrences d'une méthode dans chaque phase et P est le nombre de phases total. Cependant les valeurs de K, n et P pourraient être grandes, et rendre le calcule du NFA difficile. Pour cela, nous utilisons cette expression pour mesurer la signification d'un évènement (Khoury et al., 2016) :

$$Significatif(n,K,P) = -\frac{1}{n}\log(NFA(n,K,P))$$

Un ensemble d'événements est considéré significatif si signification(n,K,P) est supérieur à un seuil.

L'algorithme Helmholtz est présenté dans l'algorithme suivante.

Algorithme 1 L'algorithme de Helmholtz (Khoury et al., 2016)

Requérir: les segments de la trace comme divisé dans l'étape 1
1: **pour** toutes les segments de la trace **faire**
2: calculé le nombre d'occurrences de chaque méthode
3: **fin pour**
4: **pour** chaque les segments de la trace **faire**
5: calculer le nombre d'occurrence de chaque méthode dans chaque segment
6: calculer Significatif(n,K,P)
7: **if** Significatif(n,K,P) > seuil **then**
8: ajouter la méthode à la liste des méthodes significatives
9: **fin if**
10: **fin pour**

3.3 ÉLIMINATION D'UTILITAIRES ET MINIMISATION DE RÉSUMÉ

Des auteurs (Hamou-Lhadj et Lethbridge, 2004; Pirzadeh et al., 2009b) définissaient les méthodes utilitaires comme des méthodes appelées à partir de plusieurs méthodes différentes. Cette définition fournit une base algorithmique pour détecter les méthodes utilitaires et les supprimer des résumés. Nous établissons expérimentalement à 20 le seuil des appels de méthodes distincts pour désigner qu'une méthode est utilitaire et la supprimer du résumé. Dans nos expériences ultérieures seul un très petit nombre de méthodes (beaucoup moins de 1%) ont atteint ce seuil, et les résumés finaux contiennent encore de nombreux appels de méthodes utilitaires. Cependant, même ce petit nombre est suffisant pour améliorer la qualité de nos résultats au point où le comportement de chaque segment peut toujours être extrait du résumé du segment.

Pour finir, tout résumé comprenant plus de 10 méthodes est réduit en sélectionnant uniquement les dix premières méthodes (en utilisant le classement créé par les algorithmes TF-IDF et Helmholtz) et en supprimant toutes les autres. Cette étape permet de s'assurer que le résumé a toujours une taille pouvant être traitée manuellement par une analyse humaine. Dans les cas où le résumé comprend moins de 10 méthodes, cette étape est complètement ignorée. Il est simple d'examiner la documentation de 10 méthodes ou moins pour avoir une idée du comportement sous-jacent du segment.

3.4 IMPLÉMENTATION DE L'APPROCHE

Notre outils de segmentation est basé sur un outil existante de Pirzadeh et al. (Pirzadeh et Hamou-Lhadj, 2011a) sur laquelle nous avons ajouter au processus de résumé la sélection des éléments clé des phases avec l'algorithme Helmholtz et nous avons ajouter aussi le troisième

facteur de segmentation (l'utilisation des paramètres et valeurs de retour des méthodes dans le processus de segmentation).

L'outil est implémenté avec le langage java. Nous avons charger en mémoire la trace sous forme d'une structure d'arbre. chaque noeud de l'arbre est un objet qui représente une méthode. un objet(noeud) contient les informations sur la méthode, le nom de la méthode, le niveau d'imbrication , les paramètres et les valeurs. La position de la méthode est représenté dans l'objet sous forme de variable qui contient initialement l'ordre d'appel de la méthode. Après l'application des trois facteurs de rapprochement(par nom de méthode, par niveau d'imbrication et enfin par paramètres et valeurs de retour des méthodes) cette variable contiendra la position finale de la méthode.

Au niveau du processus de segmentation notre contribution et de rapprocher les éléments de la trace selon la similarité dans les paramètres et les valeurs de retour des méthodes. la fonction de rapprochement est lister dans le code suivant.

```
private static double calculateSimilarity_params(Node currentNode)
{
    double finalPosition = 0.0;
    double position, distance;
    double minPosition = currentNode.getVariableOrder();
    ArrayList<String> params = currentNode.getParams();
    for (int i = 0; i < params.size(); i++) {
        if (similarityTable.containsKey(params.get(i)))
        {
            position = (Double)similarityTable.get(params.get(i));
                distance = currentNode.getVariableOrder() - position;
                if (distance < trace.getSize())
```

```
                finalPosition = position + distance/similarityParametresDenominator;
                else finalPosition = currentNode.getVariableOrder();
                if (finalPosition < minPosition){
                        minPosition = finalPosition;
                }
            similarityTable.put(params.get(i), new Double(finalPosition));
        }
        else
        {
                finalPosition = currentNode.getVariableOrder();
                if (finalPosition < minPosition){
                        minPosition = finalPosition;
                }
                similarityTable.put(params.get(i), new Double(finalPosition));
        }
    }
    currentNode.setVariableOrder(minPosition);
    return finalPosition;
}
```

Le code de rapprochement selon les paramètres construit une HashMap qui contient les paramètre des méthodes comme clé et la position de la méthode comme valeur, et à chaque fois qu'il trouve un paramètre dans le HashMap similaire au paramètre de la noeud en cous de traitement il divise la distance entre ces deux paramètres par la valeur de la variable "similarityParametresDenominator". Ce HashMap est est ensuite clusturer avec l'algorithme k-means de weka pour construire les phases d'exécution de la trace.

CHAPITRE 4

EXPÉRIMENTATION

Le précédent chapitre s'est intéressé à présenter notre méthode de segmentation et de résumé des traces d'exécution. Dans ce chapitre, nous allons étudier expérimentalement l'efficacité des algorithmes présentés afin de vérifier que la solution présentée était efficace et précise.

Tout d'abord, nous allons expérimenter quantitativement les algorithmes de segmentation et de résumé par un ensemble de tests. Ensuite, une étude qualitative des résumés des phases obtenues par notre méthode de segmentation.

4.1 SYSTÈMES CIBLES

Nous avons sélectionné quatre systèmes logiciels Java pour ces expérimentations, ArgoUML [1], JHotDraw [2], Weka [3], DrawSWF [4] et JlGUI [5]. Les quatre systèmes sont bien documentés en ligne, ceci nous a permis de valider nos résultats en l'absence des concepteurs d'origine de ces applications. De plus, les systèmes sont accessibles au public ce qui permet la reproduction

1. www.argouml.tigris.org
2. www.jhotdraw.org
3. www.cs.waikato.ac.nz/ml/weka/
4. www.drawswf.sourceforge.net
5. http ://www.javazoom.net/jlgui/jlgui.html

de cette étude. Le choix des applications programmées en java nous a permis de générer facilement des traces qui contiennent toutes les informations nécessaires pour cette étude à l'aide de langage AspectJ (Laddad, 2003) décrit dans la section suivante.

ArgoUML est un logiciel libre d'aide à la conception orientée objet, de création de diagrammes UML et le code source correspondant. C'est un logiciel entièrement écrit en Java. Il prend en charge de tous les diagrammes UML standard.

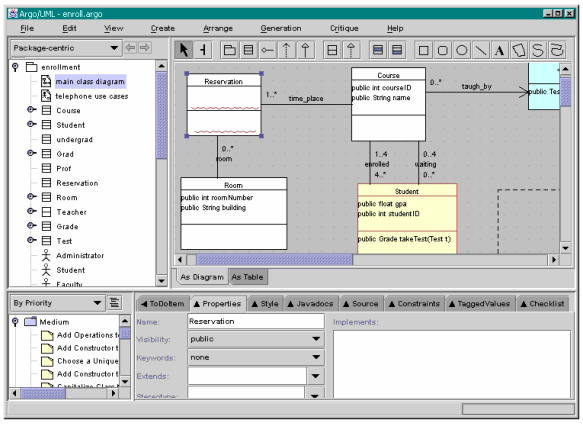

Figure 4.1 – ArgoUML screenshot

JHotDraw est un framework Java avec une interface graphique à deux dimensions assez puissant pour les graphiques techniques et structurés. Il a été développé au début comme un exercice de conception. Sa conception s'appuie fortement sur certains modèles de conception bien connus.

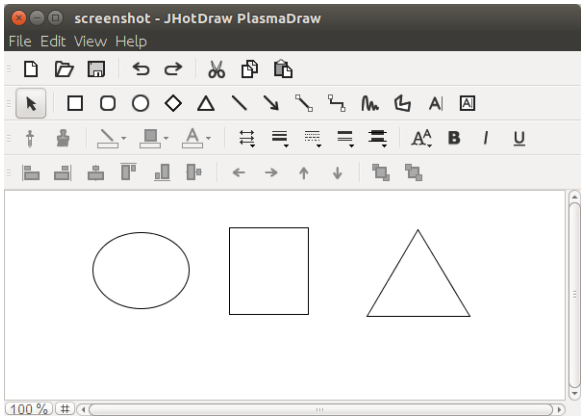

Figure 4.2 – JHotDraw screenshot

WEKA est un logiciel libre d'apprentissage automatique permettant de résoudre des problèmes de fouille de données dans le monde réel. IL est implémenté en java et fonctionne sur presque toutes les plateformes. Il contient des outils pour la préparation des données, la classification, la mise en cluster et la visualisation, etc.

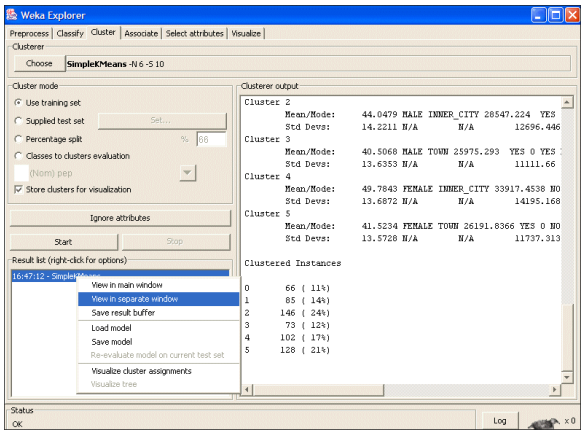

Figure 4.3 – WEKA screenshot

Drawswf est un programme simple de dessin à deux dimensions écrit en Java pour enregistrer les dessins en tant que fichier d'animation flash.

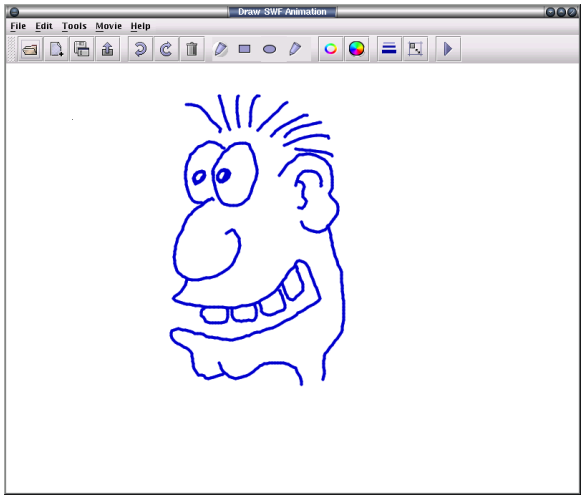

Figure 4.4 – DrawSWF screenshot

jlGui est un lecteur de musique graphique qui prend en charge les fonctionnalités de l'api JavaSound. L'objectif est de fournir un lecteur de musique en temps réel pour la plateforme Java avec prise en charge de la diffusion en continu.

Figure 4.5 – jlGui screenshot

4.2 GÉNÉRER LES TRACES

Au cours de ces expérimentations, nous avons généré des fichiers de traces collectées à partir de l'exécution des quatre systèmes en utilisant le langage de programmation AspectJ. AspectJ est une extension orientée aspect pour le langage de programmation Java qui permet d'intercepter les appels de méthodes au cours de l'exécution. Il nous a permis de collecter et de sauvegarder dans un fichier texte des informations sur chaque appel de méthode intercepté comme le nom de la méthode appelée, son niveau d'imbrication et les paramètres et valeurs de retour de chaque méthode appelée.

Nous avons installé le plugin AspectJ Development Tools (AJDT) dans l'IDE Eclipse. Ensuite, nous avons écrit un aspect qui collecte les informations de chaque appel de méthode et l'écrit dans un fichier de trace.

4.3 ÉTUDE QUANTITATIVE

Pour démontrer l'efficacité de notre approche de segmentation par rapport à celle de Pirzadeh et al. (2011) qui utilise seulement deux facteurs (noms de méthode et imbrication) nous utilisons AspectJ (Laddad, 2003) pour générer quatre fichiers de traces à partir de l'exécution de 4 systèmes différents : argoUML, JHotDraw, Weka et DrawSWF.

Nous avons généré deux traces pour chacun de ces programmes contenant respectivement 3 (3-phases) et 4 (4-phases) actions utilisateur distinctes. Chaque ligne consiste en un appel de méthode ou un retour de méthode et indique le nom de la méthode, le type et la valeur des paramètres de la méthode, son niveau d'imbrication, ainsi que le type et la valeur renvoyés par la méthode. Les valeurs des chaînes de caractères et les types élémentaire sont fournies explicitement, mais celles des objets sont fournies par des références.

Par exemple, nous avons généré à partir d'ArgoUML une trace contenant 3 actions d'utilisateur en appliquant le scénario suivant : Démarrage d'ArgoUML, création d'un diagramme de classes et fermeture d'ArgoUML. La trace résultante contient 71 035 appels de méthode.

```
org.argouml.util.JavaRuntimeUtility&isJreSupported&4&boolean&true&Rien&Rien
org.argouml.application.Main&checkJVMVersion&3&void&null&Rien&Rien
org.argouml.notation.Notation&findNotation&2&interface org.argouml.notation.NotationName&UML 1.4&class java.lang.String&UML 1.4
org.argouml.uml.diagram.static_structure.ui.UMLClassDiagram&getLabelName&1&class java.lang.String&Class Diagram&Rien&Rien
org.argouml.i18n.Translator&getName&2&class java.lang.String&menu&class java.lang.String&menu.popup.delete
org.argouml.i18n.Translator&loadBundle&3&void&null&class java.lang.String&menu
org.argouml.i18n.Translator&localize&4&class java.lang.String&Delete&class java.lang.String&menu.popup.delete
org.argouml.uml.ui.UMLComboBoxModel2&getElementAt&4&class java.lang.Object&null&int&0
org.argouml.ui.ProjectBrowser&tryExit&3&boolean&true&Rien&Rien
org.argouml.ui.ProjectBrowser&trySave&4&void&null&boolean*boolean*boolean&false*false*true
org.argouml.i18n.Translator&getName&3&class java.lang.String&optionpane&class java.lang.String&optionpane.exit-save-changes-to
```

Figure 4.6 – Exemple de lignes d'une trace

Chaque ligne dans la trace contient 7 informations séparées par le caractère "&".

1. Le nom de la classe.
2. Le nom de la méthode.
3. Le niveau d'imbrication de la méthode.
4. Les types de valeur de retour séparés par le caractère "*".
5. Les valeurs de retour séparés par le caractère "*".
6. Les types de paramètres séparés par le caractère "*".
7. Les valeurs de paramètres séparés par le caractère "*".

L'expérience comportait cinque étapes.

1. Tout d'abord, nous avons effectué une segmentation manuelle de la trace en un ensemble de segments représentant les actions effectuées par l'utilisateur. Cela était possible car nous avions généré les traces nous-mêmes et nous pouvions ainsi rechercher des appels de méthode spécifiques dans la trace supposés coïncider avec le début et la fin des actions d'utilisateur que nous avions effectués. Ces segments servent de

référence pour évaluer l'efficacité de l'algorithme de segmentation automatique. Nous considérons qu'un algorithme de segmentation A est plus précis qu'un autre algorithme de segmentation B si les segments identifiés par A sont plus similaires à cette référence que ceux identifiés par B.

2. Deuxièmement, nous avons généré des segments en utilisant l'algorithme de Pirzadeh et al. (2011) sur les 8 traces.

3. De même, nous avons généré des segments en utilisant notre algorithme de segmentation sur les 8 traces.

4. Comme nous l'avons vu dans le chapitre précédent, notre approche de segmentation rapproche les éléments de la trace pour former des groupes de méthodes. En pratique, pour rapprocher les éléments l'algorithme de segmentation divise la distance entre deux éléments de la trace par un nombre que l'on appelle le dénominateur. Ainsi, l'algorithme divise la distance entre deux méthodes qui ont un nom similaire par un dénominateur et la distance entre deux méthodes imbriquées (une méthode qui fait appel a une autre méthode) par un autre dénominateur, et enfin la distance entre deux méthodes qui ont les mêmes valeurs de retour ou les mêmes paramètres par un troisième dénominateur. Par exemple si une méthode située à la première position dans la trace et une autre méthode qui a le même nom situé à la cinquième position dans la même trace le rapprochement sera par la division de la distance entre les deux méthodes, qui est quatre dans ce cas, par la valeur choisie de dénominateur. Nous avons expérimenté dans cette étape différentes valeurs de ce dénominateur.

5. Enfin, pour chacun des segments identifiés par les deux algorithmes de segmentation, nous avons calculé la différence symétrique entre ce segment et les segments correspondants identifiés lors de la première étape. Dans ce qui suit, nous appelons *l'exactitude* de segment le nombre d'éléments résultat du calcul de la différence symétrique. Autant

que ce nombre soit petit, le segment est similaire au segment de référence.

Les tableaux 4.1 et 4.2 montrent les résultats du calcul de la différence symétrique pour les scénarios 3-phases et 4-phases respectivement avec plusieurs valeurs de dénominateur différentes. Les valeurs des dénominateurs sont indiquées dans la colonne à gauche par «i» identifiant la valeur du dénominateur pour les noms de méthodes, «n» le dénominateur pour le niveau d'imbrication et le «p» dénominateur pour les paramètres.

Tableau 4.1 – Scénario 3 phases.

	ArgoUML		jhotDraw		Weka		DrawSWF	
	3-f	2-f	3-f	2-f	3-f	2-f	3-f	2-f
$i = 2$	**22 751**	22 869	38 432	30 321	37 250	2	**382**	2018
$n = 2$	**36 980**	37 181	**31 418**	33 098	41 728	10 036	**2 623**	4 964
$p = 1.5$	**14 314**	14 314	**18 155**	24 586	**4 588**	10 034	**2 243**	2 948
$i = 2$	**22 471**	22 869	38 643	30 321	37 324	2	**1 453**	2 018
$n = 1.5$	**36 769**	37 181	**31 469**	33 098	41 837	10 036	**4 221**	4 964
$p = 2$	**14 300**	14 314	**17 866**	24 586	**4 633**	10 034	**2 770**	2 948
$i = 1.5$	**22 471**	22 869	38 449	30 321	29 388	2	**962**	2018
$n = 2$	**24 129**	37 181	**30 743**	33 098	34 012	10 036	**3 411**	4 964
$p = 2$	**12 077**	14 314	**18 813**	24 586	**4 740**	10 034	**2 481**	2 948
$i = 2$	**22 290**	22 869	38 451	30 321	39	2	**523**	2 018
$n = 3$	**36 202**	37 181	**31 751**	33 098	**4 713**	10 036	**2 679**	4 964
$p = 2$	**13 914**	14 314	**17 803**	24 586	**7 376**	10 034	**2 158**	2 948
$i = 2$	**22 217**	22 869	36 216	30 321	110	2	**760**	2 018
$n = 2$	**36 195**	37 181	33 938	33 098	**9 664**	10 036	**2 560**	4 964
$p = 3$	**13 984**	14 314	**17 851**	24 586	**9 664**	10 034	**1 802**	2 948
$i = 3$	**22 290**	22 869	38 451	30 321	39	2	**523**	2 018
$n = 3$	**36 202**	37 181	**31 751**	33 098	**7 413**	10 036	**2 679**	4 964
$p = 2$	**13 914**	14 314	**17 803**	24 586	**7 376**	10 034	**2 158**	2 948
$i = 2.5$	**22 290**	22 869	38 451	30 321	39	2	**523**	2 018
$n = 3$	**36 202**	37 181	**31 751**	33 098	**7 413**	10 036	**2 679**	4 964
$p = 2$	**13 914**	14 314	**17 803**	24 586	**7 376**	10 034	**2 158**	2 948

Les segments dans lesquels notre approche donne des résultats plus performants que ceux de Pirzadeh et al. (2011) apparaissent en rouge et en gras. Comme nous pouvons le constater, à partir de ces tableaux, notre approche expose des résultats sensiblement meilleurs dans

Tableau 4.2 – Scénario 4 phases.

	ArgoUML		jhotDraw		Weka		DrawSWF	
	3-f	2-f	3-f	2-f	3-f	2-f	3-f	2-f
$i = 2$	**27 354**	27 589	17 217	7 940	30 351	3	**460**	2 018
$n = 3$	**44 698**	45 112	**54 833**	55 429	124 083	76 483	**7 415**	7 999
$p = 2$	**38 215**	38 729	**54 518**	60 549	136 240	143 472	**8 676**	9 507
	15 553	16 654	**31 904**	34 128	**50 772**	66 990	**8 733**	8 778
$i = 3$	**27 105**	27 589	23 942	7 940	104 098	3	**559**	2 018
$n = 3$	**44 601**	45 112	66 034	55 429	**52 886**	76 483	**7 093**	7 999
$p = 3$	**37 605**	38 729	62 457	60 549	175 486	143 472	**6 803**	9 507
	15 419	16 654	**26 831**	34 128	**29 493**	66 990	10 669	8 778
$i = 4$	**26 489**	27 589	18 622	7 940	102 352	3	**1 090**	2 018
$n = 4$	45 158	45 112	61 170	55 429	111 779	76 483	**7 465**	7 999
$p = 4$	**35 599**	38 729	**49 523**	60 549	**118 339**	143 472	**7 615**	9 507
	14 266	16 654	**29 205**	34 128	**29 493**	66 990	9 942	8 778
$i = 1.5$	**21 474**	27 589	13 316	7 940	115 548	3	**776**	2 018
$n = 1.5$	**33 262**	45 112	**54 741**	55 429	**40 746**	76 483	8 568	7 999
$p = 1.5$	**21 556**	38 729	60 594	60 549	176 176	143 472	**8 975**	9 507
	9 784	16 654	**29 431**	34 128	**29 493**	66 990	**7 737**	8 778
$i = 1.1$	29 324	27 589	12 520	7 940	166	3	**513**	2 018
$n = 1.1$	**36 725**	45 112	57 835	55 429	**52 325**	76 483	**8 681**	7 999
$p = 1.1$	48 071	38 729	**59 343**	60 549	**81 654**	143 472	**7 786**	9 507
	18 006	16 654	**28 352**	34 128	**29 493**	66 990	**8 580**	8 778

presque tous les cas. En outre, bien que le choix du dénominateur ait un impact sur l'exactitude des segments, il n'existe aucune valeur pour laquelle l'approche proposée par Pirzadeh et al. (2011) est plus performant que la nôtre sur tous les segments. Expérimentalement, nous avons constaté que les valeurs de i = 2, n = 3 et p = 2 semblent offrir des résultats optimaux.

4.3.1 DEUXIÈME EXPÉRIMENTATION

Nous avons effectué une deuxième expérience afin d'évaluer l'efficacité de l'utilisation de notre approche de segmentation de trace à 3 facteurs lorsqu'elle est appliquée à une tâche de compréhension de trace. Cette expérience se déroule en deux étapes : premièrement, un

«résumé» de chaque segment est généré, en utilisant un algorithme approprié. Deuxièmement, en n'examinant que les résumés, nous avons tenté d'identifier le comportement du programme au niveau le plus élevé dans chaque segment. Une expérience similaire a été réalisée dans la recherche de Pirzadeh et al. (2011). Nous utilisons dans cette expérimentation les deux algorithmes TF-IDF et Helmholtz pour réaliser le résumé. Ces algorithmes peuvent être vus comme des algorithmes d'abstraction et leur utilisation est souvent essentielle pour gérer la taille considérable des traces d'exécution.

Les traces utilisées dans cette expérience sont générées à partir d'ArgoUML (version 0.35.2) en utilisant AspectJ de la manière décrite à la section 4.2. Nous avons généré 40 traces dont chacune contient entre 3 et 4 phases, chaque phase consistant à dessiner un des 6 diagrammes possibles : un diagramme de classes, un diagramme de séquence, un diagramme de collaboration, un diagramme d'activité, un diagramme de cas d'utilisation et un diagramme de déploiement. De plus, chaque trace commence par une phase de démarrage et se termine par une phase d'arrêt, pour un total de 8 types de segments (les six diagrammes, en plus des phases de démarrage et d'arrêt).

Les traces sont ensuite segmentées à l'aide de la méthode de segmentation proposée dans cette recherche, ainsi que de la méthode proposée par Pirzadeh et al. (2011). Au total, les 40 traces ont été divisées en 140 segments.

- 40 segments type segment de démarrage
- 40 segments type segment d'arrêt
- 11 segments type diagramme de séquence
- 9 segments type diagramme de cas d'utilisation
- 12 segments type diagramme de classes
- 9 segments type diagramme de déploiement

— 9 segments type diagramme de collaboration

— 10 segments type diagramme d'activité

Dans le cadre de cette expérience, nous avons généré les événements clés pour chaque segment en utilisant à la fois les algorithmes TF-IDF et Helmholtz. Une seule occurrence de chaque type de segment a été choisie au hasard et utilisée comme référence pour construire une bibliothèque de 8 événements clés. Nous avons ensuite tenté d'associer chacun des 132 ensembles d'événements clés restants à sa correspondante de la bibliothèque. Soit K_a l'ensemble des événements clés d'un segment A. Nous calculons la similarité entre le segment A et le segment B de la bibliothèque comme suit :

$$similarit_{a-b} = (K_a \cap K_b) \backslash (K_a - (K_a \cap K_b))$$

Puis, nous avons calculé la similarité de chacun des 132 segments avec chacun des 8 segments dans la bibliothèque. Notons qu'un segment est du type pour lequel l'évaluation de la fonction de similarité donne le nombre le plus élevé d'éléments similaires. Nous utilisons les résultats renvoyés par la fonction de similarité pour mesurer la confiance de nos assignations. Ces résultats sont présentés dans le tableau 4.3.

Tableau 4.3 – Confiance pour les assignations correctes et incorrectes

	correct	Confiance pour correct assignations	incorrect	Confiance pour incorrect assignations
TF-IDF	100	84%	40	39%
Helmholtz	105	87%	35	46%

Comme le montre le tableau, les résultats générés par l'algorithme de Helmholtz sont correctement appariés 75% du temps (105 sur 132), tandis que les résultats générés par l'algorithme

TF-IDF sont correctement appariés dans 71,42% du temps (100 sur 132). Le tableau 4.4 montre le nombre d'identifications correctes et incorrectes pour les traces segmentées selon notre approche et celle proposée dans l'article de Pirzadeh et al. (2011).

Tableau 4.4 – Nombre d'identifications correctes et incorrectes selon notre approche et celle proposée dans l'article de Pirzadeh et al. (2011)

	Correct (TF-IDF)		Incorrect (TF-IDF)		Correct (Helmholtz)		Incorrect (Helmholtz)	
	3-f	2-f	3-f	2-f	3-f	2-f	3-f	2-f
Démarrage	40	27	0	1	40	23	1	1
Arrêt	37	40	2	2	39	37	40	1
Diagramme de séquence	6	5	12	29	5	4	2	20
Diagramme de cas d'utilisation	3	1	0	1	3	1	1	1
Diagramme de classes	4	3	1	1	7	2	5	3
Diagramme de déploiement	2	2	5	0	3	3	12	13
Diagramme de collaboration	6	5	17	19	6	6	10	16
Diagramme d'activité	2	1	13	3	2	2	7	7

Les deux méthodes sont efficaces pour identifier les phases de démarrage et d'arrêt. Cela n'est pas surprenant puisque ces phases sont les plus distinctes. En particulier, notons qu'avec l'identification des phases segmentée par notre méthode (3 facteurs), les phases de démarrage sont toujours correctement identifiées.

L'algorithme a eu des difficultés à différencier les types de segments de diagramme de séquence, de déploiement et de diagramme de collaboration. Cependant, même dans ces cas, l'identification incorrecte est plus faible avec une identification à trois valeurs qu'avec l'approche de Pirzadeh et al. (2011). Une explication possible de la non efficacité de l'approche dans ces cas est qu'ArgoUML utilise les même méthodes pour les trois types de diagrammes.

Une inspection manuelle du code source confirme cette explication.

Les tableaux 4.5 et 4.6 montrent les résultats de la précision, de rappel et de l'erreur systématique (en anglais accuracy) pour les algorithmes Helmholtz et TF-IDF. Comme on peut le voir dans ces deux tableaux, la segmentation des traces volumineuses par la méthode à 3 facteurs donne généralement de meilleurs résultats que la méthode de Pirzadeh et al. (2011). De plus, ceci est le cas quelle que soit la méthode utilisée pour résumer la segment (TF-IDF ou Helmholtz).

Tableau 4.5 – Précision, Rappel et Accuracy (Helmholtz)

	Precision (Helmholtz)		Recall (Helmholtz)		Accuracy (Helmholtz)	
	3-f %	2-f %	3-f %	2-f %	3-f %	2-f %
Démarrage	97.50	95.65	100.0	56.41	99.23	86.25
Arrêt	90.47	97.29	97.43	92.30	96.18	96.94
Diagramme de séquence	66.66	13.04	40.0	30.000	93.89	79.38
Diagramme de cas d'utilisation	66.66	0.0	25.0	0.0	94.65	93.12
Diagramme de classes	54.54	25.0	54.54	9.09	92.36	90.07
Diagramme de déploiement	14.28	13.33	25.0	25.0	86.25	85.49
Diagramme de collaboration	33.33	23.80	62.5	62.5	90.07	85.49
Diagramme d'activité	100.0	12.5	11.11	11.11	93.89	88.54

De plus, lorsqu'on résume les segments en utilisant l'algorithme Helmholtz notre méthode de segmentation a donné une moyenne de précision de 56%, une moyenne de rappel de 51% et une moyenne de l'accuracy de 92%. Tandis que la méthode de segmentation à 2 facteurs a donné une moyenne de précision de 34%, une moyenne de rappel de 35%, et une moyenne de l'accuracy de 87%. Cela montre l'efficacité de notre méthode de segmentation.

Tableau 4.6 – Précision, Rapell and Accuracy(TF-IDF)

	Precision (TF-IDF)		Recall (TF-IDF)		Accuracy (TF-IDF)	
	3-f %	2-f %	3-f %	2-f %	3-f %	2-f %
Démarrage	100.0	96.29	100.0	66.66	100.0	89.31
Arrêt	94.73	95.12	92.30	100.0	96.18	98.47
Diagramme de séquence	29.41	12.12	50.0	40.0	87.02	73.28
Diagramme de cas d'utilisation	100.0	0.0	25.0	0.0	95.41	93.12
Diagramme de classes	75.0	66.66	27.27	18.18	93.12	92.36
Diagramme de déploiement	16.66	100.0	12.5	12.5	90.83	94.65
Diagramme de collaboration	22.72	17.39	62.5	50.0	84.73	82.44
Diagramme d'activité	25.0	0.0	11.11	0.0	91.60	90.83

De plus, lorsqu'on résume les segments en utilisant l'algorithme TF-IDF notre méthode de

segmentation a donné une moyenne de précision de 57%, une moyenne de rappel de 47% et une moyenne de l'accuracy de 92%. Tandis que la méthode de segmentation à 2 facteurs a donné une moyenne de précision de 48%, une moyenne de rappel de 35%, et une moyenne de l'accuracy de 88%. Cela montre l'efficacité de notre méthode de segmentation.

4.4 ÉTUDE QUALITATIVE

Nous avons testé l'efficacité de notre approche en générant des traces à partir de programmes Java disponibles. Nous nous référons au code source de ces programmes et à leur documentation pour analyser les méthodes de chaque phase de cette trace. Cette analyse nous permettra de savoir si les méthodes de chaque phase représentent réellement les actions effectuées par l'utilisateur.

Nous avons effectué plusieurs tests avec une grande variété de programmes, mais nous n'avons sélectionné que trois expériences représentatives parmi eux. Les programmes que nous avons sélectionnés sont JHotDraw, ArgoUML et JlGUI. Nous avons utilisé AspectJ (Laddad, 2003) pour générer des traces d'exécution de ces programmes, contenant respectivement 3 ou 4 actions utilisateur distinctes.

Chaque ligne de la trace consiste en un appel de méthode ou un retour de méthode et indique le nom de la méthode, le type et la valeur de chacun de ses paramètres, son niveau d'imbrication, ainsi que le type et la valeur renvoyés par la méthode. Les types primitifs et les types chaîne de caractères sont fournis explicitement, mais ceux des objets sont fournis par des références. La longueur des traces varie de 17 000 lignes à 143 298 lignes.

Nous avons segmenté chaque trace en utilisant l'approche à 2 facteurs de Pirzadeh ainsi que l'approche à 3 facteurs présenté au chapitre précédent. Nous avons ensuite généré des résumés

de chaque segment en utilisant à la fois TF-IDF et Helmholtz. Ainsi, pour chaque segment nous avons généré 4 résumés.

Nous avons procédé à l'élimination des méthodes utilitaires de la manière décrite dans la section 3.3 et chaque fois qu'un résumé comptait 10 méthodes nous avons considéré uniquement les 10 méthodes les mieux classées. De cette manière nous utilisons toujours des résumés d'une taille exploitable par un utilisateur humain.

Comme on pourra le voir dans cette section l'approche à trois facteurs génère toujours un résumé à partir duquel le comportement du segment sous-jacent peut être déterminé avec précision, en examinant simplement les méthodes présentées dans le résumé et en consultant la documentation du programme. L'approche à trois facteurs ignore toute méthode utilitaire ou insuffisamment spécifique au comportement du programme lors de la génération des résumés. Nous nous référons au méthodes dont la présence dans un résumé contribue à la compréhension du comportement sous-jacent du programme en tant que méthode *spécifique au comportement*.

Comme mentionné ci-dessus, chaque résumé de segment contient plusieurs appels de méthodes utilitaires. En fait, les appels de méthodes utilitaires sont souvent plus nombreux que les appels de méthodes spécifiques au comportement. Cependant, cela ne constitue pas un obstacle au processus de compréhension des traces. La raison est que les appels de méthodes utilitaires peuvent simplement être ignorés.

Notons que tant que tous les appels de méthodes spécifiques au comportement sont présents dans un résumé, le comportement sous-jacent de la trace peut être facilement découvert. L'analyse échoue uniquement dans deux cas : lorsque chaque méthode du résumé est un utilitaire, ou si le résumé contient par erreur des méthodes spécifiques au comportement non-liées à son comportement réel (c'est-à-dire des méthodes liées au comportement présent dans un autre segment de trace, généralement adjacent). Nous appelons ces méthodes des

méthodes mal attribuées. Comme nous le montrerons dans cette section, ces cas ne se sont produits dans notre expérience que lorsque nous avons utilisé l'approche à deux facteurs proposée par Pirzadeh et al.

Nous avons également constaté que l'approche à 3 facteurs est plus performante par rapport à l'approche à 2 facteurs de Pirzadeh selon les deux critères suivants : la plupart des 10 méthodes présentes dans les résumés sont spécifiques au comportement plutôt que des utilitaires, et de plus, dans nos ensembles de tests, l'approche à 3 facteurs n'a jamais mal attribué une méthode spécifique au comportement dans le segment incorrect. Ceci arrive souvent en utilisant l'approche à 2 facteurs.

Enfin, nos résultats indiquent que par les mêmes mesures, l'algorithme de résumé de Helmholtz est plus performant que l'algorithme TF-IDF, quelle que soit la méthode utilisée pour effectuer la segmentation de trace (à 3 facteurs ou à 2 facteurs). Le reste de cette section détaille nos résultats.

4.4.1 JHOTDRAW

Nous avons généré une trace de JHotDraw à l'aide d'un scénario en 4 phases (lancement de JHotDraw, tracé d'une ligne, tracé d'un rectangle et fermeture de JHotDraw).

Phase 1 La première phase consiste à initialiser l'interface graphique de JHotDraw (menus, barre d'outils, barre d'état et le panneau principal). La Figure 4.7 présente une liste détaillée des méthodes présentées dans chaque résumé. Elle liste chaque méthode qui apparaît dans au moins un résumé et indique à droite les résumés qui le contiennent. Les méthodes spécifiques au comportement sont mises en gras et en rouge.

Plusieurs méthodes traitent de la création de l'interface graphique et leur présence dans le

Méthodes	3-facteurs Helmholtz	3-facteurs TF-IDF	2-facteurs Helmholtz	2-facteurs TF-IDF
org.jhotdraw.draw.action.ToolBarButtonFactory$7.compare	x	x		x
org.jhotdraw.draw.action.AbstractSelectedAction.setEditor	x	x		x
org.jhotdraw.draw.action.JPopupButton.add	x	x		x
org.jhotdraw.draw.action.JPopupButton.getColumnCount	x	x		x
org.jhotdraw.draw.action.PaletteMenuItemUI.installDefaults	x	x		
org.jhotdraw.draw.action.Colors.shadow	x	x		
org.jhotdraw.util.ResourceBundleUtil.configureAction	x	x		
org.jhotdraw.draw.action.JPopupButton.updateFont	x			
org.jhotdraw.util.ResourceBundleUtil.getLAFBundle	x			x
org.jhotdraw.util.ResourceBundleUtil.configureToolBarButton	x			
org.jhotdraw.draw.action.AbstractSelectedAction$EventHandler.propertyChange		x	x	x
org.jhotdraw.draw.DefaultDrawingView.getSelectionCount		x		
org.jhotdraw.draw.action.AbstractSelectedAction.access$0		x		x
org.jhotdraw.app.action.AbstractApplicationAction.updateApplicationEnabled			x	
org.jhotdraw.app.action.RedoAction.updateProject			x	
org.jhotdraw.util.ResourceBundleUtil.getImageIcon			x	
org.jhotdraw.draw.action.ToolBarButtonFactory.addToolTo			x	
org.jhotdraw.app.action.UndoAction.updateProject			x	
org.jhotdraw.draw.ArrowTip.getDecoratorPath			x	
org.jhotdraw.geom.BezierPath$Node.getControlPoint			x	
org.jhotdraw.app.action.AbstractApplicationAction.createApplicationListener			x	
org.jhotdraw.beans.AbstractBean.addPropertyChangeListener				x
org.jhotdraw.draw.DefaultDrawingView.addFigureSelectionListener				x
org.jhotdraw.draw.action.JPopupButton.getPopupMenu				x

Figure 4.7 – Liste détaillée des résumés de la phase 1 de JHotDraw

résumé rend immédiatement facile a comprendre le comportement sous-jacent de la trace. Ceci inclut les méthodes suivantes :

— org.jhotdraw.draw.action.JPopupButton.add

— org.jhotdraw.draw.action.PaletteMenuItemUI.installDefaults

— org.jhotdraw.draw.action.JPopupButton.add

— org.jhotdraw.draw.action.JPopupButton.updateFont

En utilisant la méthode à 3 facteurs, les 3 premières méthodes apparaissent dans les deux résumés du premier segment et les 2 dernières dans le résumé de Helmholtz uniquement. Toute autre méthode présente dans le résumé est un utilitaire et peut être exclue en tant que telle par un analyste. Par conséquent, le comportement sous-jacent du segment est immédiatement déduit après une lecture rapide de la documentation.

Un nombre considérable de ces méthodes semble partager des valeurs de paramètres communes, ainsi que d'autres appels de méthodes liés à l'initialisation de l'interface graphique. Il en résulte que lors de la segmentation à l'aide de la méthode à 3 facteurs ces méthodes forment un cluster étroit et apparaissent correctement dans les résumés des algorithmes TF-IDF et Helmholtz de la phase 1. Cependant, il est intéressant de noter que la méthode *.installDefaults mentionnée ci-dessus ne partage pas un paramètre ou une valeur de retour avec d'autres méthodes de la phase 1. Les avantages d'inclure les paramètres dans le processus de segmentation s'étendent donc à d'autres méthodes car l'attraction gravitationnelle entre deux méthodes aura pour effet de rapprocher toutes les méthodes intermédiaires.

Toutefois, lorsque la segmentation est effectuée à l'aide de l'approche à 2 facteurs, aucune des dix méthodes présentes dans le résumé de Helmholtz n'est liée au comportement sous-jacent de la trace (démarrage et initialisation de l'interface graphique), tandis que le résumé avec l'algorithme TF-IDF ne contient que 2 méthodes liées au comportement sous-jacent. Le résumé généré avec l'algorithme Helmholtz de la méthode à 2 facteurs contient également 2 méthodes mal-assignées, à savoir les méthodes utilisées par JHotDraw pour dessiner une ligne à l'écran. Ces méthodes devraient en fait figurer dans le résumé de la phase suivante.

Phase 2 La deuxième phase consiste à tracer une seule ligne. La plupart des méthodes significatives sont présentes dans les classes : org.jhotdraw.geom
.BezierPath et org.jhotdraw.draw.BezierFigure. Dans ce cas, les résumés générés à l'aide de l'algorithme Helmholtz étaient particulièrement informatifs et contenaient les méthodes spécifiques au comportement suivant (et plusieurs autres de la même classe) :

— org.jhotdraw.draw.BezierFigure.basicSetEndPoint
— org.jhotdraw.draw.BezierFigure.basicSetStartPoint
— org.jhotdraw.draw.BezierFigure.basicSetBounds

— org.jhotdraw.draw.BezierFigure.basicSetPoint

Plusieurs de ces méthodes étaient absentes des résumés de l'algorithme TF-IDF. Ceci montre l'efficacité de l'algorithme Helmholtz par rapport à l'algorithme TF-IDF. Au total, 8 des 10 méthodes présentes dans le résumé par l'algorithme Helmholtz de la méthode à 3 facteurs sont spécifiques au comportement, alors que 4 des 10 méthodes du résumé par l'algorithme Helmholtz de la méthode à 2 facteurs sont spécifiques au comportement. De manière unique, dans toutes les expériences effectuées, le résumé par TF-IDF de la méthode à 2 facteurs est meilleur que le résumé TF-IDF de la méthode à 3 facteurs. La Figure 4.8 présente une liste détaillée des méthodes présentées dans chaque résumé de cette phase.

Methodes	3-facteurs Helmholtz	3-facteurs TF-IDF	2-facteurs Helmholtz	2-facteurs TF-IDF
org.jhotdraw.util.prefs.ToolBarPrefsHandler.ancestorRemoved	x	x		
org.jhotdraw.xml.DefaultDOMFactory.getName	x	x		
org.jhotdraw.geom.BezierPath.clone	x			x
org.jhotdraw.geom.BezierPath.getPathIterator	x		x	
org.jhotdraw.geom.BezierPath$Node.clone	x			
org.jhotdraw.draw.BezierFigure.basicSetEndPoint	x			x
org.jhotdraw.draw.BezierFigure.basicSetStartPoint	x		x	x
org.jhotdraw.draw.BezierFigure.basicSetBounds	x			x
org.jhotdraw.draw.BezierFigure.basicSetPoint	x		x	x
org.jhotdraw.draw.BezierFigure.drawCaps	x		x	
org.jhotdraw.draw.action.AbstractSelectedAction.access$0		x		
org.jhotdraw.draw.action.DefaultAttributeAction.access$0		x		
org.jhotdraw.draw.DefaultDrawingView.getDOMFactory		x		
org.jhotdraw.beans.AbstractBean.removePropertyChangeListener		x		
org.jhotdraw.draw.BezierNodeHandle.basicGetBounds		x		
org.jhotdraw.app.DefaultApplicationModel.getAction		x		
org.jhotdraw.app.AbstractApplication.getModel		x		
nanoxml.XMLElement.setAttribute		x		
org.jhotdraw.draw.BezierFigure.invalidate			x	
org.jhotdraw.draw.BezierFigure.invalidateCappedPath			x	
org.jhotdraw.draw.BezierFigure.drawFill			x	
org.jhotdraw.geom.BezierPath.invalidatePath			x	
org.jhotdraw.draw.BezierFigure.getPointCount			x	
org.jhotdraw.draw.DefaultDrawing.figureAreaInvalidated			x	
org.jhotdraw.app.action.AbstractProjectAction$2.propertyChange				x
org.jhotdraw.undo.UndoRedoManager.firePropertyChange				x
org.jhotdraw.undo.UndoRedoManager.setHasSignificantEdits				x
org.jhotdraw.samples.draw.DrawProject.access$0				x
org.jhotdraw.undo.UndoRedoManager.hasSignificantEdits				x

Figure 4.8 – Liste détaillée des résumés de la phase 2 de JHotDraw

Phase 3 Les résultats de la phase 3 sont semblables à ceux de la phase 2. La dite phase concerne le dessin d'un rectangle. Une fois encore, le résumé de la phase généré à l'aide de l'algorithme Helmholtz de la méthode à 3 facteurs fournit un résumé à partir duquel nous pouvons déduire le comportement du programme. Dans ce cas, le résumé contient les méthodes suivantes qui sont très spécifiques à la tâche à accomplir :

— org.jhotdraw.draw.RectangleFigure.basicSetBounds
— org.jhotdraw.draw.RectangleFigure.getFigureDrawBounds
— org.jhotdraw.draw.RectangleFigure.drawFill
— org.jhotdraw.draw.RectangleFigure.drawStroke
— org.jhotdraw.draw.AbstractHandle.drawRectangle

La plupart de ces méthodes ne sont présentes que dans le résumé généré à l'aide de l'algorithme Helmholtz et uniquement lorsque la trace est segmentée à l'aide de l'approche à 3 facteurs. Plus particulièrement, chacun des deux résumés générés par la méthode à 2 facteurs ne contient qu'une seule méthode spécifique au comportement. En conséquence, le comportement du programme devient difficile à comprendre. Au total, 5 des 10 méthodes présentes dans le résumé Helmholtz à l'aide de l'approche à 3 facteurs ainsi que 3 des 10 méthodes présentes dans le résumé TF-IDF à l'aide de l'approche à 2 facteurs sont spécifiques au comportement et sont liées au comportement du segment sous-jacent. En revanche, les résumés Helmholtz à 2 facteurs et TF-IDF contiennent une seule méthode spécifique au comportement. Ils contiennent également des méthodes qui semblent appartenir au segment précédent, telle que la méthode org.jhotdraw.draw.BezierFigure.contains et org.jhotdraw.geom.BezierPath.outlineContains.

En examinant la documentation du programme il est facile de comprendre pourquoi les résultats de l'approche à 3 facteurs sont meilleurs que ceux de l'approche à 2 facteurs. En effet, un bon nombre des méthodes utilisées pour dessiner une figure manipulent les coordonnées de

la figure. La présence de ces valeurs en tant que paramètres et valeurs de retour produit un regroupement entre les méthodes manipulant le même objet. La Figure 4.9 présente une liste détaillée des méthodes présentées dans chaque résumé de cette phase.

Methodes	3-facteurs Helmholtz	3-facteurs TF-IDF	2-facteurs Helmholtz	2-facteurs TF-IDF
org.jhotdraw.draw.action.AbstractSelectedAction$EventHandler.selectionChanged	x	x		
org.jhotdraw.draw.lectangleFigure.basicSetBounds	x	x		
org.jhotdraw.draw.FigureLayerComparator.compare	x	x		
org.jhotdraw.draw.RectangleFigure.getFigureDrawBounds	x	x	x	x
org.jhotdraw.draw.AttributeKeys.$SWITCH_TABLE orgjhotdraw$draw$AttributeKeys $Underfill	x	x		
org.jhotdraw.draw.BezierNodeHandle.getBezierFigure	x		x	
org.jhotdraw.draw.RectangleFigure.drawFill	x			
org.jhotdraw.draw.AttributeKeys.getPerpendicularFillGrowth	x			
org.jhotdraw.draw.RectangleFigure.drawStroke	x	x		
org.jhotdraw.draw.AbstractHandle.drawRectangle	x			
org.jhotdraw.draw.DefaultDrawingView.getSelectionCount		x		
org.jhotdraw.draw.AttributeKeys.$SWITCH_TABLE$org$jhotdraw$draw$AttributeKeys $StrokePlacement		x		x
org.jhotdraw.draw.AttributeKeys.getPerpendicularDrawGrowth		x		x
org.jhotdraw.draw.AbstractHandle.contains		x		x
org.jhotdraw.draw.SelectAreaTracker.clearRubberBand			x	
org.jhotdraw.draw.AbstractFigure.removeFigureListener			x	
org.jhotdraw.geom.Geom.lineContainsPoint			x	x
org.jhotdraw.draw.BezierNodeHandle.getLocation			x	
org.jhotdraw.draw.DefaultDrawing.findFigure			x	
org.jhotdraw.draw.AbstractHandle.getOwner			x	
org.jhotdraw.draw.DefaultDrawingView.getContainer			x	
org.jhotdraw.draw.DefaultDrawing.getFiguresFrontToBack			x	
org.jhotdraw.draw.AbstractHandle.getBounds				x
org.jhotdraw.draw.AttributeKeys.getPerpendicularHitGrowth				x
org.jhotdraw.draw.AbstractFigure.getLayer				x
org.jhotdraw.draw.BezierFigure.contains				x
org.jhotdraw.geom.BezierPath.outlineContains				x

Figure 4.9 – Liste détaillée des résumés de la phase 3 de JHotDraw

Phase 4 La phase finale présente des résultats encore plus marqués. Dans cette phase, le programme sauvegarde les propriétés du projet (la taille et la position des formes dessinées) dans le fichier de propriétés du projet. Il enregistre ensuite le projet si l'utilisateur le lui demande et le ferme. Le processus de sauvegarde utilise des méthodes manipulant des objets XML puisqu'il s'agit du format utilisé pour enregistrer les informations. La présence de méthodes qui écrivent des objets XML, telles que les suivantes, est donc un indicateur clair du comportement du segment :

— nanoxml.XMLElement.write

— org.jhotdraw.xml.NanoXMLLiteDOMOutput.addAttribute

— org.jhotdraw.xml.NanoXMLLiteDOMOutput.addAttribute

Le résumé Helmholtz de la segmentation à 3 facteurs fonctionne particulièrement bien à cet égard. Sept des dix méthodes présentées dans ce résumé sont liées au comportement sous-jacent, ce qui permet à un mainteneur logiciel de comprendre très facilement le comportement du système à partir de la trace. Seules 2 méthodes spécifiques au comportement apparaissent dans le résumé à 3 facteurs généré avec TF-IDF et aucun des deux résumés (Helmholtz ou TF-IDF) générés par de l'approche à 2 facteurs ne contient de méthodes spécifiques au comportement. La Figure 4.10 présente une liste détaillée des méthodes présentées dans chaque résumé de cette phase.

Methodes	3-facteurs Helmholtz	3-facteurs TF-IDF	2-facteurs Helmholtz	2-facteurs TF-IDF
nanoxml.XMLElement.write	x	x		
org.jhotdraw.draw.RectangleFigure.contains	x	x		
org.jhotdraw.draw.AbstractHandle.drawe	x	x		
org.jhotdraw.draw.AbstractHandle.drawCircle	x	x		
nanoxml.XMLElement.iterateChildren	x	x		
nanoxml.XMLElement.toString	x			
org.jhotdraw.xml.NanoXMLLiteDOMOutput.addAttribute	x			
org.jhotdraw.gui.JSheet.isShowAsSheet	x			
org.jhotdraw.gui.JSheet.getWindowForComponent	x			
org.jhotdraw.xml.NanoXMLLiteDOMOutput.closeElement	x			
org.jhotdraw.draw.action.AbstractSelectedAction$EventHandler.propertyChange		x		x
org.jhotdraw.draw.AbstractHandle.contains		x		x
org.jhotdraw.app.action.AbstractProjectAction$2.propertyChange		x		
nanoxml.XMLElement.writeEncoded		x		x
org.jhotdraw.draw.AttributeKeys.$SWITCH_TABLE$org$jhotdraw$draw$AttributeKeys $StrokePlacements		x		x
org.jhotdraw.util.ReversedList.get			x	
org.jhotdraw.app.DefaultSDIApplication$2.propertyChange			x	
org.jhotdraw.gui.JSheet.dispose			x	
org.jhotdraw.xml.NanoXMLLiteDOMOutput.getPrototype			x	
org.jhotdraw.draw.AbstractHandle.setView			x	
org.jhotdraw.draw.LocatorHandle.getLocation			x	
org.jhotdraw.draw.AttributeKeys.getPerpendicularFillGrowth			x	
org.jhotdraw.gui.JSheet.installSheet			x	
org.jhotdraw.draw.RectangleFigure.drawFill			x	
org.jhotdraw.undo.UndoRedoManager.setHasSignificantEdits			x	
org.jhotdraw.draw.DefaultDrawingView.removeFigureSelectionListener				x
org.jhotdraw.draw.action.AbstractSelectedAction.access$0				x
org.jhotdraw.draw.AbstractHandle.getBounds				x
org.jhotdraw.draw.AttributeKeys.getPerpendicularDrawGrowth				x
org.jhotdraw.draw.AttributeKeys.getPerpendicularHitGrowth				x
org.jhotdraw.draw.RectangleFigure.getFigureDrawBounds				x

Figure 4.10 – Liste détaillée des résumés de la phase 4 JhotDraw

4.4.2 ARGOUML

Dans un deuxième exemple, nous avons généré la trace d'un scénario en 4 phases avec ArgoUML, une application de création de diagrammes UML écrite en Java. Les phases consistent à lancer ArgoUML pour dessiner un diagramme de cas d'utilisation, dessiner un diagramme de classes et fermer ArgoUML.

Phase 1 La première phase consiste à lancer l'application et à initialiser l'interface graphique d'ArgoUML (menus, barre d'outils, barre d'état et panneaux principaux). Les résultats obtenus

dans cette phase sont distincts de nos autres résultats dans le sens où il y a un chevauchement minimal entre les méthodes jugées les plus significatives des résumés de l'approche à 2 facteurs et à 3 facteurs, avec seulement 3 méthodes apparaissant à la fois dans les résumés à 2 facteurs et à 3 facteurs. En outre, les deux résumés générés à l'aide de l'approche à deux facteurs étaient complètement différents, ce qui laisse planer un doute sur l'exactitude des résumés générés.

Cependant, les deux résumés générés à l'aide de l'approche à 3 facteurs sont largement similaires et partagent 7 méthodes. Ces deux résumés contiennent deux méthodes distinctes à partir desquelles le comportement sous-jacent (démarrage de l'application) de la trace peut être obtenu presque immédiatement :

— org.argouml.application.Main$1.i18nmessageFormat
— org.argouml.application.helpers.ApplicationVersion.getStableVersion

Les packages org.argouml.application et org.argouml.application.helpers contiennent la classe principale et le point de départ de l'application ArgoUML et fournissent des classes «auxiliaires» rendant disponibles les fonctionnalités de base de l'application. La méthode getStableVersion reçoit la version de l'application au moment de l'initialisation.

De même, la méthode i18nmessageFormat fournit une traduction des chaînes de caractères qui doit être personnalisée en fonction de la langue de l'utilisateur - une tâche effectuée au démarrage. Toutes les autres méthodes présentes dans les quatre résumés semble être des méthodes utilités, à l'exception des méthodes spécifiques au comportement mal-assignée et mal-placée dans le récapitulatif de Helmholtz à 2 facteurs. La Figure 4.11 présente une liste détaillée des méthodes présentées dans chaque résumé de cette phase.

Phase 2 La deuxième phase consiste à dessiner un diagramme de cas d'utilisation. Dans ce

Methodes	3-facteurs Helmholtz	3-facteurs TF-IDF	2-facteurs Helmholtz	2-facteurs TF-IDF
org.argouml.uml.cognitive.critics.CrUML.getLocalizedString	x	x		x
org.argouml.cognitive.Critic.addSupportedDecision	x	x		
org.argouml.cognitive.Decision.toString	x	x		
org.argouml.cognitive.Critic.addTrigger	x	x		
org.argouml.application.helpers.ApplicationVersion.getStableVersion	x	x		
org.argouml.application.helpers.ApplicationVersion.getStableVersion	x	x		
org.argouml.cognitive.Critic.defaultMoreInfoURL	x	x		
org.argouml.cognitive.Translator.messageFormat	x			
org.argouml.cognitive.Critic.setDescription	x			
org.argouml.cognitive.Critic.setHeadline	x			
org.argouml.cognitive.Critic.toString		x		x
org.argouml.cognitive.Agency.theAgency		x		x
org.argouml.cognitive.Designer.getAgency		x		x
org.argouml.uml.ui.PropPanel.setTitleLabel			x	
org.argouml.ui.ProjectBrowser.addPanel			x	
org.argouml.uml.cognitive.critics.CrUML.getClassSimpleName			x	
org.argouml.ui.SplashPanel$1.paint			x	
org.argouml.ui.DetailsPane.enableTabs			x	
org.argouml.ui.explorer.rules.GoCompositeStateToSubvertex.getRuleName			x	
org.argouml.cognitive.ui.ToDoItemAction.updateEnabled			x	
org.argouml.profile.internal.ocl.CompositeModelInterpreter.addModelInterpreter			x	
org.argouml.ui.explorer.rules.GoStereotypeToTagDefinition.getRuleName			x	
org.argouml.ui.DetailsPane.addTab			x	
org.argouml.cognitive.Critic.getHeadline				x
org.argouml.ui.cmd.ActionWrapper.getActionName				x
org.argouml.ui.cmd.ShortcutMgr$1.compare				x
org.argouml.cognitive.Agency.register				x
org.argouml.ui.explorer.rules.AbstractPerspectiveRule.toString				x
org.argouml.cognitive.ListSet.add				x

Figure 4.11 – Liste détaillée des résumés de la phase 1 de ArgoUML

cas, notre expérience a fourni la preuve de l'efficacité de l'algorithme Helmholtz par rapport à l'algorithme TF-IDF. Comme mentionné précédemment, chacun des résumés est réduit à 10 méthodes, en considérant uniquement les entrées les plus pertinentes. Le résumé Helmholtz à 3 facteurs contient 2 méthodes issues des packages org.argouml.uml.diagram. use_case.ui. Ces deux méthodes suffisent pour comprendre le comportement sous-jacent de la trace car aucune des autres méthodes présentes dans les résumés n'est spécifique au comportement. Elles peuvent donc être ignorées en essayant de comprendre le comportement du programme à partir de sa trace d'exécution.

Parallèlement on examinant la documentation et comme son nom l'indique le package org.argouml.uml.diagram. use_case.ui contiennent les classes qui implémentent un diagramme

de cas d'utilisation à l'aide du GEI (Graph Editing Framework) UCI. Aucun des deux résumés TD-IDF ne contenait de méthode spécifique au comportement.

Le résumé Helmholtz à 2 facteurs contenait une méthode unique de org.argouml.uml.diagram.use_case.ui. Cependant, il contenait également une méthode spécifique au comportement mal-attribuée, qui fait en réalité référence au comportement de la phase de programme suivante. En conséquence, une détermination du comportement de la trace sous-jacente serait presque impossible avec une analyse autre que l'approche à 3 facteurs proposée dans cette recherche. La Figure 4.12 présente une liste détaillée des méthodes présentées dans chaque résumé de cette phase.

Methodes	3-facteurs Helmholtz	3-facteurs TF-IDF	2-facteurs Helmholtz	2-facteurs TF-IDF
org.argouml.ui.explorer.rules.AbstractPerspectiveRule.toString	x	x		
org.argouml.ui.cmd.ShortcutMgr.putDefaultShortcut	x	x		
org.argouml.ui.explorer.ExplorerPerspective.addRule	x	x		
org.argouml.ui.cmd.GenericArgoMenuBar.setMnemonic	x	x		
org.argouml.uml.diagram.use_case.ui.SelectionActor.getIcons	x		x	
org.argouml.ui.cmd.ShortcutMgr.assignAccelerator	x			
org.argouml.uml.diagram.use_case.ui.SelectionActor.getInstructions	x			
org.argouml.ui.explorer.ExplorerTreeModel.addToMap	x			
org.argouml.ui.explorer.ExplorerTreeModel.insertNodeInto	x			
org.argouml.ui.explorer.ExplorerTreeModel.addNodesToMap	x			
org.argouml.cognitive.Critic.toString		x		x
org.argouml.ui.cmd.ShortcutMgr$1.compare		x		
org.argouml.configuration.Configuration.makeKey		x		
org.argouml.cognitive.Critic.setEnabled		x		
org.argouml.cognitive.Critic.getCriticCategory		x		
org.argouml.cognitive.Critic.getCriticKey		x		
org.argouml.application.events.ArgoEvent.getEventEndRange			x	
org.argouml.uml.ui.UMLComboBoxModel2.setElements			x	
org.argouml.cognitive.checklist.CheckManager.lookupChecklist			x	
org.argouml.ui.explorer.rules.GoProfileConfigurationToProfile.getChildren			x	
org.argouml.uml.diagram.static_structure.ui.UMLClassDiagram.getLabelName			x	
org.argouml.uml.diagram.ui.ModeLabelDrag.mouseReleased			x	
org.argouml.ui.explorer.ExplorerTreeModel.mergeChildren			x	
org.argouml.uml.ui.UMLChangeDispatch.dispatch			x	x
org.argouml.uml.ui.TabStyle.findPanelFor			x	
org.argouml.uml.diagram.ArgoDiagramImpl.toString				x
org.argouml.application.events.ArgoStatusEvent.getEventStartRange				x
org.argouml.uml.diagram.ui.FigNodeModelElement.hit				x
org.argouml.ui.explorer.ExplorerTreeNode.nodeModified				x
org.argouml.ui.explorer.ExplorerTreeModel.traverseModified				x
org.argouml.cognitive.Critic.getHeadline				x
org.argouml.uml.ui.PropPanel.collectTargetListeners				x
org.argouml.cognitive.ToDoList.elementListForOffender				x

Figure 4.12 – Liste détaillée des résumés de la phase 2 de ArgoUML

Phase 3 La troisième phase consiste à dessiner un diagramme de classes. Comme on peut le constater, les résultats de l'analyse des résumés de cette phase soulignent l'efficacité de l'approche à 3 facteurs par rapport à l'approche à 2 facteurs.

Chaque méthode présentée dans le résumé de Helmholtz de l'approche à 3 facteurs permet de comprendre le comportement sous-jacent du segment de trace. De même, toutes les méthodes présentées dans le résumé TF-IDF (à l'exception d'une méthode) ont une signification similaire. Voici quelques exemples de méthodes présentes dans ces deux résumés :

— org.argouml.uml.diagram.static_structure.ui.FigClassifierBox.
updateListeners
— org.argouml.uml.diagram.static_structure.ui.FigClassifierBox
.updateCompartment
— org.argouml.uml.diagram.static_structure.ui.FigClass.updateNameText

Ces méthodes appartiennent au package org.argouml.uml.diagram.static_structure.ui qui contient des classes qui implémentent un diagramme de classe. Plusieurs méthodes du package org.argouml.uml.diagram.ui. sont également présentes. Ce paquet fournit divers supports pour les diagrammes : actions, propriétés de panneaux pour les diagrammes, support de sélection GEF, etc.

Le résumé TF-IDF de l'approche à 2 facteurs donne également de bons résultats, 9 méthodes sur 10 figurant dans le résumé font partie de l'un des deux packages mentionnés ci-dessus. Cependant, une seule des 10 méthodes présentes dans le résumé à 2 facteurs de l'algorithme Helmholtz est spécifique au comportement. La Figure 4.13 présente une liste détaillée des méthodes présentées dans chaque résumé de cette phase.

Methodes	3-facteurs Helmholtz	3-facteurs TF-IDF	2-facteurs Helmholtz	2-facteurs TF-IDF
org.argouml.uml.diagram.ui.SelectionClassifierBox.hitHandle	x	x		x
org.argouml.uml.diagram.ui.FigCompartment.setExternalSeparatorFigBounds	x	x		
org.argouml.uml.diagram.ui.FigCompartmentBox.setCompartmentBounds	x	x		
org.argouml.uml.diagram.ui.FigCompartmentBox.calculateCompartmentBoxDimensions	x	x		
org.argouml.uml.diagram.ui.FigEdgeModelElement.propertyChange	x			
org.argouml.uml.diagram.ui.FigCompartment.setBoundsImpl	x	x		
org.argouml.uml.diagram.static_structure.ui.FigClassifierBox.updateListeners	x			
org.argouml.uml.diagram.static_structure.ui.FigClassifierBox.updateCompartment	x			
org.argouml.uml.diagram.static_structure.ui.FigClass.updateNameText	x			
org.argouml.uml.diagram.ui.FigOperationsCompartment.getUmlCollection	x			
org.argouml.uml.diagram.static_structure.ui.FigClassifierBox.propertyChange		x		x
org.argouml.uml.diagram.ui.FigCompartment.getMinimumSize		x		x
org.argouml.uml.ui.UMLComboBoxModel2.getSize		x		
org.argouml.uml.diagram.ui.RadioAction.getAction		x		
org.argouml.uml.diagram.ui.FigCompartmentBox.getVisibleCompartmentCount		x	x	
org.argouml.uml.diagram.DiagramUndoManager$DiagramCommand.toString			x	
org.argouml.uml.ui.UMLLinkedListCellRenderer.getListCellRendererComponent			x	
org.argouml.util.ThreadUtils.checkIfInterrupted			x	
org.argouml.ui.ProgressMonitorWindow$1.run			x	
org.argouml.configuration.Configuration.setString			x	
org.argouml.taskmgmt.ProgressEvent.getPosition			x	
org.argouml.uml.diagram.ArgoDiagramImpl.figDescription			x	
org.argouml.uml.diagram.ProjectMemberDiagram.getType			x	
org.argouml.persistence.PersistenceManager.getBaseName			x	
org.argouml.uml.diagram.ui.FigCompartment.getBigPort				x
org.argouml.uml.diagram.ui.FigNodeModelElement.getBigPort				x
org.argouml.uml.diagram.ui.FigNodeModelElement.propertyChange				x
org.argouml.persistence.AbstractFilePersister.getExtension				x
org.argouml.uml.diagram.ui.FigCompartmentBox.getLineWidth				x
org.argouml.uml.diagram.ui.FigNodeModelElement.hit				x
org.argouml.uml.diagram.ui.FigNodeModelElement.getStereotypeFig				x

Figure 4.13 – Liste détaillée des résumés de la phase 3 de ArgoUML

Phase 4 Dans la quatrième phase, le programme enregistre le paramétrage du projet (appelé persistance) et se ferme. Là encore, plusieurs méthodes permettant de mettre en évidence le comportement du segment sont présentes dans les résumés ; En particulier, les trois méthodes suivantes :

— org.argouml.persistence.UmlFilePersister.hasAnIcon

— org.argouml.persistence.PgmlUtility.getEnclosingId

— org.argouml.persistence.AbstractFilePersister.getExtension

Le package org.argouml.persistence contient le support pour l'enregistrement de projets sur le disque. En utilisant notre méthode à 3 facteurs, ces méthodes apparaissent correctement

dans le résumé généré à la fois en utilisant Helmholtz et TF-IDF. Aucun des deux résumés à 2 facteurs ne contient une méthode de ce package, ni aucune autre méthode pouvant être utilisée pour comprendre le comportement sous-jacent de la trace. La Figure 4.14 présente une liste détaillée des méthodes présentées dans chaque résumé de cette phase.

Methodes	3-facteurs Helmholtz	3-facteurs TF-IDF	2-facteurs Helmholtz	2-facteurs TF-IDF
org.argouml.persistence.UmlFilePersister.hasAnIcon	x	x		
org.argouml.kernel.ProjectImpl.getMembers	x	x		
org.argouml.persistence.PgmlUtility.getEnclosingId	x	x		
org.argouml.uml.diagram.ProjectMemberDiagram.getDiagram	x	x		
org.argouml.persistence.AbstractFilePersister.getExtension	x			
org.argouml.uml.diagram.ui.FigNodeModelElement.getName	x			
org.argouml.uml.ui.ActionReopenProject.getFilename	x			
org.argouml.ui.SwingWorker.access$0	x			
org.argouml.ui.SwingWorker$TimerListener.actionPerformed	x			
org.argouml.ui.ProgressMonitorWindow.isCanceled	x			
org.argouml.persistence.AbstractFilePersister.accept		x		
org.argouml.persistence.PgmlUtility.getId		x		
org.argouml.persistence.PgmlUtility.getVisibility		x		
org.argouml.uml.diagram.ui.FigCompartment.getMinimumSize		x		
org.argouml.uml.UUIDHelper.getUUID		x		
org.argouml.uml.diagram.static_structure.ui.FigClassifierBox.propertyChange		x		
org.argouml.configuration.Configuration.setInteger			x	x
org.argouml.configuration.ConfigurationHandler.setInteger			x	
org.argouml.configuration.Configuration.save			x	
org.argouml.ui.ProjectBrowser.getInstance				x
org.argouml.configuration.ConfigurationHandler.saveDefault				x
org.argouml.ui.SwingWorker.finished				x
org.argouml.ui.SwingWorker$ThreadVar.clear				x
org.argouml.ui.SwingWorker.access$1				x
org.argouml.ui.SwingWorker.access$3				x
org.argouml.ui.ProgressMonitorWindow$2.run				x
org.argouml.configuration.ConfigurationProperties.setValue				x
org.argouml.swingext.GlassPane.setVisible				x

Figure 4.14 – Liste détaillée des résumés de la phase 4 de ArgoUML

4.4.3 JLGUI

Comme troisième exemple, nous avons généré une trace de jlGui, un lecteur de musique, en utilisant un scénario en 3 phases (Lancer jlGui, sélectionner et lire un élément et fermer jlGui).

Phase 1 La première phase consiste à initialiser l'interface graphique de jlGui (chargement du lecteur, liste de lecture du chargement) et à lancer l'application. Tous les types de résumés

fonctionnent bien sur cette phase.

La classe javazoom.jlgui.player.amp.util.BMPLoader et le package javazoom.jlgui.player.amp.skin. contiennent les méthodes nécessaires au chargement de l'interface graphique personnalisée de l'application. Plusieurs méthodes de ce package et de cette classe apparaissent dans chacun des 4 résumés. Cependant, la présence de méthodes issues de ces deux sources dans l'approche à 3 facteurs de l'algorithme Helmholtz donne au mainteneur une meilleure compréhension du contenu du résumé. La Figure 4.15 présente une liste détaillée des méthodes présentées dans chaque résumé de cette phase.

Methodes	3-facteurs Helmholtz	3-facteurs TF-IDF	2-facteurs Helmholtz	2-facteurs TF-IDF
javazoom.jlgui.player.amp.util.BMPLoader.readScanLine	x	x		x
javazoom.jlgui.player.amp.util.BMPLoader.unpack	x	x		x
javazoom.jlgui.player.amp.equalizer.ui.Cubic.eval	x	x		
javazoom.jlgui.player.amp.util.FileNameFilter.accept	x	x		
javazoom.jlgui.player.amp.equalizer.ui.ControlCurve.boundY	x	x		
javazoom.jlgui.player.amp.util.BMPLoader.readInt	x	x		x
javazoom.jlgui.player.amp.util.BMPLoader.readShort	x	x		x
javazoom.jlgui.player.amp.skin.Taftb.getBanner	x	x		
javazoom.jlgui.player.amp.skin.Skin.getAcEqSliders	x	x		x
javazoom.jlgui.player.amp.skin.AbsoluteLayout.addLayoutComponent	x	x		x
javazoom.jlgui.player.amp.util.ini.Configuration.getInt			x	
javazoom.jlgui.player.amp.skin.Skin.getMainHeight			x	
javazoom.jlgui.player.amp.skin.Skin.getAcPlAddPopup			x	
javazoom.jlgui.player.amp.skin.ActiveJToggleButton.setConstraints			x	
javazoom.jlgui.player.amp.skin.ActiveJLabel.getConstraints			x	
javazoom.jlgui.player.amp.skin.Skin.getAcPlList			x	
javazoom.jlgui.player.amp.skin.Skin.getAcEqualizer			x	
javazoom.jlgui.player.amp.skin.Skin.getAcPlDown			x	
javazoom.jlgui.player.amp.skin.Skin.getAcRepeat			x	
javazoom.jlgui.player.amp.skin.ActiveSliderUI.setThumbImage			x	
javazoom.jlgui.player.amp.skin.AbsoluteConstraints.toString				x
javazoom.jlgui.player.amp.util.ini.Configuration.get				x
javazoom.jlgui.player.amp.visual.ui.SpectrumTimeAnalyzer.getColor				x
javazoom.jlgui.player.amp.util.BMPLoader.getBMPImage				x

Figure 4.15 – Liste détaillée des résumés de la phase 1 de jlGui

Phase 2 La deuxième phase consiste à charger une liste de lecture et à lire un élément. Dans ce cas, l'approche à trois facteurs donne de meilleurs résultats si on la compare à l'approche à deux facteurs et les résumés de l'algorithme Helmholtz sont également meilleurs que les

résumés de l'algorithme TF-IDF. En particulier, lors de la segmentation de la trace à l'aide de l'approche à 3 facteurs, 7 méthodes sur 10 présentées dans le résumé Helmhotlz et 5 méthodes sur 10 présentées dans le résumé TF-IDF sont spécifiques au comportement, ce qui rend le comportement sous-jacent de la trace facile à comprendre dans les deux cas (toutes les autres méthodes du résumé sont des utilitaires).

Pour l'approche à deux facteurs, les résultats sont respectivement 3 et 4 pour Helmholtz et TF-IDF. Les résumés de l'approche à 2 facteurs semblent également contenir des appels de méthodes liés à l'ouverture des fichiers, ce qui n'aide pas à la compréhension du comportement sous-jacent du système.

Des exemples de méthodes spécifiques au comportement présentes dans les résumés de cette phase incluent les méthodes de la classe javazoom.jlgui.player .amp.PlayerUI. Cette classe est la classe principale de l'interface utilisateur. Il implémente BasicPlayerListener et contient les méthodes nécessaires à la lecture du contenu multimédia.

Plusieurs méthodes de la classe SpectrumTimeAnalyzer concernent également la lecture d'un fichier multimédia. Ils apparaissent dans les deux résumés générés en utilisant notre approche à 3 facteurs, mais sont absents des résumés générés en utilisant l'approche à 2 facteurs. Les deux résumés à 2 facteurs contiennent également des méthodes mal assignées. La Figure 4.16 présente une liste détaillée des méthodes présentées dans chaque résumé de cette phase.

Phase 3 La troisième phase concerne la fermeture du programme et la sauvegarde des propriétés de l'utilisateur. Pour cette phase, comme pour la phase initiale de la trace, les deux méthodes de segmentation semblent fonctionner correctement, mais la méthode à 3 facteurs surpasse toujours celle de la méthode à 2 facteurs.

Par exemple, une méthode qui contribue au processus d'arrêt se présente comme suit : confi-

Methodes	3-facteurs Helmholtz	3-facteurs TF-IDF	2-facteurs Helmholtz	2-facteurs TF-IDF
javazoom.jlgui.player.amp.skin.DragAdapter.mouseMoved	x			
javazoom.jlgui.player.amp.visual.ui.SpectrumTimeAnalyzer.stereoMerge	x	x		
javazoom.jlgui.player.amp.visual.ui.SpectrumTimeAnalyzer.process	x			
javazoom.jlgui.player.amp.visual.ui.SpectrumTimeAnalyzer.drawSpectrumAnalyser	x			
javazoom.jlgui.player.amp.PlayerUI.progress	x			
javazoom.jlgui.player.amp.visual.ui.SpectrumTimeAnalyzer.writeDSP	x	x		
javazoom.jlgui.player.amp.PlayerUI.processProgress	x	x	x	
javazoom.jlgui.player.amp.playlist.PlaylistItem.getLength	x	x	x	
javazoom.jlgui.player.amp.equalizer.ui.EqualizerUI.setBands	x			
javazoom.jlgui.player.amp.skin.ActiveJNumberLabel.setAcText	x			
javazoom.jlgui.player.amp.skin.Skin.getAcVolume		x		
javazoom.jlgui.player.amp.skin.ImageBorder.paintBorder		x	x	
javazoom.jlgui.player.amp.visual.ui.SpectrumTimeAnalyzer.paintComponent		x		
javazoom.jlgui.player.amp.util.ini.SortedStrings.stringAt		x		
javazoom.jlgui.player.amp.util.ini.SortedStrings.insert		x		
javazoom.jlgui.player.amp.util.ini.Alphabetizer.lessThan		x		
javazoom.jlgui.player.amp.skin.PopupAdapter.checkPopup			x	
javazoom.jlgui.player.amp.tag.MpegInfo.loadInfo			x	
javazoom.jlgui.player.amp.PlayerUI.setCurrentSong			x	
javazoom.jlgui.player.amp.skin.PlaylistUIDelegate.paintBackground			x	x
javazoom.jlgui.player.amp.equalizer.ui.ControlCurve.boundY			x	x
javazoom.jlgui.player.amp.util.Config.getTaginfoPolicy			x	
javazoom.jlgui.player.amp.skin.ActiveFont.getImage			x	
javazoom.jlgui.player.amp.equalizer.ui.Cubic.eval				x
javazoom.jlgui.player.amp.util.FileNameFilter.accept				x
javazoom.jlgui.player.amp.equalizer.ui.ControlCurve.addPoint				x
javazoom.jlgui.player.amp.playlist.ui.PlaylistUI.getPlaylist				x
javazoom.jlgui.player.amp.playlist.BasePlaylist.getPlaylistSize				x
javazoom.jlgui.player.amp.playlist.ui.PlaylistUI.getTopIndex				x
javazoom.jlgui.player.amp.playlist.BasePlaylist.getSelectedIndex				x
javazoom.jlgui.player.amp.util.FileSelector.getInstance				x

Figure 4.16 – Liste détaillée des résumés de la phase 2 de jlGui

guration.save dans le package javazoom.jlgui.player.amp.util

.ini. Cette méthode enregistre les propriétés du projet sur le disque et sa présence rend le comportement sous-jacent de la trace compréhensible. Il n'apparaît que dans le résumé généré à l'aide de l'algorithme de Helmholtz de notre approche à 3 facteurs.

Cependant, chacun des 4 résumés contient plusieurs occurrences de méthodes de la classe : SortedStrings. Lorsque le programme est fermé, un objet de configuration est utilisé pour enregistrer un ensemble de propriétés de configuration. Les propriétés sont écrites sur le disque par paires "nom = valeur". À partir d'une analyse de la documentation, nous avons découvert que ces paires sont écrites sur le disque à l'aide de méthodes de la classe .SortedStrings. En conséquence, la présence de ces méthodes dans le résumé de la phase facilite beaucoup la

compréhension de la phase.

Enfin, une autre méthode importante est SpectrumTimeAnalyzer.stopDSP dans le package javazoom.jlgui.player.amp.util.ini. Cette méthode arrête les processeurs de signal numérique (DSP) lorsque l'utilisateur clique sur le bouton de fermeture, ce qui aide à comprendre le comportement du programme. Ladite méthode apparaît dans les deux résumés générés par la méthode à 3 facteurs, mais uniquement dans le résumé TF-IDF de l'approche à 2 facteurs. Globalement, le résumé à 3 facteurs de Helmholtz contient 8 méthodes spécifiques au comportement, tandis que les 3 autres résumés contiennent entre 3 et 5. Une méthode d'initialisation est également présente dans les résumés à 2 facteurs rendant le résumé flou. La Figure 4.17 présente une liste détaillée des méthodes présentées dans chaque résumé de cette phase.

Methodes	3-facteurs Helmholtz	3-facteurs TF-IDF	2-facteurs Helmholtz	2-facteurs TF-IDF
javazoom.jlgui.player.amp.util.ini.SortedStrings.stringCount	x			x
javazoom.jlgui.player.amp.util.ini.SortedStrings.search	x			x
javazoom.jlgui.player.amp.util.ini.SortedStrings.insert	x	x	x	x
javazoom.jlgui.player.amp.util.ini.SortedStrings.add	x			x
javazoom.jlgui.player.amp.util.ini.SortedStrings.stringAt	x	x	x	x
javazoom.jlgui.player.amp.util.ini.SortedStrings.clear	x			
javazoom.jlgui.player.amp.visual.ui.SpectrumTimeAnalyzer.stopDSP	x	x	x	
javazoom.jlgui.player.amp.util.ini.Configuration.save	x			
javazoom.jlgui.player.amp.visual.ui.SpectrumTimeAnalyzer.getDisplayMode	x	x	x	x
javazoom.jlgui.player.amp.util.ini.Alphabetizer.lessThan	x	x	x	x
javazoom.jlgui.player.amp.visual.ui.SpectrumTimeAnalyzer.stereoMerge		x	x	
javazoom.jlgui.player.amp.visual.ui.SpectrumTimeAnalyzer.drawSpectrumAnalyser		x	x	
javazoom.jlgui.player.amp.visual.ui.SpectrumTimeAnalyzer.process		x	x	
javazoom.jlgui.player.amp.util.ini.Alphabetizer.greaterThan		x	x	x
javazoom.jlgui.player.amp.PlayerUI.actionPerformed		x		
javazoom.jlgui.player.amp.visual.ui.SpectrumTimeAnalyzer.drawSpectrumAnalyserBar			x	
javazoom.jlgui.player.amp.util.ini.Alphabetizer.compare				x
javazoom.jlgui.player.amp.skin.DragAdapter.mouseMoved				x

Figure 4.17 – Liste détaillée des résumés de la phase 3 de jlGui

4.4.4 COMPARAISON DES RÉSULTATS

Dans cette section nous comparons les résultats de notre approche de segmentation à trois facteurs avec les résultats obtenu par la segmentation à deux facteurs de Pirzadeh et al. (2011). Ainsi que l'impacte de l'utilisation des deux approches sur les résumés avec les algorithmes TF-IDF et Helmhotlz. Cette comparaison est fait sur les quatre systèmes utilisé dans les expérimentations a savoir ArgoUML, jhotDraw, Weka et DrawSWF.

La segmentation des traces de ArgoUML et DrawSWF en utilisant notre approche de segmentation à trois facteurs donne toujours des segments plus similaire au segments de référence et ça pour les deux scénario de segmentation en 3 phases et 4 phases. Tandis que pour le système Weka on a presque une égalité dans le nombre de segments les plus similaire au segments de référence, avec une remarque importante que la premier phase de l'approche à deux facteurs est toujours très similaire à la premier phase de référence.

Concernant le système jhotDraw la segmentation avec notre approche a trois facteurs est meilleur que la segmentation a deux facteurs dans le cas du scénario en 3 phases tandis que pour le scénario a 4 phases il y a une égalité dans le nombre de phases les plus similaire au segments de référence.

Le tableau 4.7 résume le nombre de segments les plus similaire au segments de référence pour les deux approches dans le cas du scénario en 3 phases et 4 phases.

Tableau 4.7 – Nombre de segments les plus similaire au segments de référence pour les deux approches

	ArgoUML		jhotDraw		Weka		DrawSWF	
	3-f	2-f	3-f	2-f	3-f	2-f	3-f	2-f
Segmentation en 3 phases	**21**	0	**13**	8	**11**	10	**21**	0
Segmentation en 4 phases	**16**	4	10	10	10	10	**17**	3

Ce qui concerne la qualité des résumés le tableau 4.8 montre les nombre totale des méthodes spécifiques au comportement de chaque phase pour les deux approches 3-facteurs et 2-facteurs. Nous remarquants que les résumés de l'approche à trois facteurs contient plus de méthodes qui aident a comprendre le contenu de la phase.

Tableau 4.8 – Confiance pour les assignations correctes et incorrectes

	3-facteurs	2-facteurs
phase 1 de JHotDraw	8	2
phase 2 de JHotDraw	8	14
phase 3 de JHotDraw	8	2
phase 4 de JHotDraw	9	0
phase 1 de ArgoUML	4	0
phase 2 de ArgoUML	2	1
phase 3 de ArgoUML	19	10
phase 4 de ArgoUML	8	0
phase 1 de jlGui	14	17
phase 2 de jlGui	11	7
phase 3 de jlGui	11	8

4.4.5 SYNTHÈSE DES RÉSULTATS

Le tableau 4.9 résume les résultats de notre expérimentation. Pour chaque résumé de segment nous listons la taille du résumé (S), le nombre de méthodes spécifiques au comportement liées au comportement sous-jacent du segment qu'il contient (A, pour accuracy), ainsi que le nombre de méthodes mal-assignées dans ce segment résumé (M).

Comme le montre tableau, la méthode de segmentation à 3 facteurs est meilleure que celle de la segmentation à 2 facteurs dans le sens suivant : les résumés qu'elle génère contiennent plus de méthodes spécifiques au comportement et moins de méthodes placées par erreur. En utilisant la même métrique, l'extraction d'éléments clés de la trace en utilisant l'algorithme Helmhotlz est généralement meilleure que l'algorithme TF-IDF. En fait, dans nos expériences,

les résumés générés par l'approche à 3 facteurs n'ont jamais contenu de méthode mal-assignée.

Lors de l'examen de plusieurs traces du même programme, nous avons constaté qu'à chaque fois que le comportement sous-jacent du programme est identique pour plusieurs segments d'une trace, les méthodes relatives au comportement présentes dans le résumé de trace tendent également à être les mêmes. Par exemple, deux segments capturant tous les deux le démarrage de deux exécutions différentes du même programme ont tendance à contenir les mêmes méthodes relatives au comportement. Ceci est particulièrement intéressant du fait que cela peut aider à automatiser le processus d'analyse.

	3-facteurs						2-facteurs					
	Helmholtz			TF-IDF			Helmholtz			TF-IDF		
	S	A	M	S	A	M	S	A	M	S	A	M
JHotdraw phase 1	10	5	0	10	3	0	9	0	2	10	0	0
JHotdraw phase 2	10	8	0	10	1	0	10	9	0	10	5	0
JHotdraw phase 3	10	5	0	10	3	0	10	1	1	10	1	2
JHotdraw phase 4	10	7	0	10	2	0	10	0	1	10	0	2
ArgoUML phase 1	10	2	2	10	2	0	10	0	1	10	0	0
ArgoUML phase 2	10	2	2	10	0	0	10	1	1	10	0	1
ArgoUML phase 3	10	10	0	10	9	0	10	1	0	10	9	0
ArgoUML phase 4	10	3	0	10	2	0	3	0	0	10	0	0
jlGui phase 1	10	7	0	10	7	0	10	9	0	10	8	0
jlGui phase 2	10	7	0	10	5	0	10	3	1	10	3	1
jlGui phase 3	10	8	0	10	3	0	10	3	0	10	4	0

Tableau 4.9 – Résumé des résultats

CONCLUSION

Avant de faire une tâche de maintenance logicielle, le mainteneur doit premièrement comprendre le système qu'il veut maintenir. Pour comprendre le système en question le mainteneur, peut analyser les traces d'exécutions des méthodes. Cependant, la taille considérable des traces d'exécution rend difficile cette analyse.

Dans ce travail, nous avons proposé une technique permettant de segmenter une grande trace d'exécution en phases d'exécution constitutives et d'extraire de chaque phase un résumé d'une taille exploitable par les mainteneurs logiciels qui est composé de ses événements clés. Nous montrons que notre méthode de segmentation des traces surpasse la méthode existants de Pirzadeh et al., car les résumés produits contiennent plus de méthodes liées au comportement et moins de méthodes mal placées peu importe la méthode utilisée pour l'extraction d'éléments-clés. Notre méthode de segmentation se distingue de la méthode de Pirzadeh et al. par l'application d'un troisième facteur de rapprochement des méthodes. En plus de rapprochement en fonction des noms des méthodes et le niveau d'imbrication des méthodes nous avons rapproché les méthodes en fonction de la similarité dans les valeurs de retour des méthodes et la similarité des paramètres. Nous comparons en outre deux algorithmes d'extraction d'éléments clés et montrons que l'algorithme Helmholetz surpasse généralement

l'algorithme TF-IDF en utilisant la même métrique.

Notre méthode de segmentation peut être améliorée de plusieurs manières. En particulier, dans notre expérimentation, la similarité entre les segments est calculée par une simple fonction faisant intervenir l'intersection des ensembles d'éléments clés dans les deux segments. Des méthodes plus élaborées (utiliser des modèles tirés du domaine de fouille de données) pourraient améliorer l'efficacité de l'approche. Notre méthode de segmentation nécessite une intervention humaine : un utilisateur doit spécifier le nombre attendu de segments avant d'appliquer l'algorithme K-means. Des recherches antérieures suggéraient d'expérimenter plusieurs valeurs et d'appliquer le critère d'information bayésien (BIC) pour déterminer la valeur la plus susceptible d'être exacte. La métrique de similarité peut constituer une méthode plus efficace, car l'utilisation de la valeur correcte pour k peut conduire à des segments présentant un niveau élevé de similarité avec les segments de référence. Afin d'évaluer l'aspect pratique de la segmentation de traces il est nécessaire d'inclure des études qualitatives impliquant des développeurs logiciels.

Le fait que le processus de segmentation repose désormais sur trois facteurs ouvre également la possibilité d'attribuer une pondération à chacun des trois schémas de segmentation (similarité, continuité et paramètres). Cela va orienter l'analyse à donner plus d'importance au facteur pris en compte dans ce schéma. Le choix de cette valeur peut être déterminé par une combinaison d'analyse de code et d'expérimentation. En pratique, un utilisateur peut deviner lequel des trois facteurs est susceptible d'être le plus déterminant. Par exemple, un compilateur peut appliquer plusieurs transformations successives aux mêmes objets. En conséquence, les valeurs des paramètres seraient un mauvais indicateur de la phase d'exécution, car le même gestionnaire d'objet apparaîtrait comme paramètre pour les méthodes se produisant dans différentes phases d'exécution. Inversement, pour un programme récursif, la similarité et la continuité porteront des informations plus significatives.

BIBLIOGRAPHIE

Basili, V. R. 1996. *Evolving and packaging reading technologies*. Coll. « Achieving Quality in Software2 », p. 3–13. Springer.

Becker, D., F. Wolf, W. Frings, M. Geimer, B. J. Wylie, et B. Mohr. 2007. « Automatic trace-based performance analysis of metacomputing applications ». In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, p. 1–10. IEEE.

Benomar, O., H. Sahraoui, et P. Poulin. 2014. « Detecting program execution phases using heuristic search ». In Le Goues, C. et S. Yoo, éditeurs, *Search-Based Software Engineering*. T. 8636, p. 16–30, Cham. Springer International Publishing.

Boyer, R. S. et J. S. Moore. 1977. « A fast string searching algorithm », *Communications of the ACM*, vol. 20, no. 10, p. 762–772, ACM.

Brooks, R. 1977. « Towards a theory of the cognitive processes in computer programming », *International Journal of Man-Machine Studies*, vol. 9, no. 6, p. 737–751, Elsevier Ltd.

Corbi, T. A. 1989. « Program understanding : Challenge for the 1990s », *IBM Systems Journal*, vol. 28, no. 2, p. 294–306.

Cornelissen, B., A. Zaidman, A. van Deursen, L. Moonen, et R. Koschke. 2009. « A systematic survey of program comprehension through dynamic analysis », *IEEE Transactions on Software Engineering*, vol. 35, no. 5, p. 684–702.

Dadachev, B., A. Balinsky, H. Balinsky, et S. Simske. 2012. « On the helmholtz principle for data mining ». In *Emerging Security Technologies (EST), 2012 Third International Conference on*, p. 99–102. IEEE.

Desolneux, A., L. Moisan, et J.-M. Morel. 2007. *From gestalt theory to image analysis : a probabilistic approach*. T. 34. Springer Science & Business Media.

Hamou-Lhadj, A. 2005. « The concept of trace summarization ». In *Proceedings of the 1st International Workshop on Program Comprehension through Dynamic Analysis*, p. 43–47. IEEE Computer Society Press.

———. 2006. « Techniques to simplify the analysis of execution traces for program comprehension ». Thèse de doctorat, Ottawa, Ont., Canada, Canada. AAINR15024.

Hamou-Lhadj, A., E. Braun, D. Amyot, et T. Lethbridge. 2005. « Recovering behavioral design models from execution traces ». In *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*, p. 112–121. IEEE.

Hamou-Lhadj, A. et T. Lethbridge. 2003. « Techniques for reducing the complexity of object-oriented execution traces ». In *Proceedings of the 2nd IEEE international workshop on visualizing software for understanding and analysis*, p. 35–40. Citeseer.

———. 2004. « Reasoning about the concept of utilities ». In *ECOOP International Workshop on Practical Problems of Programming in the Large, Oslo, Norway, Lecture Notes in Computer Science (LNCS)*. T. 3344, p. 10–22. Springer-Verlag.

———. 2006. « Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system ». In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, p. 181–190. IEEE.

Hamou-Lhadj, A., T. C. Lethbridge, et L. Fu. 2004. « Challenges and requirements for an effective trace exploration tool ». In *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*, p. 70–78. IEEE.

IEEE-1219. 1998. « Ieee standard for software maintenance », *IEEE Std 1219-1998*, p. 1–56.

Jerding, D. F. et J. T. Stasko. 1998. « The information mural : A technique for displaying and navigating large information spaces », *IEEE Transactions on Visualization and Computer Graphics*, vol. 4, no. 3, p. 257–271, IEEE.

Joachims, T. 1998. « Text categorization with support vector machines : Learning with many relevant features ». In Nédellec, C. et C. Rouveirol, éditeurs, *Machine Learning : ECML-98*. T. 1398, p. 137–142, Berlin, Heidelberg. Springer Berlin Heidelberg.

Khoury, R., L. Shi, et A. Hamou-Lhadj. 2016. « Key elements extraction and traces comprehension using gestalt theory and the helmholtz principle ». In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, p. 478–482. IEEE.

Koffka, K. 1935. *Principles of Gestalt Psychology*. Coll. « International library of psychology ». Routledge.

Koskimies, K., T. Systa, J. Tuomi, et T. Mannisto. 1998. « Automated support for modeling oo software », *IEEE software*, vol. 15, no. 1, p. 87–94, IEEE.

Kuhn, A. et O. Greevy. 2006. « Exploiting the analogy between traces and signal processing ». In *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*, p. 320–329. IEEE.

Laddad, R. 2003. *AspectJ in Action : Practical Aspect-Oriented Programming*. Greenwich, CT, USA : Manning Publications Co.

Letovsky, S. 1986. « Cognitive processes in program comprehension ». In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, p. 58–79, Norwood, NJ, USA. Ablex Publishing Corp.

Lowe, D. 2012. *Perceptual organization and visual recognition*. T. 5. Springer Science & Business Media.

Maalej, W., R. Tiarks, T. Roehm, et R. Koschke. 2014. « On the comprehension of program comprehension », *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, p. 31, ACM.

MacQueen, J. 1967. « Some methods for classification and analysis of multivariate observations ». In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*. T. 1, p. 281–297. Oakland, CA, USA.

Medini, S., G. Antoniol, Y. G. Guéhéneuc, M. D. Penta, et P. Tonella. 2012. « Scan : An approach to label and relate execution trace segments ». In *2012 19th Working Conference on Reverse Engineering*, p. 135–144. IEEE.

Medini, S., V. Arnaoudova, M. Di Penta, G. Antoniol, Y.-G. Guéhéneuc, et P. Tonella. 2014. « Scan : An approach to label and relate execution trace segments », *J. Softw. Evol. Process*, vol. 26, no. 11, p. 962–995, John Wiley & Sons, Inc.

Müller, H. A., S. R. Tilley, et K. Wong. 1993. « Understanding software systems using reverse engineering technology perspectives from the rigi project ». In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research : software engineering-Volume 1*, p. 217–226. IBM Press.

Pelleg, D. et A. W. Moore. 2000. « X-means : Extending k-means with efficient estimation of the number of clusters ». In *Proceedings of the Seventeenth International Conference on Machine Learning*. Coll. « ICML '00 », p. 727–734, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Pirzadeh, H. 2012. « Trace abstraction framework and techniques ». Thèse de doctorat, Concordia University Montreal, Quebec, Canada.

Pirzadeh, H., L. Alawneh, et A. Hamou-Lhadj. 2009a. « Quality of the source code for design and architecture recovery techniques : Utilities are the problem ». In *2009 Ninth International Conference on Quality Software*, p. 465–469.

Pirzadeh, H., L. Alawneh, et A. Hamou-Lhadj. 2009b. « Quality of the source code for design and architecture recovery techniques : Utilities are the problem ». In *Proceedings of the 2009 Ninth International Conference on Quality Software*. Coll. « QSIC '09 », p. 465–469, Washington, DC, USA. IEEE Computer Society.

Pirzadeh, H. et A. Hamou-Lhadj. 2011a. « A novel approach based on gestalt psychology for abstracting the content of large execution traces for program comprehension ». In *Proceedings of the 2011 16th IEEE International Conference on Engineering of Complex Computer Systems*. Coll. « ICECCS '11 », p. 221–230, Washington, DC, USA. IEEE Computer Society.

———. 2011b. « A software behaviour analysis framework based on the human perception systems (nier track) ». In *Proceedings of the 33rd International Conference on Software Engineering*. Coll. « ICSE '11 », p. 948–951, New York, NY, USA. ACM.

Pirzadeh, H. et A. Hamou-Lhadj. 2011c. « A software behaviour analysis framework based on the human perception systems : Nier track ». In *2011 33rd International Conference on Software Engineering (ICSE)*, p. 948–951.

Pirzadeh, H., A. Hamou-Lhadj, et M. Shah. 2011. « Exploiting text mining techniques in the analysis of execution traces ». In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, p. 223–232. IEEE.

Radev, D. R., E. Hovy, et K. McKeown. 2002. « Introduction to the special issue on summarization », *Comput. Linguist.*, vol. 28, no. 4, p. 399–408, MIT Press.

Reiss, S. P. 2005. « Dynamic detection and visualization of software phases ». In *Proceedings of the Third International Workshop on Dynamic Analysis*. Coll. « WODA '05 », p. 1–6, New York, NY, USA. ACM.

Shafiee, A. 2013. « Phase flow diagram : A new execution trace visualization technique ». Thèse de doctorat, Concordia University.

Storey, M.-A. 2006. « Theories, tools and research methods in program comprehension : past, present and future », *Software Quality Journal*, vol. 14, no. 3, p. 187–208, Springer.

Systa, T. 2000. « Understanding the behavior of java programs ». In *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, p. 214–223. IEEE.

Watanabe, Y., T. Ishio, et K. Inoue. 2008. « Feature-level phase detection for execution trace using object cache ». In *Proceedings of the 2008 international workshop on dynamic analysis : held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, p. 8–14. ACM.