Université de Montréal

Analyse du comportement des programmes à l'aide des matrices d'adjacence

par
Samah Rached

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Octobre, 2005

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

Analyse du comportement des programmes à l'aide des matrices d'adjacence

présenté par:

Samah Rached

a été évalué par un jury composé des personnes suivantes:

Mostapha Aboulahamid
président-rapporteur

Yann-Gaël Guéhéneuc
directeur de recherche

Petko Valtchev
codirecteur

Stefan Monnier
membre du jury

Mémoire accepté le

RÉSUMÉ

La maintenance des programmes orientés objets est difficile car elle dépend de la compréhension du programme par les mainteneurs. Nous cherchons à faciliter la compréhension des programmes par la visualisation des données statiques et dynamiques. Nous utilisons une représentation graphique sous forme de matrices d'adjacence qui permet de visualiser les relations statiques et dynamiques entre les classes de grands programmes. Nous lions les données dynamiques avec des données statiques pour aider les mainteneurs à comprendre les relations entre les classes, à avoir une vue architecturale du système tout au long des investigations des mainteneurs, à identifier les fonctionnalités des programmes et à détecter les traitements répétitifs en vue d'avoir des abstractions de plus haut niveau.

Nous avons conçu un système, DRAM (Dynamic Relational Adjacency Matrix), qui utilise l'analyse dynamique pour générer des traces des programmes, nous l'enrichissons avec l'analyse statique puis développons différents algorithmes de conversion des données sous forme de matrices d'adjacence : groupement par classes et/ou méthodes, recherche de fonctionnalités du programme. Enfin, nous expérimentons notre approche sur JUnit, JHotdraw pour montrer l'intérêt et les limites de notre approche.

Mots-clés : Analyses dynamiques, analyses statiques, matrices d'adjacences, fonctionnalités du programme, regroupement par classes/méthodes.

ABSTRACT

The maintenance of the oriented object programs is difficult because it depends on the comprehension of the program by maintainers. We want to ease the comprehension of programs by the visualization of static and dynamic data. We use a graphic representation in the form of adjacency matrices to visualize the static and dynamic relations among classes of large programs. We link dynamic data with static data to help maintainers to understand the relations among classes, to have a architectural view of programs throughout the maintainers' investigations, to identify the functionalities of the programs, and to detect the repetitive treatments to obtain an higher abstraction levels.

We conceive a system, DRAM (Dynamic Relational Adjacency Matrix), which uses dynamic analysis to generate traces of a program, we enrich this with the static analysis, then develop various algorithms of conversion of the data in the form of adjacency matrix : grouping by classes and/or methods, search for functionalities of the program. Finally, we experiment our approach on JUnit, JHotdraw to show its interest and its limits.

Keywords : Dynamic analysis, static analysis, adjacency matrix, functionalities of the programs, grouping by classes/methods.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

CHAPITRE 1

INTRODUCTION

Les systèmes logiciels sont de plus en plus grands et complexes. La maintenance de ces systèmes a un coût élevé [Antoniol et Guéhéneuc, 2005] à cause de la difficulté à comprendre le code, les fonctionnalités des programmes ou à retrouver l'architecture des systèmes. Les mainteneurs, qui sont souvent peu familiers avec les programmes qu'ils doivent maintenir, font face à des difficultés pour la compréhension des structures et des relations dynamiques qui existent pendant l'exécution des programmes [Pauw *et al.*, 1993]. L'identification des choix réalisés lors de la conception et de l'implémentation des programmes en vue de corriger des bogues, d'ajouter de nouvelles fonctionnalités ou d'améliorer la réutilisabilité reste compliquée.

Soloway *et al.* [Soloway *et al.*, 1988] caractérisent les activités de compréhension des logiciels comme des épisodes d'enquête, pendant lesquels un mainteneur lit le code source d'un programme, pose une question sur ce code, conjecture une réponse et recherche dans le code et la documentation la confirmation de sa conjecture.

Cependant, Johnson et Erdem [Johnson et Erdem, 1995] expliquent que "la recherche dans la compréhension de logiciel est sujette à erreur ; un mainteneur ne sait pas toujours où chercher l'information pour soutenir ses conjectures"[1]. En plus, la documentation existante n'est souvent plus à jour et l'architecture originale du

[1] "Search in software understanding is very error-prone ; a maintainer does not always know where to look for information to support their conjectures".

programme n'est plus disponible ou n'est plus à jour à cause des changements faits par d'autres mainteneurs.

Par conséquent, les outils pour aider les mainteneurs dans leur compréhension du code sont devenus essentiels. Parmi ces outils, les outils de visualisation permettent de faciliter la compréhension des programmes qui sont à l'étude. L'affichage visuel permet aux mainteneurs d'étudier les aspects multiples de problèmes complexes. Ware [Ware, 2000] affirme que "la visualisation fournit une capacité de comprendre des quantités énormes de données"[2] et Harel [Harel, 1992] déclare que "l'utilisation de formalismes visuels appropriés peut avoir un effet spectaculaire sur des ingénieurs et des programmeurs.".

1.1 Utilisation des outils de visualisation

Les domaines de la visualisation scientifique et de la visualisation des programmes [Kimelman et Ngo, 1991, Nielson *et al.*, 1990, Upson *et al.*, 1989] ont démontré à plusieurs reprises que la manière la plus efficace de présenter de grands volumes de données aux utilisateurs est le mode visuel continu. L'affichage visuel continu permet aux utilisateurs une assimilation rapide des informations et l'identification des tendances et des anomalies.

Plusieurs définitions de la visualisation de programmes ont été proposées dans la littérature. Le travail de Bassil et Keller [Bassil, 2000] a largement couvert cet aspect. Nous retenons trois définitions de la visualisation de programmes qui nous paraît recouvrir les principaux aspects de la visualisation. La première est de Do-

[2] "Visualization provides an ability to comprehend huge amounts of data"

mingue *et al.* [Domingue *et al.*, 1992] : “la visualisation de logiciels décrit des systèmes qui utilisent des médias visuels (et autres), et ceci pour augmenter la compréhension d'un programmeur du travail effectué par une autre personne (ou bien par lui-même)”. La deuxième est celle de Price *et al.* [Price *et al.*, 1993] : “la visualisation de logiciel est l'utilisation du métier de la typographie, du design graphique, de l'animation et de la cinématographie avec la technologie moderne de l'interaction homme–machine et ceci pour faciliter la compréhension humaine et l'utilisation pertinente des logiciels informatiques”. Finalement, celle présentée par Muthukumarasamy et de Stasko [Muthukumarasamy et Stasko, 1995] est la suivante : “la visualisation de logiciel est l'utilisation des techniques de la visualisation et de l'animation pour aider les personnes à comprendre les caractéristiques et les exécutions de programmes informatiques.”

En conclusion, la visualisation d'un programme facilite la capacité de former une image mentale du programme visualisé et d'avoir une représentation plus lisible et facile à comprendre. Cette représentation permet une meilleure compréhension des programmes et une assimilation rapide et aisée des différentes fonctionnalités des programmes.

La visualisation des logiciels se divise en deux catégories [Stasko *et al.*, 1998] :

- la visualisation des programmes : c'est la visualisation du code du programme ou de ses structures de données sous une forme statique ou dynamique, par exemple l'utilisation d'Eclipse permet entre autres d'afficher les structures et les hierarchies des classes ;
- la visualisation des algorithmes : c'est la visualisation des abstractions de haut niveau qui décrivent le logiciel. Un bon exemple est l'animation d'algorithme,

qui est la visualisation dynamique d'un algorithme et de son exécution. Ceci a été principalement fait pour expliquer le fonctionnement interne des algorithmes, par exemple le fonctionnement des algorithmes de tris.

La visualisation des algorithmes a perdu de l'importance [Lanza, 2003], principalement parce que l'avancement dans le matériel informatique et la possibilité d'utiliser des bibliothèques standards contenant de tels algorithmes ont déplacé l'attention loin de l'implémentation de tels algorithmes. C'est pourquoi, dans notre approche, nous nous intéressons à la visualisation des programmes. Nous proposons l'utilisation des matrices d'adjacence pour représenter les données statiques et dynamiques des programmes et pallier aux limites de techniques de visualisation existantes.

1.2 Limites des approches existantes

Les outils de visualisation des programmes utilisent souvent les graphes nœuds–liens ou les diagrammes d'UML pour visualiser les données des programmes étudiés. Ceci est dû principalement au fait que ces représentations sont les plus intuitives et qu'elles sont bien comprises et connues des mainteneurs [Ghoniem *et al.*, 2005a].

1.2.1 Diagrammes UML

UML [Object Management Group, Inc., 2003] est un langage graphique dérivé de plusieurs notations antérieures généralement employé pour décrire la conception des logiciels orientés objets. UML utilise plusieurs types de diagrammes pour décrire un système logiciel. Le diagramme le plus populaire est le diagramme de classes qui est utilisé pour représenter la structure statique des classes et les rela-

tions qui existent entre elles [Rumbaugh *et al.*, 1999]. Les classes sont représentées par des rectangles contenant une liste d'attributs et d'opérations. Il y a deux types de relations : les relations d'héritage (qui représente les relations de généralisations–spécialisations) et les associations. Un diagramme peut être vu comme un graphe où les nœuds sont représentés par les classes et les liens par les relations.

Dans son travail, Seeman [Seemann, 1997] présente un algorithme pour la mise en page automatique des diagrammes de classes UML en utilisant une extension de l'algorithme de Sugiyama [Sugiyama *et al.*, 1981] ainsi que des techniques de dessin orthogonal. L'algorithme de mise en page différencie les types de liens (héritage, associations) en les traitant de différentes manières. La stratégie adoptée est de placer d'abord les classes impliquées dans des relations d'héritage dans une structure hiérarchique, puis de placer les classes restantes tout en préservant la structure de base. Seeman s'est basé sur l'algorithme de Sugiyama qui est une technique largement répandue pour tracer les graphes orientés [Sugiyama *et al.*, 1981] et qui a été bien analysée [Eades et Sugiyama, 1990, Frick, 1997] et améliorée de plusieurs manières dans les années récentes [Gansner *et al.*, 1993, Mutzel, 1997]. Seeman a utilisé l'algorithme de Sugiyama avec quelques modifications pour le placement des classes impliquées dans des relations d'héritage. Les classes restantes sont placées en utilisant une extension de l'algorithme de base avec un algorithme de placement incrémental jusqu'à ce que toutes les classes soient placées. Ce placement peut influencer le premier placement par Sugiyama. Ensuite, une mise en page orthogonale[3] des liens d'association est appliquée. Seeman a eu de bons résultats pour des programmes orientées objets avec de nombreuses relations d'héritage mais des

[3]Orthogonal Drawing

résultats peu satisfaisant quand le programme contient de nombreuses associations. Eichelberger [Eichelberger, 2002] utilise aussi l'algorithme de Sugiyama en étendant le travail de Seeman mais sans réussir à complètement supprimer le problème quand le programme contient de nombreuses associations.

1.2.2 Diagrammes nœuds–liens

La visualisation des graphes s'est principalement concentrée sur les diagrammes nœuds–liens. Cette métaphore a été déclinée sous diverses formes tant en 2D qu'en 3D. De très nombreux travaux issus de la communauté de graphes portent sur diverses techniques de placement de nœuds en tenant compte de règles d'esthétique, telles que la minimisation du nombre d'intersections, la minimisation du rapport entre le lien le plus long et le lien le plus court et la révélation de symétries [Battista *et al.*, 1999]. La majorité de ces travaux s'est attachée à l'optimisation des algorithmes répondant à ces critères esthétiques, tandis que des études moins nombreuses se sont consacrées à leur validation expérimentale sur le plan cognitif [Ghoniem *et al.*, 2004]. Cependant, les diagrammes nœuds–liens ne passent pas bien à l'échelle pour l'affichage de grands graphes voir par exemple figure 1.1 : leur mise en page devient lente et instable et les graphes sont rapidement illisibles quand la taille du graph et la densité des liens augmentent.

1.3 Notre approche

Pour remédier aux problèmes soulevés par les représentations UML et les représentations nœuds–liens, l'utilisation d'approches alternatives semble intéressante. Nous nous inspirons des travaux de Ghoniem *et al.* [Ghoniem *et al.*, 2004, Ghoniem

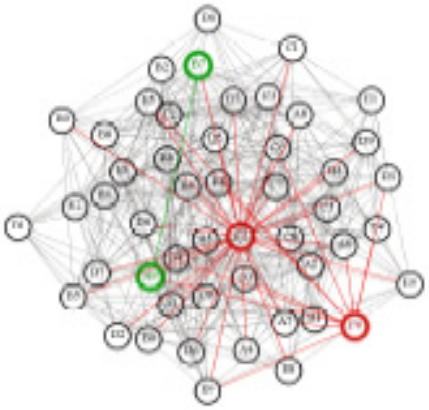

FIG. 1.1 – diagramme nœuds–liens d'un graphe non orienté comportant 50 sommets et 400 arcs. construit à l'aide du programme "neato" [AT&T Labs Research., 2004]

et al., 2005a] pour représenter les programmes orientés–objets sous forme de matrices d'adjacence. Nous proposons une approche pour aider les mainteneurs à comprendre l'exécution des programmes orientés objet en utilisant une représentation graphique des données statiques et dynamiques d'un programme sous forme de matrices d'adjacence. Une matrice d'adjacence est une matrice NxN où N est le nombre de sommets et où chaque ligne et colonne présente un sommet. Une cellule [i,j] dans la matrice représente un lien entre le sommet i et le sommet j (voir figure 1.2). La densité de la matrice dépend du nombre de liens qui existent entre les sommets.

Les matrice d'adjacence semblent une approche alternative intéressante pour la visualisation des programmes pour :

– cibler l'information à comprendre : représenter le programme à étudier sous forme matricielle permet d'avoir une vision plus claire des relations qui existent entre ses classes. De plus, notre approche permet de regrouper les informations concernant des classes afin de voir l'évolution de l'exécution du pro-

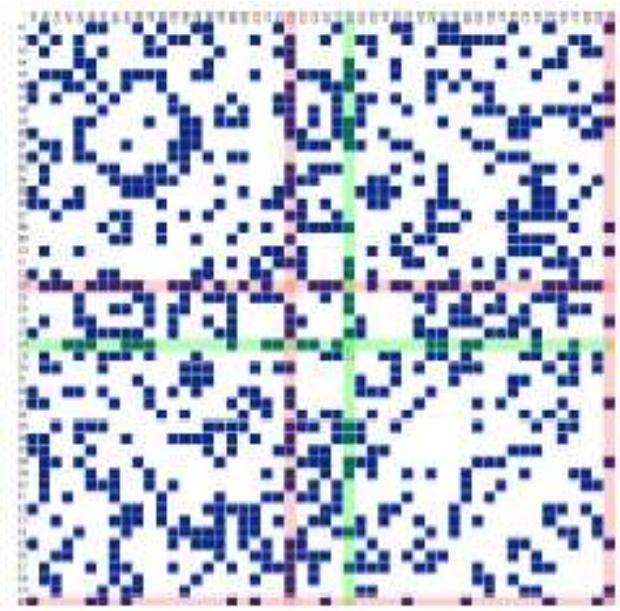

FIG. 1.2 – Matrice d'adjacence contenant 50 sommets et 400 liens réalisée par le programme VisAdj [Ghoniem *et al.*, 2004]

gramme dans le temps ;

- avoir une vue architecturale du programme : la compréhension des relations statiques et dynamiques qui existent entre les différentes classes du programme permet l'identification des choix réalisés lors de la conception et de l'implémentation.
- identifier les fonctionnalités[4] du programme : permettre aux mainteneurs d'identifier les classes du programme qui coopèrent pour réaliser une tâche donnée du programme ;
- détecter les traitements répétitifs et les abstractions éventuelles : la représentation de l'évolution de la trace dans le temps permet de remarquer les traitements répétitifs et de décrire l'architecture des programmes avec des abstractions de plus haut niveau.

[4]Le terme fonctionnalité décrit le mot anglais "Feature".

Pour atteindre ces objectifs, nous utilisons les matrices d'adjacence pour visualiser les données statiques et dynamiques qui portent sur les différentes classes de programmes. Nous avons implémenté un outil DRAM qui permet de filtrer les données à visualiser des traces d'exécution et de découper ces traces filtrées pour visualiser le déroulement dans le temps de l'exécution des programmes et enfin de comparer des traces d'exécution pour identifier les fonctionnalités du programme. Nous utilisons deux études de cas (JUnit et JHotDraw) pour illustrer notre approche et son implémentation.

1.4 Organisation du mémoire

Ce mémoire est divisé en 7 sections : l'état de l'art, la présentation des matrices d'adjacences, la génération des traces et des données à visualiser, la présentation des traces sous forme de matrices d'adjacence, l'implémentation et l'expérimentation et enfin la conclusion.

Dans un premier temps nous présentons au chapitre 2 l'état de l'art, où nous passons en revue quelques travaux représentatifs qui traitent de la visualisation des logiciels. Dans le chapitre 3, nous présentons les matrices d'adjacence, leur utilisation, leur points forts et leur faiblesses. Ensuite, dans le chapitre 4, nous traitons des données à visualiser et de l'utilisation de l'analyse dynamique et de l'analyse statique pour la génération des traces à visualiser. Dans le chapitre 5, nous présentons les données des traces sous forme de matrices d'adjacence et les manipulations faites sur les données pour une meilleure compréhension, notamment le découpage de la trace et la recherche des fonctionnalités dans les programmes. Par

la suite, dans le chapitre 6, nous décrivons l'implémentation et l'application de notre approche sur deux études de cas pour évaluer les résultats obtenus. Finalement, dans le chapitre 7, nous concluons par un récapitulatif des aspects majeurs de ce travail, les contributions majeures de ce mémoire, les limitations de l'approche proposée et les travaux futurs.

CHAPITRE 2

ÉTAT DE L'ART

Plusieurs techniques et outils de visualisation sont disponibles pour aider les mainteneurs dans la compréhension des programmes. Ces outils peuvent être des outils d'analyse, de conception, de test, de déboggage ou de maintenance. Plusieurs travaux se sont intéressés aux différents aspects de la visualisation, des différents problèmes qui y existent et ont présenté des solutions. La plupart de ces travaux se sont basés sur des approches pour aider les mainteneurs dans la compréhension des programmes. Les problèmes résolus grâces aux approches utilisant des techniques de visualisation les plus représentatifs sont : la détection du code responsable des fonctionnalités, la compréhension des programmes, l'aide au déboggage, la recherche des patrons de conception. Le tableau 2.1 résume ces différents approches. Dans ce chapitre, nous présentons les travaux recensés dans le tableau 2.1 leurs contributions et leurs limitations.

2.1 Détection des fonctionnalités

Lukoit et *al.* présentent TraceGraph [Lukoit *et al.*, 2000], un outil de visualisation qui permet la localisation rapide de code dans de grands programmes en cours de modification. Il fournit une visualisation de la trace qui est produite en instrumentant un programme cible et en exécutant des tests pour distinguer facilement les changements d'exécution. Le mainteneur peut exécuter la fonctionnalité à laquelle il s'intéresse et visualiser immédiatement les variations des exécutions

Article	Fonctionnalité	Compréhension des programmes	Patrons	Déboggage
TraceGraph : Immediate Visual Location of Software Features [Lukoit *et al.*, 2000]	X			
Visualizing the Behavior of Object-Oriented Systems [Pauw *et al.*, 1993]		X		
Bridging the Gulf Between Code and Behavior in Programming [Lieberman et Fry, 1995]				X
Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information [Richner et Ducasse, 1999]		X		
A Hierarchy of Dynamic Software Views : from object-interactions to feature-interactions [Salah et Mancoridis, 2004]	X			
Pattern Visualization for Software Comprehension [Schauer et Keller, 1998]			X	
Almost : Exploring Program Traces [Renieris et Reiss, 1999]		X		
Using Multilevel Call Matrices in Large Software Projects [van Ham, 2003]		X		

TAB. 2.1 – Récapitulatif des travaux étudiés dans l'état de l'art

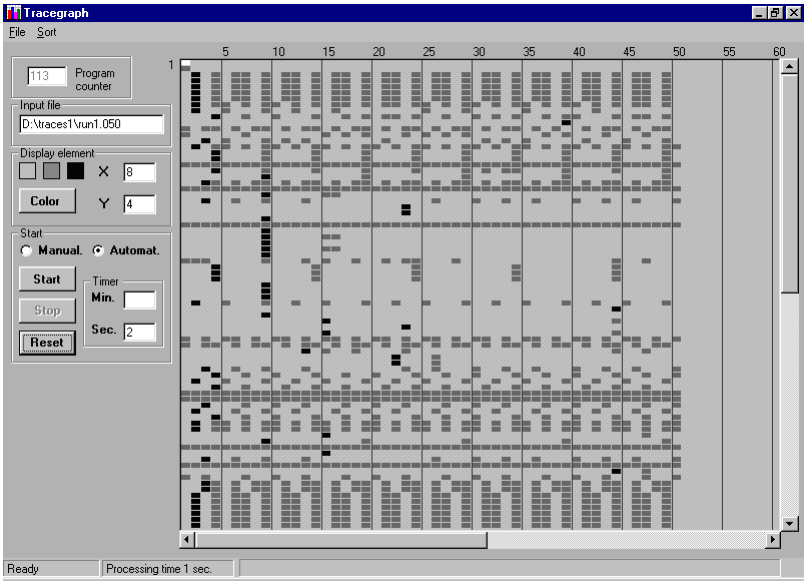

FIG. 2.1 – Affichage de TraceGraph [Lukoit *et al.*, 2000] à partir du cas d'étude *Joint STARS* : échange de messages entre deux processus

du programme. En effet, TraceGraph surveille le programme cible pendant son exécution et génère des fichiers de trace qui sont analysés en continu et qui peuvent être affichés en temps réel comme la trace d'un oscilloscope qui se prolonge lentement de gauche à droite pendant que le programme s'exécute (figure 2.1). Ainsi, TraceGraph permet au mainteneur d'essayer différentes entrées dans le programme et d'obtenir une image rapide montrant la réponse du programme.

Les auteurs affirment que l'outil tire profit de la capacité de l'œil humain à identifier des patrons dans de grandes quantités d'informations présentées sous forme graphique. L'outil fournit une réponse visuelle rapide aux opérations des mainteneurs. Cependant, comme sur la figure 2.1, cette affirmation n'est pas évidente.

Toutefois, TraceGraph souffre de quelques limitations. D'abord, il ne peut être utilisé que pour localiser les fonctionnalités que l'utilisateur du programme peut contrôler. En effet, la plupart des programmes contiennent une quantité significa-

tive de code commun qui est toujours exécuté avec chaque test non-trivial tandis qu'un mainteneur a besoin de localiser une partie bien spécifique du code exécuté (en général, hors du code commun mais entrelacé avec le code commun). En second lieu, comme pour toutes les techniques d'analyse dynamiques, les résultats dépendent des cas d'essai utilisés. Finalement, il ne permet de fournir qu'un point de départ pour l'exploration du code. Le mainteneur doit étudier chaque classe du programme et comprendre comment elle interagit avec le reste du programme.

Salah and Mancoridis [Salah et Mancoridis, 2004] ont proposé une approche qui repose sur une hiérarchie de vues dynamiques qui est construite par analyse des traces d'exécution pour identifier les fonctionnalités dans les programmes et assurer leurs traçabilités dans le code. Cet ensemble de vues associe les fonctionnalités au code et établit les interactions entre les parties responsables des diverses fonctionnalités automatiquement.

L'approche est centrée sur le concept de marquage des traces d'exécution, qu'utilisent les mainteneurs pour définir les fonctionnalités du programme. Le marquage de la trace est établi manuellement pendant l'exécution du programme en indiquant le début et la fin de la trace associée à une fonctionnalité.

L'ensemble des vues fournit des niveaux de détails variés que les mainteneurs peuvent exploiter. L'outil permet la navigation à travers les différentes vues aussi bien que l'extension des nœuds et des liens du graphe des vues pour permettre aux mainteneurs de voir les détails spécifiques des données codées dans les nœuds et les liens. Selon la vue, un nœud peut être une fonctionnalité, un fil d'exécution[1], une classe, un objet ou une méthode. De même, un lien encode le type de relation et les objets et les classes qui participent à cette relation.

[1] "thread" en anglais.

Il y a plusieurs niveaux d'abstraction dans la hiérarchie des vues. Le niveau le plus bas représente les événements d'exécution qui sont générés durant l'exécution de la trace. Le 2^{e} niveau d'abstraction, "object-interaction view", inclut les objets exécutés et les relations qui existent entre ces objets. Ce niveau est la base des niveaux plus abstraits. Le 3^{e} niveau, "class-interaction view", une abstraction du niveau précèdent, inclut les classes et les relations d'exécution qui existent entre elles. Le 4^{e} niveau, "feature-interaction view", illustre les interactions qui existent entre les fonctionnalités du programme. Elle est dérivée à partir de la vue "object-interaction" automatiquement. Enfin, la dernière vue, "feature-implementation view", est une correspondance entre les fonctionnalités du programme et les classes qui les implémentent. Cependant, La lisibilité des différentes vues devient difficile si le nombre de classes ou d'objets est très important.

2.2 Compréhension des programmes

De Pauw *et al.* [Pauw *et al.*, 1993] présentent un système pour la présentation visuelle dynamique du comportement des programmes orientés objets afin de remédier aux difficultés de la compréhension, de la réutilisation, de la correction et du réglage de ces programmes. Les auteurs présentent des vues du comportement des programmes objets et une architecture pour la création et l'animation de ces vues (voir l'exemple de vue figure 3.2 qui représente la matrice d'appel inter-class qui fournit des informations cumulatives et quantitatives. Les carrés colorés dans cette visualisation représente le nombre d'appels d'une classe sur l'axe vertical à une classe sur l'axe horizontal. La clef de couleur le long du fond indique le nombre relatif des appels. Les lignes verticales indiquent les classes qui s'appellent par beaucoup

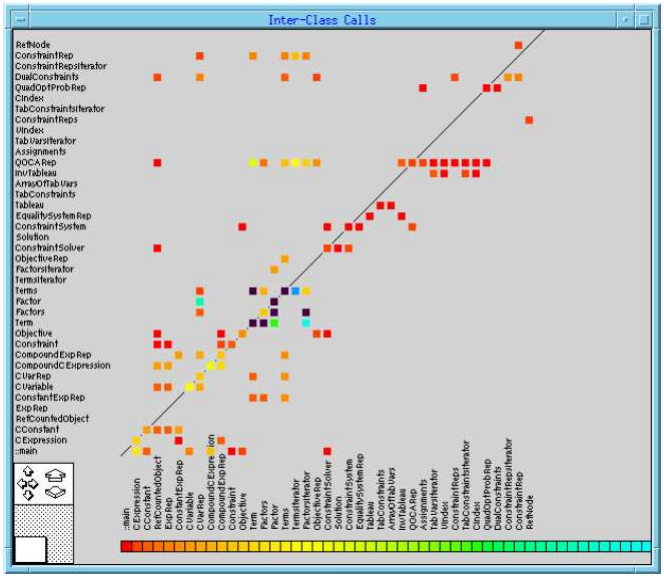

FIG. 2.2 – Matrice Inter-class [Pauw *et al.*, 1993]

d'autres classes. Les lignes verticales au-dessus de la diagonale tendent à indiquer les classes abstraites principales dans le framework ou la bibliothèque. Les lignes horizontales indiquent une classe qui appelle beaucoup d'autres classes. Les groupes près de la diagonale peuvent indiquer les classes ou les sous-ensembles étroitement couplés). Ils décrivent des techniques pour instrumenter les programmes objets indépendamment de la plateforme utilisée, un langage indépendant pour surveiller leur exécution et une structure pour découpler l'exécution d'un programme de sa visualisation. L'ensemble des vues qu'ils présentent inclut des affichages qui groupent les classes par rapport au degré avec lequel elles agissent l'une sur l'autre, des histogrammes variés qui montrent les instances des classes et leurs niveaux d'activité et des matrices de references croisées indiquant le degré de diverses formes de références inter- et intra-classes. Ce travail nous a inspiré pour l'utilisation des matrices d'adjacence et nous y avons ajouté la manipulation des données pour une meilleure interaction avec les programmes étudiés.

Richner and Ducasse [Richner et Ducasse, 1999] présentent un environnement qui permet la génération de vues ajustables[2] de programmes orientés objets à partir de données statiques et dynamiques. Leur approche est basée sur une combinaison de requêtes en utilisant un langage de programmation logique (Prolog) qui permettent de créer des abstractions de plus haut niveau et de produire des vues présentant ces abstractions.

L'utilisation des données statiques et dynamiques permet de répondre à une gamme étendue de questions au sujet d'un programme : là où les données statiques sont moins précises elles sont complétées par les données dynamiques ; les données statiques sont employées pour grouper les données dynamiques dans des composants plus maniables et les données dynamiques fournissent des réponses aux questions qui ne peuvent pas être répondues avec des données statiques seulement.

Les auteurs ont employé un langage de programmation logique pour interroger les données statiques et dynamiques et pour générer une gamme de vues à un niveau plus élevé d'abstraction. Les données statiques sur le programme sont décrites par un méta-modèle qui réifie les notions orientées objets. Ces données structurelles sont complétées par l'information comportementale décrites sous forme de traces d'exécution du programme. Ainsi, en se basant sur ce modèle, la recherche dans le code est orientée à travers un processus itératif semblable à celui décrit par Murphy [Murphy et Notkin, 1997].

Différentes vues de plus haut niveau peuvent être générées. Une première vue est générée qui répond à quelques questions comme la communication entre les différentes instances du scénario exécuté. Cette vue peut être raffinée en faisant

[2] "tailorable" en anglais.

des regroupements qui permettent d'avoir des vues de granularités variables allant du regroupement des ensembles de classes à l'interaction entre les instances. Ces vues générées sont comme des catalyseurs pour générer des questions au sujet du programme étudié et pour aider au processus de recouvrement de la conception. Cependant, il n'y a pas d'échange d'informations entre les différentes vues.

Renieris and Reiss [Renieris et Reiss, 1999] présentent un outil pour visualiser et explorer l'exécution de la trace d'un programme pour aider les mainteneurs dans la compréhension des programmes. Les données sont collectées et interrogées pour fournir des informations lisibles fournissant ainsi la trace à visualiser. La trace est affichée sous forme linéaire et sous forme d'une spirale. La spirale fait une meilleure utilisation de l'écran en fournissant une image plus dense que des méthodes linéaires et les méthodes linéaires permettent un meilleur "zoom in" et "zoom out" visuel. Le code du programme est représenté par une vue du texte codé pour procurer un affichage actif pour aller des données affichées au code source. Les vues sont reliées avec un modèle MVC pour permettre la coordination entre les différentes vues (figure 3.3).

Cependant, il n'est pas évident de repérer les boucles ou les patrons avec l'affichage en spirale. De plus, ce mode d'affichage n'est pas commode pour la visualisation et la spirale ne peut pas être enrichie avec les noms des fonctions dans le graphe. Enfin, la coordination entre les différentes vues reste difficile.

van Ham [van Ham, 2003] a étudié l'utilisation des matrices d'appels en tant qu'aides visuelles dans la gestion de grands projets logiciels. Il présente un certain nombre de problèmes de visualisation, en utilisant comme exemple un très

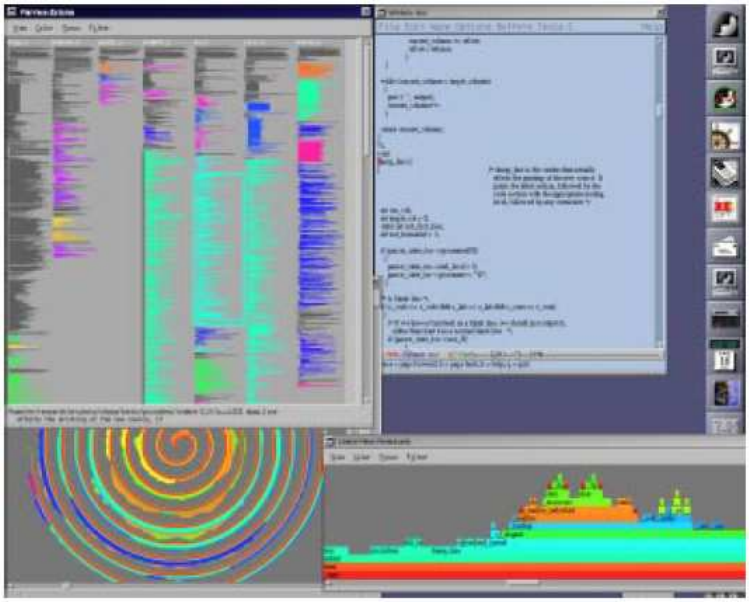

FIG. 2.3 – Ensembles des 4 vues de la trace du programme [Renieris et Reiss, 1999]

grand programme en cours de développement : un système médical de Philips qui implémente une plateforme d'image médicale.

L'auteur a utilisé les matrices d'appels car elles ont un certain nombre d'avantages par rapport aux diagrammes nœuds–liens quand l'objet principal d'intérêt est le lien (l'appel) au lieu du nœud (les objets origines et destinations d'un appel). Les diagrammes nœuds–liens ne peuvent habituellement pas fournir des dispositions significatives et lisibles pour plus de cent nœuds à la fois et ont un problème avec le visionnement à multiniveaux et ne sont pas toujours stables : l'addition ou le déplacement d'un simple nœud peut mener à une disposition radicalement différente. Par contre, les matrices d'appels fournissent une representation visuelle uniforme puisque le seul élément visuel utilisé dans la représentation est une cellule de la matrice. Elles ont une structure recursive qui permet de construire une visualisation à plusieurs niveaux. Enfin, ils permettent une stabilité de la disposition.

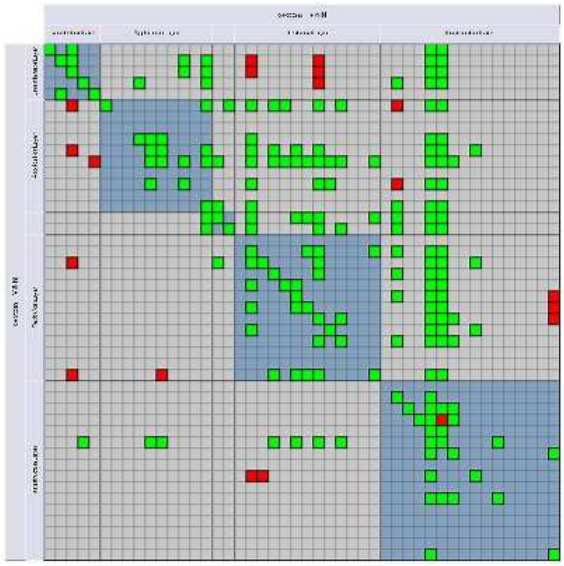

FIG. 2.4 – Représentation matricielle [van Ham, 2003]

L'auteur soutient que la construction d'une matrice d'appels n'est pas difficile en tant que telle, mais une conception soigneuse est exigée pour un résultat optimal. Ainsi, il prend en compte la subdivision de la matrice pour pouvoir afficher différents niveaux d'abstraction d'un programme. Le nombre de niveaux d'abstractions à afficher est limité par les moyens d'affichage et en pratique, l'affichage de deux niveaux d'abstractions est un bon compromis entre afficher les détails et maintenir la vue d'ensemble. Ceci permet d'utiliser le nombre des sous-composants des composants du programme actuellement activés comme mesure de la taille d'affichage. Il a utilisé la structure recursive des matrices d'appels pour fournir un zoom sémantique entre les niveaux d'abstraction, qui facilite la maintenance du contexte étudié et une compréhension immediate des relations qui existent entre les différents niveaux de visualisation. Enfin, il a exploité le fait que la visualisation par matrices offre un espace suffisant pour afficher des informations additionnelles, telles que les attributs des nœuds.

L'auteur a eu des résultats satisfaisant. En effet, il peut repérer les appels non désirés entre sous-systèmes, determiner les dépendances entre les différents composants, avoir une stabilité de la mise en page pour des relations complexes.

2.3 Déboggage

Le travail de Lieberman et Fry [Lieberman et Fry, 1995] présente un environnement de déboguage des programmes conçu pour aider les mainteneurs à mieux comprendre la correspondance entre le code et l'exécution d'un programme. Cet environnement aide un mainteneur à assortir ses attentes au sujet du comportement du code avec le comportement réel de ce code et à combler la différence entre ce que Donald Norman [Norman, 1986] a nommé le "fossé d'évaluation"[3] et le "fossé d'exécution"[4] entre les attentes de l'utilisateur, l'exécution de la commande soumise à l'ordinateur et le résultat produit.

La solution adoptée par les auteurs et implémenté dans leur outil ZStep 94 fournit une structure réversible de contrôle qui permet de garder un historique d'exécution complet et incrémental de l'exécution du programme. Lieberman et Fry se sont concentrés sur les détails de réduire au minimum l'espace requis pour l'historique et de dépister les effets secondaires des structures des données, dont tout les deux sont importants, mais secondaire aux aspects interface utilisateur. Il est non seulement important de relier les variables à leurs valeurs précédentes mais également de retourner à une vue cohérente de l'interface utilisateur, en incluant le code statique, les données dynamiques et la sortie graphique de sorte que l'utilisateur retrouve l'image fidèle de l'exécution du programme. ZStep94 facilite

[3] "Gulf of Evaluation" en anglais
[4] "Gulf of Execution" en anglais

grandement la compréhension du comportement des programmes. Cependant, les mainteneurs doivent vérifier leurs conjonctures en ce qui concerne le comportement du programme à la main. En plus, les requêtes utilisées ne sont pas extensibles et ZStep94 a été conçu pour LISP, ce qui rend son utilisation pour les programmes Java impossible.

2.4 Recherche de patrons

Schauer and Keller [Schauer et Keller, 1998] présentent un outil de prototypage pour visualiser les patrons de conception existants dans la littérature, les patrons de conception génériques et les solutions de conception ad hoc (de circonstance), donnant la retro-conception du code source d'un système.

L'idée fondamentale est de réifier les patrons comme des composants à part entière pour les utiliser en tant qu'artifacts tangibles de conception de logiciels. Ces composants de conception peuvent servir de base à un procédé compositionnel de développement de programmes, dans lequel les composants de conception de base sont assemblés pour fournir d'autres composants, qui a leur tour peuvent servir comme composants de base à une conception plus complète et plus complexe. L'évolution des composants de conception peut être tracée, permettant d'avoir un historique du programme et de son évolution architecturale.

Les auteurs se sont concentrés sur la visualisation de la conception de logiciel basée sur les patrons et leur raisonnement. Ils ont envisagé des techniques et des outils qui organisent le programme autour des patrons, leur visualisation et le zoom in et le zoom out des composants formels et informels de ces patrons. L'outil traite quatre questions principales : la retro-conception du code source dans le but

d'extraire un modèle à partir du code source ; le referentiel de conception utilisé pour le stockage, la manipulation et l'interrogation des patrons de conception abstraits et des patrons de conception implémentés dans le modèle du code source ; la représentation de la conception utilisé pour la visualisation et l'amélioration interactive des patrons du code source, des patrons de conception abstraits et des patrons implémentés ; et enfin le regroupement des artefacts de la conception[5] dans le but d'aider à structurer un diagramme de classes pour devenir un "diagramme de patron" en envisageant d'utiliser trois techniques : automatic pattern clustering, manual pattern clustering, et semi-automatic pattern clustering. Cependant, les auteurs n'ont utilisé que l'analyse statique pour leur étude.

En conclusion, Bassil and Keller [Bassil et Keller, 2001] présentent une étude pour évaluer les outils de visualisation des programmes en menant un sondage auprès d'utilisateurs qui manipulent ce type d'outils. Ce sondage prend en compte divers aspects des outils de visualisation : aspects fonctionnels, pratiques et cognitifs en plus des aspects d'analyse de code que les utilisateurs peuvent rechercher dans ces outils.

En général, les utilisateurs sont satisfaits des résultats obtenus à partir des outils de visualisation, notamment en ce qui concerne le gain de temps dans la réalisation des taches spécifiques et une meilleure compréhension des programmes visualisés. Cependant, de nombreux aspects et améliorations de ces outils de visualisation en l'état actuel ont été indiqués par les participants. Les améliorations les plus demandés sont l'intégration dans les outils de visualisation d'outils spécifique comme un générateur de code, la possibilité d'importer/d'exporter vers d'autres formats

[5] "design clustering" en anglais

d'outils de visualisation, doivent être plus adaptés au(x) système(s) de compilation utilisé(s) pour le développement de code, ajouter plus de fonctionnalités...

D'après cette analyse de l'état de l'art, il ressort que les approches utilisées lors de la visualisation des programmes servent principalement à résoudre les problèmes de la compréhension des programmes et de la détection des fonctionnalités du programme (voir tableau 2.1). Dans notre approche, nous essayons d'atteindre les mêmes objectifs car ils nous paraissent les plus pertinents pour aider les mainteneurs dans la compréhension des programmes. Cependant, nous abordons le problème de manière différente. Nous proposons l'utilisation des matrices d'adjacence qui nous semble un moyen très intéressant pour l'affichage de gros volumes de données tout en remédiant aux problèmes des approches existantes (présentations nœuds-liens, diagrammes UML) à savoir, l'illisibilité des représentations, chevauchement... De plus, nous proposons un ensemble d'algorithmes de filtrage et de manipulation de données qui permet une meilleure compréhension des programmes qui sont à l'étude.

CHAPITRE 3

MATRICES D'ADJACENCES

3.1 Définition et présentation

La visualisation matricielle [Ghoniem *et al.*, 2005b] des graphes repose d'un point de vue théorique sur la représentation d'un graphe par sa matrice d'adjacence. Cette technique se fonde sur la propriété qu'un graphe peut être représenté par sa matrice de connectivité, qui est une matrice NxN, où N est le nombre de sommets dans le graphe et où chaque ligne ou colonne dans la matrice représente un sommet. Quand deux sommets V_i et V_j sont liés, le coefficient m_{ij} correspondant dans la matrice est placé à 1, autrement il est placé à 0. En plus, un codage peut être utilisé quand le poids des liens est considéré : plus le poids est grand, plus le coefficient m_{ij} peut avoir une grande valeur, par exemple.

Les matrices d'adjacence sont utilisées depuis longtemps pour représenter des graphes en mathématiques et dans la théorie des graphes [Buckley et Haray, 1990, Battista *et al.*, 1999]. À la différence des diagrammes nœuds–liens, les représentations des graphes qui sont basées sur les matrices ne souffrent pas des chevauchement des nœuds et des liens qui peuvent rendre les graphes nœuds–liens illisibles, voir par exemple figure 3.1. Pratiquement, chaque lien dans le graphe peut être vu séparément. Pour les graphes dirigés, les colonnes représentent l'origine du lien et les rangées représentent leur sommet d'arrivée par conventions. La densité des matrices dépend des nombres de liens qui existent entre les sommets.

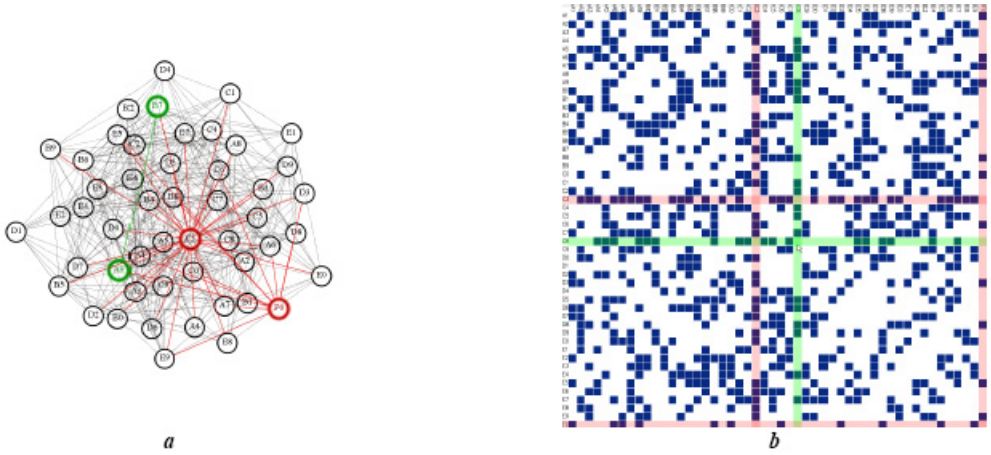

FIG. 3.1 – Deux représentations du même graphe non orienté comportant 50 sommets et 400 arcs. Le diagramme noeuds–liens (a) est construit à l'aide du programme "neato" [AT&T Labs Research., 2004] et la représentation matricielle (b) est construite à l'aide de l'outil VisAdj [Ghoniem *et al.*, 2005b].

Les matrices d'adjacences offrent plusieurs avantages : avec cette technique, autant de liens peuvent être montrés que la résolution d'affichage du matériel le permet. Un fisheye paramétrable [Carpendale et Montagnese, 2001] ainsi que l'affichage des labels des arêtes balayées selon la technique du labeling excentrique [Fekete et Plaisant, 1999] facilitent sensiblement l'exploration du graphe. De plus, les lignes et les colonnes d'une matrice peuvent être ordonnées selon des critères donnés ce qui est difficile dans les graphes nœuds–liens. Toutefois, les sommets dans les matrices d'adjacence sont distribués sur les deux axes de la matrice, ce qui nécessite un entraînement préalable de la part des utilisateurs pour se familiariser avec la métaphore des matrices.

En conclusion, les avantages des matrices d'adjacence rendent leur utilisation intéressante par rapport aux représentations déjà existantes.

3.2 Applications

Ghoniem et *al.* [Ghoniem *et al.*, 2005a] présentent un outil pour visualiser l'activité dynamique des graphes générés par des programmes basés sur les contraintes. Ils se concentrent sur la visualisation des graphes dynamiques en utilisant des matrices d'adjacence et montrent que les matrices d'adjacence peuvent fournir des indices intéressants sur le comportement des solveurs de contraintes en présentant l'activation des contraintes pendant le résolution du problème. La présentation sous forme de matrices d'adjacence permet une meilleure compréhension du comportement dynamique des solveurs et améliore l'analyse et l'optimisation des programmes basés sur les contraintes.

La programmation par contraintes (PPC) est un paradigme de programmation déclarative consistant, pour un problème donné, à décrire les contraintes qu'une solution doit satisfaire. La résolution est confiée à des solveurs spécialisés, sortes de boites noires munies d'une panoplie de méthodes et d'algorithmes de résolution. Un problème est modélisé par un ensemble de variables à valeurs discrètes dans un domaine, reliées entre elles par des contraintes. Les sommets du graphe représentant ce problème sont les variables et les contraintes du problème. Les arêtes du graphe relient les contraintes aux variables sur lesquelles elles portent.

L'espace de recherche peut être représenté par un arbre avec une largeur exponentiellement croissante où chaque niveau correspond à une variable du problème. Par conséquent, la visualisation de telles hiérarchies est impossible. Des recherches ont été menées pour représenter et visualiser ces arbres mais tous les outils de vi-

sualisation se sont heurtés au défi principal de représenter les structures de données exponentiellement croissantes sur un écran avec une résolution limitée. L'approche adoptée par les auteurs est de considérer le réseau des variables et des contraintes qui collaborent ensemble pour résoudre le problème au lieu de se concentrer sur l'arbre de recherche. L'activité dans un réseau est simplement le nombre de fois qu'une variable ou une contrainte est utilisée pendant un intervalle de temps. L'activité peut être représentée par un attribut d'ensemble (un ensemble où l'activité s'est produite) associé aux sommets du réseau ou à ses arcs ou aux deux.

Pour visualiser les données, les auteurs utilisent les matrices d'adjacence au lieu des diagrammes nœuds–liens pour visualiser et explorer interactivement de grands graphes, avec des milliers de nœuds et un nombre possiblement très grand de liens. Tous les liens sont de plus pondérés avec le nombre de fois où la contrainte est activée dans la résolution. Ceci aide à améliorer la structure statique avec l'information dynamique en précisant l'activation des contraintes. Plus précisément, un historique d'activité est gardé avec le graphe. Ainsi, le graphe peut être interrogé dynamiquement sur les liens qui sont en activité dans une période de temps contrôlée par l'utilisateur et comparer la quantité d'activité entre les liens pendant cette période. L'utilisateur peut visualiser l'activité dans le graphe à travers tout le processus de résolution ou dans une plus petite période de temps dont les limites et l'ampleur sont interactivement paramètrées. En particulier, les auteurs balayent la période du temps du début à la fin de l'historique et peuvent revenir sur le processus de résolution et voir quels liens ont été établis, quand, et combien de fois. Ils s'intéressent plus particulièrement à l'activité entre solutions successives.

Un autre travail de Ghoniem et Fekete [Ghoniem et Fekete, 2002] montrent également que la représentation par matrices d'adjacence permet de déceler clairement la structure sous-jacente d'un graphe de co-activité et d'apprécier l'activité en son sein au fil du temps alors que la représentation équivalente en nœuds–liens est inexploitable.

Dans notre travail, nous nous sommes basés sur le travail de Ghoniem et *al.* [Ghoniem *et al.*, 2005a] pour l'utilisation des matrices d'adjacence et leurs manipulations.

3.3 Avantage des matrices d'adjacence

Pour évaluer la lisibilité des matrices d'adjacence qu'ils ont utilisées dans leur travaux, Ghoniem *et al.* [Ghoniem *et al.*, 2004] ont procédé à une expérimentation contrôlée visant à évaluer la lisibilité de deux représentations de graphes : les matrices d'adjacences et les diagrammes nœuds–liens mis en page par un algorithme de champ de forces, sur un jeu de neuf graphes aléatoires et un ensemble de sept tâches d'exploration. Les tâches concernent les caractéristiques élémentaires des sommets (le nombre de sommets, les sommets isolés), les caractéristiques élémentaires des chemins (nombre de liens, existence d'un voisin commun) et les caractéristiques élémentaires des sous-graphes (sous-graphe quelconque).

Dans les limites des paramètres de cette expérience contrôlée, les résultats montrent que : pour de petits graphes (une vingtaine de sommets), les diagrammes nœuds–liens sont toujours les plus lisibles car plus familiers. Pour des graphes de

taille plus importante, les performances des diagrammes nœuds–liens se dégradent rapidement tandis que les matrices d'adjacence restent lisibles avec des écarts allant jusqu'à 30% de bonnes réponses en plus pour des temps de réponses comparables sinon meilleurs.

En revanche, les matrices requièrent un apprentissage mais qui, une fois acquis, permet certainement d'améliorer encore les performances qui ont été mesurées. Pour des tâches plus compliquées comme la recherche de chemins, une interaction est toujours préférable quelle que soit la représentation utilisée, par exemple en sélectionnant un nœud et en affichant tous les chemins ou le meilleur chemin. Dans la représentation matricielle, cela peut se faire à l'aide de courbes connectant les liens connexes, c'est-à-dire les cellules représentant les nœuds.

De plus, Bertin [Bertin, 1967] montre qu'il est possible de révéler la structure sous-jacente des réseaux lorsque ceux-ci sont représentés sur une matrice d'adjacence moyennant un jeu de permutations des lignes et des colonnes. En effet, la détection de la structure fondamentale des problèmes peut accélérer de manière significative la recherche. Dans le travail de Bertin, les outils de visualisation aident à classifier l'information susceptible d'être intégrée dans un comportement adaptatif en indiquant la structure fondamentale des problèmes.

3.4 Mise en oeuvre des matrices d'adjacence

Nous utilisons les matrices d'adjacence pour représenter les données statiques et dynamiques des programmes en cours d'étude pour essayer d'en faciliter la

compréhension à savoir : cibler l'information à comprendre, avoir une vue architecturale du système, identifier les fonctionnalités du programme et détecter les traitements répétitifs.

Nous nous intéressons aux données relatives aux classes et non aux objets (à leurs instances) parce que les objets peuvent être trop nombreux et parce que les classes abstraient leurs instances. De plus, les matrices d'adjacence montrent ainsi les mêmes types de données (types de relation entre les classs) que les diagrammes de classes UML habituels.

Les données que nous affichons dans la matrice d'adjacence sont : *classe appelante, classe appelée, poids de la relation.*

où :

- la *classe appelante* représente la classe dont une instance a fait un appel de méthode ;
- la *classe appelée* représente la classe dont une instance reçoit un appel de de la part de la *classe appelante* ;
- le *poids de la relation* est un identifiant qui représente le type de relation qui existe entre une *classe appelante* et une *classe appelée* (par exemple aggregation, composition...).

À ces données affichées, nous avons ajouté un incrément qui permet de voir l'évolution de ces données dans le temps pour permettre une meilleure manipulation des données.

Les lignes et les colonnes dans la matrice d'adjacence représentent les classes utilisées dans la représentation où par convention :

- les colonnes représentent les classes appelantes ;
- les lignes représentent les classes appelées ;
- deux classes liées par une relation implique l'activation de la cellule d'intersection correspondante avec un nombre représentant le type de relation.

L'utilisation des matrices d'adjacence nous permet :

- d'afficher les noms des classes qui sont utilisées par les jeux d'essais utilisés sur un programme en cours d'étude en ordre alphabétique ;
- d'utiliser un incrément de temps pour les données affichées pour représenter l'évolution de relations entre classes dans le temps ;
- de sauvegarder l'évolution des données dans le temps pour une manipulation ultérieure (comparaison de représentations) ;
- d'ajouter les noms des classes qui manquent dans une représentation même si elles ne participent pas dans l'exécution des jeux d'essaies pour une meilleure comparaison avec d'autres représentations. Ceci, permet de faciliter la détection des différences qui existent entre les représentations ;
- d'utiliser des couleurs pour représenter le type de relation qui existe entre classes appelantes et appelées et ainsi d'enrichir la représentation d'informations qui en facilite la compréhension.

CHAPITRE 4

EXTRACTION DES DONNÉES À VISUALISER

Dans ce chapitre, nous discutons des données à visualiser et les méthodes que nous avons utilisées pour les extraire et les manipuler afin de les visualiser.

La visualisation dynamique [Price *et al.*, 1993] d'un programme se compose de trois phases : collection des données sur le comportement du programme ; analyses des données collectées et présentation des résultats d'analyses. Les données peuvent être extraites statiquement en analysant le code du programme ou dynamiquement à partir d'une trace d'événements de l'exécution du programme [Pacione *et al.*, 2003]. Une trace d'événement peut être produite en instrumentant le code source, le code objet ou l'environnement d'exécution, ou en exécutant le programme sous le contrôle d'un déboggueur ou d'un profileur. Une trace est un historique d'événements d'exécutions. Les événements sont reliés aux constructeurs du langage. Ils reflètent la sémantique des flux des contrôles et des flux des données du programme. Le modèle d'une exécution d'un programme Java consiste en une trace des événements exécutés. Le modèle considéré est le modèle d'objets de Java, donc les événements se relient aux : classes (chargement et déchargement) ; champs (accès et modification) ; constructeurs, "finalizers" et méthodes (entrée et sortie).

Ernst [Ernst, 2003] discute les synergies et les dualités entre l'analyse statique et l'analyse dynamique. Il soutient que l'analyse statique et l'analyse dynamique doivent être complémentaire et devraient inspirer différentes approches au même problème. Quelques approches statiques et dynamiques sont semblables du fait

que chacune considère, et généralise, un sous-ensemble de toutes les exécutions possibles.

L'analyse statique examine le code du programme et infère tous les comportements possibles qui pourraient surgir pendant son exécution. Typiquement, l'analyse statique est conservatrice et robuste. La robustesse garantit que les résultats de l'analyse sont une description précise du comportement du programme, sans se préoccuper des entrées ou de l'environnement dans lequel le programme est exécuté. L'analyse statique emploie habituellement un modèle abstrait des états du programme qui peut entraîner une perte d'informations, mais qui est plus compact et plus facile à manipuler.

L'analyse dynamique fonctionne en exécutant un programme et en observant son comportement. L'analyse dynamique est précise parce qu'aucune approximation ou abstraction n'est faite. Les inconvénients de l'analyse dynamique sont que ses résultats peuvent ne pas être valides pour les exécutions futures, qu'il n'y a pas d'abstractions et que la taille des données générées devient très difficile à gérer.

L'analyse statique et l'analyse dynamique ont des forces et des faiblesses complémentaires. Les deux approches peuvent être appliquées à un seul problème, produisant des résultats qui sont utiles dans différents contextes. L'exécution d'une analyse puis de l'autre analyse (et peut-être en réitérant) est plus intéressante que l'exécution de l'une ou de l'autre seule. Alternativement, les différentes analyses peuvent rassembler une variété d'informations pour lesquelles elles sont plus adaptées. Par consequent, Ernst propose une analyse alternative, l'analyse hybride, qui combine l'analyse dynamique et l'analyse statique. Une telle analyse sacrifierait un peu de la robustesse de l'analyse statique et un peu de l'exactitude de l'analyse dynamique pour obtenir des nouvelles techniques dont les propriétés conviendront

mieux pour des usages plus poussés. Les analyses hybrides comblent l'espace entre l'analyse statique et l'analyse dynamique.

Dans notre approche, nous utilisons en premier lieu l'analyse dynamique pour extraire des données précises sur le déroulement de l'exécution de jeux d'essais. Les résultats de cette analyse sont affinés à l'aide d'analyses statiques pour avoir des données précises sur le comportement du programme.

4.1 Analyse dynamique

Il y a plusieurs méthodes pour la collecte des donnée dynamiques. Salah et Mancoridis [Salah et Mancoridis, 2004] ont cité trois méthodes. La première est l'instrumentation du code source. La deuxième implique l'instrumentation du code compilé. La troisième est basée sur le déboggage et le profilage.

La technique la plus commune pour collecter les données est l'instrumentation du code source. Plusieurs travaux ont utilisé cette technique, par exemple, le travail de Ernst *et al.* [Ernst *et al.*, 2001] qui découvre les invariants qui existent dans les traces d'exécutions. Harrold *et al.* [Harrold *et al.*, 1998] présentent une approche et une analyse empirique des relations qui existent à travers un programme spectra à savoir la distribution des paths dérivés à partir de l'exécution du programme et le comportement du programme. Jeffrey et Auguston présentent le système UFO d'analyses dynamiques [Jeffrey et Auguston, 2003] qui combine un language et un moniteur d'architecture pour une construction facile d'outils d'analyses dynamiques.

Une autre technique consiste à instrumenter l'environnement d'exécution des programmes [Guéhéneuc *et al.*, 2002]. L'exécution d'un programme écrit en Java a lieu dans une JVM, qui à son tour s'exécute dans le système d'exploitation d'un ordinateur. Cette architecture superposée simplifie le déboggage des programmes Java. En effet, la JVM fournit une interface standard pour déboguer le programme qu'elle exécute, la Java Platform Debug Architecture (JPDA) [Microsystems., 2002].

La JPDA se décompose de trois parties principales. Sur l'ordinateur local, la JVM local fournit une entrée (front-end) pour le déboggage à distance, la Java debug Interface (JDI). La JDI est une interface 100% Java pour le débogguage des programmes Java s'exécutant sur une JVM à distance. Elle permet la connection avec une JVM à distance déjà existante ou la création d'une JVM à distance, et l'interaction avec elle. La JDI dialogue avec la JVM à distance qui s'exécute sur l'ordinateur à distance par l'interface native spécialisée de Java pour les trois parties des outils de débogguage, la JVM Debug Interface (JVMDI). Le Java Debug Protocol (JDWP) abstrait la couche de communication entre la JVM locale et la JVM distante.

Dans la JVM local, La JDI représente la JVM à distance comme une instance qui implémente l'interface com.sun.jdi.VirtualMachine. À travers l'interface VirtualMachine, le programme local reçoit des événements et accède aux instances, aux classes, et aux threads du programme à distance. Quand le programme à distance émet des événements, la JVM à distance suspend son activité (tous ses threads) et attend le programme local pour qu'elle reprend son exécution, par un appel à la méthode appropriée de l'interface VirtualMachine.

Dans notre approche, nous utilisons la JPDA parce qu'elle offre une API puissante pour l'instrumentation de l'environnement d'exécution des programmes et parce qu'elle est portable à travers des plateformes et des versions de J2SDK. La JDI est utilisée pour créer une JVM à distance, pour exécuter sur cette JVM distante le programme à analyser, et pour obtenir les événements reliés à l'exécution du programme analysé.

Cependant, un des problèmes de la collecte des données à l'aide de l'analyse dynamique est la gestion des larges quantités des données générées. Plusieurs travaux ont discuté ce problème, par exemple Hamou-Lhadj et Lethbridge [Hamou-Lhadj et Lethbridge, 2002] ont présenté un algorithme de compression qui permet de détecter les redondances qui se trouvent dans une trace dynamique et de les supprimer. L'algorithme de compression présenté permet de mieux comprendre une trace d'exécution en réduisant de manière significative sa taille et en même temps en gardant son contenu lisible. L'algorithme de compression est réversible car il permet de reconstruite la trace originale à partir de sa version compressée.

La trace est d'abord prétraitée en détectant et en supprimant les redondances contiguës non-recouvertes produites par les boucles et la récursivité. Les appels redondants provoqués par les boucles et la récursivité tendent en effet à encombrer la trace. De plus, si la trace est perçue comme un arbre, en enlevant les redondances contiguës, la profondeur de l'arbre et le degré de ses nœuds sont réduits.

Après le prétraitement, la trace est représentée sous forme d'un arbre étiqueté et ordonné avec une seule racine et l'algorithme commun des subexpressions [Downey *et al.*, 1980, Flajolet *et al.*, 1990] est appliqué pour enlever toutes les redondances restantes à savoir, les redondances non-contiguës, afin de représenter les parties répétées une seule fois.

Dans notre approche, nous n'utilisons pas de mécanisme de compression de données mais cette alternative semble très intéressante à développer pour gérer les grandes quantités de données.

4.1.1 Génération de la trace

Dans notre approche, nous utilisons l'architecture de déboggage de la plate-forme Java pour générer une trace. Un moteur Prolog est utilisé pour exécuter des requêtes à travers la trace obtenue. Les événements sont décrits en prédicats Prolog, sous forme d'un functor et d'un ensemble de paramètres. La table 4.1 présente la liste des événements possibles dans la trace d'exécution d'un programme. Le moteur Prolog s'exécute comme une co-routine du programme qui est analysé. Il exécute une requête qui contient des prédicats pour mener l'exécution du programme et pour obtenir les prochains événements. Prolog est utilisé parce qu'il possède un mécanisme d'unification et des capacités de pattern-matching de haut niveau, et parce qu'il est expressif car déclaratif et en plus il a un mécanisme de retour en arrière.

Nous ne considérons pas la trace post-mortem : il n'est pas supposé que l'historique complet de l'exécution soit entièrement disponible à tout moment de l'analyse. Les prochains événements d'exécution disponibles peuvent être interrogés, mais les précédents ne le peuvent pas. Il est possible, cependant, que le programme d'analyse stocke une partie de la trace passée qui intéresse le mainteneur.

Dans notre approche, nous nous intéressons aux constructeurs et aux méthodes (entrée et sortie) car nous cherchons à représenter les appels de méthodes qui

Java execution event	Definition and parameters
fieldAccess(<Field name>, <Event unique ID>, <Unique ID of the instance possessing this field>)	The instance field <Field name> of the instance identified by its ID <Unique ID of the instance possessing this field> shall be read.
fieldModification(<Field name>, <Event unique ID>, <Unique ID of the instance possessing this field>, <Unique ID of the new object assigned to this field>, <Fully qualified name of the class of the new object>)	The instance field <Field name> of the instance identified by its ID <Unique ID of the instance possessing this field> shall be modified. The ID of the object to be assigned to the field is <Unique ID of the new object assigned to this field>. For visualization purpose, the fully-qualified name of the object class is given.
classLoad(<Class name>, <Event unique ID>)	The program requires the class <Class name>.
constructorEntry(<Declaring class name>, <Event unique ID>, <Unique ID of the newly created instance>)	A new instance of class <Declaring class name> is being created. This instance is uniquely identified by <Unique ID of the newly created instance> for the rest of the program execution.
finalizeEntry(<Declaring class name>, <Event unique ID>, <Unique ID of the being-finalized instance>)	The instance of class <Declaring class name> uniquely identified by <Unique ID of the being-finalized instance> is being finalized.
methodEntry(<Method name>, <Event unique ID>, <Unique ID of the caller instance>, <Unique ID of the receiver instance>, <Fully-qualified name of the class declaring this method>)	The method <Method name> of class <Fully-qualified name of the class declaring this method> is called on the instance uniquely identified by <Unique ID of the receiver instance>.

Identical definitions for dual events : classUnload ; constructorExit ; and, finalizerExit.

Definition changes for event methodExit :

methodExit(<Method name>, <Event unique ID>, <Unique ID of the caller instance>, <Unique ID of the receiver instance>, <Fully-qualified name of the class declaring this method>, <The value returned by this method>)	The method <Method name> of class <Fully-qualified name of the class declaring this method> is completing its call on the instance uniquely identified by <Unique ID of the receiver instance>. The returned value is provided in <The value returned by this method>.

TAB. 4.1 – Liste des événements possibles

existent entre les classes. Les constructeurs permettent d'indiquer l'instanciation de classes par les instances appelantes et les entrées et sorties des méthodes permettent d'indiquer l'émission et le traitement des messages reçus par les instances des classes.

Les événements de la trace contiennent les informations suivantes :

Identificateur : identificateur d'événements ;

Type : type d'événement (dans notre approche se sont les <*cEntry*>, <*cExit*>, <*mEntry*>ou <mExit>) où :

- *mEntry* : l'entrée de la méthode ;
- *mExit* : la sortie de la méthode ;
- *cEntry* : le début de l'instanciation ;
- *cExit* : la fin de l'instanciation.

Origine : classe qui a fait appel aux constructeurs ou aux méthodes d'une autre classe ;

Commentaire : classe dont une instance reçoit l'appel de méthode ;

Nom : nom de l'événement (si le *Type* est une méthode alors le *Nom* sera le nom de la méthode et si c'est un constructeur se sera le nom de la classe dont une instance est construite) ;

Valeur retournée : la valeur retournée par la méthode appelée s'il y a lieu.

4.1.2 Exemples

Prenons un programme simple de 6 classes A, B, C, D, E, F où :

- la méthode operation1() de la classe A fait appel à la méthode operation2() de la classe B ;
- la méthode operation3() de la classe C fait appel à la méthode operation4() de la classe D ;
- la méthode operation5() de la classe E fait appel à la méthode operation6() de la classe F ;
- A est en relation de composition avec B ;
- C est en relation d'aggrégation avec D ;
- E est en relation d'aggrégation avec F.

```
public class TestMain {
    public static void main(final String args[]) {
        A a = new A();
        D d = new D();
        C c = new C(d);
        E e = new E();
    }
}
public class A {
    B b = new B();
    public A(){
        operation1(b);
    }
    public void operation1(B b){
        b.operation2();
    }
}
class B{
    public void operation2(){
    }
}
class C {
    public D d;
    public C(D d) {
        this.d = d;
        operation3();
    }
    public void operation3(){
        this.d.operation4();
    }
}
```

```
class D{
    public void operation4(){
    }
}
class E {
    public F f = new F();
    public E() {
        operation5();
    }
    public void operation5(){
        this.f.operation6();
    }
}
class F{
    public void operation6(){
    }
}
```

La trace dynamique générée par une exécution de la méthode *main()* de la classe *Testmain* qui n'inclut que les appels des méthodes et des constructeurs est[1] :

```
mEntry(JVM,TestMain,main)
cEntry(TestMain,A)
cEntry(A,B)
cExit(A,B)
mEntry(A,A,operation1)
mEntry(A,B,operation2)
mExit(A,B,operation2)
mExit(A,A,operation1)
cExit(TestMain,A)
cEntry(TestMain,D)
cExit(TestMain,D)
cEntry(TestMain,C)
mEntry(C,C,operation3)
mEntry(C,D,operation4)
mExit(C,D,operation4)
mExit(C,C,operation3)
cExit(TestMain,C)
cEntry(TestMain,E)
cEntry(E,F)
cExit(E,F)
mEntry(E,E,operation5)
mEntry(E,F,operation6)
```

[1]Pour simplifier la lecture de la trace nous avons omis de mentionner les noms des packages pour représenter les classes.

```
mExit(E,F,operation6)
mExit(E,E,operation5)
cExit(TestMain,E)
mExit(JVM,TestMain,main)
```

4.1.3 Requêtes sur les traces

Après avoir utilisé l'architecture de déboggage de la plateforme Java pour générer les traces d'exécution, nous avons conçu un outil qui permet la manipulation de cette trace générée. En effet, pour mieux comprendre le programme qui est à l'étude, à savoir, cibler l'information à comprendre, avoir une vue architecturale du système, identifier les fonctionnalités du programme et détecter les traitements répétitifs, nous nous intéressons aux classes et aux méthodes qui nous semblent les plus impliquées dans une exécution. Nous pouvons extraire les informations à analyser dans des sous-traces et les visualiser sous forme de matrice d'adjacence selon un certain nombre de choix :

- **searchAll()** : permet d'obtenir toute la trace d'exécution afin de la visualiser.
- **classSearch(Noms de classes)** : permet d'obtenir de la trace la sous-trace contenant uniquement les classes incluses dans *Noms de classes* plus la classes auxquelles elles ont envoyé un message (soit par une instanciation "cEntry" ou par l'appel des méthodes "mEntry")

  Exemple : classSearch ({A,C})

  ```
  cEntry(A,B)
  mEntry(A,A,operation1)
  mEntry(A,B,operation2)
  mEntry(C,C,operation3)
  mEntry(C,D,operation4)
  ```

- **methodSearchAll(Nom de classe)** : permet d'obtenir de la trace pour une classe donnée tous les messages qui sont envoyés à ses instances.

Exemple : methodSearchAll(“A”)

```
cEntry(A,B)
cExit(A,B)
mEntry(A,A,operation1)
mEntry(A,B,operation2)
mExit(A,B,operation2)
mExit(A,A,operation1)
```

– **methodSearch(Nom de classe, Noms de méthodes)** : permet d’obtenir de la trace pour une classe donnée les événements qui sont générés par ses méthodes (blocs d’événements inclus entre mEntry des méthodes et leurs mExit).

Exemple : methodSearch(“A”, {“operation1”})

```
mEntry(A,A,operation1)
mEntry(A,B,operation2)
mExit(A,B,operation2)
mExit(A,A,operation1)
```

– **methodSearchB(Nom de classe, Noms de méthodes)** : permet d’obtenir de la trace pour une classe donnée les classes qui lui ont envoyées un message à travers les méthodes choisies.

Exemple : methodSearch(“A”, {“operation1”})

```
mEntry(A,A,operation1)
```

– **constructorSearch(Nom de classe)** : permet d’obtenir de la trace pour une classe donnée les événements qui sont générés par son constructeur (blocs d’événements inclus entre le cEntry et le cExit).

Exemple : constructorSearch(“TestMain”)

```
cEntry(TestMain,A)
cEntry(A,B)
cExit(A,B)
```

```
mEntry(A,A,operation1)
mEntry(A,B,operation2)
mExit(A,B,operation2)
mExit(A,A,operation1)
cExit(TestMain,A)
cEntry(TestMain,D)
cExit(TestMain,D)
cEntry(TestMain,C)
mEntry(C,C,operation3)
mEntry(C,D,operation4)
mExit(C,D,operation4)
mExit(C,C,operation3)
cExit(TestMain,C)
cEntry(TestMain,E)
cEntry(E,F) cExit(E,F)
mEntry(E,E,operation5)
mEntry(E,F,operation6)
mExit(E,F,operation6)
mExit(E,E,operation5)
cExit(TestMain,E)
```

– **constructorSearchB(Nom de classe)** : permet d'obtenir de la trace pour une classe donnée les classes qui lui ont envoyées un message à travers son constructeur.

Exemple : constructorSearchB(“A”) ;

```
cEntry(TestMain,A)
```

4.1.4 Discussion et conclusion

L'analyse dynamique est utilisée comme point de départ à l'étude du comportement des programmes. Elle permet d'obtenir des données sur l'exécution d'un programme. Elle est efficace et précise : elle n'exige pas des analyses coûteuses, bien qu'elle exige le choix de jeux de tests, et elle donne des résultats fortement détaillés concernant ces jeux [Ernst, 2003]. La manipulation des données générées lors de l'analyse dynamique par les requêtes que nous avons proposées permet l'extraction des traces des données qui semblent les plus pertinents pour le programme

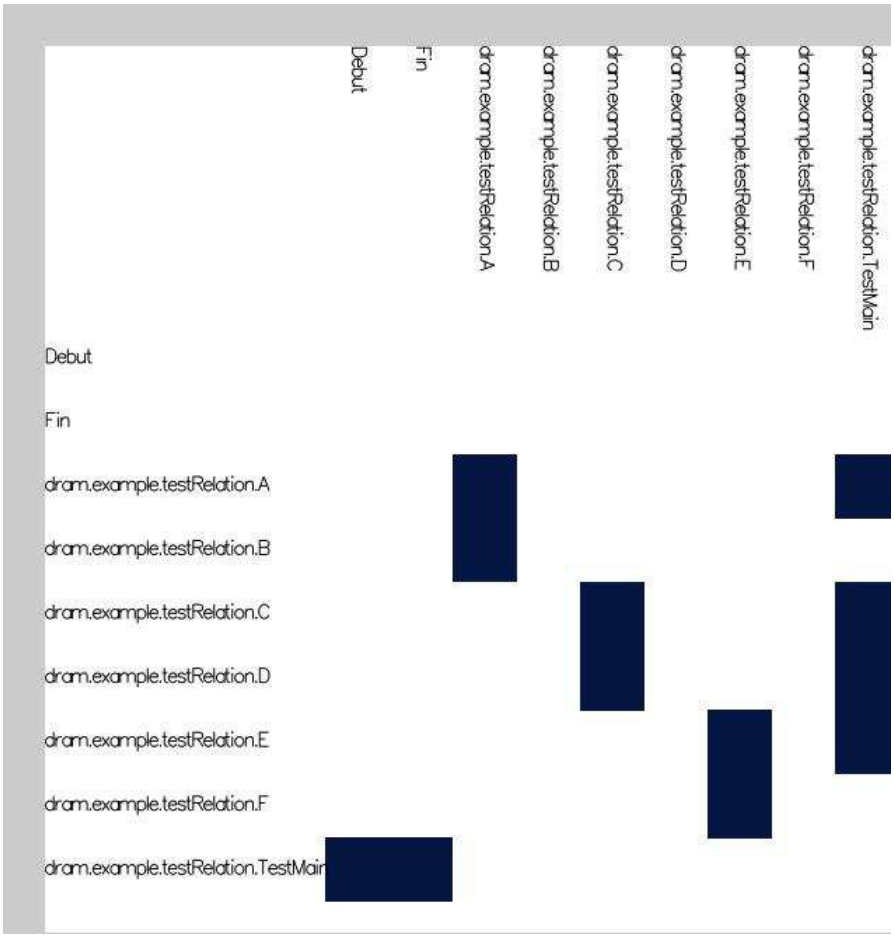

FIG. 4.1 – Représentation matricielle de la trace filtrée (nous avons choisi de travailler avec toute la trace avec le filtre *searchAll()*).

à étudier sous forme de sous-traces. Les résultats ainsi obtenus peuvent être visualisés sous forme de matrices d'adjacence pour une meilleure lisibilité, voir par exemple la figure 4.1 qui représente une sous-trace de la trace du programme de la page 40 obtenue en utilisant la requête *searchAll()* (la sous-trace correspond à la trace du programme car la requête *searchAll()* permet d'avoir toute la trace).

La représentation de la figure 4.1 permet d'avoir les liens seulement entre les classes de la trace filtrée du programme de la page 40 obtenue en utilisant la requête *searchAll()*. Ceci permet aux mainteneurs de se concentrer sur les informations du

jeux d'essais exécuté en faisant abstractions des autres informations qui ne sont pas nécessaires à l'exécution. De plus, la représentation ainsi faite offre plus de lisibilité car elle ne souffre pas de chevauchement des nœuds et des liens.

4.2 Analyse statique

L'analyse dynamique et l'analyse statique sont complémentaires. La combinaison des résultats des deux types d'analyses permet de rassembler les différents types d'informations qui sont propres à chaque type d'analyse. Dans notre approche, nous avons utilisé l'analyse dynamique pour obtenir la trace d'exécution du programme à étudier. Nous voulons enrichir ces informations avec des analyses statiques qui permettent de fournir le type de relations qui existent entre les différentes classes afin d'avoir une vision globale et générale du programme étudié et une vision plus abstraite que les envois de messages fournie par l'analyse dynamique.

Nous utilisons un métamodèle PADL [Albin-Amiot *et al.*, 2002] qui permet de représenter les modèles de programmes aux niveaux idiomatique et conception et les motifs de conception au niveau idiomatique comme des entités uniques et à part entière. Nous utilisons PADL pour avoir le type de relations qui existe entre les différentes classes de la trace générée dynamiquement (voir figure 4.2). Nous associons à chaque type de relation un nombre unique qui permet de le représenter. Ainsi, la trace sera enrichie par le type de relations qui existent entre les différentes données de la trace.

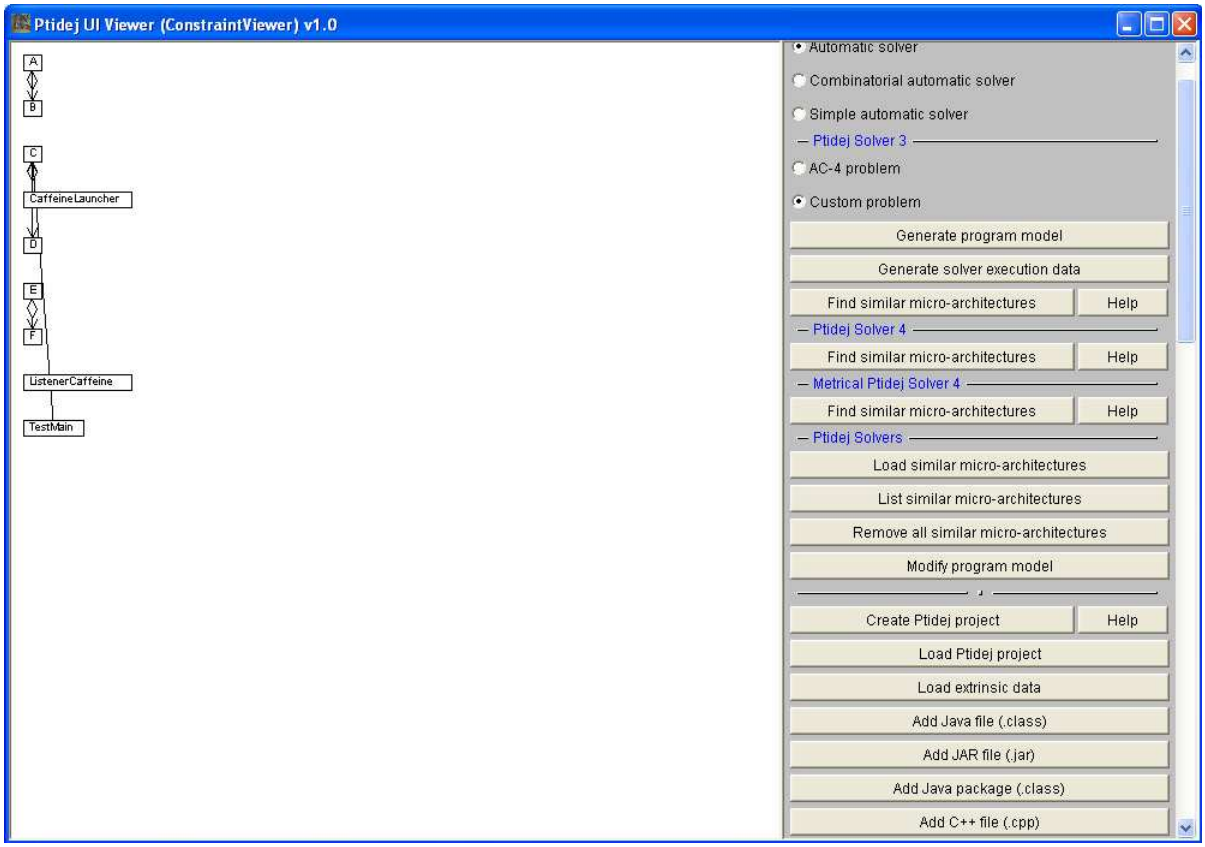

FIG. 4.2 – Interface graphique Ptidej

Exemple : La trace dynamique générée à la page 42 est modifiée comme suit :

```
JVM TestMain 1 1
TestMain A 1 1
A B 6 1
A B 6 1
A A 5 1
A B 6 1
A B 6 1
A A 5 1
TestMain A 1 1
TestMain D 1 1
TestMain D 1 1
TestMain C 1 1
C C 5 1
C D 6 1
C D 6 1
C C 5 1
TestMain C 1 1
TestMain E 1 1
E F 6 1
E F 6 1
E E 5 1
E F 6 1
E F 6 1
E E 5 1
TestMain E 1 1
JVM TestMain 1 1
```

où chaque nombre représente un type de relation entre classes [Guéhéneuc, 2004] :

- 1 : indéfini (exemple d'appel par référence, l'appel entre les classes ne se fait connaître qu'au moment de l'analyse dynamique) ;
- 2 : héritage : lien représente l'héritage entre deux classes ;
- 3 : création : lien créé quand une classe instancie une autre classe ;
- 4 : utilisation : quand une classe utilise une autre classe ;
- 5 : association : lien de dépendance entre deux classes. Chaque classe peut avoir de multiples associations ;
- 6 : agrégation : une relation binaire entre deux classes, respectivement le tout

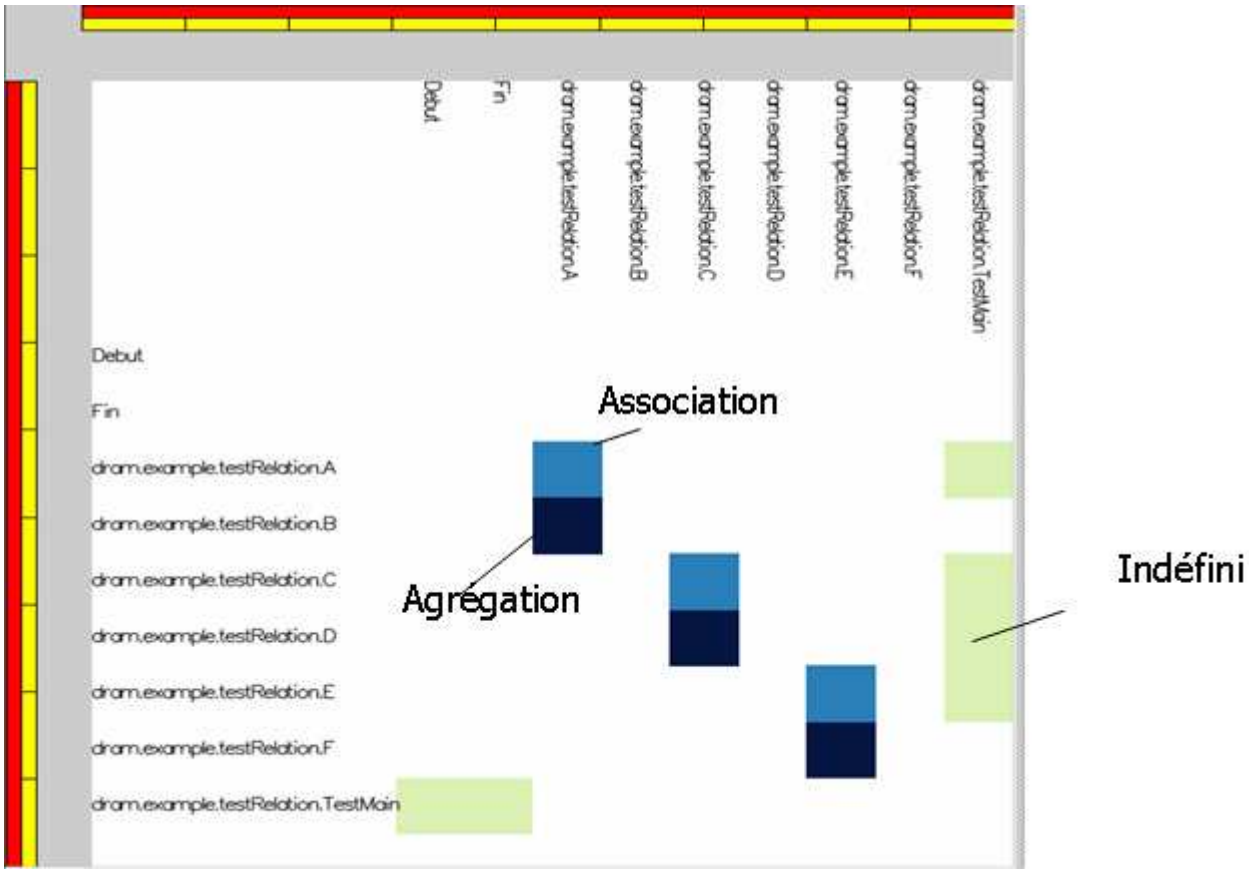

FIG. 4.3 – Matrice d'adjacence enrichie par le type de relations entre classes

et la partie. Conceptuellement, une partie n'a pas d'existence en dehors de son tout ;

– 7 : composition : une relation d'agrégation particulière pour laquelle les parties contenues dans le tout disparaissent quand le tout disparaît.

La trace ainsi modifiée est représentée sous forme de matrice d'adjacence en utilisant les couleurs pour différencier les types de relations qui existent entre les différentes classes du programme, comme montré sur la figure 4.3.

Interprétation des résultats

L'utilisation de l'analyse statique permet d'enrichir les informations obtenues lors de l'analyse dynamique en indiquant le type de relation qui existent entre les différentes classes dont les instances collaborent (appels de méthodes) pendant l'exécution. Ainsi, nous pouvons avoir, à chaque moment de notre investigation pour comprendre le comportement du programme, une vue architecturale du programme étudié.

4.3 Conclusion

Nous avons utilisé les matrices d'adjacence pour représenter les données obtenues à l'aide de l'analyse dynamique et de l'analyse statique pour visualiser les données du programme en cours d'étude. Dans le prochain chapitre, nous raffinons l'utilisation des matrices d'adjacence pour avoir une meilleure manipulations des données affichées.

CHAPITRE 5

VISUALISATION ET MATRICES D'ADJACENCE

5.1 Choix de données à visualiser

La manipulation des données à visualiser peut réduire considérablement l'effort du mainteneur dans la compréhension d'un programme. En effet, permettre au mainteneur de manipuler de plusieurs manières les données obtenues lors de l'analyse dynamique et l'analyse statique l'aide à mieux focaliser son attention lors de la maintenance du programme.

Dans le chapitre précèdent, nous avons offert des possibilités de manipulations de la trace (avoir des sous-traces) pour permettre au mainteneur de cibler l'information à visualiser.

Nous raffinons ces manipulations en permettant au mainteneur de parcourir ces sous-traces et de voir l'évolution de l'exécution du programme. L'évolution de l'exécution des sous-traces permet d'avoir des informations plus détaillées quand au déroulement de l'exécution du programme. Les manipulations permettent d'obtenir le déroulement chronologique selon des critères choisis sur tout le comportement représenté par les sous-traces.

Cependant, les critères pour découper les sous-traces ne sont pas triviaux car la sous-trace ne contient pas explicitement de points de découpage. Nous proposons plusieurs manières de découper la sous-trace : découpage par événement, découpage

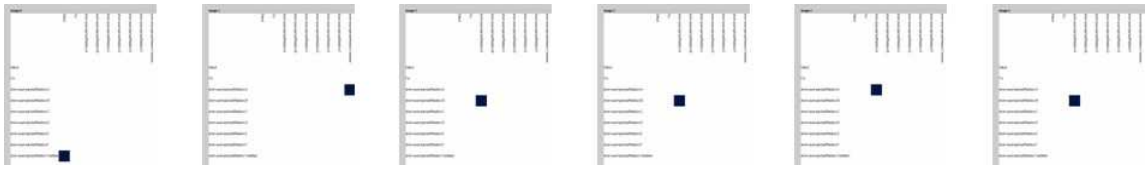

FIG. 5.1 – Representation matricielle de la trace filtrée découpé par événements

par classes, découpage par méthodes, découpage par fonctionnalités programmes.

5.2 Découpage dans le temps

Nous proposons plusieurs méthodes pour découper les sous-traces générées afin de permettre aux mainteneurs une meilleure manipulation et une meilleure compréhension des données.

5.2.1 Découpage par événements

Nous découpons les sous-traces par événements en isolant chaque appel de méthode ou de constructeur et en le représentant seul sous forme de matrice d'adjacence. Nous aurons ainsi, une suite de matrices d'adjacence où chaque matrice représente un événement de la trace, comme montré sur la figure 5.1.

Toutefois, la visualisation de la trace suivant ce découpage n'est pas intéressante car chaque événement est représenté dans une matrice d'adjacence isolé des autres sans donner une idée des relations qui peuvent exister avec les autres événements. Pour remédier à ce problème, nous proposons un regroupement des sous-traces par classes et/ou méthodes.

5.2.2 Regroupement par classes

Après avoir obtenu une sous-trace à visualiser selon les critères décrit dans la section 4.1.3, nous pouvons raffiner notre choix en procédant à un regroupement des événements par classes. Ce regroupement permet de voir l'évolution de l'exécution d'un programme par rapport à des classes "pivots" montrant ainsi le déroulement de la sous-trace avant, pendant et après les appels de méthodes de ces classes. Pour pouvoir représenter ce déroulement, nous utilisons un indice (incrément) qui est incrémenté à chaque fois que les classes du regroupement sont trouvées dans la sous-trace. Cet indice ou cet incrément ne reflète pas la profondeur des éléments dans l'arbre des appels, car les appels inclus dans les classes de regroupement ont le même indice ou incrément.

Exemple : si l'on considère l'exemple de la page 40, nous prenons le choix de travailler sur toute la trace avec searchAll(). Nous regroupons par rapport à la classe "*A*". Au début, l'incrément est initialisé à 1. Par la suite, la trace est parcourue et à chaque fois que des événements sont associés aux appels de méthodes ou de constructeurs (bloc d'événements inclus entre un *cEntry* et un *cExit* ou entre *mEntry* et *mExit* par la classe "*A*" sont trouvés, l'incrément est modifié en lui ajoutant une unité. La sous-trace obtenue est :

```
JVM TestMain 1 1
TestMain A 1 1
A B 6 2
A B 6 2
A A 5 2
A B 6 2
A B 6 2
A A 5 2
```

```
TestMain A 1 3
TestMain D 1 3
TestMain D 1 3
TestMain C 1 3
C C 5 3
C D 6 3
C D 6 3
C C 5 3
TestMain C 1 3
TestMain E 1 3
E F 6 3
E F 6 3
E E 5 3
E F 6 3
E F 6 3
E E 5 3
TestMain E 1 3
JVM TestMain 1 3
```

La procédure se résume comme suit :

- le mainteneur choisit les classes qui lui semblent les plus intéressantes dans l'exécution du programme à visualiser ;
- l'algorithme parcourt la sous-trace du début à la fin :
  - à chaque fois que des événements concernant les classes que nous voulons regrouper sont trouvées, à savoir les blocs d'événements inclus entre un *cEntry* et un *cExit* ou entre un *mEntry* et un *mExit* de la classe de regroupement, l'incrément est incrémenté ;
  - une fois l'incrément modifié, la trace est aussi modifiée : chaque événement de la sous-trace est modifié en lui ajoutant l'incrément correspondant ;
  - la sous-trace initiale est découpée en sous-traces qui contiennent les événements qui ont le même incrément ;
  - chaque élément de l'ensemble des sous-traces est représenté sous forme d'une matrice d'adjacence formant ainsi une suite de matrices d'adjacence ;
  - le mainteneur peut visualiser cette suite de matrices et défiler les affichages

des matrices jusqu'à la fin de la suite de matrices, comme montré sur la figure 5.2.

Ce regroupement est fait par classes, nous pouvons généraliser l'algorithme pour avoir un regroupement par classes et par méthodes.

5.2.3 Interprétation des résultats

En effectuant le découpage de la trace par classes, nous remarquons les relations qui s'établissent entre classes à chaque fois que les classes choisies pour le regroupement déclarent des appels aux instances des autres classes du programme. Sur la figure 5.2, nous montrons que :

- la 1ère matrice d'adjacence représente les relations qui existent entre les instances des classes avant qu'une instance de la classe A ne soit construite ;
- la 2ème matrice d'adjacence représente les relations qui existent entre les classes choisies pour le regroupement et les autres classes du jeu d'essais. Ainsi, dans la 2ème matrice sur la figure 5.2, la classe A envoie des messages aux classes B et A ;
- la 3ème matrice d'adjacence représente les relations qui existent entre les classes après la fin de l'appel de méthode de l'instance de la classe A.

Ainsi, en choisissant les classes à étudier, nous pouvons connaître les collaborations qui se produisent entre les classes de regroupement et les autres classes du jeu d'essais exécuté. Ceci nous permet de bien cibler l'information à comprendre.

FIG. 5.2 – Visualisation d'une suite de matrices en regroupant par la classe "A"

5.2.4 Intérêts et limitations

L'intérêt de cet algorithme est de voir l'évolution de l'exécution d'un programme par rapport aux classes qui semblent au mainteneur les plus intéressantes. Le mainteneur peut voir les événements qui se déroulent avant l'instanciation des classes choisies, après leurs instanciations, pendant que leurs méthodes s'exécutent et après la fin de l'exécution de leurs méthodes. De plus, le déroulement de l'évolution de l'exécution du programme fait ressortir des patrons qui peuvent être intéressants pour le mainteneur. Ainsi, le mainteneur peut dégager des informations qui semblent plus pertinentes pour la compréhension des programmes à étudier. De plus, la visualisation des matrices d'adjacence selon le regroupement des classes peut permettre de détecter des traitements répétitifs qui se produisent à chaque fois qu'il y a appel ou fin d'appel des classes de regroupement et de dégager des abstractions éventuelles de plus haut niveau. Pour détecter les patrons et dégager des abstractions de haut niveau, les mainteneurs, dans un premier temps, doivent le faire de manière visuelle et à la main.

Cependant, cet algorithme souffre de certaines limitations à savoir le choix des classes de regroupement. Le choix est difficile car choisir les classes les plus intéressantes pour faire le regroupement nécessite un minimum de compréhension préalable (un point de départ) du programme à étudier. À défaut d'acquérir ce début de connaissance, le mainteneur est amené à faire plusieurs essais pour pouvoir faire ressortir les classes les plus intéressantes. De plus, si le programme étudié a un nombre limité de classes et que certaines classes contiennent toute l'information pertinente, le déroulement de l'exécution de la trace ne sera pas très représentative car le nombre de matrices d'adjacence sera limité et l'information condensée. Le

mainteneur peut remédier à ce problème en regroupant par classes et méthodes, l'information est découpée et la lisibilité des classes et des matrices est plus grande.

5.3 Identification des fonctionnalités d'un programme

La compréhension des fonctionnalités d'un programme et de leurs implémentations est une étape fondamentale pour la compréhension du programme, en particulier si le but est de modifier ou d'étendre ces fonctionnalités. Pour cela, le mainteneur doit localiser le code source responsable de l'implémentation de ces fonctionnalités. Une fonctionnalité est un sous-ensemble du code source d'un programme qui participe à la réalisation d'une fonctionnalité observée par un utilisateur du programme. Wong *et al.* [Wong *et al.*, 1999] dans leur travail proposent deux méthodes pour la localisation des fonctionnalités des programmes : d'abord, une approche systématique qui exige une compréhension complète du comportement du programme avant modification ; en second lieu, une approche "au besoin" qui exige seulement une compréhension partielle du programme afin de localiser, aussi rapidement que possible, certains segments du code qui doivent être changés pour l'amelioration du code ou pour corriger des bogues.

L'approche systématique fournit une bonne compréhension des interactions existantes entre les fonctionnalités dans le programme, mais est souvent impraticable pour les programmes grands et complexes qui peuvent contenir des millions de lignes de code. L'approche "au besoin", bien que moins chère et moins longue, tend à manquer certaines des interactions non-locales parmi les fonctionnalités. Ces interactions peuvent être critiques pour éviter des effets secondaires inattendus pendant la modification du code.

Une autre approche a été proposé par Wilde et Scully [Wilde et Scully, 1995] pour identifier les fonctionnalités d'un programme en analysant les traces d'exécution de jeux d'essais. Les auteurs utilisent deux ensembles de jeux d'essais pour avoir deux traces d'exécutions : une trace d'exécution où la fonctionnalité est utilisée et une autre trace d'exécution où la fonctionnalité n'est pas utilisée. Ensuite, ils comparent les deux traces pour identifier le code source (classes, méthodes) responsable des différences entre traces. Cependant, les auteurs n'ont utilisé que l'analyse dynamique pour identifier les fonctionnalités des programmes.

Dans notre approche, nous nous sommes inspirés des travaux de Wilde et Scully. Une fois les differences localisées, nous utilisons l'analyse statique pour enrichir les résultats et pour permettre par la suite la représentations des différences entre les traces sous forme de matrices d'adjacence.

5.3.1 Comparaison de traces

Nous utilisons la trace dynamique pour localiser les fonctionnalités programmes comme suit :

- obtenir une trace d'exécution (T1) où la fonctionnalité est utilisé ;
- obtenir une trace d'exécution (T2) où la fonctionnalité n'est pas utilisé ;
- comparer les deux traces pour générer les sous-traces incluant les points communs et les différences.

La comparaison des deux traces permet de localiser les différences qui existent et de déduire les appels entre les classes qui sont responsables de la fonctionnalité étudiée. La comparaison se fait dans les deux sens : extraire les différences qui

existent dans les deux traces (même si à priori T1 doit être incluse dans T2 mais parfois des appels peuvent exister dans T1 et n'existent pas dans T2 ou vice versa, exemple le cas du multithreading où le déplacement de la souris n'est pas reproduit de la même façon dans les deux exécutions).

Les différences localisées sont visualisées sous forme de matrices d'adjacence où chaque matrice représente un bloc de différences contiguës entre les deux traces de comparaison.

5.3.2 Interprétation des résultats

La comparaison des fonctionnalités du programme permet de faire ressortir les différences entre les scenarios et de voir les classes qui en sont responsables. En plus, l'affichage des matrices d'adjacence qui permettent l'affichage des differences entre les traces selon l'ordre de leur apparition, nous permet de nous focaliser sur chaque différence et de comprendre ses principales classes et les relations qui existent avec les autres classes. Le chapitre suivant permettra d'illustrer ces résultats avec un exemple concret.

CHAPITRE 6

IMPLÉMENTATION ET ÉTUDES DE CAS

Ce chapitre décrit notre outil DRAM (Dynamic Relational Adjacency Matrix) qui permet de traiter les matrices d'adjacence, les différents outils qui y ont été intégrés, l'expérimentation faite sur deux systèmes JUnit et JHotDraw pour visualiser les informations à comprendre, identifier les fonctionnalités du programme et détecter les traitements répétitifs et présenter les résultats obtenus. Dans la première section, nous présentons les technologies et les outils utilisés pour l'implémentation et les principales interfaces utilisées. Ensuite, nous présentons notre experimentation et les résultats obtenus.

6.1 Implémentation

DRAM a été réalisé en Java. Il utilise plusieurs outils :

- Caffeine [Guéhéneuc *et al.*, 2002] : pour l'analyse dynamique du programme à étudier.
- Ptidej [Guéhéneuc, 2005] : pour l'analyse statique du programme à étudier.
- VisAdj [Ghoniem *et al.*, 2005a] : pour représenter les données sous forme de matrice d'adjacence.

La figure 6.1 montre les relations entre DRAM, Caffeine et Ptidej.

6.1.1 Caffeine

Pour générer la trace dynamique des programmes, nous avons utilisé Caffeine, un assistant qui aide le mainteneur dans le test de ses conjectures au sujet du

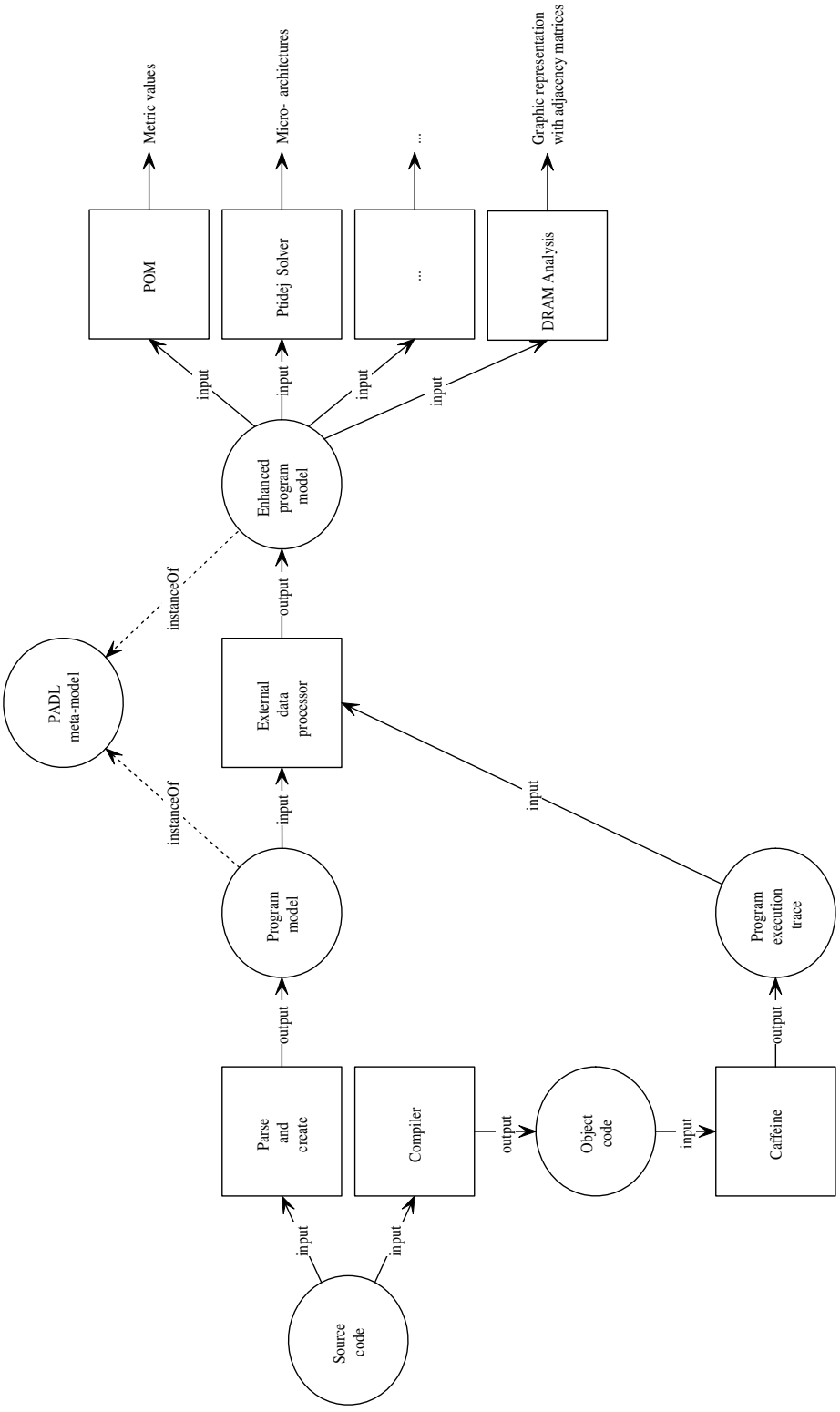

FIG. 6.1 – Utilisation de la suite d'outils Ptidej et de DRAM

comportement de programmes Java. Caffeine est un outil d'analyse dynamique 100% Java qui utilise l'architecture du déboggueur de la plateforme Java pour générer une trace de l'exécution d'un programme java.

Ainsi, Caffeine génère et analyse à la volée la trace d'une exécution du programme Java, selon une requête écrite en Prolog. Le prédicat pour contrôler l'exécution du programme à analyser est :

```
nextEvent([<liste des classes filtres>],
  [<liste des evenements desires a partir du programme>], E)
```

Ce prédicat surveille les classes (et leurs instances) spécifiées par les classes filtres et exécute le programme jusqu'à ce que le prochain événement désiré se produit dans les classes et les instances surveillées. L'événement est alors unifié avec la variable E.

Caffeine se décompose en deux parties principales : la classe *EventManager* et le moteur de Prolog. La classe EventManager utilise la JDI (Java Debug Interface) pour contrôler la JVM (Java Virtual Machine) à distance : elle indique à la JVM distante les événements prévus ; elle collecte les événements du JVM distante ; elle traduit les événements en faits Prolog et elle contrôle l'exécution de la JVM distante comme demandé par le moteur Prolog. Le moteur Prolog résout la requête qui décrit l'analyse à exécuter ; requête qui est exprimée en terme de modèle trace et de modèle d'exécution java. Le modèle trace est définit comme un historique des événements d'exécution du programme. Le modèle de l'exécution est modélisé comme une séquence d'événements d'exécution.

Génération des traces avec Caffeine

Dans notre approche, nous utilisons Caffeine pour générer la trace dynamique des programmes qui sont à l'étude. Nous utilisons Prolog pour exécuter des requêtes concernant les appels aux constructeurs et aux méthodes. Pour obtenir la trace avec le format désiré, nous implémentons un *Listener* qui permet d'intercepter les événements générés par la JVM et pour pouvoir les traiter. Le listener filtre les événements générés par la JVM et retourne seulement les événements concernant les constructeurs et les entrées sorties des méthodes selon le format désiré. Les informations sont enregistrés dans un fichier texte.

La génération de la trace consiste à l'exécution de la classe Caffeine avec les arguments appropriés :

```
public final class CaffeineLauncher {
    public static void main(final String[] args) {
3       ListenerCaffeine listenerCaffeine = new ListenerCaffeine();
        Caffeine.getUniqueInstance().addCaffeineListerner(listenerCaffeine);
        Caffeine
               .getUniqueInstance()
               .start(
8                   "R:/workspace/DRAMTests/src/dram/example/
                     Junit/LookAhead.pl",
10                   <Classpath omitted here>,
11                    "junit.samples.money.MoneyTest1",
12                       new String[] { "junit.*" },
13                       Constants.GENERATE_CONSTRUCTOR_ENTRY_EVENT
                                | Constants.GENERATE_METHOD_ENTRY_EVENT
                                | Constants.GENERATE_METHOD_EXIT_EVENT
                                | Constants.GENERATE_CONSTRUCTOR_EXIT_EVENT);
    }
}
```

– ligne 3 : instantiation du *Listener* qui intercepte les événements de la JVM distente pour les traiter par la suite ;

- ligne 8 : le nom du fichier qui contient la requête Prolog à verifier ;
- ligne 10 : le chemin d'exécution du programme à étudier ;
- ligne 11 : la classe main du programme à analyser ;
- ligne 12 : les filtres d'événements ;
- ligne 13 : les événements requis pour l'analyse. Dans notre approche, l'analyse demande à la JVM distante de générer les événements concernant les constructeurs et les méthodes.

Pour retracer ces événements de la trace qui interceptent les messages des constructeurs et les entrées et sorties des méthodes, nous avons utilisé la requête Prolog suivante :

```
query(N, M) :-
2    nextConstructorEntryStaticEvent(_, _, _),
3    nextMethodEntryStaticEvent(_, _, _, _, _),
4    nextMethodExitStaticEvent(_, _, _, _, _, _),
5    nextConstructorExitStaticEvent(_, _, _),
    write(' time(s)'),
    nl,
    query(N1, M).
    query(N, N).

    main(N1, N2) :-
        query(N1, N2).
```

- ligne 2 : nextConstructorEntryStaticEvent/3 : permet d'avoir chaque instanciation d'un objet ;
- ligne 3 : nextMethodEntryStaticEvent/5 : permet d'avoir chaque appel de méthode ;
- ligne 4 : nextMethodExitStaticEvent/5 : permet d'avoir chaque fin d'appel de méthode ;
- ligne 5 : nextConstructorExitStaticEvent/3 : permet d'avoir chaque fin d'instanciation d'un objet.

La trace est stockée dans un fichier texte pour être exploiter ultérieurement. Le fichier texte contient les informations suivantes :

– dans le cas des constructeurs :

(constructorEntry/constructorExit,classe appelante,classe appelée)

– dans le cas des méthodes :

(methodEntry/methodExit,classe appelante,classe appelée,nom méthode)

Une fois la trace dynamique est générée par Caffeine, nous l'enrichissons avec l'information statique que nous obtenons avec Ptidej.

6.1.2 Ptidej

Pour obtenir des informations statiques, nous utilisons Ptidej (Pattern Trace Identification, Detection, and Enhancement in Java), une suite d'outils qui vise à développer des outils pour évaluer et améliorer la qualité des programmes orientés objets en favorisant l'utilisation des patrons au niveau du language, de la conception et de l'architecture [Amnon et Kazman, 2003].

La suite d'outil permet, par son interface utilisateur (voir figure 6.2), de créer un modèle d'un programme à partir de son code source, pour identifier les relations d'association, d'aggrégation et de composition entre classes et les micro-architectures semblables à un motif de conception et d'appeler sur le modèle divers générateurs d'analyses et outils externes.

Le projet Ptidej est composé de :

– un meta-model, PADL (Pattern and Abstract- level Description Language), pour décrire la structure des motifs (la partie de "solution" dans les definitions des patrons de conception) et les programmes orientés objets ;

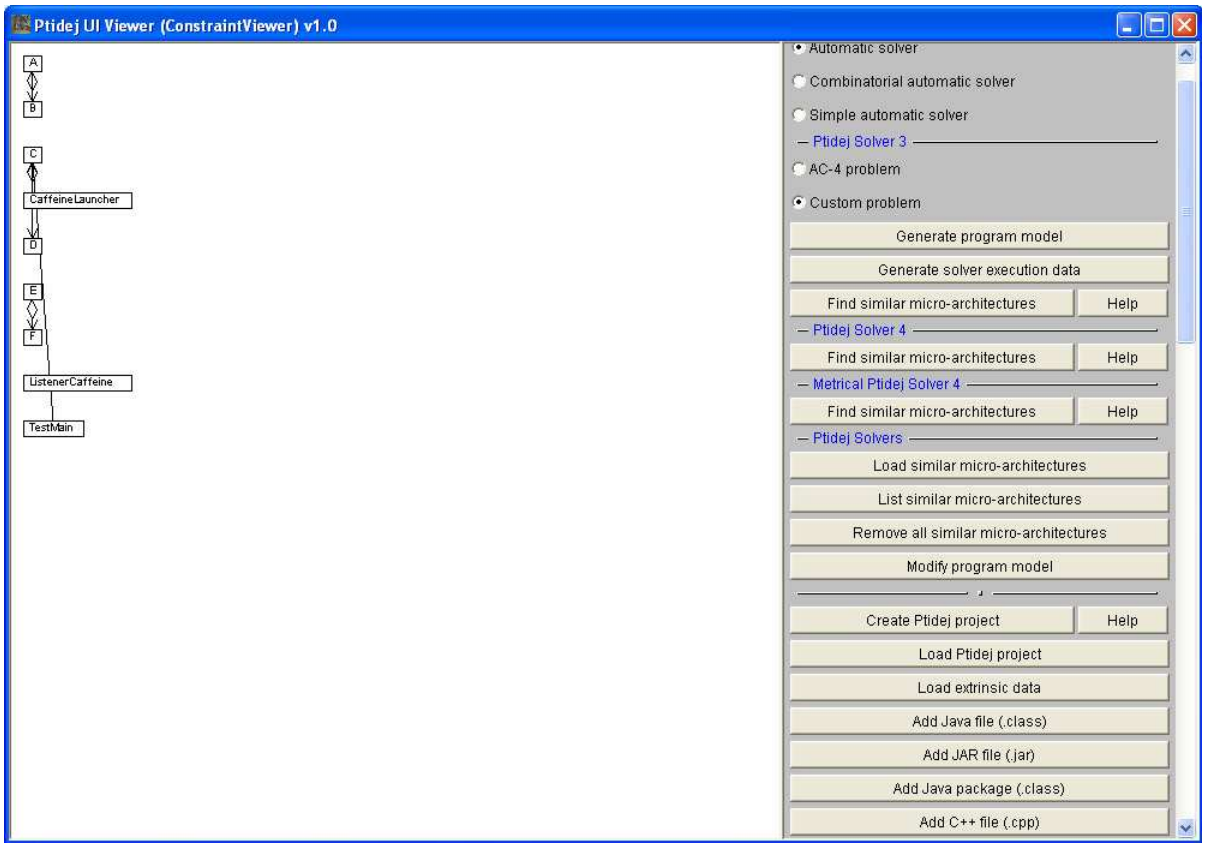

FIG. 6.2 – Interface graphique Ptidej

- une bibliothèque des motifs de conception des patrons de conception [Gamma *et al.*, 1994], incluant **Chain of responsability, Composite, Observer, Visitor**... ;
- plusieurs analyseurs syntaxiques pour construire les modèles des programmes à partir de différentes representations du code source, incluant AOL [Antoniol *et al.*, 1998], C++ et java ;
- une bibliothèque de métriques logiciels, POM (Primitives, Operators, Metrics) [Guéhéneuc *et al.*, 2004] pour calculer des métriques bien connue sur des modèles de programmes, tels que les métrique de Chidamber et de Kemerer [Chidamber et Kemerer, 1994] ;
- une bibliothèque de générateurs et d'analyses à appliquer sur des modèles de programmes et des motifs ;
- un solveur de contraintes avec explications [10], Ptidej Solver, pour identifier les micro-architectures similaire au modèles des motifs dans les modèles de programmes ;
- l'analyseur dynamique pour Java et Caffeine ;
- une bibliothèque graphique, Ptidej UI, pour afficher le modèle des motifs, des programmes et les données dynamiques de Caffeine ;
- plusieurs interfaces utilisateurs pour accéder aux fonctionnalités fournies par la suite d'outils Ptidej.

Nous utilisons la suite d'outils Ptidej pour avoir le type de relations qui existe entre les différentes classes de la trace générée dynamiquement. Ainsi, la trace sera enrichie par le type de relations qui existent entre les différents données de la trace. Nous avons implémenté la classe **DRAMAnalysis** qui permet d'utiliser en entrée

la trace générée par Caffeine et de parcourir le métamodel PADL pour avoir le type de relations qui existent entre les classes de la trace. L'information ainsi obtenue permet de modifier la trace générée par Caffeine et de l'enrichir par le poids des relations qui existent entre les classes (voir exemple page 49).

Toutefois, les informations retournées par Caffeine, par analyse dynamique, et par Ptidej, par analyse statique, ne sont pas concordantes car elles ne fournissent pas le type de relation exacte entre les classes au moment de chaque exécution : avec Caffeine, nous avons une trace sous la forme : *classe appelante - classe appelée - méthode appartenant à la classe appelée et utilisée par classe appelante*, alors qu'avec Ptidej, nous avons les informations suivantes : *classe appelante - classe appelée - méthode appartenant à classe appelante et à travers laquelle on a fait appel à classe appelée*. Les méthodes des classes utilisées ne sont pas les mêmes ce qui rend difficile la correspondance entre les informations dynamiques et les informations statiques. Par exemple : soit deux classes A et B et une relation d'agrégation entre ces deux classes. Lors de l'exécution d'un jeu de test où une fonctionnalité X est exécutée, nous supposons qu'il y a une relation d'association entre ces deux classes. Nous voulons lors de l'analyse statique, avoir cette information et la représenter au moment de l'affichage par matrices d'adjacence en ayant le type de relation exacte entre ces deux classes au moment de l'exécution. Par conséquent, à cause de cette discordance entre les données lors de l'analyse statique et de l'analyse dynamique, nous utilisons le même type de relation entre les classes pour touts les jeux d'essais exécutés.

6.1.3 Visadj

Nous utilisons la suite d'outils Visadj qui a été conçue pour explorer des relations entre les contraintes et les variables dans des problèmes de contrainte. La figure 6.3 montre l'interface graphique de Visadj. Cette interface permet :

- de charger le fichier des contraintes et des variables avec le menu *File* et d'afficher les relations qui existent entre elles ;
- d'ajuster la taille d'affichage des labels avec *Label Size* ;
- de montrer le nombre de liens établis dans chaque représentation matricielle avec *Visible Edges* ;
- d'utiliser les couleurs pour montrer le nombre de fois où les relations contraintes/variables ont été établies avec *Intensity* ;
- de naviguer entre les différentes solutions trouvées avec l'option *Time Range*.

Dans notre approche, nous avons utilisé Visadj pour construire notre outil DRAM. Nous y avons apporté les modifications nécessaires pour mieux répondre à nos besoins.

6.1.4 DRAM

DRAM est une représentation graphique sous forme de matrices d'adjacence des relations qui existent entre les différentes classes d'un programme. Elle permet de représenter une trace dynamique du programme étudié et de parcourir cette trace générée pour étudier l'évolution dans le temps des relations qui existent entre les différentes classes du programme.

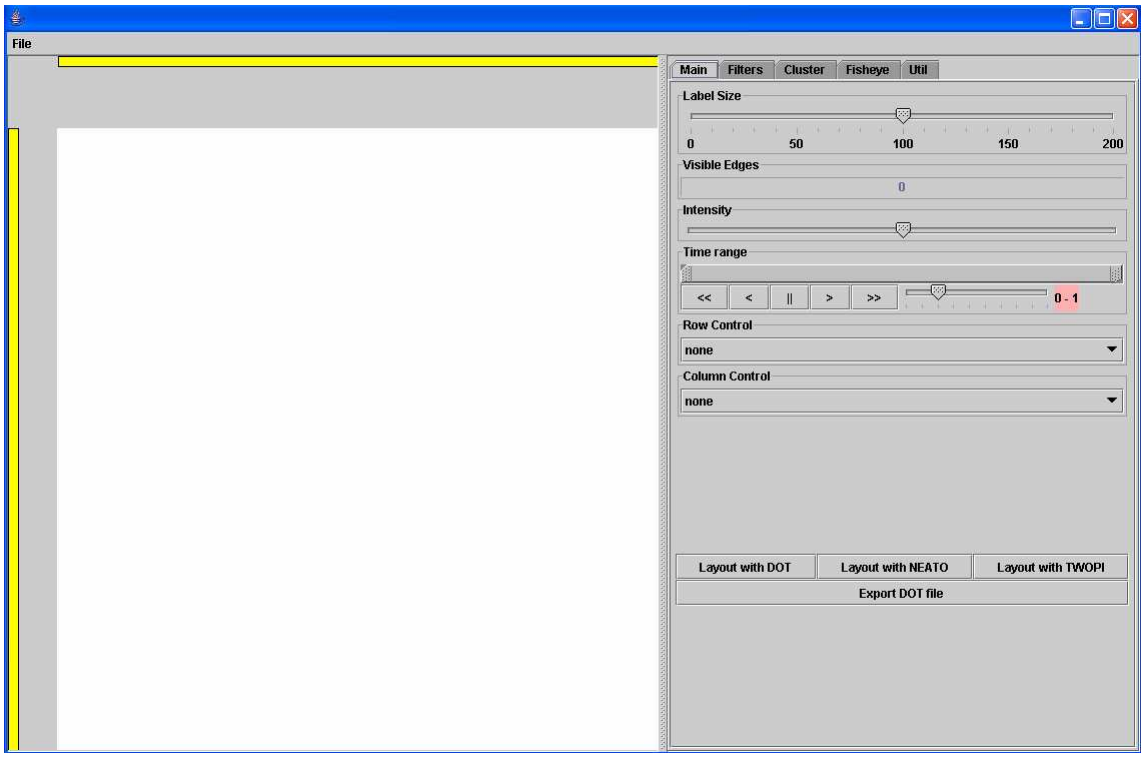

FIG. 6.3 – Interface graphique de Visadj

Nous avons apporté les modifications suivantes par rapport à Visadj :

- nous avons changé l'interface graphique pour ajouter des options au menu principal : en plus de l'option *File* qui permet de charger le fichier, nous avons ajouter les options de regroupement par classes et par méthodes ;
- nous avons changé les paramètres de chargement des graphes pour ne traiter que les graphes dirigés ;
- nous avons une nouvelle classe qui permet générer des traces avec Caffeine ;
- nous avons implémenté les filtres qui permettent d'avoir les événements désirés étudiés selon le choix de l'utilisateur : searchAll(), classSearch(Noms de classes), methodSearchAll(Nom de classe), methodSearch(Noms de classes, Noms de méthodes), methodSearchB(Nom de classe, Noms de méthodes), constructor-Search(Nom de classe), constructorSearchB(Nom de classe) ;
- nous avons changé le poids associé au lien qui existe entre deux sommets : au lieu d'avoir le nombre de fois que le lien a été établi, nous utilisons le poids associé au type de relation qui existent entre deux classes ;
- nous avons rajouté la possibilité de regrouper par classes ou classes et méthodes la trace étudiée ;
- nous avons conçu un algorithme de comparaison entre deux fichiers et un outil pour visualiser les differences entre les fichiers en modifiant (en incrémentant) l'increment à chaque fois qu'il y a un bloc de différences de données. Cette incrémentation dans le temps permet de visualiser l'évolution des differences qui existent entre les deux fichiers.
- nous avons ajouté la possibilité d'enregistrer les représentations matricielles générées par le defilement du temps dans des fichiers pour un emploi et une étude ultérieure.

Le code source de DRAM est disponible sur le site web *http ://ptidej.iro.umontreal.ca/participants/Members/rachedsa*.

Étapes d'exécution du programme DRAM

Pour obtenir les représentations matricielles des programmes en cours d'étude, les étapes suivantes doivent être suivies :

- exécuter Caffeine pour avoir la trace dynamique du programme à étudier. L'exécution se fait à l'aide des classes *CaffeineLauncher* et *ListenerCaffeine* ;
- filtrer la trace ainsi générée en utilisant la classe *TxtReader* pour avoir une sous–trace avec les informations que nous désirons traiter. Le filtre se fait à l'aide et selon le choix des méthodes : searchAll(), classSearch(Noms de classes), methodSearchAll(Nom de classe), methodSearch(Noms de classes, Noms de méthodes), methodSearchB(Nom de classe, Noms de méthodes), constructorSearch(Nom de classe), constructorSearchB(Nom de classe) ;
- soumettre la sous–trace générée à la suite d'outils Ptidej pour avoir le type de relations qui existent entre les différentes classes. Les informations obtenues modifient le fichier de la trace avec le poids de relation entre les classes correspondantes. Après avoir chargé le projet avec *Load Ptidej project* l'utilisateur choisit le bouton DRAM Analysis pour obtenir une sous–trace étendue avec les poids des relations qui existent entre les classes.
- exécuter DRAM pour obtenir la représentation matricielle de la trace modifiée à l'aide de la classe *DRAMAdjacencyMatrix*. La figure 6.4 montre l'interface graphique de notre outil : l'interface permet d'ouvrir la trace étendue. Il

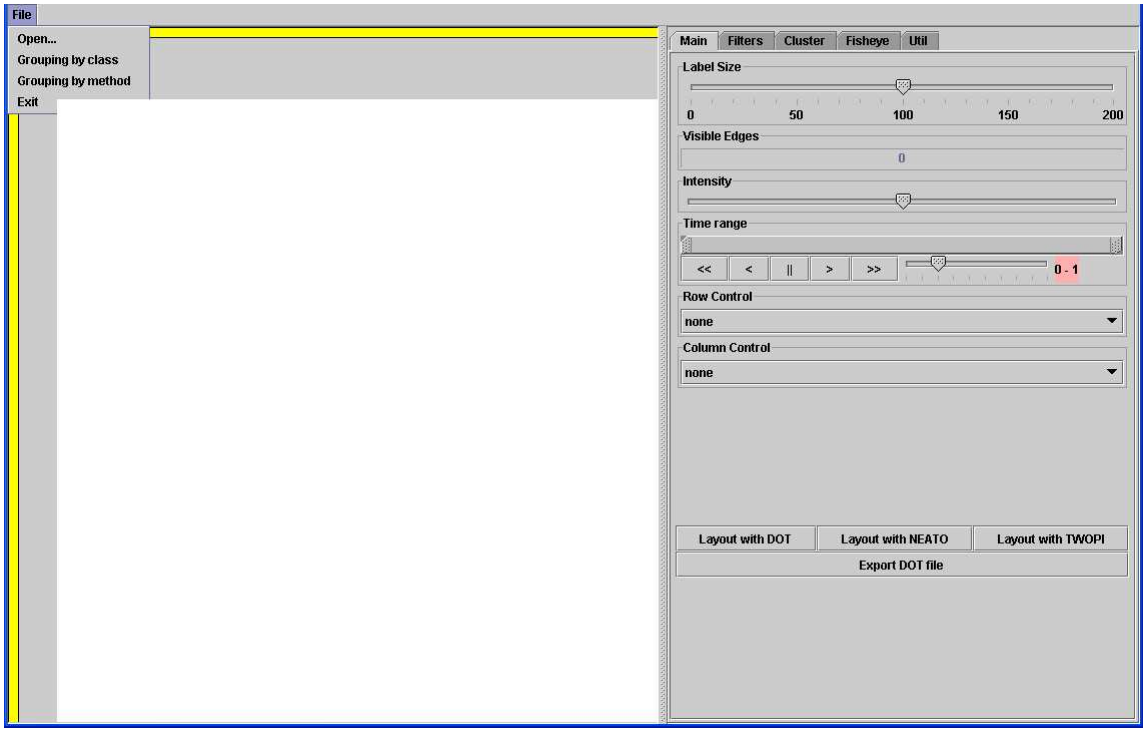

FIG. 6.4 – Interface graphique de DRAM

permet aussi de regrouper la trace selon les classes choisies. L'option *Time Range* qui se trouve dans le menu droit de l'interface permet de parcourir la trace en sauvegardant les différents regroupements dans des fichiers *JPEG* pour un traitement ultérieur ;

– de comparer des traces à l'aide de la classe *FileComparaison* qui permet d'identifier les blocs où les appels aux constructeurs et aux méthodes sont différents et de les ordonner chronologiquement pour les visualiser comme à l'étape précédente.

6.2 Études de cas

Nous utilisons deux études de cas : JUnit et JHotDraw pour visualiser les informations à comprendre, identifier les fonctionnalités du programme et détecter les traitements répétitifs. Nous avons choisi ces programmes car ils sont connus et des informations architecturales les concernant sont disponibles : diagrammes de classes, études [Gamma et Eggenschwiler, 1998, Gamma et Beck, 1998, Seemann et von Gudenberg, 1998, Microsystems., 2002, Kaiser, 2001, Gamma et Beck, 2002]. Dans cette section, nous utilisons JUnit pour tester les regroupement par classes afin de dégager l'information à comprendre et les traitements répétitifs qui sont dans le programme et JHotDraw pour la recherche des fonctionnalités du programme.

6.2.1 JUnit

JUnit est un framework qui permet d'écrire des tests répétitifs. C'est une instance de l'architecture xUnit pour les frameworks de tests d'unité. JUnit [Gamma et Beck, 1998] définit comment structurer les tests et fournit des outils pour les exécuter.

Nous utilisons JUnit pour comprendre les comportements des classes du jeux d'essais utilisés et détecter les traitements répétitifs en regroupant par classes et en visualisant les résultats sous forme de matrices d'adjacence. Nous utilisons l'exemple fournit avec JUnit à savoir les tests unitaires *MoneyTest* du package *junit.samples.money*. C'est un exemple qui permet de résoudre le problème

de représentation arithmétique avec des devises multiples. *MoneyTest* est un ensemble de tests que nous exécutons en mode textuel. Pour plus de lisibilité des représentations, nous n'exécutons qu'un seul test de l'ensemble des tests fournit par *MoneyTest* : *testBagMultiply()*. Nous étudions le regroupement par classes du test unitaire *testBagMultiply()*. Nous présentons les différentes matrices d'adjacence des regroupement et nous analysons les résultats obtenus. Cependant, le mainteneur doit avoir une idée au préalable des classes qui lui sembleront les plus intéressantes pour la compréhension du programme afin de faire le regroupement sinon, il peut faire plusieurs essais pour dégager les classes les plus pertinentes dans la compréhension du programme en cours d'étude.

Les matrices d'adjacence sont obtenues de la manière suivante :

- nous générons la trace dynamique avec Caffeine en exécutant *MoneyTest* avec : *testBagMultiply()* ;
- nous faisons une première extraction avec la méthode *methodSearch* de *TxtReader* avec la classe *junit.textui.TestRunner* et la méthode *run*. La figure 6.5 montre la représentation matricielle de cette sous-trace ;
- nous exécutons Ptidej pour enrichir la trace dynamique avec les données statiques et nous pouvons voir le résultat en utilisant l'icône *Intensity* ;
- nous regroupons la trace dynamique selon plusieurs choix de classes : *TestSuite* dans la figure 6.6, *Assert* dans la figure 6.7, *Money* dans la figure 6.8.

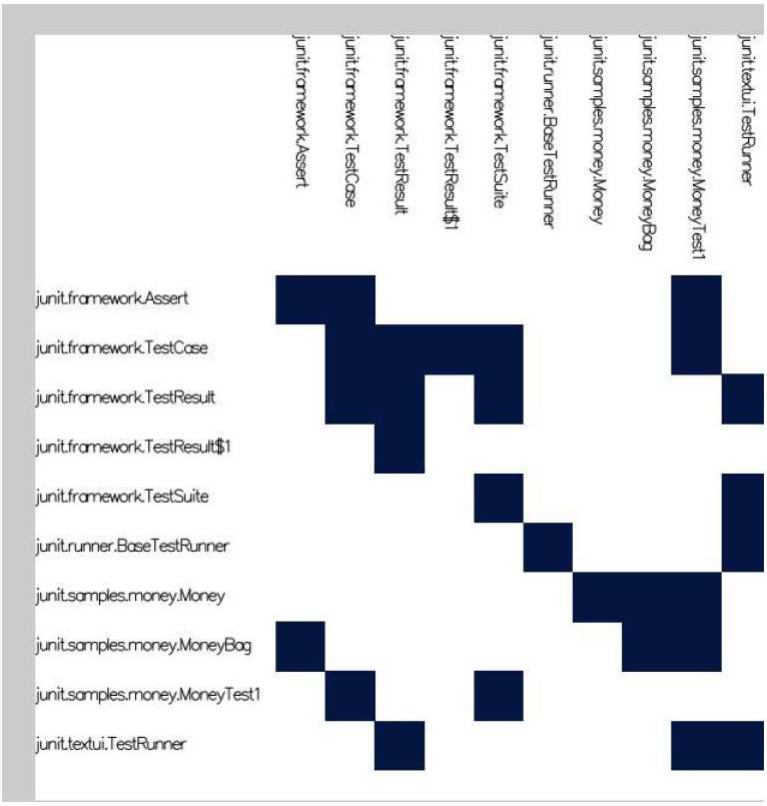

FIG. 6.5 – Matrice d'adjacence de la classe junit.samples.money.MoneyTest de JUnit avec une seule Méthode : *testBagMultiply()*

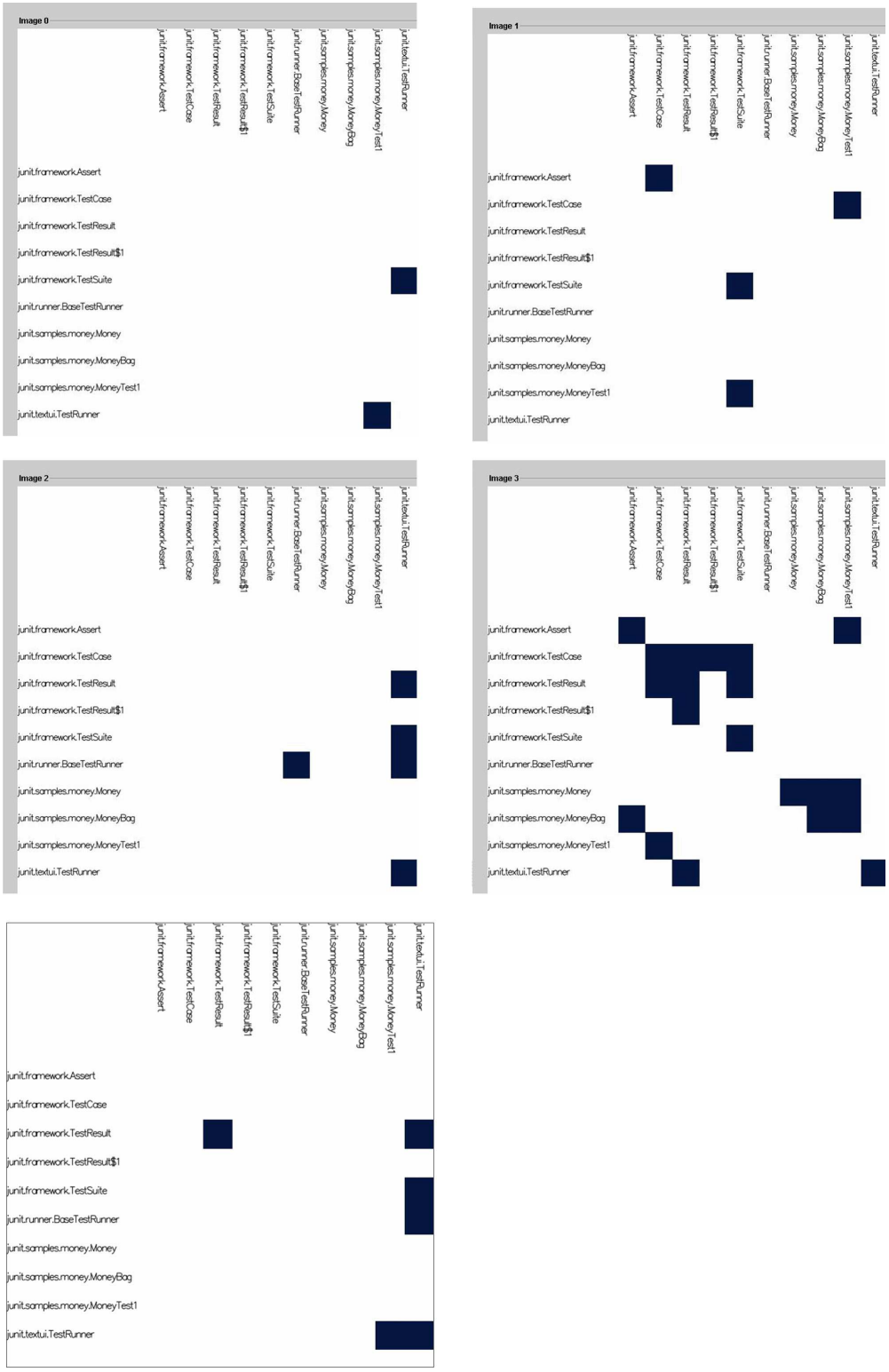

FIG. 6.6 – Matrice d'adjacence de la classe junit.samples.money.MoneyTest de JUnit avec une seule Méthode *testBagMultiply()* groupé par la classe *TestSuite*

FIG. 6.7 – Matrice d'adjacence de la classe junit.samples.money.MoneyTest de JUnit avec une seule Méthode *testBagMultiply()* groupé par la classe *Assert*

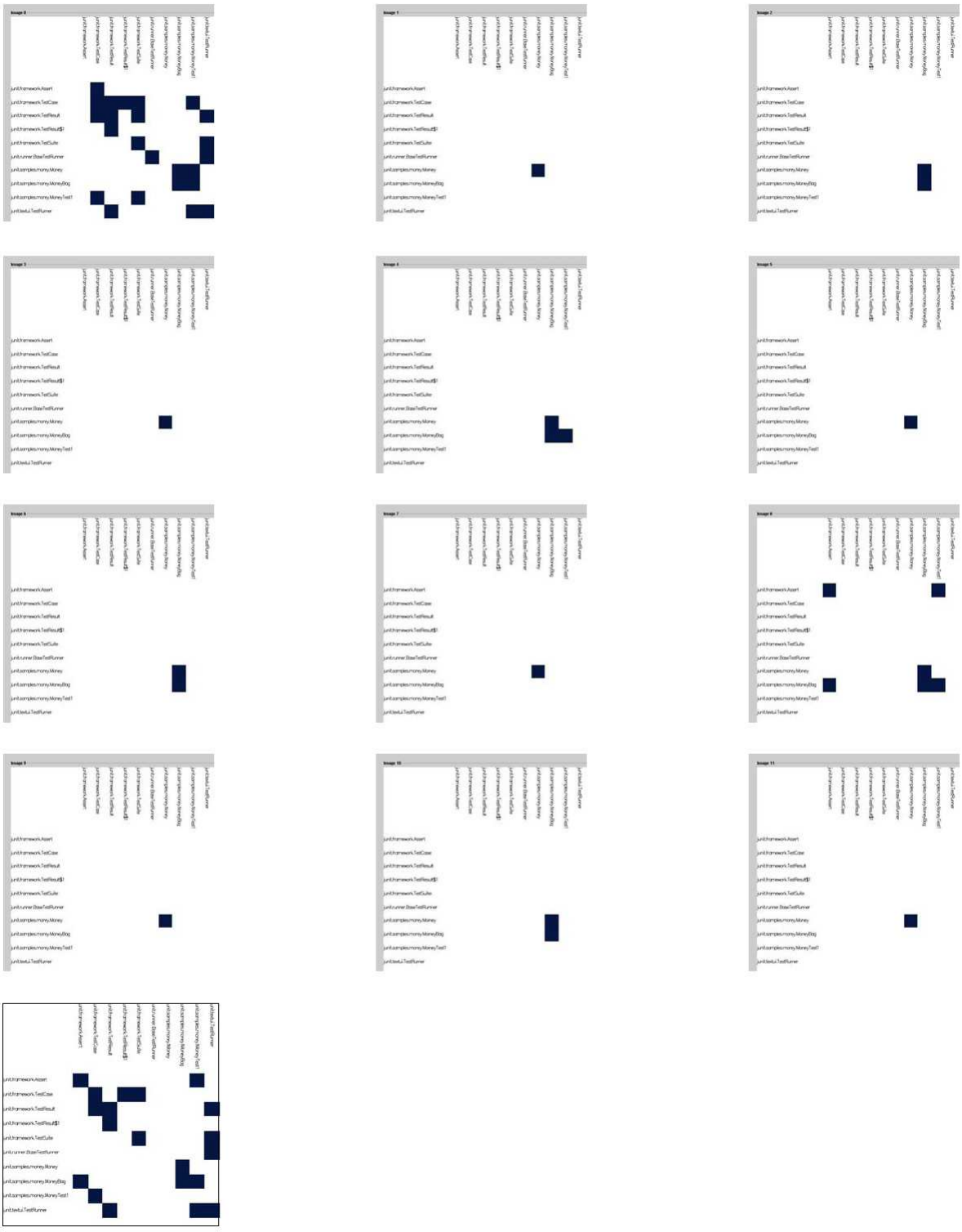

FIG. 6.8 – Matrice d'adjacence de la classe junit.samples.money.MoneyTest de JUnit avec une seule Méthode *testBagMultiply()* groupé par la classe *Money*

Résultat

Les représentations des figures 6.6, 6.7 et 6.8 visualisent le regroupement par classes du test unitaire *testBagMultiply()*. Ces figures représentent l'évolution de l'exécution de la trace dynamique par rapport aux classes choisies pour le regroupement. Chaque présentation montre les événements générés avant, pendant et après l'appel des classes du regroupement qui est fait soit par une instanciation ou une entrée/sortie des méthodes qui lui appartiennent. Par exemple, la figure 6.6 montre que la classe *TestSuite* du regroupement est responsable des événements générés des représentations 2 et 4 de la figure 6.6. La figure 6.8 montre que la classe *Money* dans les représentations 2, 4, 6, 8, 10 et 12 fait toujours appel à à elle -même selon les tests du jeu d'essais utilisés. Nous pouvons déduire que la classe *Money* ne fait appel qu'à elle même et ne fait appel à aucune autre classe selon le jeu d'essai utilisé. De plus, ces représentations montrent que le même traitement est effectué à chaque fois que la classe de regroupement est appelée, ce qui permet de dégager les traitements répétitifs qui se déroulent dans une exécution (ce dégagement se fait, dans un premier temps, visuellement).

En conclusion, le regroupement par classes permet de réduire considérablement le temps de compréhension du programme ou des "classes" qui sont à l'étude. Le déroulement dans le temps de la trace dynamique permet de savoir les traitements qui sont fait avant, pendant et après le traitement fait par les classes du regroupent.

6.2.2 JHotDraw

JHotDraw [Gamma et Eggenschwiler, 1998] est un programme de dessin vectoriel dont l'architecture met en oeuvre de nombreux patrons de conception. Ses

auteurs originels, Erich Gamma et Thomas Eggenschwiler, ont commencé son implémentation pour expérimenter l'utilisation des patrons et ont continué son développement devant les succès rencontrés. Les patrons utilisés sont décrits dans la documentation du programme. En particulier, la conception et l'implémentation de JHotDraw utilise le patron de conception Composite. C'est maintenant un logiciel libre, partagé sur SourceForge.net. Nous utilisons JHotDraw pour plusieurs raisons :

- sa conception et implantation utilisent de nombreux patrons de conception ;
- sa documentation fournit une liste des patrons de conception utilisés ;
- sa taille est raisonnable : 155 classes réparties dans 11 paquetages pour un total de 16 015 lignes de code commenté ;
- il est utilisé et étendu dans de nombreux programmes [Kaiser, 2001] ;
- il a été conçu et implémenté indépendamment de nos recherches, il constitue donc un cas réel d'utilisation ;
- il est disponible librement, accessible depuis members.pingnet.ch/gamma/JHD-5.1.zip ou jhotdraw.sourceforge.net/.

Nous utilisons JHotDraw pour chercher les classes responsables de fonctionnalités du programme en comparant des traces d'exécution. Dans notre exemple, nous cherchons les classes responsables de la fonctionnalité *dessein d'un rectangle*. Les matrices d'adjacence correspondantes sont obtenues de la manière suivante :

- nous générons deux traces dynamiques avec Caffeine en exécutant la classe JavaDrawApp de l'application JHotDraw et en utilisant une instance de la classe Robot du package java.awt.Robot :
  - une trace où nous lançons l'exécution de JHotDraw et ensuite nous la

fermons avec l'option exit ;

– une trace où nous lançons l'exécution de JHotDraw, nous dessinons un rectangle et ensuite nous fermons JHotDraw avec l'option exit.

Pour faire la comparaison entre deux traces et pour éviter d'avoir des événements différents générés par le déplacement de la souris, nous avons utilisé une instance de la classe Robot du package java.awt.Robot pour exécuter à notre place le programme à étudier. Cette classe permet de prendre la main et d'exécuter plusieurs scénarios du programme en respectant (à peu prés) les mêmes étapes évitant ainsi de générer des déplacements de la souris qui seront superflus lors de la comparaison des traces.

– nous exécutons Ptidej pour enrichir les deux traces dynamiques avec les données statiques ;
– nous comparons les deux traces avec la classe FileComparaison ;
– nous présentons toutes les différences qui existent entre les traces dans une matrice d'adjacence (voir figure 6.9)[1].

[1]La figure 6.9 contient 39 sommets, ce qui rend la lisibilité un peu difficile sur le papier. Les noms des classes qui sont représentés sont : CH.ifa.draw.application.DrawApplication, CH.ifa.draw.application.DrawApplication$7, CH.ifa.draw.figures.AttributeFigure, CH.ifa.draw.figures.FigureAttributes, CH.ifa.draw.figures.GroupCommand, CH.ifa.draw.figures.RectangleFigure, CH.ifa.draw.figures.UngroupCommand, CH.ifa.draw.framework.DrawingChangeEvent, CH.ifa.draw.framework.FigureChangeEvent, CH.ifa.draw.samples.javadraw.AnimationDecorator, CH.ifa.draw.samples.javadraw.BouncingDrawing, CH.ifa.draw.samples.javadraw.JavaDrawApp, CH.ifa.draw.standard.AbstractFigure, CH.ifa.draw.standard.AbstractTool, CH.ifa.draw.standard.AlignCommand CH.ifa.draw.standard.BringToFrontCommand CH.ifa.draw.standard.BufferedUpdateStrategy CH.ifa.draw.standard.CompositeFigure CH.ifa.draw.standard.CopyCommand CH.ifa.draw.standard.CreationTool CH.ifa.draw.standard.CutCommand CH.ifa.draw.standard.DecoratorFigure CH.ifa.draw.standard.DeleteCommand CH.ifa.draw.standard.DuplicateCommand CH.ifa.draw.standard.FigureChangeEventMulticaster CH.ifa.draw.standard.FigureEnumerator CH.ifa.draw.standard.SendToBackCommand CH.ifa.draw.standard.StandardDrawing CH.ifa.draw.standard.StandardDrawingView CH.ifa.draw.standard.ToolButton CH.ifa.draw.util.ColorMap CH.ifa.draw.util.Command CH.ifa.draw.util.CommandMenu CH.ifa.draw.util.Geom CH.ifa.draw.util.PaletteButton CH.ifa.draw.util.PaletteIcon junit.framework.AssertionFailedError junit.framework.TestFailure

– nous pouvons parcourir les différences entre les traces selon leur ordre d'apparition (voir figure 6.10).

Résultat

Les représentations de la figure 6.10 présentent les différences entre les traces dynamiques exécutées. En examinant ces différentes présentations, il en ressort les classes responsables du dessein du rectangle comme les classes *CH.ifa.draw.figures.RectangleFigure*, *CH.ifa.draw.figures.FigureAttributes*, *CH.ifa.draw.util.PaletteIcon*... Ainsi, il est plus aisé de trouver et de comprendre les classes responsables de la fonctionnalité du dessein du rectangle dans l'application JHotDraw. Nous pouvons, après avoir trouvé les classes responsables de la fonctionnalité *dessein du rectangle*, regrouper par ces classes ou certaines de ces classes pour savoir les relations qui sont établies au moment de l'exécution du programme et voir l'évolution de cette exécution dans le temps. Ainsi, nous comprendrons mieux le déroulement de l'exécution de la trace et le rôle de chaque classe ou groupe de classes dans ce déroulement (pour voir un exemple, se référer à l'exemple donné avec JUnit dans la sous-section 6.2.1 de la page 76).

En conclusion, la comparaison des traces permet de détecter les différences entre deux exécutions du jeu d'essaies permettant ainsi de dégager les classes responsables des fonctionnalités désirées à trouver. Toutefois, la comparaison peut faire apparaître des différences qui ne sont pas très importantes pour la compréhension des fonctionnalités que nous voulons étudier, ce qui nécessite l'application de filtres pour limiter l'apparition de ce type d'informations.

---

robot.RJhotDrawFigure

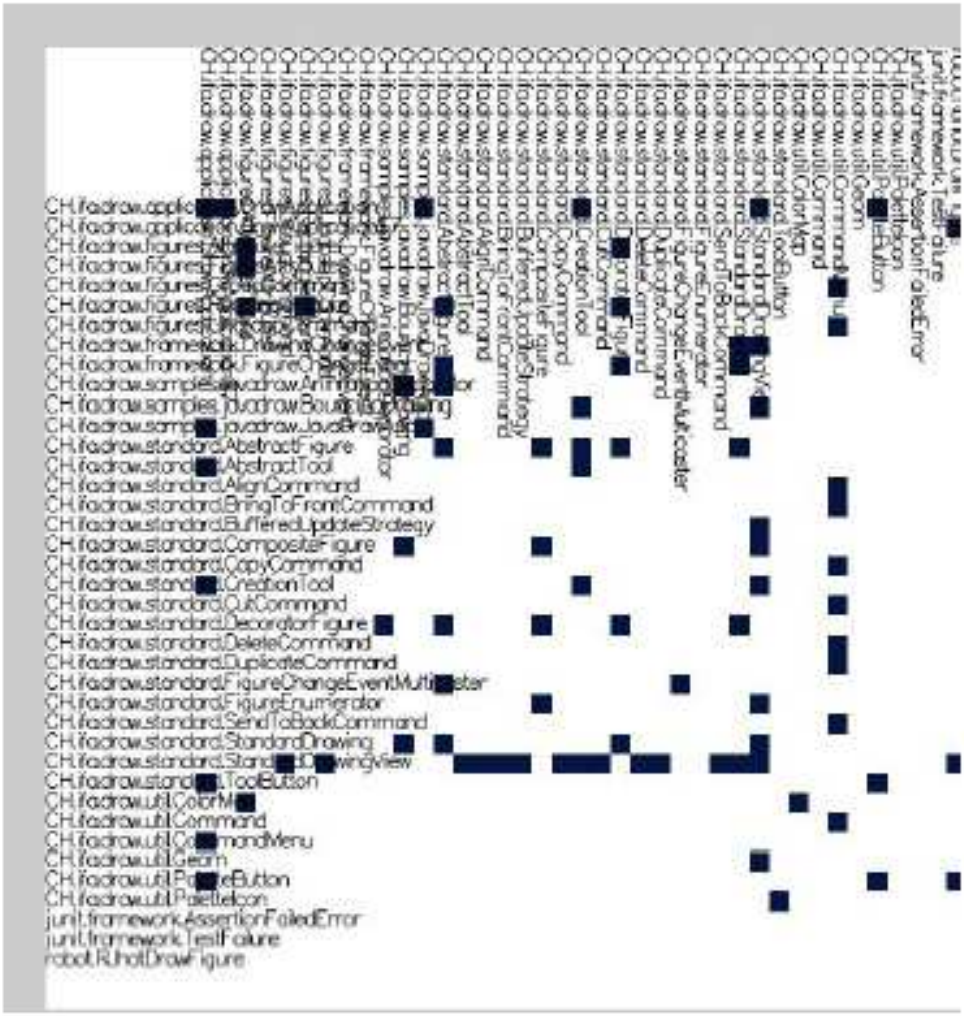

FIG. 6.9 – Matrice d'adjacence des différences entre les traces d'exécution de JHotDraw[1]

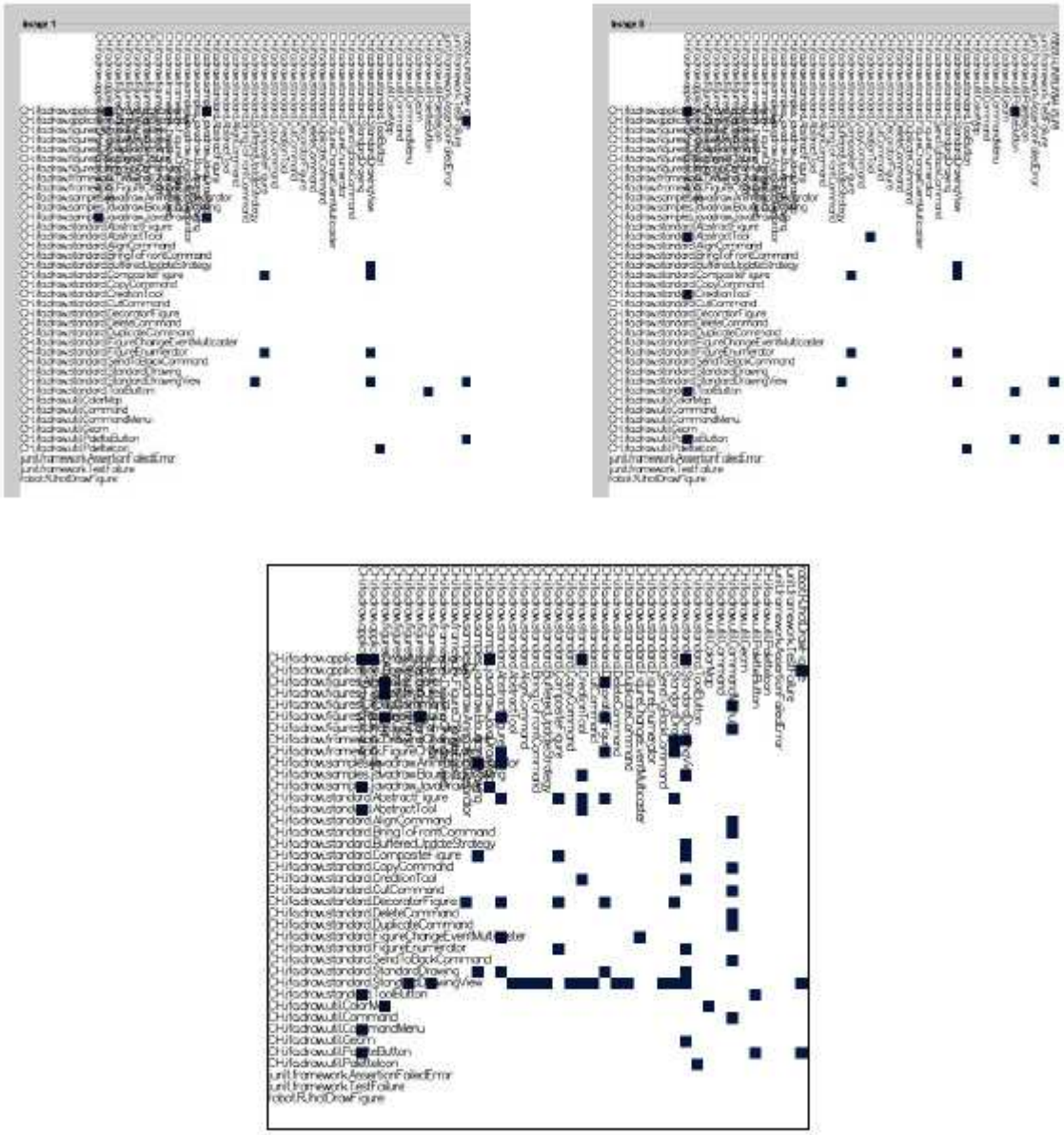

FIG. 6.10 – Matrices d'adjacences des différences entre les traces d'exécution de JHotDraw selon leur ordre chronologique

CHAPITRE 7

CONCLUSION ET TRAVAUX FUTURS

Dans ce mémoire, nous avons présenté DRAM, un système d'analyse du comportement des programmes à l'aide des matrices d'adjacence, dont le but est de répondre aux problèmes de compréhension des programmes par les mainteneurs. La difficulté consiste à comprendre un programme qui est, le plus souvent, écrit par autrui afin d'y apporter des modifications ou des extensions de fonctionnalités. DRAM permet de visualiser plus rapidement les informations dynamiques et statiques dans une représentation compacte, d'avoir une vue architecturale du programme étudié tout au long de l'investigation du mainteneur, de détecter les traitements répétitifs et, enfin, identifier les fonctionnalités du programme auxquelles s'intéressent les mainteneurs.

Nous avons présenté dans le chapitre de l'état de l'art plusieurs travaux qui traitent des mêmes problèmes (ou en partie) que DRAM. La plupart de ces travaux se sont principalement intéressés à aider les mainteneurs dans la compréhension des programmes et à les aider à chercher les classes responsables des fonctionnalités du programme que les mainteneurs désirent comprendre. Nous nous sommes inspirés de ces travaux et nous avons conçu DRAM pour présenter les informations recherchés sous forme visuel pour une meilleure perception et une meilleure visibilité. Nous avons utilisé les matrices d'adjacence car elles offrent de meilleures manipulations que les représentations traditionnelles (nœuds–liens ou UML). Nous avons générés des traces d'exécution à l'aide de l'analyse dynamique et nous avons enrichi

ces traces à l'aide de l'analyse statique pour avoir les relations statiques et dynamiques entre les classes des programmes étudiés. Nous avons développé plusieurs algorithmes pour permettre une manipulation plus facile des données à visualiser à savoir : filtrer les données à visualiser, regrouper par classes et/ou méthodes et enfin comparaison de traces d'exécution pour trouver les classes responsables des fonctionnalités du programme. Nous avons testé nos algorithme sur deux applications JUnit et JHotDraw et nous avons obtenus des résultats satisfaisant qui nous permettent de chercher à améliorer notre outil DRAM.

Perspectives

DRAM nous a aidé à mieux comprendre les programmes étudiés, toutefois plusieurs améliorations peuvent être apportés pour mieux répondre aux besoins des mainteneurs.

An niveau de l'analyse dynamique, nous avons utilisé *Caffeine* pour générer les traces d'exécution des jeux d'essais utilisés, toutefois, *Caffeine* intercepte tous les événements générés par la JVM, ce qui rend la trace très volumineuse et peut contenir des informations qui ne sont pas très intéressantes pour la compréhension des programmes. Par conséquent, nous proposons d'instrumenter le code source pour n'avoir que les événements d'intérêts pour les mainteneurs. De plus, une technique de compression de données semble nécessaire pour la gestion des quantités de données générées lors de l'exécution des programmes.

Au niveau de Ptidej, nous voudrions fournir le type de relation entre deux classes au moment de chaque exécution, ce qui permettra d'avoir une vue architecturale plus exacte à chaque exécution du programme.

An niveau de la détection des traitements répétitifs, nous pouvons améliorer cette détection en utilisant l'analyse formelle des concepts (AFC) [Ganter et Wille, 1999, Wille, 1982] qui constitue une technique puissante de découverte et de structuration de connaissances. Elle fournit une méthode universelle de dérivation de hiérarchies d'abstractions à partir d'un ensemble d'entités. L'AFC représente les entités par le biais d'un *context formel* composé d'un ensemble d'individus (*objets formels*), d'un ensemble de caractéristiques (*attributs formels*) et d'une relation binaire permettant de spécifier l'incidence objet-attribut.

Les algorithmes de l'AFC permettent l'identification de groupements d'objets formels ayant des attributs formels communs. Ces groupements, appelés *concepts formels* sont organisés en une hiérarchie conceptuelle, appelée *treillis de concepts*, où les attributs formels sont factorisés de façon maximale.

Les mécanismes d'analyse de l'AFC sont basés sur la construction, visualisation et exploration du treillis ou de ses structures dérivées. Depuis l'établissement du paradigme de l'AFC, la technique a été utilisé avec succès dans beaucoup de domaines tels que l'analyse de données, le génie logiciel, le data-mining et la recherche d'information.

En génie logiciel, traditionnellement, l'AFC a fourni des outils pour la résolution de multiples problèmes nécessitant le regroupement d'un ensemble d'entités où l'organisation et la réutilisation de l'information sont des critères de bonne qualité.

L'intérêt derrière cette utilisation réside dans les qualités importantes de la méthode de base de l'AFC et du résultat obtenu. En effet, les difficultés dans la création et la maintenance des modèles UML permet à l'AFC de fournir un environnement naturel de représentation de connaissance sur la structure d'un ensemble d'individus.

Ainsi, pour notre approche, nous pouvons améliorer la recherche des traitements répétitifs et avoir des abstractions de plus haut niveaux dans les programmes à étudier d'une manière plus formelle en utilisant l'AFC qui semble un aspect intéressant à développer.

Au niveau de la comparaison des traces, des filtres pour éliminer les informations superflues à la compréhension des programmes semblent nécessaires.

Avoir une vue quantitative des algorithmes utilisés pour mieux illustrer les résultats.

Enfin, notre approche devra être testée sur des applications plus grandes avec un nombre de classes plus grand pour cohorborer les résultats déjà obtenus avec JUnit et JHotDraw.

BIBLIOGRAPHIE

[Albin-Amiot *et al.*, 2002]

Hervé Albin-Amiot, Pierre Cointe et Yann-Gaël Guéhéneuc. Un méta-modèle pour coupler application et détection des design patterns. Michel Dao et Marianne Huchard, éditeurs, *actes du* 8^e *colloque Langages et Modèles à Objets*, volume 8, numéro 1-2/2002 de *RSTI – L'objet*, pages 41–58. Hermès Science Publications, janvier 2002.

Disponible à : `www.yann-gael.gueheneuc.net/Work/Publications/`.

[Amnon et Kazman, 2003]

H.Eden Amnon and Rick Kazman. Architecture, design, implementation. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 149–159, Washington, DC, USA, 2003. IEEE Computer Society. ISBN: 0-7695-1877-X.

[Antoniol *et al.*, 1998]

G. Antoniol, R. Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. In *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, page 153, Washington, DC, USA, 1998. IEEE Computer Society. ISBN: 0-8186-8560-3.

[Antoniol et Guéhéneuc, 2005]

Giuliano Antoniol and Yann-Gaël Guéhéneuc. Feature identification: A novel approach and a case study. In Tibor Gyimóthy and Vaclav Rajlich, editors, *proceedings of the* 21^{st} *International Conference on Software Maintenance*. IEEE Computer Society Press, September 2005. To appear.

[AT&T Labs Research., 2004]

AT&T Labs Research. Graphviz - open source graph drawing software, 2004. Available at: `http://www.research.att.com/sw/tools/graphviz/`.

[Bassil et Keller, 2001]

S. Bassil and R. K. Keller. Software visualization tools: Survey and analysis. In *IWPC '01: Proceedings of the 9th International Workshop on Program Comprehension*, page 7, Washington, DC, USA, 2001. IEEE Computer Society.

[Bassil, 2000]

Sarita Bassil. Évaluation qualitative et quantitative doutils de visualisation logicielle. Thèse de mastère, Faculté des Études Supérieures, Université de Montréal, Décembre 2000.

[Battista *et al.*, 1999]

G. Battista, P. Eades, R. Tamassia, and I. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, Upper Saddle River, 1999.

[Bertin, 1967]

J. Bertin. *Sémiologie graphique : Les diagrammes -Les réseaux - Les cartes*. Editions de l'Ecole des Hautes Etudes en Sciences, Paris, France, 1967.

[Buckley et Haray, 1990]

F. Buckley and F. Haray. *Distance In Graphs*. Addison-Wesley, p.117, 1990.

[Carpendale et Montagnese, 2001]

M. S. T. Carpendale and Catherine Montagnese. A framework for unifying presentation space. In *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 61–70, New York, NY, USA, 2001. ACM Press. ISBN: 1-58113-438-X.

[Chidamber et Kemerer, 1994]

S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, Piscataway, NJ, USA, 1994. IEEE Press.

[Domingue *et al.*, 1992]

John Domingue, Blaine Price, and Marc Eisenstadt. A framework for describing and implementing software visualization systems. In *Proceedings of the conference on Graphics interface '92*, pages 53–60, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc. ISBN: 0-9695338-1-0.

[Downey *et al.*, 1980]

Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, New York, NY, USA, 1980. ACM Press.

[Eades et Sugiyama, 1990]

Peter Eades and Kozo Sugiyama. How to draw a directed graph. *J. Inf. Process.*, 13(4):424–437, Tokyo, Japan, Japan, 1990. Information Processing Society of Japan.

[Eichelberger, 2002]

Holger Eichelberger. Sugibib. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Proc. Graph Drawing, 9th International Symposium, GD '02*, volume 2265 of *Lecture Notes in Computer Science*, pages 467–468. Springer, Springer, 2002.

[Ernst *et al.*, 2001]

Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution.

IEEE Trans. Softw. Eng., 27(2):99–123, Piscataway, NJ, USA, 2001. IEEE Press.

[Ernst, 2003]

M. Ernst. Static and dynamic analysis: synergy and duality, 2003.

Available at: `citeseer.ist.psu.edu/ernst03static.html`.

[Fekete et Plaisant, 1999]

Jean-Daniel Fekete and Catherine Plaisant. Excentric labeling: dynamic neighborhood labeling for data visualization. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 512–519, New York, NY, USA, 1999. ACM Press. ISBN: 0-201-48559-1.

[Flajolet *et al.*, 1990]

Philippe Flajolet, Paolo Sipala, and Jean-Marc Steyaert. Analytic variations on the common subexpression problem. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 220–234, New York, NY, USA, 1990. Springer-Verlag New York, Inc. ISBN: 0-387-52826-1.

[Frick, 1997]

Arne Frick. Upper bounds on the number of hidden nodes in sugiyama's algorithm. In *GD '96: Proceedings of the Symposium on Graph Drawing*, pages 169–183, London, UK, 1997. Springer-Verlag. ISBN: 3-540-62495-3.

[Gamma *et al.*, 1994]

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1^{st} edition, 1994. ISBN: 0-201-63361-2.

[Gamma et Beck, 1998]

Erich Gamma and Kent Beck. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50. SIGS Publications, July 1998.

Available at: `junit.sourceforge.net/doc/testinfected/testing.htm`.

[Gamma et Beck, 2002]

Erich Gamma and Kent Beck. JUnit. Web site, 2002.

Available at: `www.junit.org/`.

[Gamma et Eggenschwiler, 1998]

Erich Gamma and Thomas Eggenschwiler. Jhotdraw. Web site, 1998.

Available at: `members.pingnet.ch/gamma/JHD-5.1.zip`.

[Gansner *et al.*, 1993]

E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng.*, 19(3):214–230, Piscataway, NJ, USA, 1993. IEEE Press.

[Ganter et Wille, 1999]

B. Ganter and R. Wille. *Formal Concept Analysis, Mathematical Foundations*. Springer, Berlin, 1999.

[Ghoniem *et al.*, 2004]

Mohammad Ghoniem, Jean-Daniel Fekete, and Philippe Castagliola. A comparison of the readability of graphs using node-link and matrix-based representations. In *INFOVIS*, pages 17–24, 2004.

[Ghoniem *et al.*, 2005a]

Mohammad Ghoniem, Hadrien Cambazard, Jean-Daniel Fekete, and Narendra Jussien. Peeking in solver strategies using explanations visualization of dynamic

graphs for constraint programming. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 27–36, New York, NY, USA, 2005. ACM Press. ISBN: 1-59593-073-6.

[Ghoniem *et al.*, 2005b]

Mohammad Ghoniem, Jean-Daniel Fekete, and Philippe Castagliola. On the readability of graphs using node-link and matrix-based representations: a controlled experiment and statistical analysis. *Information Visualization*, 4(2):114–135, 2005.

[Ghoniem et Fekete, 2002]

Mohammad Ghoniem and Jean-Daniel Fekete. Visualisation de graphes de coactivité; par matrices d'adjacence. In *IHM '02: Proceedings of the 14th French-speaking conference on Human-computer interaction (Conférence Francophone sur l'Interaction Homme-Machine)*, pages 279–282, New York, NY, USA, 2002. ACM Press. ISBN: 1-58113-615-3.

[Guéhéneuc *et al.*, 2002]

Yann-Gaël Guéhéneuc, Rémi Douence, and Narendra Jussien. No java without caffeine: A tool for dynamic analysis of java programs. In *ASE '02: Proceedings of the 17 th IEEE International Conference on Automated Software Engineering (ASE'02)*, page 117, Washington, DC, USA, 2002. IEEE Computer Society. ISBN: 0-7695-1736-6.

[Guéhéneuc *et al.*, 2004]

Yann-Gaël Guéhéneuc, Houari Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. In Eleni Stroulia and Andrea de Lucia, editors, *proceedings of the* 11^{th} *Working Conference on Reverse Engineering*, pages 172–181. IEEE

Computer Society Press, November 2004.

Available at: www.yann-gael.gueheneuc.net/Work/Publications/.

[Guéhéneuc, 2004]

Yann-Gaël Guéhéneuc. A systematic study of uml class diagram constituents for their abstract and precise recovery. In Doo-Hwan Bae and William C. Chu, editors, *proceedings of the 11th Asia-Pacific Software Engineering Conference*. IEEE Computer Society Press, November-December 2004. To appear.

Available at: www.yann-gael.gueheneuc.net/Work/Publications/.

[Guéhéneuc, 2005]

Yann-Gaël Guéhéneuc. Ptidej: Promoting patterns with patterns. In *proceedings of the* 1st *ECOOP workshop on Building a System using Patterns*. Springer-Verlag, July 2005.

[Hamou-Lhadj et Lethbridge, 2002]

Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. Compression techniques to simplify the analysis of large execution traces. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, page 159, Washington, DC, USA, 2002. IEEE Computer Society. ISBN: 0-7695-1495-2.

[Harel, 1992]

David Harel. Biting the silver bullet: Toward a brighter future for system development. *Computer*, 25(1):8–20, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[Harrold *et al.*, 1998]

Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical

investigation of program spectra. *SIGPLAN Not.*, 33(7):83–90, New York, NY, USA, 1998. ACM Press.

[Jeffrey et Auguston, 2003]

C. Jeffrey and M. Auguston. Some axioms and issues in the ufo dynamic analysis framework. *ICSE Workshop on Dynamic Analysis (WODA 2003)*, Portland, OR, USA, 2003. Portland, OR, USA.

[Johnson et Erdem, 1995]

W. L. Johnson and A. Erdem. Interactive explanation of software systems. In *KBSE '95: Proceedings of The 10th Knowledge-Based Software Engineering Conference*, page 155, Washington, DC, USA, 1995. IEEE Computer Society. ISBN: 0-8186-7204-8.

[Kaiser, 2001]

Wolfram Kaiser. Become a programming picasso with JHotDraw – Use the highly customizable GUI framework to simplify draw application development. In Carolyn Wong, editor, *JavaWorld*. IDG Publications, February 2001.

Available at: `www.javaworld.com/javaworld/jw-02-2001/jw-0216-jhotdraw.html`.

[Kimelman et Ngo, 1991]

D. N. Kimelman and T. A. Ngo. The rp3 program visualization environment. *IBM J. Res. Dev.*, 35(5-6):635–651, Riverton, NJ, USA, 1991. IBM Corp.

[Lanza, 2003]

Michele Lanza. *Object-Oriented Reverse Engineering - Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. Ph.D. thesis, University of Berne, Switzerland, Mai 2003.

[Lieberman et Fry, 1995]

Henry Lieberman and Christopher Fry. Bridging the gulf between code and behavior in programming. In *CHI '95: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 480–486, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co. ISBN: 0-201-84705-1.

[Lukoit *et al.*, 2000]

Kazimiras Lukoit, Norman Wilde, Scott Stowell, and Tim Hennessey. Tracegraph: Immediate visual location of software features. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 33, Washington, DC, USA, 2000. IEEE Computer Society. ISBN: 0-7695-0753-0.

[Microsystems., 2002]

I. Sun Microsystems. Java platform debug architecture, 2002.

Available at: `http:// java.sun.com/ products/ jpda/`.

[Murphy et Notkin, 1997]

Gail C. Murphy and David Notkin. Reengineering with reflexion models: A case study. *Computer*, 30(8):29–36, Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[Muthukumarasamy et Stasko, 1995]

Jeyakumar Muthukumarasamy and John T. Stasko. Visualizing program executions on large data sets using semantic zooming. Technical Report GIT-GVU-95-02, Graphics, Visualization, and Usability Center Georgia Institute of Technology, Atlanta, GA, January 1995.

[Mutzel, 1997]

Petra Mutzel. An alternative method to crossing minimization on hierarchical

graphs. In *GD '96: Proceedings of the Symposium on Graph Drawing*, pages 318–333, London, UK, 1997. Springer-Verlag. ISBN: 3-540-62495-3.

[Nielson *et al.*, 1990]

G.M. Nielson, B.D. Shriver, and J. Rosenblum. Visualization in scientific computing. *IEEE Computer Society Press, Washington*, 1990.

[Norman, 1986]

Donald A. Norman. *User Centered System Design; New Perspectives on Human-Computer Interaction.* Lawrence Erlbaum Associates, Inc., Mahwah, NJ, USA, 1986. ISBN: 0898597811.

[Object Management Group, Inc., 2003]

Object Management Group, Inc. *Unified Modeling Language Specification*, Marsh 2003. Version 1.5.

Available at: `http://www.omg.org`.

[Pacione *et al.*, 2003]

M J Pacione, M Roper, and M Wood. A comparative evaluation of dynamic visualisation tools. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE), Victoria, BC*, pages 80–89, New York, NY, USA, 2003. Los Alamitos, CA: IEEE CS Press.

[Pauw *et al.*, 1993]

Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the behavior of object-oriented systems. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 326–337, New York, NY, USA, 1993. ACM Press. ISBN: 0-89791-587-9.

[Price *et al.*, 1993]

B. A. Price, R. M. Baecker, and I. S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–226, 1993.

[Renieris et Reiss, 1999]

Manos Renieris and Steven P. Reiss. Almost: exploring program traces. In *NPIVM '99: Proceedings of the 1999 workshop on new paradigms in information visualization and manipulation in conjunction with the eighth ACM internation conference on Information and knowledge management*, pages 70–77, New York, NY, USA, 1999. ACM Press. ISBN: 1-58113-254-9.

[Richner et Ducasse, 1999]

Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, page 13, Washington, DC, USA, 1999. IEEE Computer Society. ISBN: 0-7695-0016-1.

[Rumbaugh *et al.*, 1999]

J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*, addison wesley edition, 1999.

[Salah et Mancoridis, 2004]

Maher Salah and Spiros Mancoridis. A hierarchy of dynamic software views: From object-interactions to feature-interactions. In *ICSM*, pages 72–81, 2004.

[Schauer et Keller, 1998]

R. Schauer and R. Keller. Pattern visualization for software comprehension. In *IWPC '98: Proceedings of the 6th International Workshop on Program Compre-*

hension, page 4, Washington, DC, USA, 1998. IEEE Computer Society. ISBN: 0-8186-8560-3.

[Seemann et von Gudenberg, 1998]

Jochen Seemann and Jürgen Wolff von Gudenberg. Pattern-based design recovery of java software. In Bill Scherlis, editor, *proceedings of* 5^{th} *international symposium on Foundations of Software Engineering*, pages 10–16. ACM Press, November 1998.

Available at: `www.informatik.uni-trier.de/~ley/db/indices/a-tree/s/Seemann:Jochen.html`.

[Seemann, 1997]

Jochen Seemann. Extending the sugiyama algorithm for drawing uml class diagrams: Towards automatic layout of object-oriented software diagrams. In *GD '97: Proceedings of the 5th International Symposium on Graph Drawing*, pages 415–424, London, UK, 1997. Springer-Verlag. ISBN: 3-540-63938-1.

[Soloway *et al.*, 1988]

Elliot Soloway, Robin Lampert, Stan Letovsky, David Littman, and Jeannine Pinto. Designing documentation to compensate for delocalized plans. *Commun. ACM*, 31(11):1259–1267, New York, NY, USA, 1988. ACM Press.

[Stasko *et al.*, 1998]

J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price. Software visualization - programming as a multimedia experience. In *The MIT Press*, pages (11, 12, 21, 54, 92, 113), 1998. ISBN: 0-89791-587-9.

[Sugiyama *et al.*, 1981]

K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of

hierarchical system structures. *IEEE Trans. on Systems, Man, and Cybernetic-sSMC*, 11(2):109–125, 1981.

[Upson *et al.*, 1989]

C. Upson, T. Faulhaber, D. Kammins, D. Laidlaw, D. Schlegert, J. Vroom, R. Gurwitz, and A. Van Dam. The application visualization system: a computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42. IEEE Publication, July 1989.

[van Ham, 2003]

Frank van Ham. Using multilevel call matrices in large software projects. In *IEEE INFOVIS*, pages 227–232, Seattle, Washington, USA, 2003. IEEE Computer Society Press.

[Ware, 2000]

Colin Ware. *Information visualization: perception for design.* Morgan Kaufmann Publishers Inc. pp(15, 54, 59, 93, 113), San Francisco, CA, USA, 2000. ISBN: 1-55860-511-8.

[Wilde et Scully, 1995]

Norman Wilde and Michael C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance*, 7(1):49–62, New York, NY, USA, 1995. John Wiley & Sons, Inc.

[Wille, 1982]

R. Wille. Restructuring the lattice theory: an approach based on hierarchies of concepts. *Ordered Sets*, pages 445–470. Reidel, 1982.

[Wong *et al.*, 1999]

W. Eric Wong, Joseph R. Horgan, Swapna S. Gokhale, and Kishor S. Trivedi.

Locating program features using execution slices. In *ASSET '99: Proceedings of the 1999 IEEE Symposium on Application - Specific Systems and Software Engineering and Technology*, page 194, Washington, DC, USA, 1999. IEEE Computer Society. ISBN: 0-7695-0122-2.