Titre: Title:	Extraction des sous-graphes : identification des microarchitectures dans les logiciels évolutifs orientés objets
Auteur: Author:	Ahmed Belderrar
Date:	2011
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Belderrar, A. (2011). Extraction des sous-graphes : identification des microarchitectures dans les logiciels évolutifs orientés objets [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/658/

UNIVERSITÉ DE MONTRÉAL

EXTRACTION DES SOUS-GRAPHES : IDENTIFICATION DES MICROARCHITECTURES DANS LES LOGICIELS ÉVOLUTIFS ORIENTÉS OBJETS.

AHMED BELDERRAR
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2011

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

EXTRACTION DES SOUS-GRAPHES : IDENTIFICATION DES MICROARCHITECTURES DANS LES LOGICIELS ÉVOLUTIFS ORIENTÉS OBJETS.

présenté par : BELDERRAR, Ahmed
en vue de l'obtention du diplôme de : Maîtrise ès Sciences Appliquées
a été dûment accepté par le jury d'examen constitué de :

M. QUINTERO, Alejandro, Doct., président.
M. ANTONIOL, Giuliano, Ph.D., membre et directeur de recherche.
M. DESMARAIS, Michel C., Ph.D., membre.

À mes très chers parents,
ma femme et mes enfants...

REMERCIEMENTS

En premier lieu, je tiens à remercier Monsieur le Professeur Giuliano Antoniol pour avoir accepté et assuré la direction de mes travaux, pour la qualité de son encadrement, et ses remarques pertinentes. Je suis très reconnaissant de son savoir et son expérience partagés, et de son soutien scientifique.

Je remercie aussi Monsieur Yann-Gaël Guéhéneuc, de son aide, de ses commentaires, et de son outil Ptidej et son méta-modèle PADL pour les données utilisées dans ce travail.

Je veux aussi adresser mes remerciements à Monsieur Philippe Galinier, qui nous a aidé à valider notre algorithme.

Je veux également exprimer mes sincères remerciements, et témoigner de ma grande reconnaissance à Monsieur Segla Kpodjedo pour son soutien, et qui par son expérience, m'a donné beaucoup de propositions tout au long de ce projet.

Mes plus profonds remerciements vont à ma femme. Tout au long de la période d'études, elle m'a toujours soutenu, encouragé et aidé. Elle a su me donner toutes les chances pour réussir.

La réalisation de ce travail s'appuie également sur le bon environnement de travail des laboratoires "soccerlab"' et "Ptidej". A ce titre, je tiens à remercier tous ceux qui m'ont aidé de près ou de loin afin d'achever ce travail, ainsi que je remercie toutes les personnes intéressées par notre travail, en espérant qu'elles puissent trouver dans ce rapport des explications utiles pour leurs propres travaux.

RÉSUMÉ

Les développeurs introduisent des nouvelles microarchitectures, et des microarchitectures non documentées lorsqu'ils effectuent des tâches d'évolution sur les applications orientées objets. Nous nous intéressons à chercher la relation entre les microarchitectures et les propriétés telles que la stabilité et les défauts. Nous proposons une nouvelle approche basée sur l'extraction des sous-graphes, un nouvel algorithme, et un outil SGFinder permettant de recenser d'une manière efficace et exhaustive les microarchitectures dans le diagramme des classes des petits et moyens systèmes orientés objets. Une fois que nous énumérons toutes les occurrences des microarchitectures, nous les exploitons pour identifier leurs propriétés souhaitables, comme la stabilité, ou leurs propriétés indésirables, comme les changements et la prédisposition aux défauts. Nous avons effectué une étude empirique pour vérifier la faisabilité de notre approche, en appliquant l'outil SGFinder sur le diagramme des classes de plusieurs versions de deux systèmes orientés objets Rhino et ArgoUml. Nous identifions les microarchitectures les plus et les moins prédisposées aux défauts, et les plus et les moins prédisposées aux changements. Finalement, nous concluons que le nouvel outil SGFinder ouvre plusieurs voies pour d'autres éventuelles recherches.

Mots-clefs : Microarchitectures, changements et défauts des logiciels, maintenance et évolution des logiciels.

ABSTRACT

Developers introduce novel and undocumented micro-architectures when performing evolution tasks on object-oriented applications. We are interested in understanding whether those organizations of classes and relations can bear, much like cataloged design and anti-patterns, potential harm or benefit to an object-oriented application. We present SGFinder, a sub-graph mining approach and tool based on an efficient enumeration technique to identify recurring micro-architectures in object-oriented class diagrams. Once SGFinder has detected instances of micro-architectures, we exploit these instances to identify their desirable properties, such as stability, or unwanted properties, such as change or fault proneness. We perform a feasibility study of our approach by applying SGFinder on the reverse-engineered class diagrams of several releases of two Java applications: ArgoUML and Rhino. We characterize and highlight some of the most interesting micro-architectures, the most fault prone and the most stable, and conclude that SGFinder opens the way to further interesting studies.

Keywords : Micro-architectures, software changes and faults, software maintenance and evolution.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ALGORITHMES

LISTE DES ANNEXES

LISTE DES SIGLES ET ABRÉVIATIONS

SGFinder	Engin d'énumération des sous-graphes (Sub-Graph Finder).
SGViewer	Engin de visualisation des sous-graphes (Sub-Graph Viewer).
OO	Orienté objet.
mAs	Microarchitectures.
PADL	Méta-modèle (Pattern and Abstract-level Description Language).
Ptidej	Suite d'outils de rétro-conception (Pattern Trace Identification, Detection, and Enhancement in Java).
DP	Un patron de conception est un concept permettant de décrire des solutions standards pour résoudre des problèmes récurrents et spécifiques de conception dans des contextes bien précis.
AP	Les anti-patrons sont de "mauvaises" solutions à des problèmes récurrents de conception logicielle.
AST	Arbre syntaxique abstrait (AST est un arbre dont les nœuds internes marquent les opérateurs, et dont les feuilles représentent les opérandes des opérateurs.).
CSP	Problèmes de satisfaction de contraintes.
nauty	Est un ensemble de fonctions permettant de déterminer le groupe d'automorphismes d'un graphe. Il peut aussi fournir la matrice d'adjacence représentant la forme canonique d'un graphe.
Q1, Q3	Les quartiles permettent de séparer une série des données en quatre groupes de même nombre d'éléments. Un quart des valeurs sont inférieures au premier quartile Q1, et un autre quart des valeurs sont supérieures au troisième quartile Q3.
Médiane	Lorsqu'on ordonne les valeurs d'une variable, la médiane est défini par le point milieu de cette liste ordonnée (c'est-à-dire que 50 % des valeurs sont supérieures à la médiane et 50 % lui sont inférieures).
Résumé en 5 chiffres	est la représentation d'une série de données sous le format "Min, Q1, Médiane, Q3, Max".

CHAPITRE 1

INTRODUCTION

Le travail présenté dans ce mémoire est la mise en place d'une nouvelle approche et d'un outil SGFinder pour recenser des microarchitectures à partir de diagramme des classes. Nous modélisons un diagramme des classes par un graphe orienté et étiqueté, et nous définissons une microarchitecture comme un sous-graphe induit par un sous-ensemble de classes. Cette technique de modélisation nous permet de recenser les sous-graphes d'un graphe orienté et étiqueté au lieu de recenser les microarchitectures du diagramme des classes d'un système OO. Pour grouper les sous-graphes isomorphes dans leurs propres catégories, nous utilisons la librairie existante nauty (voir McKay, 1981). Notre outil SGFinder inclut une représentation visuelle des sous-graphes correspondant aux microarchitectures. Cet outil permet un recensement efficace des sous-graphes induits d'ordre prédéfini (nombre de sommets).

Définition de microarchitecture : Le concept de microarchitecture peut varier en fonction du domaine ou du contexte de son utilisation. Pour le diagramme des classes d'un système orienté objet, une microarchitecture est définie comme un ensemble de composants et de ses connecteurs. Autrement dit, cette microarchitecture comprend un ensemble de classes et de ses interactions (les relations entre les classes), où aucune des classes n'est isolée du reste des classes participantes à cette microarchitecture.

Nous utilisons l'outil SGFinder pour recenser des microarchitectures de taille trois, quatre et cinq sur plusieurs versions de deux systèmes OO (Rhino et ArgoUml). Ensuite, nous reportons les détails concernant le nombre total de toutes les microarchitectures, et le nombre total des microarchitectures identiques. Aussi, nous identifions les microarchitectures les plus et les moins prédisposées aux changements et aux défauts.

Dans ce chapitre, nous commençons par décrire le contexte de notre projet de recherche, définir les éléments de la problématique, et les objectifs à atteindre. Ainsi, nous présentons la méthodologie utilisée pour répondre à nos objectifs. Par la suite, nous rappelons brièvement les notions de base en théorie des graphes et des ensembles. Finalement nous terminons par le plan d'organisation de ce mémoire.

1.1 Contexte

La mise en production d'un logiciel est l'aboutissement de plusieurs phases où différentes équipes collaborent entre elles pour réaliser ce logiciel. Ces phases sont découpées selon un cycle bien déterminé dans l'objectif de valider et de vérifier la conformité de ce logiciel avec les besoins et les exigences de l'utilisateur final. Les erreurs commises durant le développement, et/ou l'évolution du logiciel peuvent engendrer des coûts très élevés lors de l'opération de maintenance après sa livraison. En effet, l'activité de maintenance est une phase importante et cruciale dans le cycle de vie d'un logiciel.

Aujourd'hui, les ressources relatives à la maintenance d'un logiciel et de la gestion de son évolution représentent plus de 50% des ressources du projet. En effet, plusieurs études ont été effectuées pour définir ces ressources comme le budget, l'effort, et le temps. Ces ressources sont récapitulées dans le tableau 1.1.

En industrie, les chefs de projets, les architectes logiciels, et les concepteurs consacrent une partie importante de leur temps à la conception des systèmes informatiques solides et stables. Donc, il est essentiel pour ces gestionnaires de bien gérer la conception et la maintenabilité pour pouvoir maîtriser le coût de développement des logiciels, et accélérer leurs réalisations pour satisfaire les besoins de l'utilisateur final.

Les entreprises développant des systèmes informatiques cherchent d'un côté (i) des moyens et des nouvelles techniques pour assurer la qualité de leurs logiciels, et d'un autre côté (ii) des solutions pour réduire le coût des changements apportés au code après la livraison de leurs logiciels aux clients. Pour atteindre les deux objectifs visés par ces entreprises, il faut prendre en compte certains éléments dont :

i. Les développeurs doivent comprendre le code, y compris la structure interne des classes et les relations entre elles, afin d'effectuer correctement les changements.
ii. Pendant la phase de conception détaillée du logiciel, les concepteurs des classes ne doivent pas créer des structures prédisposées aux défauts (bogues), ou qui peuvent créer beaucoup de changements.

Éléments de la problématique

Après la mise en œuvre des logiciels, des activités de maintenance seront ultérieurement nécessaires pour corriger les bogues rencontrés par les utilisateurs finaux, ajouter des nouvelles

Tableau 1.1 Coûts, efforts, et temps de maintenance des logiciels.

Ressource	Description	Référence
Coût	Selon une étude effectuée en 1986 aux États-Unis auprès de 55 entreprises indique que 53% du coût total de la réalisation d'un logiciel est réservé à sa maintenance. Ce coût est réparti de la manière suivante : – 34% de ce coût est attribué aux modifications des spécifications initiales (évolution du logiciel). – 17% est affecté aux corrections des bogues. – 10% pour adapter le logiciel aux nouveaux utilisateurs, ou aux nouveaux environnements de travail. – Le reste du coût (39%) est reparti sur le contrôle de la qualité, l'amélioration de la performance du logiciel, et l'assistance aux utilisateurs.	(voir Audibert, 2009)
	Jusqu'à **90%** du coût total du projet est consacré à la maintenance d'un système existant, et à son évolution.	(voir Erlikh, 2000; Moad, 1990)
	Environ **75%** du budget est consommé pendant la phase de maintenance selon une étude effectuée auprès de 1000 compagnies.	(voir Eastwood, 1993)
	Le coût de maintenance peut consommer entre **60%** et **70%** du budget de gestion et de fonctionnement du projet.	(voir Huff, 1990; Port, 1988)
Effort	De **65%** à **75%** de l'effort logiciel est consacré à la maintenance des systèmes.	(voir McKay, 1984)
Temps	Selon une étude effectuée auprès de 487 organisations, plus de **50%** du temps total du projet est consacré à la maintenance des systèmes.	(voir Lientz et Swanson, 1981)

fonctionnalités, améliorer l'efficacité, et adapter le code selon un nouvel environnement de travail. Ces travaux sont donc inévitables pour augmenter la durée de vie des logiciels.

La taille du logiciel croît au fur et à mesure durant la phase de développement des composants et leur intégration pour produire la version finale du logiciel. Donc, les logiciels deviennent de plus en plus complexes, et par conséquent, leur maintenance, et leur mise au point deviennent des tâches extrêmement difficiles pour les développeurs.

Afin d'effectuer efficacement les activités de maintenance, et avant de procéder à n'importe quelle modification, les responsables de la maintenance doivent comprendre l'objectif de leurs systèmes OO, ainsi que les concepts représentés dans ces systèmes. La compréhension com-

prend évidemment les différents choix qui ont été pris pendant les phases de conception et d'implémentation. Dans les systèmes orientés objets, les choix de conception comprennent la structure interne des classes et les relations entre elles. Par exemple, les patrons de conception (voir Gamma *et al.*, 1994) sont des structures de classes permettant de décrire des solutions standards à des problèmes de conception spécifiques et récurrents dans des contextes bien précis. Ils sont conçus dans l'objectif de faire une conception plus souple, réutilisable et robuste, ainsi que améliorer la qualité des logiciels. Par contre, les anti-patterns (voir Brown *et al.*, 1998) sont de "mauvaises" solutions à des problèmes récurrents de conception logicielle.

Cependant, il y a des types de microarchitectures comme les patrons de conception et les anti-patterns qui ont été déjà documentés. En effet, certaines microarchitectures ne sont pas documentées parce qu'elles sont méconnues, ou bien leur domaine ou leur application sont spécifiques. Nous supposons que ces microarchitectures peuvent avoir des propriétés utiles, comme la stabilité, et les défauts (les bogues).

Dans les systèmes orientés objets, la plupart des approches existantes se focalisent sur les patrons de conception, les anti-patterns et plus précisément sur une librairie d'abstraction de microarchitectures existantes (ex. patrons de conception (voir Guéhéneuc et Antoniol, 2008)).

1.2 Objectifs de la recherche

L'objectif principal du travail présenté dans ce mémoire est d'implémenter un algorithme et un outil efficace permettant d'une part, de recenser toutes les microarchitectures d'une taille donnée, et d'autre part de regrouper les microarchitectures identiques. De plus, d'autres objectifs sont pris en considération :

i. Chercher la relation entre les microarchitectures et les propriétés telles que la stabilité et les défauts.

ii. Tracer l'évolution des microarchitectures entre les différentes versions d'un système OO.

iii. Caractériser les microarchitectures (définir les rôles joués par les classes, et identifier les microarchitectures dans les trois côtés du tunnel [1]) (voir la section 3.3).

1.3 Esquisse de la méthodologie

Pour atteindre les objectifs, tout d'abord le diagramme des classes d'un système orienté objet est transformé en un graphe orienté et étiqueté où :

1. un tunnel est un ensemble de classes communes entre plusieurs versions

- Les classes sont représentées par des sommets (nœuds).
- Les relations entre les classes sont représentées par des arcs.
- Les étiquettes associées aux arcs représentent les relations simples (agrégation, association, et héritage) et composées.

Avec cette technique, nous modélisons le problème d'énumération des microarchitectures comme un problème d'énumération des sous-graphes.

Par la suite, nous implémentons un algorithme efficace pour énumérer toutes les microarchitectures (sous-graphes) d'une taille trois, quatre, et cinq.

Nous identifions ensuite les classes boguées et les classes modifiées (voir la section 4.2.1), et nous les exploitons pour déterminer les microarchitectures les plus et les moins prédisposées aux défauts et aux changements.

1.4 Notations et notions de base en théorie des graphes et des ensembles

Afin d'avoir une terminologie cohérente, nous allons introduire les principaux concepts de graphes et d'ensembles que nous utiliserons dans l'algorithme d'énumération des sous-graphes.

1.4.1 Les graphes

Un graphe orienté et étiqueté G est un quadruplet $G(V, A, L, l)$ où :

- V est un ensemble $\{ v_1, v_2, .., v_n \}$ fini non vide dont les éléments sont appelés sommets ou nœuds.
- A est un ensemble $\{ a_1, a_2, .., a_m \}$ dont les éléments sont appelés arcs. Un arc a de l'ensemble A est défini par une paire ordonnée de sommets. Lorsque $a = (u, v)$, on dira que l'arc a va de u à v. On dit aussi que u est l'extrémité initiale et v est l'extrémité finale de a.
- L est un ensemble d'étiquettes.
- $l : A \rightarrow L$ est une fonction d'étiquetage vérifiant la condition suivante : $\forall a \in A \Rightarrow l(a) \in L$.

Deux sommets d'un graphe sont dits adjacents ou voisins s'il existe un arc qui les relie. Le nombre de sommets d'un graphe G est appelé ordre de ce graphe, et on le note par $|V|$. La taille d'un graphe, notée $|A|$ est le nombre de ses arcs.

Une chaîne $\mu = (v_1, a_1, v_2, .., v_{p-1}, a_p, v_{p+1})$ est une suite de sommets dans laquelle deux éléments successifs quelconques sont reliés par un arc. La longueur d'une chaîne est le nombre de sommets moins un et on la note par $length(u) = p$. La distance $dist(u, v)$ entre deux sommets u et v est la longueur de la plus petite chaîne reliant ces deux sommets. Un graphe connexe G est un graphe dans lequel il existe au moins une chaîne $\mu = (u, .., v)$ entre toute paire de sommets u et v de $V(G)$. Il est utile de noter qu'un graphe orienté et étiqueté peut être cyclique, et donc ce graphe peut avoir plusieurs chaînes entre deux sommets données.

Un sous graphe partiel d'un graphe G est un graphe $G_S = (V_S, A_S, L, l^S)$ composé de certains sommets de G et de certains arcs reliant ces sommets dans G. En effet, le graphe $G_S = (V_S, A_S, L, l^S)$ est un sous graphe partiel de $G(V, A, L, l)$ ssi $V_S \subset V$, $A_S \subset A \cap (V_S \times V_S)$, et la fonction d'étiquetage $l^S : A_S \to L$ vérifie la condition suivante : $\forall (a, b) \in A_S$, $l^S(a, b) = l(a, b)$.

Un graphe $G'(V', A', L, l')$ est un sous-graphe induit d'un graphe $G(V, A, L, l)$ si V' est un sous-ensemble de V et si pour tout couple de sommets (u, v) de V', le sommet u est connecté au sommet v dans G' si et seulement si le sommet u est connecté au sommet v dans G, et l'étiquette de l'arc (u, v) est la même dans G et G'. Autrement dit, le sous-graphe G' est induit si $V' \subset V$ et $A' = \{(u, v) \in A, u \in V' \wedge v \in V'\}$. De même, le sous-graphe $G'(V', A', L, l')$ est dit induit par le sous-ensemble des sommets V' de V. Un k-sous-graphe est un sous-graphe induit d'ordre k.

Deux sous-graphes $G_S = (V_S, A_S, L, l^S)$ et $G_H = (V_H, A_H, L, l^H)$ sont isomorphes s'ils sont identiques. Formellement, les deux sous-graphes G_S et G_H sont isomorphes si seulement si $|V_S| = |V_H|$, et s'il existe une fonction de bijection $\phi : V_S \to V_H$ telle que, $\forall (u, v) \in A_S$ $\Rightarrow (\phi(u), \phi(v)) \in A_H$. La décision de l'existence d'un isomorphisme entre deux sous-graphes est un problème NP-complet (voir Garey et Johnson, 1990).

Une matrice d'adjacence M est une structure de données permettant de représenter un graphe G. La matrice M est une matrice carrée ayant pour taille le nombre de sommets du graphe G. Le couple (i, j) désigne l'intersection de la ligne i et de la colonne j. Dans une matrice d'adjacence, les lignes et les colonnes représentent les sommets du graphe. La valeur $m_{ij} \neq 0$ à la position (i, j) signifie que le sommet i est adjacent au sommet j, et elle représente l'étiquette sur l'arc reliant le sommet i et le sommet j. Donc, pour un graphe orienté et étiqueté, $M = (m_{ij})$ où $m_{ij} = \begin{cases} l(v_i, v_j), & \text{si } (v_i, v_j) \in A \\ 0, & \text{sinon} \end{cases}$

1.4.2 Les ensembles

Soit S un ensemble contenant n éléments distincts et soit k un entier positif. Un k-sous-ensemble est une combinaison de k éléments de l'ensemble S. Les k éléments sont pris sans répétition et ne sont pas ordonnés. Le nombre total des combinaisons possibles est donné par le coefficient binomial suivant :

$$C_n^k = \binom{n}{k} = \begin{cases} \frac{n!}{k!(n-k)!}, & \text{si } 0 \leq k \leq n \\ 0, & \text{sinon} \end{cases}$$

Exemple : les combinaisons de 2 éléments pris dans $\{1, 2, 3, 4\}$ sont $\{1, 2\}$, $\{1, 3\}$, $\{1, 4\}$, $\{2, 3\}$, $\{2, 4\}$, $\{3, 4\}$. Donc, il y a 6 combinaisons possibles ($C_4^2 = \frac{4!}{2!(4-2)!} = 6$).

1.5 Organisation du mémoire

Ce mémoire est composé de six chapitres. Ce premier chapitre d'introduction met en évidence le contexte de notre étude, les éléments de la problématique, la méthodologie, et le plan de ce mémoire. Dans le deuxième chapitre, nous présentons une revue de littérature sur les travaux effectués sur les algorithmes de recensement des sous-graphes (microarchitectures), et les types des microarchitectures traitées. Dans le troisième chapitre, nous décrivons la conception et l'implémentation de la technique proposée pour réaliser ce travail. Dans le quatrième chapitre, nous présentons la validation empirique. Le cinquième chapitre est consacré à l'analyse et à l'interprétation des résultats obtenus par la nouvelle technique. Finalement le dernier chapitre conclut ce mémoire, révèle les limitations de notre travail, et présente les travaux futurs.

CHAPITRE 2

REVUE DE LITTÉRATURE

Ce chapitre présente une revue de littérature en deux parties sur les travaux réalisés précédemment. La première partie de cette revue de littérature est consacrée aux algorithmes de recensement des motifs de réseau [1] et des sous-graphes. La deuxième partie est consacrée aux études effectuées sur les patrons de conception, les anti-patrons, et les microarchitectures des systèmes OO.

2.1 Algorithmes de recensement des motifs de réseau et des sous-graphes

Des travaux de recensement des motifs ont été réalisés dans les différents domaines d'applications comme les réseaux sociaux et les réseaux biologiques. Plus précisément, les réseaux d'interaction protéine-protéine (PPI), les réseaux génétiques, et les réseaux métaboliques, sont les réseaux les plus largement étudiés dans le domaine de la biologie (voir Milo *et al.*, 2002; Batagelj et Mrvar, 2003; Schreiber et Schwöbbermeyer, 2005; Sebastian, 2006; Razaghi et Kashani, 2009). En effet, des techniques mathématiques et informatiques sont appliquées pour analyser et modéliser les données à cause de la complexité de ces réseaux et de la quantité des données qu'ils contiennent. Pour permettre l'établissement d'un modèle mathématique convenable pour l'analyse de ces réseaux complexes, il est nécessaire d'utiliser les notions de la théorie des graphes. Les éléments d'un réseau en traitement sont représentés par des sommets (nœuds), et l'interaction entre eux sont représentés par des arcs ou des arêtes. Les algorithmes peuvent ensuite être utilisés pour analyser, simuler et visualiser le réseau traité. En effet, des méthodes puissantes de calcul, permettant l'extraction des informations pertinentes à partir d'une grande quantité de données, doivent être développées. Ces méthodes sont basses sur des algorithmes de recensement des motifs de réseau, qui sont très coûteuses en temps d'exécution et en consommation de mémoire. Ces algorithmes sont soumis à des restrictions sur la taille des motifs de réseau, et la taille et les types des réseaux traités. Généralement, les algorithmes proposés permettent soit d'énumérer exhaustivement tous les motifs de réseau (voir Milo *et al.*, 2002; Schreiber et Schwöbbermeyer, 2005; Sebastian, 2006; Razaghi et Kashani, 2009), ou énumérer seulement les sous-graphes les plus fréquents (voir Kuramochi et Karypis, 2004). Aussi, une autre caractéristique pertinente pour l'évaluation

1. Dans les réseaux biologiques, Les motifs de réseau sont définis par des sous-graphes, qui se trouvent dans le réseau original beaucoup plus souvent que dans les réseaux randomisés (voir Milo *et al.*, 2002).

des algorithmes existants est la représentation visuelle des résultats.

Les algorithmes de recensement des motifs de réseau sont basés sur le même principe. Chaque algorithme implémente les trois tâches suivantes :

1. **Enumération** : Chercher des motifs de réseau (sous-graphes) d'une taille donnée qui se trouvent dans un réseau original (graphe en entrée).
2. **Classement** : Regrouper les sous-graphes identiques dans des catégories. La majorité des algorithmes (voir Milo *et al.*, 2002; Sebastian, 2006; Razaghi et Kashani, 2009) utilise l'outil nauty (voir McKay, 1981) pour chercher la forme canonique de ces sous-graphes. La forme canonique est un code unique permettant de distinguer les sous-graphes isomorphes (identiques).
3. **Randomisation** : Générer des graphes aléatoires pour s'assurer que les motifs retrouvés caractérisent bien les réseaux considérés. Un motif est défini par un petit sous-graphe connexe qui se trouve dans le réseau considéré (original) avec une fréquence plus élevée que dans les réseaux générées aléatoirement (voir Milo *et al.*, 2002). Le recensement et le classement sont effectués de nouveau sur les graphes générés aléatoirement.

Dans les sections suivantes, nous introduisons quelques algorithmes de recensement des sous-graphes ou motifs de réseaux. Pour chaque algorithme, nous mettons en exergue les raisons pour lesquelles il ne répond pas à nos besoins.

2.1.1 Algorithme *NeMoFinder*

NeMoFinder est un algorithme proposé par J. Chen et al (voir Chen *et al.*, 2006) pour chercher les motifs de réseau qui sont récurrents et uniques dans les réseaux d'interaction protéine-protéine (PPI). Ces réseaux PPI sont représentés par des graphes non orientés et non étiquetés. En effet, l'algorithme *NeMoFinder* est le premier algorithme utilisé pour extraire les motifs de réseau d'une taille allant jusqu'à 12 dans les levures Saccharomyces cerevisiae industrielles.

L'algorithme *NeMoFinder* utilise une technique de recherche basée sur les arbres récurrents afin de partitionner le réseau PPI en un ensemble de graphes. Premièrement, il cherche les sous-graphes fréquents de taille k dans le réseau PPI. Il commence par les arbres de taille 2, puis, il les élargit en ajoutant des sommets voisins afin d'atteindre les tailles 3, 4, et ainsi de suite jusqu'à obtenir la taille k. Par la suite, il utilise un algorithme basé sur les chaines de Markov (voir Maslov et Sneppen, 2002) pour générer des graphes aléatoires en échangeant les

arêtes aléatoirement entre les sommets du graphe original. Chaque sommet dans le graphe aléatoire contient le même nombre de sommets voisins que le sommet correspondant dans le graphe original. La procédure de recensement des sous-graphes dans les graphes aléatoires est la même que dans le graphe original. Ensuite, l'algorithme *NeMoFinder* vérifie si les sous-graphes du graphe original sont des motifs de réseau.

L'algorithme *NeMoFinder* ne peut traiter ni les graphes orientés et étiquetés, ni les boucles sur les sommets des graphes non orientés.

2.1.2 Algorithme MFinder

L'algorithme MFinder (voir Milo *et al.*, 2002) est le premier algorithme utilisé pour extraire des motifs de réseau. Il fournit deux méthodes d'exploration qui sont :

1. Énumération exhaustive des motifs de réseau.
2. Énumération partielle de certains motifs de réseau. Le problème du recensement exhaustif est que le nombre des motifs croît exponentiellement avec la taille du réseau considéré, et la taille des motifs eux-mêmes. Un algorithme d'approximation probabiliste est utilisé pour énumérer certains motifs de réseau du réseau considéré.

L'algorithme MFinder commence par choisir une arête e. Donc, le premier motif de réseau est constitué des deux sommets de l'arête e. À chaque itération, il ajoute un nouveau sommet situé à l'extrémité d'une arête reliée au motif de réseau généré partiellement. Une fois que la taille du motif de réseau généré atteint la taille désirée, il génère un code unique basé sur l'isomorphisme de ce motif de réseau. Le processus de génération des réseaux aléatoires et de vérification des motifs de réseau est le même que l'algorithme *NeMoFinder* (voir la section 2.1.1).

L'algorithme MFinder a besoin de beaucoup de mémoire pour explorer tous les motifs de réseau, ce qui entrave la recherche des motifs de réseau dans des réseaux de taille moyenne.

2.1.3 Algorithme Pajek

Pajek (voir Batagelj et Mrvar, 2003) est un outil d'analyse et de visualisation des grands réseaux. Il permet de chercher certains motifs de réseau fréquents comme les tétrades ayant certaines particularités, et les triades. Les triades et les tétrades sont des sous-graphes de taille trois et quatre respectivement. Les triades peuvent être connectés ou déconnectés, et leurs analyses proviennent de l'analyse des réseaux sociaux.

En effet, le recensement des motifs de réseau par l'outil Pajek est limité aux motifs de réseau de taille trois (triades), et au certains motifs de réseau de taille quatre (tétrades).

2.1.4 Algorithme MAVisto

MAVisto (voir Schreiber et Schwöbbermeyer, 2005) est un outil d'exploration et de visualisation des motifs de réseau trouvés dans les réseaux biologiques. Il fournit un algorithme de recherche des motifs, et différentes vues pour analyser et visualiser les motifs de réseau par une interface graphique. MAVisto est compatible avec le format Pajek-.net- (voir Batagelj et Mrvar, 2003), et le format GML (voir Himsolt, 1997). Il offre un éditeur graphique pour manipuler et créer les réseaux. L'algorithme MAVisto cherche des motifs d'une taille particulière, qui est donnée soit par le nombre de sommets, ou par le nombre d'arêtes.

L'algorithme MAVisto est particulièrement lent pour recenser les motifs de réseau de taille trois. En effet, pour un réseau de 672 sommets et 1277 arcs, l'algorithme s'exécute en 13532.0 secondes (environ 4 heures). Pour chercher des motifs de taille quatre d'un réseau social de 67 sommet et 183 arcs, l'algorithme s'exécute en 1492 secondes (environ 25 minutes).

2.1.5 Algorithme FanMod

FanMod (voir Sebastian, 2006) est un outil de recensement des motifs de réseau de taille comprise entre trois et huit sommets. L'outil FanMod énumère les motifs de réseau à l'aide de l'algorithme Rand-ESU, ce qui rend la recherche des motifs plus rapide que les autres outils basés sur d'autres algorithmes (voir Milo *et al.*, 2002). Il comprend une interface graphique pour faciliter la configuration des paramètres de l'algorithme comme la taille des motifs de réseau. Les résultats peuvent être exportés en format HTML.

Contrairement à l'algorithme MFinder (voir Milo *et al.*, 2002) qui commence par deux sommets d'une arête e, l'outil FanMod commence par un sommet u, puis il explore ses sommets successeurs ou prédécesseurs non encore visités. Ensuite d'une manière itérative, l'algorithme FanMod cherche les sommets successeurs ou prédécesseurs de chaque sommet visité précédemment. La recherche en profondeur des successeurs ou prédécesseurs s'arrête lorsque la distance entre le premier sommet et le dernier sommet successeurs ou prédécesseurs est égal à la taille désirée. Le processus de génération des réseaux aléatoires et de vérification des motifs de réseau est le même que l'algorithme *NeMoFinder* (voir la section 2.1.1).

L'outil FanMod est l'outil le plus rapide comparativement aux autres outils (voir Milo *et al.*, 2002; Batagelj et Mrvar, 2003; Schreiber et Schwöbbermeyer, 2005; Chen *et al.*, 2006).

L'inconvénient majeur de l'outil FanMod est la limitation du nombre d'étiquettes attribuées au réseau considéré (voir Rasche et Wernicke, 2006). Le tableau 2.1 montre le nombre d'étiquettes traité pour chaque taille de motif de réseau.

Tableau 2.1 Limitation de l'algorithme FanMod sur le nombre d'étiquettes attribuées aux arêtes des motifs de réseau.

Taille des motifs de réseau	Nombre d'étiquettes
3	7
4	7
5	3
6	3
7	1
8	1

2.1.6 Algorithme Kavosh

Kavosh (voir Razaghi et Kashani, 2009) est un algorithme de recensement exhaustif des motifs de réseau d'une taille allant jusqu'à 12. Il permet d'explorer des graphes orientés et non orientés pour chercher des sous-graphes de taille $k \leq 12$.

Les sommets enfants d'un sommet quelconque sont définies par les sommets successeurs ou prédécesseurs de ce sommet. Un niveau est défini par les sommets enfants d'un sommet quelconque. L'algorithme Kavosh démarre du premier niveau qui contient le sommet u, et il descend niveau par niveau pour choisir un sommet enfant qui n'a pas été visité auparavant. Donc, la démarche de cet algorithme n'a pas à prendre en considération la notion de cyclicité du graphe. Cet algorithme est basé sur la méthode de "revolving door ordering" (voir Kreher et Stinson, 1998) pour générer toutes les combinaisons possibles des sommets. Comme exemple, pour chercher des sous-graphes de taille $k = 4$, les combinaisons possibles (incluant le sommet u du premier niveau) sont :

- 3 sommets du deuxième niveau.
- 2 sommets du deuxième niveau, et 1 sommet du troisième niveau.
- 1 sommet du deuxième niveau, et 2 sommets du troisième niveau.
- 1 sommet du deuxième niveau, 1 sommet du troisième niveau, et 1 sommet du quatrième niveau.

L'outil Kavosh ne traite pas les graphes étiquetés (orientés ou non orientés), et il consomme beaucoup de mémoire pour stocker les sommets visités et non encore visités de l'arbre contenant le sommet racine u.

2.2 Patrons de conception, anti-patrons, et microarchitectures

Plusieurs approches ont été proposées pour identifier les microarchitectures similaires aux patrons de conception, et aux anti-patrons. Généralement, ces approches sont basées sur une bibliothèque des structures connues préalablement comme les patrons de conception (voir Krämer et Prechelt, 1996; Seemann et von Gudenberg, 1998; Pettersson et Löwe, 2006; Tsantalis *et al.*, 2006), les anti-patrons (voir Brown *et al.*, 1998), les plans (voir Rich et Waters, 1990), et quelques structures de microarchitectures (voir Guéhéneuc et Antoniol, 2008; Keller *et al.*, 1999). En effet, ces approches utilisent des techniques architecturales basées soit sur l'appariement des sous-graphes, soit sur les propriétés associées aux structures des motifs traités. Les algorithmes correspondants à ces techniques, ont été développés pour chercher les structures des motifs prédéfinis dans le catalogue.

En particulier, les approches proposées pour identifier les patrons de conception (voir Pettersson et Löwe, 2006), utilisent différentes techniques comme la méta-programmation, les graphes, la programmation logique, les algorithmes de reconnaissance de *clichés* (voir Krämer et Prechelt, 1996), les réseaux de raisonnement flou (voir Niere *et al.*, 2002; Jahnke et Zündorf, 1997), et les requêtes interrogeant les bases de données contenant les modèles génériques des systèmes OO (voir Kullbach et Winter, 1999).

Les approches proposées pour spécifier et détecter les anti-patrons sont basées sur les techniques manuelles d'inspection du code source (voir Travassos, 1999), les techniques de visualisation (voir Dhambri. *et al.*, 2008) pour afficher et présenter les résultats, les techniques de détections automatiques (voir Lanza. et Marinescu., 2006), et les techniques heuristiques et de mesures (voir Marinescu, 2004).

Dans les sections suivantes, nous introduisons brièvement quelques travaux majeurs pour détecter les patrons de conception, les anti-patrons, les plans, et d'autres types de microarchitectures.

2.2.1 Détection des plans [2]

Rich and Waters (voir Rich et Waters, 1990) ont proposé l'utilisation de la programmation par contraintes pour identifier les plans[2] du code source des programmes Cobol. Les systèmes

2. Un plan est un module exécutable contenant le chemin d'accès logique produit par l'optimiseur DB2. Il peut être composé d'un ou plusieurs DBRMs et packages. Le plan est stocké dans le répertoire DB2, et consulté lorsque son programme est exécuté. Les informations sur le plan sont stockées dans le catalogue DB2.

Cobol sont modélisés par des arbres syntaxiques abstraits (AST). Un plan est modélisé par les sommets de l'arbre AST, et les relations entre elles sont représentées par les contraintes comme les contrôles et les flux de données. Le plan d'un code source est converti en un problème de satisfaction de contraintes (CSP) dans lequel les sommets du plan représentent les variables, les relations entre les sommets représentent les contraintes entre les variables, et le code source de l'arbre syntaxique abstrait représente le domaine des variables.

2.2.2 Détection des patrons de conception

Kramer et Prechelt (voir Krämer et Prechelt, 1996) ont proposé une approche et développé un outil appelé ***Pat*** pour chercher les instances des patrons de conception structurels *Adapter*, *Bridge*, *Composite*, *Decoration*, et *Proxy*. Ces patrons ont été modélisés par l'outil de conception OMT (voir Rumbaugh *et al.*, 1990) et convertis dans des formats de base de connaissances de règles en Prolog. Ensuite le code source est parsé par l'outil *Paradigm Plus* pour décrire sa base de faits en Prolog. Par la suite, des requêtes sont effectuées pour déterminer la correspondance entre la base de faits du code source et les règles définissant les patrons de conception. L'outil *Pat* ne traite que des systèmes ayant de 150 à 300 classes, et il peut détecter plus que 53% des instances des patrons de conception.

Seemann et al (voir Seemann et von Gudenberg, 1998) ont proposé une approche pour détecter et distinguer les relations d'agrégation et d'association entre les classes, et une technique basée sur les graphes pour décrire et identifier les patrons de conception. La structure statique d'un système OO est représentée par un graphe orienté et étiqueté. Ce graphe est composé de trois types de sommets et de six types d'arcs. Les sommets du graphe représentent les classes, les interfaces et les méthodes. Les arcs entre les sommets représentent les différentes relations entre les classes, les interfaces, et les méthodes ($CLASS \times CLASS$, $INTERFACE \times INTERFACE$, $CLASS \times INTERFACE$, $CLASS \times METHOD$, $METHOD \times METHOD$, $METHOD \times CLASS$). Les etiquettes représentent les types : appel "*Calls*", possession "*Owns*", attributs, etc. Une technique de transformation de graphe est proposée pour détecter les sous-graphes représentant les patrons de conception.

Peterson et al (voir Pettersson et Löwe, 2006) ont proposé une technique basée sur les graphes planaires[3]. La technique de transformation d'un graphe représentant un système OO à un graphe planaire permet d'améliorer les performances de recherche des patrons de conception. Pour les graphes non planaires, une technique de filtrage est utilisée afin de

3. Un graphe est planaire s'il accepte une représentation planaire c'est-à-dire une représentation dans laquelle deux arcs (arêtes) distinctes ne se croisent pas.

supprimer la majorité des arcs affectant la planarité, et de réduire considérablement la taille du graphe. Cela permet évidement aussi de réduire le temps de recherche des patrons de conception. L'approche proposée peut détecter jusqu'à 97% des instances des patrons de conception.

Tsantalis et al. (voir Tsantalis *et al.*, 2006) ont proposé une approche basée sur la similarité entre les sommets du graphe pour détecter les patrons de conception. L'approche proposée est capable de reconnaître les patrons de conception même si leur représentation standard est modifiée. L'approche proposée prend en considération la relation d'héritage qui existe dans la plupart des patrons de conception pour réduire la taille du système traité, en le partitionnant en plusieurs sous-systèmes sans perdre aucune information structurelle. Les diagrammes des classes des sous-systèmes et des patrons de conceptions sont modélisés par des matrices carrées. Les lignes et les colonnes de ces matrices représentent les classes. Les éléments de ces matrices indiquent la présence ou l'absence des relations entre les classes. L'approche utilise un algorithme de similarité entre les matrices des sous-systèmes et les matrices des patrons de conception pour chercher les patrons de conception dans chacun des sous-systèmes séparément. L'algorithme de similarité calcule la matrice de scores (similarité) [4] pour vérifier la correspondance entre un patron de conception et les sous-systèmes. Le taux de succès de cette approche de détection des patrons de conception est de 100%.

2.2.3 Détection des anti-patrons

Brown (voir Brown *et al.*, 1998) décrit 40 anti-patrons, y compris le "*Blob*" et le code "*spaghetti*". Il a défini les anti-patrons comme des mauvaises pratiques pour résoudre les problèmes de conception. Ces mauvaises pratiques sont liées principalement aux compétences, et au manque d'expériences des développeurs, ainsi qu'une mauvaise application des patrons de conception. Les auteurs montrent comment détecter les anti-patrons, et ils présentent aussi des solutions de "refactorisation" pour chaque anti-patron présenté.

2.2.4 Détection de certaines microarchitectures

Un nouvel environnement, appelé SPOOL (voir Keller *et al.*, 1999), a été introduit pour visualiser graphiquement la représentation abstraite du code source. SPOOL présente un environnement pour la rétro-ingénierie des composants de conception basé sur la description structurelle des patrons de conception. Un patron de conception est modélisé par une structure abstraite afin de faciliter sa recherche dans le modèle abstrait du code source. La

4. Une matrice de similarité est une matrice de scores qui expriment la similarité entre deux données. Deux matrices A et B sont similaires s'il existe une matrice inversible P tel que $A = PBP^{-1}$

technique proposée permet de détecter manuellement, semi automatiquement, et automatiquement, les composants abstraits de conception à l'aide des requêtes d'interrogation sur le modèle abstrait du code source.

DeMIMA (voir Guéhéneuc et Antoniol, 2008) décrit une approche semi-automatique, et une recherche structurelle basée sur la programmation par contraintes avec explication pour identifier les microarchitectures similaires à des motifs de conception. De plus, cette technique assure la traçabilité de ces microarchitectures entre l'implémentation et la conception. En effet, cette technique comprend trois couches dans laquelle les deux premières sont consacrées à la récupération du modèle abstrait du code source, et la dernière est consacrée à la détection des patrons de conception dans le modèle abstrait. Le taux de succès de l'approche DeMIMA est de 100%.

2.3 Conclusion

Dans ce chapitre, nous avons discuté quelques approches de recherche, dans les systèmes OO, de certains types de microarchitectures prédéfinies comme les patrons de conception, les anti-patrons, et les plans. Cependant, notre technique de recherche des microarchitectures est similaire à des travaux antérieurs de Tonella et Antoniol (voir Tonella et Antoniol, 2001), dans lequel une analyse conceptuelle a été utilisée pour déduire les patrons de conception d'un domaine spécifique avec l'inspection manuelle du code source. Notre approche élimine complètement le problème d'inspection manuelle du code source en s'appuyant sur la notion des sous-graphes fréquents. De plus notre approche améliore l'évolutivité via une technique efficace de recensement des microarchitectures.

Dans le chapitre suivant, nous présentons un nouvel algorithme et un outil SGFinder qui ne se limitent ni à une bibliothèque contenant les microarchitectures connus préalablement, ni à un ensemble de règles pour détecter les instances des microarchitectures.

CHAPITRE 3

ALGORITHME DE RECENSEMENT ET DE CLASSIFICATION DES SOUS-GRAPHES

3.1 Introduction

Nous avons modélisé le diagramme des classes d'un système OO par un graphe orienté et étiqueté dont l'ensemble de ses sommets représentent l'ensemble des classes, et dont les arcs représentent les relations entre les classes. Nous considérons qu'une microarchitecture est équivalente à un sous-graphe. Donc, pour chercher les microarchitectures d'une taille donnée k, il faut chercher les sous-graphes d'ordre k.

Dans ce chapitre, nous décrivons en détails un nouvel algorithme efficace permettant, d'une part, de recenser tous les sous-graphes existants dans un graphe orienté et étiqueté, et d'une autre part, de regrouper les sous-graphes identiques. L'outil mettant en évidence ce nouvel algorithme est nommé SGFinder.

3.2 Algorithme

Dans cette section, nous présentons un nouvel algorithme permettant de recenser les sous-graphes d'ordre k (noté k-sous-graphes) d'un graphe orienté et étiqueté $G(V, A, L, l)$. Les sommets du graphe G sont numérotés par des nombres entiers positifs, uniques, et consécutifs.

Algorithme 3.1 Méthode principale de l'algorithme

Entrées : Un graphe $G(V_G, A, L, l)$ et un nombre positif k.

1: **for all** vertex $u \in V$ **do**
2: N_{ku} = GenerateNeighborsSet(u, k) ;
3: $V_{G_0} = \{\}$
4: GenerateKSubSetsAndValidate(N_{ku}, u, V_{G_0}, 0) ;
5: V.Remove(u) ;
6: **end for**

L'idée principale de l'algorithme 3.1 est de :

i. Chercher la liste des sommets voisins $N^k(u)$ d'un sommet donnée u de l'ensemble V_G (**GenerateNeighborsSet**).

ii. Générer et valider des k-sous-ensembles (**GenerateKSubSetsAndValidate**).

(a) Générer une combinaison d'un k-sous-ensemble S de la liste des sommets voisins $N^k(u)$.

(b) Si le sous-graphe G_S induit par le k-sous-ensemble S n'est pas connexe, aller à (a).

(c) Sinon, chercher la matrice d'adjacence canonique du sous-graphe G_S, pour le mettre dans sa propre catégorie (voir la section 3.2.4). Il est utile de noter que chaque catégorie contient les sous-graphes isomorphes.

(d) Répéter les étapes précédentes (a), (b), (c) jusqu'à générer toutes les combinaisons possibles des k-sous-ensembles.

iii. Lorsque tous les sous-graphes d'ordre k contenant notamment le sommet de départ u sont énumérés, nous retirons ce sommet u du graphe G pour ne pas dupliquer des sous-graphes dans les étapes suivantes (ligne 5).

Afin de chercher tous les sous-graphes d'ordre k du graphe G, nous suivons le même processus pour les autres sommets du graphe G.

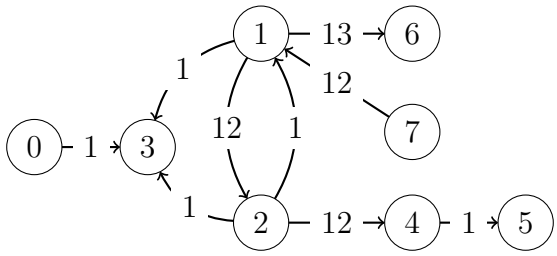


Figure 3.1 Un sous-graphe extrait de la version 1.7R1 du système Rhino. Les sommets représentent les classes, et les arcs représentent les relations entre les classes. La description des étiquettes sur les arcs est définie dans la section 4.2.1.

3.2.1 Voisinage (GenerateNeighborsSet)

L'idée principale de l'algorithme de voisinage est de parcourir le graphe orienté et étiqueté $G(V, A, L, l)$ en largeur pour chercher d'une manière itérative la liste des sommets voisins d'un sommet donné u. Ce type de parcours consiste à chercher d'une part, tous les sommets successeurs auxquels on peut accéder directement à partir du sommet u, et d'autre part, tous les sommets prédécesseurs depuis lesquels on peut arriver directement au sommet u, ensuite de chercher les sommets qui sont accessibles par le premier successeur ou prédécesseur de u, et le deuxième successeur ou prédécesseur de u et ainsi de suite.

En effet, la recherche de la liste des sommets voisins consiste à explorer les sommets voisins d'un graphe orienté et étiqueté G une couche à la fois. La première couche contient le sommet de départ u, ensuite, la deuxième couche contient les sommets voisins du sommet u, et la troisième couche contient les sommets voisins de la deuxième couche et ainsi de suite.

D'une manière formelle, nous décrivons les itérations de recherche de la liste des sommets voisins d'un sommet donné u. Cette liste est noté par $N^k(u)$ où k est l'ordre des sous-graphes. Nous notons aussi par N_i l'ensemble des sommets trouvés à la i^{eme} couche. Donc, la liste des sommets voisins $N^k(u)$ est défini par la formule suivante :

$$N^k(u) = \bigcup_{i=1}^{k} N_i(N_{i-1}) \text{ où } \begin{cases} N_1 = \{u\} \\ N_i(N_{i-1}) = \bigcup_{v \in N_{i-1}} N(v) \\ N(v) = \{w : (v,w) \in A(G) \vee (w,v) \in A(G)\} \end{cases}$$

Les sommets de l'ensemble N_i du i^{eme} couche sont insérés dans la liste des sommets voisins $N^k(u)$ juste après les sommets de N_{i-1} du $(i-1)^{eme}$ couche. Cet ordre, d'insertion des sous-ensembles N_i dans la liste des sommets voisins $N^k(u)$, est très important pour optimiser l'algorithme de génération des k-sous-ensembles représentant les sous-graphes d'ordre k. De plus, un sommet déjà visité ne sera pas inséré de nouveau dans l'ensemble des sommets N_i de la couche courante.

Exemple : Prenons comme exemple le graphe orienté et étiqueté G définie dans la figure 3.1. Dans cet exemple nous cherchons la liste des sommets voisins $N^4(0)$ du sommet 0 avec $k = 4$.

Premièrement, nous parcourons le graphe G en largeur en partant du sommet 0 qui se trouve à la couche 1. Donc, la liste des sommets voisins est : $N^4(0) = \{\overset{couche\ 1}{0}\}$.

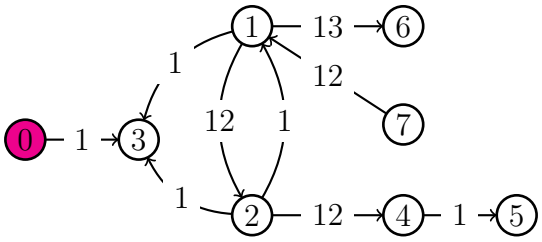


Figure 3.2 Sommets de la première couche de la liste des sommets voisins $N^4(0)$.

En effet, si nous partons du sommet 0 pour chercher tous les sommets successeurs ou prédécesseurs, nous trouvons seulement un sommet successeur qui est le sommet 3. Donc, la

liste des sommets voisins devient $N^4(0) = \{\overset{couche\ 1}{0}, \overset{couche\ 2}{3}\}$.

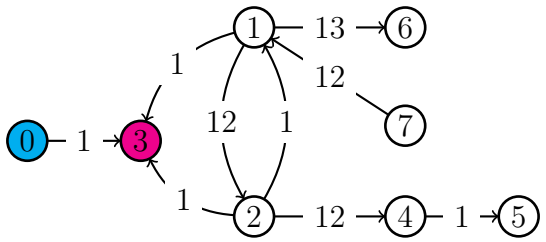


Figure 3.3 Sommets de la deuxième couche de la liste des sommets voisins $N^4(0)$.

Dans la première couche, il n'y a plus de sommets qui sont directement accessibles à partir du sommet 0. Donc, maintenant, il faut chercher les sommets qui sont accessibles à partir de la deuxième couche. Il s'agit donc de chercher les voisins du sommet 3. Dans ce cas, il y a seulement les deux sommets prédécesseurs 1 et 2. Maintenant, la liste des sommets voisins devient $N^4(0) = \{\overset{couche\ 1}{0}, \overset{couche\ 2}{3}, \overset{couche\ 3}{1,2}\}$.

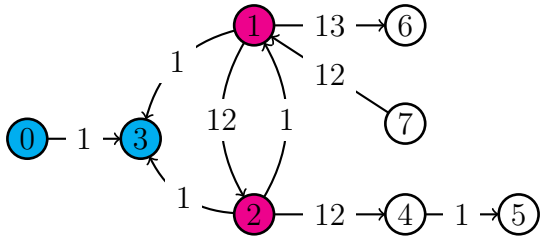


Figure 3.4 Sommets de la troisième couche de la liste des sommets voisins $N^4(0)$.

À ce stade, il faut donc chercher les voisins des sommets 1 et 2. Nous commençons par le premier sommet 1 qui a trois sommets successeurs 2, 3, et 6 et un seul sommet prédécesseur 7. Les sommets 2 et 3 ont déjà été insérés dans la liste des sommets voisins $N^4(0)$, donc, nous ne devons pas les ajouter de nouveau dans $N^4(0)$. Par contre, le deuxième sommet 2 a trois sommets successeurs 1, 3, et 4, mais les sommets 1 et 3 sont déjà présents dans la liste $N^4(0)$, donc il ne faut pas les insérer de nouveaux. Cependant dans cette couche, les seuls voisins à ajouter sont les sommets 6,7, et 4. Donc, la liste des sommets voisins $N^4(0)$ composant les sommets des sous-graphes d'ordre 4 est la suivante :

$$N^4(0) = \{\overset{couche\ 1}{0}, \overset{couche\ 2}{3}, \overset{couche\ 3}{1,2}, \overset{couche\ 4}{6,7,4}\}.$$

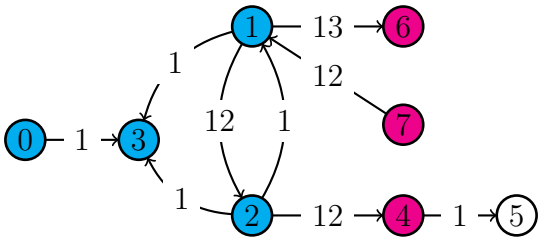


Figure 3.5 Sommets de la quatrième couche de la liste des sommets voisins $N^4(0)$.

Description de l'algorithme de voisinage "GenerateNeighborsSet" (voir les algorithmes 3.1 et 3.2) : Essayons maintenant de passer à l'implémentation de l'algorithme de voisinage (GenerateNeighborsSet). Nous utilisons une liste N_{ku} pour stocker les sommets voisins du sommet u. Au début de l'algorithme 3.2, aucun sommet n'a été visité, il faut donc commencer par stocker le sommet de départ u par lequel nous allons commencer le parcours du graphe G. Donc, le sommet u est stocké dans la liste N_{ku} et dans la liste des sommets voisins de la première couche N_i (lignes 2 & 3).

Ensuite nous cherchons les voisins de la i^{me} couche N_i (lignes 6-13). La recherche des sommets voisins d'un sommet v se fait par l'intermédiaire de la matrice d'adjacence M du graphe G. Une fois que nous découvrons tous les voisins de la i^{me} couche N_i, nous les ajoutons dans la liste N_{ku} (ligne 16). L'algorithme 3.2 s'arrête lorsqu'il trouve les sommets voisins de la $(k-1)^{me}$ couche (ligne 4).

Tous les sous-graphes incluant notamment le sommet u seront trouvés. Par la suite, le sommet u sera retiré du graphe G. Cela signifie que la prochaine liste des sommets voisins ne contient pas le sommet u pour éviter la duplication des sous-graphes trouvés. Le processus de recherche de la liste des sommets voisins $N^k(u)$ se répète sur les sommets restants du graphe G non encore traités.

3.2.2 Génération des k-sous-ensembles "GenerateKSubSetsAndValidate" (voir la section 2a)

L'idée principale de l'algorithme SGFinder est de trouver les sous-graphes d'ordre k (où k est le nombre des sommets) comportant un sommet de départ u. Pour ce faire,

1. Nous cherchons l'ensemble de tous les sommets voisins $N^k(u)$ du sommet u (voir la section 3.2.1).
2. Cet ensemble $N^k(u)$ est utilisé pour trouver toutes les combinaisons possibles des sous-ensembles des sommets de taille k (appelé aussi k-sous-ensemble).

Algorithme 3.2 Voisinage

Entrées : Un sommet u de l'ensemble des sommets du graphe $G(V, A, L, l)$.
k est l'ordre des sous-graphes à trouver.

Sorties : Une liste N_{ku} contenant les sommets voisins de u.
$Joined(v, w)$ est une fonction qui retourne vraie si les deux sommets v et w sont connectés.

```
1: function GenerateNeighborsSet (u, k)
2: N_ku.Add(u) ;
3: N_i.Add(u) ;
4: for i = 1 → k do
5:    # Chercher les voisins (N) de la ième couche (N_i)
6:    for all v ∈ N_i do
7:       N = {} ;
8:       for all w ∈ V(G) do
9:          if Joined(v, w) then
10:            N.Add(w) ;
11:         end if
12:      end for
13:   end for
14:   N_i = {} ;
15:   for all n ∈ N do
16:      N_ku.Add(n) ;
17:      N_i.Add(n) ;
18:   end for
19: end for
20: return N_ku
21: end function
```

3. Chacune de ces combinaisons contient l'ensemble des sommets d'un sous-graphe induit d'ordre k (appelé aussi k-sous-graphe) (voir la section 1.4.1).
4. La connectivité de chacun de ces sous-graphes induits d'ordre k doit être vérifiée.

L'algorithme de génération des k-sous-ensembles est basé sur une méthode récursive. En effet, les k-sous-ensembles générés représentent évidemment les ensembles des sommets des k-sous-graphes. Cependant, nous devons vérifier que les k-sous-graphes générés sont connexes tels que définis dans la section 3.2.3.

Le problème de génération des sous-ensembles de taille k est connu comme un problème de combinaison sans répétition de k éléments d'un ensemble S de taille n. Le nombre des k-sous-ensembles croit exponentiellement avec la taille de S et de k. Le nombre total de toutes les combinaisons possibles de ce problème est donné par le coefficient binomial C_n^k.

Dans le but de réduire le nombre de combinaisons possibles, nous introduisons deux critères de sélection des sommets de la liste des sommets voisins $N^k(u)$ pour générer des sous-ensembles de taille k. Grâce à ces deux critères, nous pouvons donc éviter plusieurs combinaisons inutiles.

Premier critère de sélection : L'objectif de l'algorithme dédié au voisinage (voir la section 3.2.1) est de chercher une liste des sommets voisins contenant un sommet de départ u et tous ces voisins qui sont accessibles via un chemin passant par le sommet u. Cependant, le premier élément des sous-ensembles est toujours le sommet de départ u, ce qui signifie que le sommet u doit être nécessairement inclu dans tous les k-sous-ensembles.

Deuxième critère de sélection : Pour chercher le $i^{\text{ème}}$ élément d'un k-sous-ensemble, il faut choisir seulement un sommet parmi les sommets de la première à la $i^{\text{ème}}$ couche de la liste des sommets voisins $N^k(u)$. En d'autres termes, pour l'exemple 3.2.1, il faut prendre en compte les points clés suivants :

- Le deuxième élément d'un k-sous-ensemble est toujours l'un des sommets de la deuxième couche de $N^k(u)$.
- Le troisième élément d'un k-sous-ensemble est toujours l'un des sommets de la deuxième ou de la troisième couche de $N^k(u)$.
- Le dernier élément d'un k-sous-ensemble est l'un des sommets situés entre la deuxième et la $k^{\text{ème}}$ couche de $N^k(u)$.

Description de l'algorithme de génération des k-sous-ensembles : Tout d'abord, nous avons besoin d'un vecteur V_G pour stocker les éléments du k-sous-ensemble à engendrer. Premièrement, le vecteur V_G contient le sommet de départ u, et la variable n indique le prochain sommet à prendre de la liste des sommets voisins $N^k(u)$ pour compléter le k-sous-ensemble (vecteur V_G) (ligne 3). A chaque itération, nous devons vérifier si le sous-ensemble généré est de taille k (ligne 4). Si c'est le cas, il faut passer ce k-sous-ensemble à la fonction Validate(V_G) pour vérifier sa connectivité, et enfin, le remettre dans la catégorie contenant ses sous-graphes identiques (ligne 5). Si le nombre des sommets présents dans le vecteur V_G est différent de k, nous procédons à une autre opération récursive sur le reste des sommets de la liste $N^k(u)$ (ligne 7) pour chercher le sommet candidat suivant à insérer dans V_G.

3.2.3 Connectivité

Dans cette section nous décrivons la méthode utilisée pour vérifier la connectivité des k-sous-ensembles engendrés par l'algorithme couvert à la section 3.3. Dans la littérature, il

Algorithme 3.3 Générateur des sous ensembles de taille k

Entrées : N est la liste des sommets voisins du sommet u ($N^k(u)$).
u est le sommet en traitement.
V_G est un vecteur contenant les sommets d'un sous-graphe.
n indique le prochain sommet à prendre de la liste des sommets voisins N.
Sorties : Tous les k-sous-ensembles contenant le sommet u.

```
1: Procedure GenerateKSubSets (N, u, V_G, n)
2: for i = n to |N| do
3:    V_G.Add(N[i]) ;
4:    if |V_G| == k then
5:       Validate(V_G) ;
6:    else
7:       GenerateKSubSets (N, u, V_G, n + 1) ;
8:    end if
9: end for
10: end Procedure
```

existe deux techniques d'exploration qui sont différentes pour vérifier la connectivité d'un graphe.

i. La première est la recherche en profondeur d'abord (*Depth First Search ou DFS*) qui explore immédiatement les successeurs de tout sommet visité et,

ii. La deuxième est la recherche en largeur d'abord (*Breadth First Search ou BFS*) qui visite les sommets couche par couche c'est-à-dire il ne visite aucun sommet de la couche $n+1$ avant qu'il explore tous les sommets de la couche n (une couche est un ensemble de sommets auxquels ils sont accessibles directement via un sommet donné).

Dans notre approche, nous utilisons l'algorithme de recherche *en largeur d'abord (BFS)*. Il est décrit dans l'algorithme 3.4, et la figure 3.6 met en évidence son fonctionnement sur un exemple de sous-graphe composé de l'ensemble des sommets {0, 4, 2, 3, 1} (voir la figure 3.6).

Description de l'algorithme de connectivité : L'idée principale de l'algorithme de connectivité est de construire d'une manière itérative un k-sous-graphe connexe $G_S(V_S, A_S, l, L)$, induit par un k-sous-ensemble S.

- À la première étape de l'algorithme, nous construisons V_S par le sommet de départ u, puis nous retirons ce sommet u de S (ligne 2 & 3).
- À la deuxième étape, nous ajoutons certains sommets de S à l'ensemble V_S. Ces sommets ajoutés doivent être connectés dans le graphe $G(V, A, l, L)$ (équivalent au diagramme des classes), au sommet u. Ces sommets ajoutés sont ensuite retirés de S (ligne 5 & 6).

- À chaque étape suivante, nous ajoutons certains sommets de S à l'ensemble V_S. Ces sommets ajoutés doivent être connectés dans le graphe G, au moins à un des sommets de V_S. Ces sommets ajoutés sont ensuite retirés de S (ligne 5 & 6).
- L'algorithme s'arrête si l'une des deux conditions suivantes est satisfaite :
  - Le k-sous-ensemble S est vide. Dans ce cas, le k-sous-graphe G_S est connexe.
  - Il existe au moins un sommet de S qui n'est pas connecté à aucun sommet de V_S. Dans ce cas, le k-sous-graphe G_S n'est pas connexe.

Algorithme 3.4 Connectivité

Entrées : Un k-sous-ensemble S.
Le sommet u en traitement.
Sorties : Vrai si le $k-$sous-graphe G_S induit par le k-sous-ensemble S est connexe
La fonction $Connected(v, V_S)$ retourne vrai si le sommet v est connecté à un sommet de V_S.

```
1: Function Connectivity (S, u)
2: V_S.Add(u)
3: S.Remove(u)
4: while ∃v ∈ S and Connected(v, V_S) do
5:    V_S.Add(v)
6:    S.Remove(v)
7: end while
8: if S is empty then
9:    return true
10: end if
11: return false
12: End Function
```

Exemple Considérons l'exemple illustré par la figure 3.6, que $k = 5$, et le k-sous-ensemble $S = \{0, 4, 2, 3, 1\}$. Donc, pour vérifier la connectivité du k-sous-graphe $G_S(V_S, A_S, l, L)$ induit par S, il faut premièrement commencer l'algorithme 3.4 par le sous-ensemble $V_S = \{0\}$ et donc S devient $\{4, 2, 3, 1\}$ suite au retrait du sommet 0. Ensuite :

i. Dans la deuxième étape, l'algorithme cherche les sommets de S qui sont connectés à au moins un sommet appartenant à V_S. Donc, dans ce cas, il y a seulement le sommet 4. Par conséquent, $V_S = \{0, 4\}$ et $S = \{2, 3, 1\}$.

ii. Dans la troisième étape, il faut examiner de nouveau les sommets de S. Le sommet 2 est connecté au sommet 4. Donc, V_S devient $\{0, 4, 2\}$ et $S = \{1, 3\}$.

iii. Dans la quatrième étape, le sommet 1 est connecté au sommet 2. Donc, V_S devient $\{0, 4, 2, 1\}$ et $S = \{3\}$.

iv. Dans la cinqième étape, le sommet 3 est connecté au sommet 1 ($V_S = \{0, 4, 2, 1, 3\}$ et $S = \{\}$). Donc, l'algorithme s'arrête, avec un sous-graphe connexe $G_S(\{0, 4, 2, 1, 3\}, A_S, l, L)$

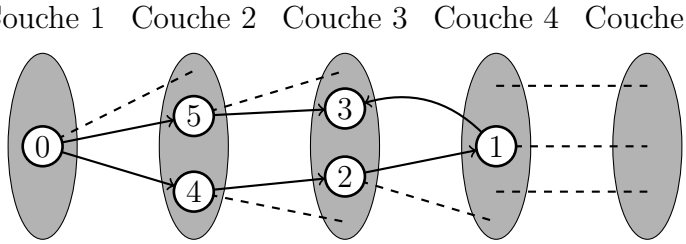


Figure 3.6 Connectivité d'un sous-ensemble de taille $k = 5$

3.2.4 Classement

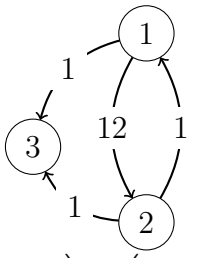
Dans cette section, nous introduisons quelques notions élémentaires utilisées dans notre algorithme pour regrouper les sous-graphes identiques dans des catégories. Il s'agit de l'isomorphisme des graphes, et de l'utilisation de la librairie Nauty de McKay(voir McKay, 1981, 2009) pour chercher la matrice d'adjacence représentant la forme canonique [1] d'un sous-graphe donné.

Un graphe G de n sommets peut-être représenté par plusieurs matrices d'adjacences d'ordre n (voir l'exemple de la figure 3.7). En fonction des permutations possibles des sommets sur les lignes et les colonnes, le nombre total des matrices d'adjacences générées est égal à $n!$. La forme canonique[1] d'un graphe G permet de représenter de manière unique cette multiplicité de matrices possibles. Cette forme canonique [1] est définie par la concaténation des lignes ou des colonnes de la matrice d'adjacence canonique. Cette matrice canonique est fournie par la librairie Nauty.

Exemple Soit la matrice d'adjacence canonique $M = \begin{pmatrix} l_{11} & l_{12} & \dots & l_{1n} \\ l_{21} & l_{22} & \dots & l_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{pmatrix}$ du graphe $G(V, A, L, l)$. Alors, la forme canonique de G est définie par l'un des deux chaines suivantes :

$$cl_1(M) = l_{11}l_{12}\dots l_{1n}l_{21}l_{22}\dots l_{2n}\dots l_{n1}l_{n2}\dots l_{nn}.$$
$$cl_1(M) = l_{11}l_{21}\dots l_{n1}l_{12}l_{22}\dots l_{n2}\dots l_{1n}l_{2n}\dots l_{nn}.$$

1. La forme canonique d'un objet est un moyen de représentation de cet objet. Pour tester si deux objets sont équivalents, il suffit de tester l'égalité de leurs formes canoniques.



$$\begin{pmatrix} 0 & 12 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 12 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 12 \\ 1 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 1 \\ 12 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 12 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 12 & 0 \end{pmatrix}$$

Figure 3.7 Matrices d'adjacences d'un graphe

.

Pour déterminer si deux graphes G_1 et G_2 sont isomorphes (identiques), il faut comparer les formes canoniques (chaines) de leurs matrices d'adjacences canoniques. Si les chaines de la forme canonique de G_1 et G_2 sont égaux (autrement dit $cl_1(G_1) = cl_1(G_2)$ ou bien $cl_2(G_1) = cl_2(G_2)$), donc les deux sous-graphes G_1 et G_2 sont isomorphes.

Afin de chercher les matrices d'adjacences canoniques, nous avons utilisé la librairie Nauty (voir McKay, 1981). Chaque matrice d'adjacence M' de chaque sous-graphe G' est fournie à la fonction nauty. La matrice d'adjacence canonique M'' retournée par nauty est transformée en une chaine $cl(M'')$. Ensuite la chaine $cl(M'')$ est transformée en un Hash Code [2] (voir Coffey, 2011). Celui-ci permet l'optimisation de l'espace mémoire alloué pour identifier les sous-graphes. Par conséquent, nous identifions les catégories contenant les sous-graphes identiques par les Hash Code.

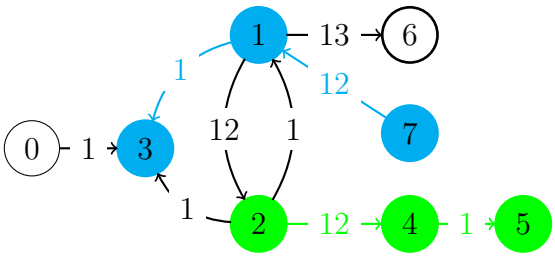


Figure 3.8 Un sous-graphe extrait de la version 1.7R1 de l'application Rhino avec deux occurrences {2,4,5} et {7,1,3} d'ordre $k = 3$

2. Une fonction de hachage utilisant 64bits peut engendrer une seule collision sur 3.7×10^7 hash code (http ://www.javamex.com/tutorials/collections/strong_hash_code.shtml).

3.3 Caractérisation des sous-graphes

Dans cette section, nous détaillons les opérations effectuées pendant la recherche des microarchitectures. Au cours de cette étude, nous nous intéressons seulement au calcul des zones définies à la section 3.3.1. Tandis que les autres opérations sont utiles pour effectuer d'éventuelles recherches, que nous mentionnons pour les réalisations futures à la section 6.3.

3.3.1 Nombre des zones

Nous avons vu à la section des classements des sous-graphes (voir 3.2.4) que l'algorithme est capable de regrouper tous les sous-graphes identiques dans des catégories. Le nombre total des occurrences des sous-graphes identiques est très grand à cause de certains sommets qui sont très connectés (voir la figure 3.9). Nous introduisons les zones pour réduire ce nombre d'occurrences qui n'a pas de signification. Le nombre des zones des sous-graphes identiques est défini par le nombre des régions disjointes qui n'ont pas d'arcs en commun. Autrement dit, deux sous-graphes isomorphes $G_1(V_1, A_1, L, l_1)$ et $G_2(V_2, A_2, L, l_2)$ sont dans la même zone si seulement s'il existe au moins $(v_i, v_j) \in A_1$ tel que $(v_i, v_j) \in A_2$.

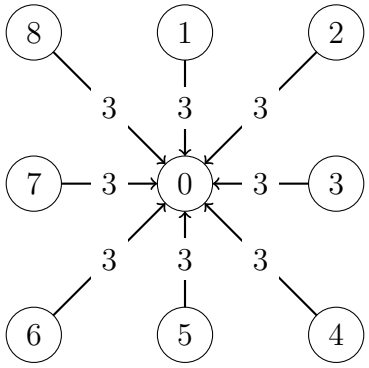


Figure 3.9 Un graphe en étoile (les sous-graphes contenant le sommet 0 et de taille supérieure à un, appartiennent à la même zone)

.

En effet, lorsqu'un nouveau sous-graphe G' est détecté par l'algorithme de recensement des sous-graphes, il sera mis dans sa propre catégorie, et nous recomptons le nombre des zones seulement si G' n'a aucun arc commun avec l'un des sous-graphes identiques à lui.

Nous allons maintenant décrire l'algorithme de calcul des zones. Donc, il s'agit d'utiliser un hashmap T_z pour stocker les arcs des sous-graphes détectés par l'algorithme 3.1 sachant que les clés représentent le code "*Hash Code*" de la forme canonique des sous-graphes.

Algorithme 3.5 Calcul des zones.

Entrées : une microarchitecture mA.

1: $nbArcs = sizeof(T_z)$;
2: insérer tous les arcs de mA dans T_z
3: **if** sizeof(T_z) = $nbArcs$+ sizeof(mA) **then**
4: $nbZones++$;
5: **end if**

3.3.2 Rôles des classes

Chaque microarchitecture est représentée par une structure composée d'un ensemble de classes coopérant entre elles par plusieurs relations. Plusieurs classes participantes peuvent jouer le même rôle dans les microarchitectures identiques. De même, une classe peut jouer différents rôles dans des microarchitectures différentes.

La figure 3.8 montre que la classe 1 et la classe 4 jouent le même rôle. L'exemple de la figure 3.10 décrit le patron de conception fabrique (*Factory*). Le rôle de la classe "Fabrique" est de créer les objets de la classe "Produit" sans exposer la logique d'instanciation.

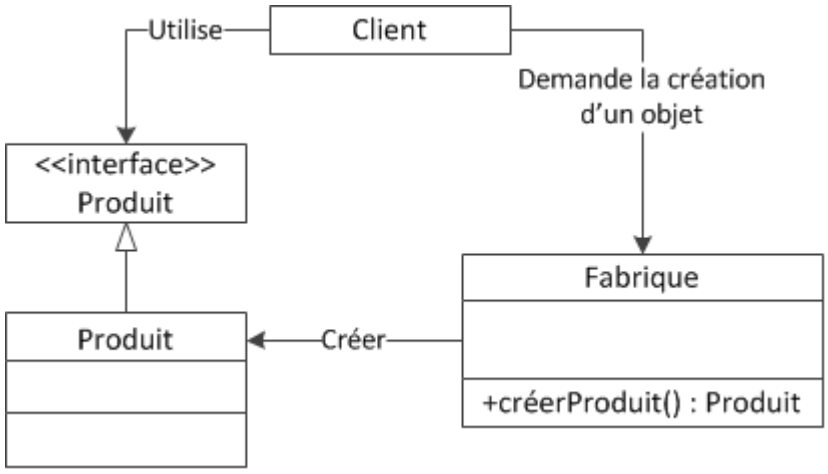


Figure 3.10 Patron de conception fabrique

Nous identifions les rôles des classes par l'intermédiaire de l'ordre des sommets de la matrice d'adjacence canonique retournée par l'algorithme nauty (voir McKay, 1981, 2009). Les rôles associés aux classes peuvent être visualisés par notre outil graphique SGViewer (voir la figure 3.11).

3.3.3 Tunnels

Le diagramme des classes évolue d'une version à une autre en subissant quelques changements sur sa structure. Étant donné que les microarchitectures sont les sous-graphes extraits

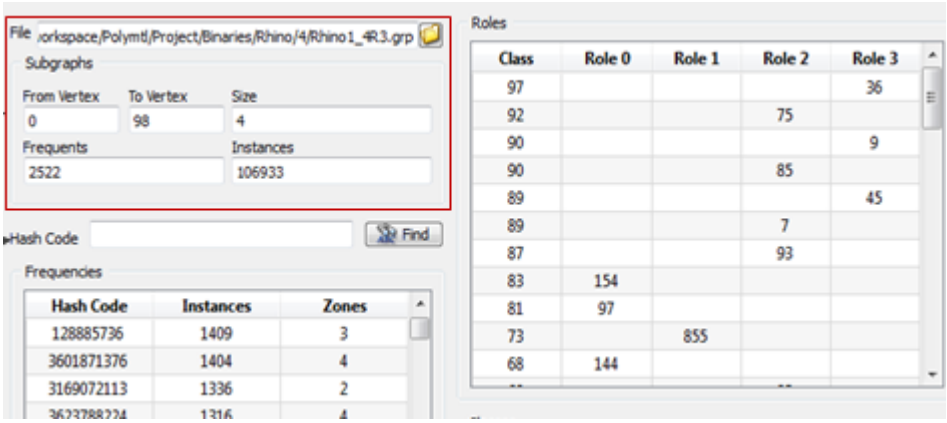


Figure 3.11 Illustration par l'outil SGViewer des rôles joués par des classes.

de ces diagrammes de classes, nous nous intéressons à la relation entre l'évolution des diagrammes et les microarchitectures extraites. Nous définissons un tunnel (voir Kpodjedo *et al.*, 2009) comme un ensemble de classes communes entre plusieurs versions. Les classes communes comportent notamment les classes renommées. En particulier, le tunnel de la première version contient toutes les classes. Dans cette section, nous allons voir comment identifier les microarchitectures dans les trois côtés du tunnel (figure 3.12) suivants :

i. **Intra-tunnel** : Microarchitectures complètement à l'intérieur du tunnel (ex : dans la figure 3.12, le sous-graphe G_1 est dans le troixième tunnel).

ii. **Extra-tunnel** : Microarchitectures complètement à l'extérieur du tunnel (ex : dans la figure 3.12, toutes les classes du sous-graphe G_2 se trouvent à l'extérieur du troixième tunnel).

iii. **Inter-tunnel** : Microarchitectures dont une partie des classes se trouve à l'intérieur du tunnel, et dont une autre partie des classes se trouve à l'extérieur de ce tunnel (ex : dans la figure 3.12, le sous-graphe G_3 a une classe à l'intérieur du troixième tunnel et deux classes à l'extérieur de ce troixième tunnel).

L'idée principale de l'algorithme est très simple. Il s'agit donc de comparer les classes d'une microarchitecture mA_i avec celles du tunnel (vecteur T) associé à une version donnée (lignes 1 & 4). Toutefois, nous utilisons un map $mapT$ comme structure de données pour stocker les occurrences des microarchitectures d'un tunnel (lignes 2, 5 & 7). Cependant, nous disons que les microarchitectures sont (voir l'algorithme 3.6) :

– À l'intérieur (IN) du tunnel si toutes ses classes sont dans le vecteur T (ligne 2).
– À l'extérieur (OUT) du tunnel si toutes ses classes ne sont pas dans le vecteur T (ligne 5).

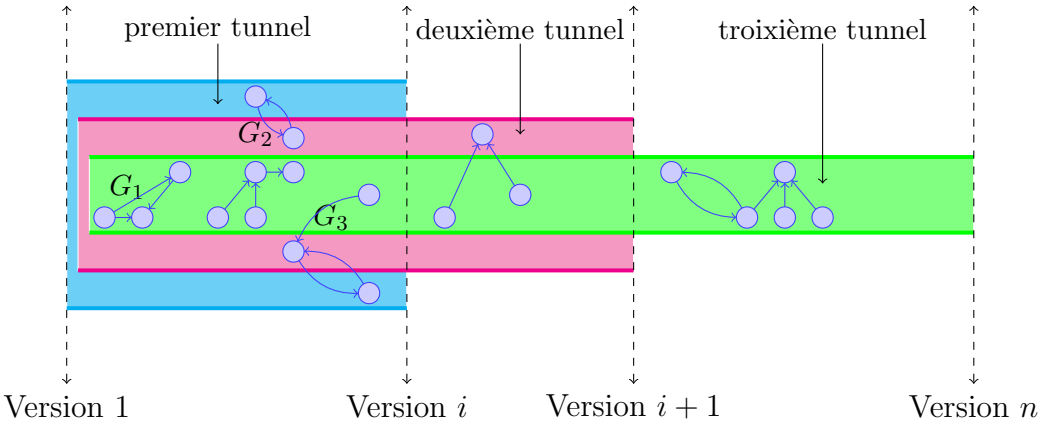


Figure 3.12 Identification des microarchitectures dans les tunnels

– Entre les deux côtés (BTW) du tunnel si seulement si certaines de ses classes se trouvent à l'intérieur du tunnel T et d'autres classes sont à l'extérieur de T (ligne 7).

Algorithme 3.6 Identification des microarchitectures dans les tunnels.

Entrées : Un vecteur contenant les classes d'un tunnel T et une microarchitecture mA.

```
1: if all classes of mA ∈ T then
2:    mapT[mA].IN++;
3: else
4:    if all classes of mA ∉ T then
5:       mapT[mA].OUT++;
6:    else
7:       mapT[mA].BTW++;
8:    end if
9: end if
```

3.3.4 Évolution des microarchitectures entre les versions

Dans cette section, nous proposons une classification des microarchitectures en sept profils selon leurs évolutions. Nous distinguons un profil par rapport à un autre par l'évolution des occurrences des microarchitectures. Cette classification tient en compte toutes les microarchitectures de toutes les versions du système traité. Donc, les sept profils sont les suivants (voir la figure 3.13) :

1. **Profil croissant :** Ce profil identifie les microarchitectures dont le nombre d'occurrences est toujours en croissance.
2. **Profil croissant instable :** Le nombre d'occurrences des microarchitectures n'est pas toujours en croissance c'est-à-dire, il peut baisser avec un taux faible au moins une fois

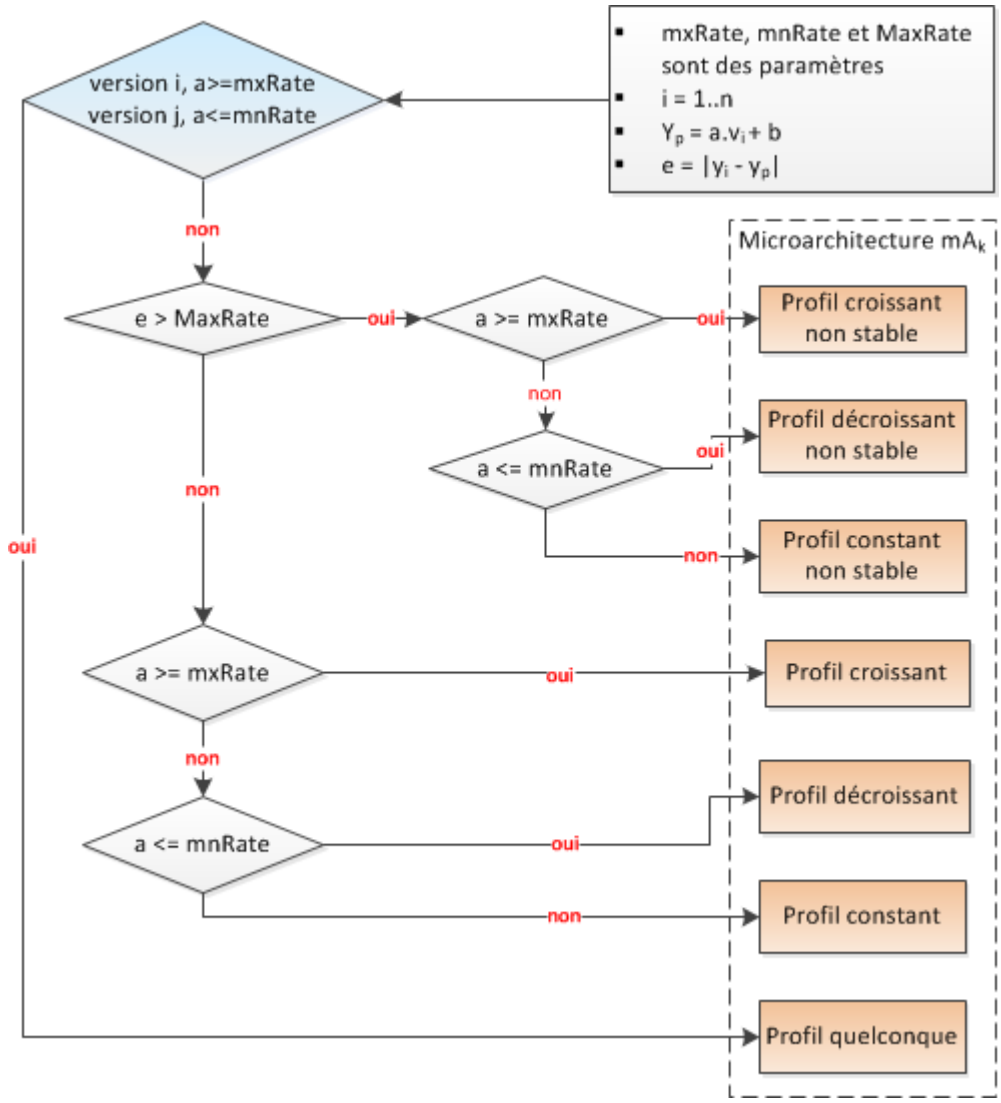


Figure 3.13 Classification des microarchitectures selon leurs évolutions entre les versions d'un système orienté objet.

dans une version, puis il continue généralement sa croissance.

3. **Profil décroissant :** Le nombre d'occurrences des microarchitectures est toujours en décroissance.
4. **Profil décroissant instable :** Ce profil définit les microarchitectures dont le nombre d'occurrences n'est pas toujours décroissant d'une version et sa suivante, mais il peut augmenter au moins une fois avec un taux faible, puis il décroît généralement.
5. **Profil constant :** Le nombre d'occurrences des microarchitectures est toujours constant.
6. **Profil constant instable :** Le nombre d'occurrences des microarchitectures peut augmenter et diminuer avec un taux faible par rapport à une valeur constante.

7. **Profil indéfini :** Pour ce profil, le nombre d'occurrences des microarchitectures peut augmenter, ou baisser avec un taux élevé entre les différentes versions du système traité.

Donc, étant donné n versions $\{v_1, v_2, \ldots, v_n\}$ d'un système orienté objet, et n nombre d'occurrences $\{y_{k1}, y_{k2}, \ldots, y_{kn}\}$ d'une microarchitecture mA_k. En effet, pour définir les sept profils, nous utilisons la droite de régression qui est donnée par la formule suivante :

$$Y = a.X + b$$

où :

- X représente les versions du système traité.
- Y représente le nombre d'occurrences de la microarchitecture mA_k .
- La pente $a = \frac{Covariance(X,Y)}{Variance(X)} = \frac{\sum_{i=1}^{n}(v_i-\overline{v})(y_{ki}-\overline{y})}{\sum_{i=1}^{n}(v_i-\overline{v})^2}$
- $b = \overline{y} - a.\overline{v}$
- $\overline{v} = \frac{\sum_{i=1}^{n} v_i}{n}$ est la moyenne des versions.
- $\overline{y} = \frac{\sum_{i=1}^{n} y_k i}{n}$ est la moyenne des occurrences des microarchitectures.

Remarque : La pente a est sensible au nombre d'occurrences, de sorte qu'avec un accroissement relatif équivalent, la covariance peut être plus forte pour la tendance avec le plus grand nombre d'occurrences. La corrélation r peut être une meilleure mesure pour définir les 7 profils.

$$r = \frac{\sum_{i=1}^{n}(v_i-\overline{v})(y_{ki}-\overline{y})}{\sqrt{\sum_{i=1}^{n}(v_i-\overline{v})^2} \times \sqrt{\sum_{i=1}^{n}(y_{ki}-\overline{y})^2}}$$

3.4 Conclusion

L'algorithme de recensement et de regroupement des sous-graphes, et les techniques décrites dans ce chapitre sont implémentés dans l'outil SGFinder. Cet outil est développé en C++. Le guide d'utilisation de cet outil est décrit dans l'annexe C.

Dans le chapitre suivant, nous décrivons la validation empirique qui est basée sur l'outil SGFinder.

CHAPITRE 4

VALIDATION EMPIRIQUE

Nous présentons dans ce chapitre les étapes suivies pour valider notre outils SGFinder. Nous effectuons une étude empirique basée sur notre outil SGFinder (voir la section 3) dont l'objectif est de vérifier son efficacité, et son applicabilité sur les petits et les moyens systèmes OO. Ainsi, nous posons des questions de recherche reliées aux propriétés de la stabilité et des défauts des microarchitectures existantes dans les systèmes OO.

4.1 Questions de recherche

Nos questions de recherche sont dérivées de nos objectifs visant à :
- Identifier les microarchitectures des systèmes OO.
- Chercher la relation entre les microarchitectures et les propriétés telles que la stabilité et les défauts.

Nous nous intéressons donc à répondre aux questions de recherche suivantes :

QR1 : SGFinder est-il capable de recenser les microarchitectures des systèmes OO de moyenne envergure ? Et quels types de microarchitectures existent dans les systèmes OO ?

Étant donné les différentes relations possibles entre les classes d'un diagramme des classes, les combinaisons des microarchitectures théoriquement possibles sont élevées. Par conséquent, il est nécessaire de vérifier l'applicabilité et l'efficacité de SGFinder sur les systèmes OO. Donc, notre premier objectif pour la question QR1 est de vérifier l'applicabilité de SGFinder sur les petits et les moyens systèmes OO. Par ailleurs, il est intéressant d'avoir un aperçu sur les types des microarchitectures existantes dans les systèmes OO.

QR2 : Existe-t-il des microarchitectures sans aucun défaut ou prédisposées aux défauts ?

En ce qui concerne la deuxième question QR2, nous voulons déterminer s'il existe des microarchitectures prédisposées aux défauts. Ainsi, nous voulons également identifier les microarchitectures qui ne peuvent pas avoir des défauts.

QR3 : Existe-t-il des microarchitectures stables ou prédisposées aux changements ?

Notre objectif pour la troisième question QR3 est de déterminer les microarchitectures stables, ainsi que les microarchitectures impliquées dans les changements effectués sur le code source. Donc une information préalable sur les microarchitectures prédisposées aux changements peut aider les concepteurs, les développeurs, et les mainteneurs des logiciels.

La complexité d'un logiciel a une conséquence directe sur les erreurs de programmation commises par les développeurs. Les architectes logiciels, les concepteurs, et les développeurs participant à la conception du diagramme des classes, peuvent engendrer des structures pouvant causer des défauts, et augmenter les chances de modification ultérieures au code source du logiciel. Il est donc important d'aider les concepteurs et les développeurs en signalant les microarchitectures prédisposées aux défauts et aux changements. De plus, une analyse qualitative (voir la section 5) pourrait donner une information suffisante sur les microarchitectures désirables ou à risque.

4.2 Objets

Pour évaluer l'efficacité et vérifier l'applicabilité de notre algorithme SGFinder, nous l'avons testé sur deux systèmes orientés objets développés en Java (Rhino et ArgoUml). Les données de ces deux systèmes sont illustrées dans le tableau 4.1.

- **Système Rhino** [1] **:** Rhino est un interpréteur JavaScript. Il implémente le langage JavaScript du standard ECMAScript (voir ECMA, 2007). Il est entièrement développé en Java et géré aujourd'hui par la communauté Mozilla. Il est capable de fonctionner avec les deux modes (i) compilé (intégré dans les applications), et (ii) interprété (script compilé en objets JavaScript). Actuellement, Rhino est le seul moteur inclus dans le noyau de JDK 1.6 et ses successeurs.
- **Système ArgoUml** [2] **:** ArgoUml est un outil d'analyse et de conception en UML des systèmes OO. Il est entièrement implémenté en Java par l'université de Southern California. Il est également capable de générer le code source à partir du diagramme des classes pour accélérer le développement des systèmes orientés objets.

4.2.1 Les données

Les deux systèmes Rhino et ArgoUml sont sélectionnés en raison de la disponibilité des données relatives à l'historique des défauts, et des changements. Nous illustrons au tableau

1. http ://www.mozilla.org/rhino/
2. http ://argouml.tigris.org/

4.1, les versions, et les données relatives aux défauts et aux changements des deux systèmes Rhino et ArgoUml.

Tableau 4.1 Sommaire des systèmes orientés objets

Systèmes	Versions	Nombre de			
		Classes	Relations	Bogues	Changements
Rhino	1.4R3	99	499	33	1466
	1.5R1	134	718	22	1020
	1.5R2	183	935	12	106
	1.5R3	178	929	41	666
	1.5R4	193	1017	115	236
	1.5R4.1	193	1017	95	835
	1.5R5	194	1027	53	808
	1.6R1	191	1118	21	217
ArgoUml	0.10	876	3491	218	1856
	0.10.1	876	3491	459	4006
	0.12	960	4232	133	2355
	0.14	1239	5609	142	1039
	0.15.6	1187	5898	102	541
	0.16	1190	5909	181	1299
	0.16.1	1190	5910	664	5048
	0.17.5	1243	7357	169	739

Nous avons téléchargé huit versions de chaque système pour recenser les microarchitectures de taille trois, quatre, et cinq.

Pour le premier système Rhino, nous avons traité les versions entre 1.4R3 et 1.6R1. Les défauts ont été récupérés via l'étude de Eaddy (voir Eaddy *et al.*, 2008), et les données concernant les changements sont extraites du fichier des journaux (logs) CSV.

Pour le deuxième système ArgoUml, nous avons traité les versions entre 0.10 et 0.17.5. Les données des défauts sont extraites du serveur de suivi des bogues "Bugzilla" et du serveur SVN, et les changements sont extraits du serveur SVN.

Relations traitées : Dans les systèmes OO, la communication entre les classes peut être exprimée par les trois types de relations d'association (1), d'agrégation (2), et d'héritage (3). De plus, dans un système, une classe A peut hériter d'une classe B, et en même temps les éléments de la classe A peuvent participer à plusieurs relations avec la classe B. Donc, nous

considérons que les relations en parallèle (dans le même sens i.e. de A vers B) entre les deux classes A et B comme une seule relation composée. En effet, un système contenant les trois relations simple 1, 2, et 3 peut contenir les quatre relations composées suivantes :

- 12 : L'association et l'agrégation
- 13 : L'association et l'héritage.
- 23 : L'agrégation et l'héritage.
- 123 : L'association, l'agrégation et l'héritage.

Notons que la relation composée entre une classe A et une classe B (de A vers B) est la concaténation de toutes les relations simples (1, 2, et 3) entre A et B (de A vers B). Cette concaténation est faite du plus petit au plus grand chiffre.

4.3 Approche

Nous avons utilisé la suite d'outils Ptidej [3] et son méta-modèle PADL [4] pour construire le diagramme des classes des deux systèmes orientés objets Rhino et ArgoUml, puis nous avons transformé le diagramme des classes de chaque système à un graphe orienté et étiqueté $G(V, A, L, l)$ tel que :

- L'ensemble des sommets V du graphe G représente l'ensemble des classes du système traité.
- L'ensemble des arcs A du graphe G représente les relations entre les classes.
- L'ensemble des étiquettes $L = \{1, 2, 3, 12, 13, 23, 123\}$ représente les relations simples et les relations composées.
- l est une fonction d'étiquetage (voir la section 1.4.1).

Ensuite, nous utilisons SGFinder (voir le chapitre 3) pour recenser toutes les microarchitectures de taille trois, quatre, et cinq. Puis, nous regroupons les microarchitectures identiques dans des catégories en utilisant l'outil nauty (voir la section 3.2.4). Et finalement, nous procédons à l'analyse de ces microarchitectures. La figure 4.1 illustre un aperçu global de cette approche.

Avec cette technique, le problème de recensement des microarchitectures d'une taille donnée k d'un système OO est modélisé comme un problème de recensements des sous-graphes

3. Ptidej (voir Guéhéneuc, 2004) est un ensemble d'outils de rétro-conception. Il permet d'extraire le diagramme des classes à partir du code source des programmes.

4. Le méta-modèle PADL (voir Albin-Amiot et Guéhéneuc, 2001) fait partie de la suite d'outils Ptidej. Il permet de donner la représentation générique d'un système OO quelconque. Il est indépendant des langages de programmation. Il comprend un analyseur de Java et un générateur des graphes.

d'ordre k d'un graphe orienté et étiqueté $G(V, A, L, l)$.

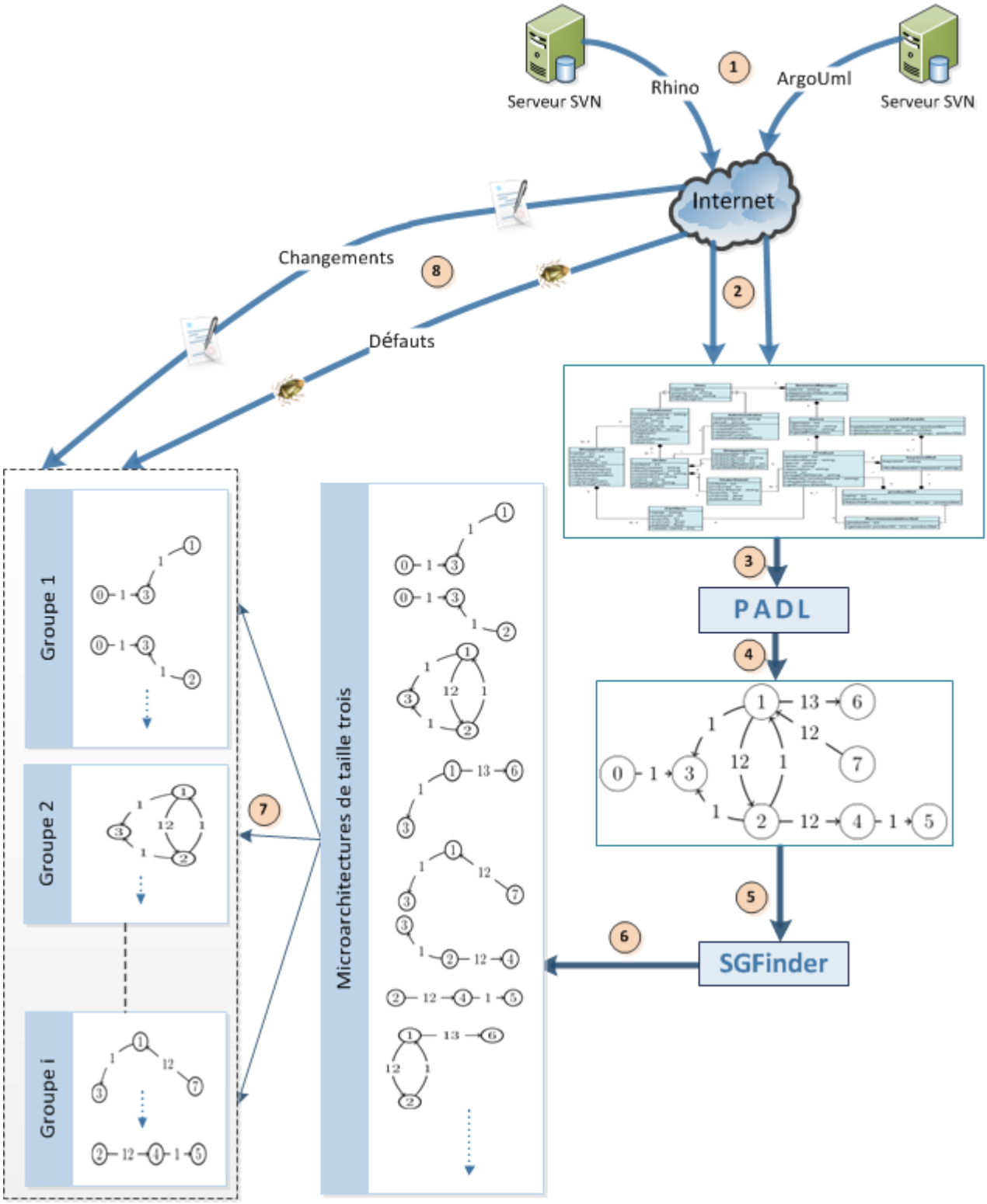


Figure 4.1 Aperçu de la nouvelle approche

4.4 Méthode d'analyse

Tout d'abord, nous présentons les informations associées aux microarchitectures, et leurs présences dans les différentes versions. Par la suite, nous décrivons l'utilisation de ces informations pour répondre à nos questions de recherche.

4.4.1 Caractéristiques des microarchitectures

Afin d'analyser les microarchitectures d'une manière exhaustive, nous introduisons les deux types d'informations suivantes :

Connectivité des classes participantes aux microarchitectures

Une microarchitecture est définie par une structure de classes reliées entre elles par des relations simples ou composées. Donc, pour le premier type d'information, nous nous focalisons sur les différents modèles de relations pouvant exister dans les microarchitectures. Nous distinguons les modèles des relations selon les variables suivantes :

i. ***nbRel*** est le nombre total des relations. Ce nombre comprend les trois types de relations simples et les quatre types de relations composées [5].

ii. ***nbAssoc*** est le nombre total des relations de type association.

iii. ***nbAggr*** est le nombre total des relations de type agrégation.

iv. ***nbInher*** est le nombre total des relations de type héritage.

v. ***nbCycl*** est le nombre total des relations cycliques. Une relation cyclique entre deux classes A et B est généralement générée par une paire d'associations c'est-à-dire la classe A appelle ou utilise la classe B et vice-versa.

Présence des microarchitectures

La présence des microarchitectures est la manière dont elles sont reparties dans les systèmes OO. Donc ce type d'information est défini par les quatre variables suivantes :

i. ***nbVersions*** est le nombre total des versions contenant une microarchitecture donnée.

ii. ***nbOccurrences*** est le nombre total des occurrences des microarchitectures identiques.

iii. ***nbZones*** est le nombre total des régions (voir la section 3.3.1) contenant les microarchitectures identiques.

5. Dans la présente étude, nous n'avons pas traité les relations qui sont sous forme d'une boucle sur une classe c'est-à-dire les relations d'une classe avec elle-même.

iv. ***nbClasses*** est la cardinalité de l'ensemble des classes participant au moins une fois dans une microarchitecture donnée.

4.4.2 Réponses aux questions de recherche

Tout d'abord, nous rappelons quelques notions de statistique descriptive, et la recherche d'information. Ces notions sont essentielles pour analyser les données représentées par les différentes variables définies dans la section 4.4.1.

Notions de la statistique descriptive

- **Médiane :** La médiane sépare une série des données en deux groupes de même nombres d'éléments. Le premier groupe contient les plus petites valeurs et le deuxième groupe contient les plus grandes valeurs.
- **Quartiles :** Les quartiles permettent de séparer une série des données en quatre groupes de même nombre d'éléments. Un quart des valeurs sont inférieures au premier quartile Q1, et un autre quart des valeurs sont supérieures au troisième quartile Q3.
- **Résumé en 5 chiffres :** Ce résumé se compose de minimum, premier quartile, médiane, troisième quartile, et maximum (Min, Q1, Médiane, Q3, Max).

Précision, Rappel, et F1

- **Précision (P) :** La précision est la proportion des éléments pertinents parmi les éléments sélectionnés.
- **Rappel (R) :** Le rappel est le rapport du nombre des éléments pertinents trouvés par le nombre des éléments pertinents disponibles. Il s'agit donc d'une proportion des éléments bien classés dans la classe des éléments pertinents.
- **F1 :** Est une mesure qui combine la précision et le rappel ($F1 = \frac{2 \times P \times R}{P+R}$).

Pour répondre à nos questions de recherche, nous utilisons les notions de la statistique descriptive pour analyser les données des variables définies dans la section 4.4.1. Plus précisément, nous utilisons la représentation de résumé en 5 chiffres pour analyser et comparer la distribution des données de ces variables.

Question QR1 : Pour répondre à la question de recherche QR1 visant à vérifier l'applicabilité de l'algorithme SGFinder sur les petits et les moyens systèmes OO, nous représentons

les données des variables (voir la section 4.4.1) sous le format de résumé en 5 chiffres. Nous représentons les microarchitectures des deux systèmes Rhino et ArgoUml séparément. Ensuite, nous représentons les microarchitectures communes entre ces deux systèmes. Étant donné que ces deux systèmes Rhino et ArgoUml sont développés par deux équipes différentes, donc le traitement des microarchitectures communes permettra de généraliser les résultats.

Pour vérifier l'efficacité de l'algorithme, nous calculons le temps d'exécution total de SG-Finder sur les huit versions des deux systèmes Rhino et ArgoUml.

Questions QR2 : Notre objectif pour cette question est d'identifier les microarchitectures prédisposées aux défauts ou celles qui n'ont aucun défaut. Donc, premièrement, nous identifions les classes possédant des défauts. Ensuite, nous utilisons pour chaque microarchitecture, son ensemble de classes, et nous calculons le pourcentage de ses classes boguées. Autrement dit, pour une microarchitecture donnée mA_i, nous devons adapter la précision P et le rappel R comme suivant :

- **Précision P :** La précision est le pourcentage des classes boguées par rapport à l'ensemble des classes participantes à la microarchitecture mA_i. Une précision de 100% signifie que la totalité des classes de la microarchitecture mA_i sont boguées, alors que une précision de 0% signifie qu'aucune des classes de mA_i n'est boguée. Donc, si la précision P est faible (respectivement élevée), la majorité des classes participantes aux microarchitectures ne sont pas boguées (sont boguées respectivement).
- **Rappel R :** Le rappel est le pourcentage des classes boguées dans la microarchitecture mA_i par rapport à la totalité des classes boguées dans le diagramme des classes.

Pour un nombre fixe de classes (ex. quatre), nous trions les microarchitectures par ordre décroissant de la précision P (c'est-à-dire les plus boguées vers les moins boguées). Nous sélectionnons ensuite les premières 10%, et les dernières 10% des microarchitectures de la liste des microarchitectures triées. Il est utile à noter que les microarchitectures sélectionnées sont les microarchitectures des deux systèmes OO Rhino et ArgoUml séparément, et les microarchitectures communes entre ces deux systèmes.

Questions QR3 : Pour répondre à cette question, nous utilisons la même méthodologie de réponse à la question QR2, sauf que nous analysons les changements effectuées sur les classes participantes aux microarchitectures au lieu d'analyser leurs défauts.

4.5 Conclusion

Dans ce chapitre nous avons présenté la validation empirique de l'outil SGFinder. Nous avons posé des questions de recherche, et présenté les réponses à ces questions. Dans le chapitre suivant, nous décrivons et discutons les résultats trouvés par notre outil SGFinder.

CHAPITRE 5

RÉSULTATS

Dans ce chapitre, nous présentons et analysons les résultats obtenus par l'outil SGFinder. Nous répondons aux trois questions de recherche qui visent à vérifier l'applicabilité de l'algorithme SGFinder sur les petits et les moyens systèmes OO, et à chercher la relation entre les microarchitectures et les propriétés telles que la stabilité et les défauts.

Remarque : Il est utile de noter que notre étude se focalise uniquement sur les versions déjà réalisées des deux systèmes Rhino et ArgoUml. Autrement dit, nous ne pouvons pas introduire des facteurs de bruit comme l'augmentation, ou la diminution de la taille du diagramme des classes, ou de modification de sa structure. Par conséquent, cette étude ne permet de prédire les mêmes résultats, ni pour les nouvelles versions à venir de ces deux systèmes, ni à d'autres systèmes autres que Rhino et ArgoUml.

5.1 QR1 : Applicabilité de l'outil SGFinder et la description des microarchitectures trouvées

Le tableau 5.1 présente le nombre total des occurrences des microarchitectures (Occ), et le nombre total des microarchitectures différentes (Dif) des deux systèmes Rhino et ArgoUml. Les microarchitectures de taille trois, quatre, et cinq sont indiquées dans l'entête du tableau.

Nous constatons que :

- Le nombre total des microarchitectures différentes des deux systèmes est presque le même, malgré que le diagramme des classes du système Rhino contient environ 10 fois moins de classes que le diagramme des classes du système ArgoUml (Ex. pour les microarchitectures de taille quatre, la deuxième version "1.5R1" du système Rhino contient 3390 microarchitectures différentes, et la deuxième version "0.10.1" du système ArgoUml contient 3432 microarchitectures différentes). Donc, le nombre total des microarchitectures différentes n'est pas strictement liée à la taille de diagramme des classes.
- Le système ArgoUml contient plus d'occurrences de microarchitectures par rapport au système Rhino.

Tableau 5.1 Les microarchitectures trouvées dans les deux systèmes Rhino et ArgoUml ("Occ" est le nombre total des occurences des microarchitectures, "Dif" est le nombre total des microarchitectures différentes).

Systèmes	Versions	Nombre	Taille des microarchitectures		
			3	4	5
Rhino	1.4R3	Occ	6352	106933	1556177
		Dif	162	2522	32572
	1.5R1	Occ	11046	230096	4255632
		Dif	197	3390	51800
	1.5R2	Occ	15417	389589	8926000
		Dif	229	4165	67237
	1.5R3	Occ	15519	398616	9254224
		Dif	226	4160	67069
	1.5R4	Occ	18203	509826	12968171
		Dif	249	4653	77010
	1.5R4.1	Occ	18203	509826	12968171
		Dif	249	4653	77010
	1.5R5	Occ	17235	453162	10841181
		Dif	275	5052	81859
	1.6R1	Occ	19782	534127	12862920
		Dif	265	4929	84538
ArgoUml	0.10	Occ	71114	2326149	76638002
		Dif	222	3432	48932
	0.10.1	Occ	71114	2326149	76638002
		Dif	222	3432	48932
	0.12	Occ	103935	4193438	174509644
		Dif	224	3833	62962
	0.14	Occ	171783	8614757	458176012
		Dif	266	5290	104706
	0.15.6	Occ	315612	38570009	4948807397
		Dif	269	5072	93893
	0.16	Occ	316138	38716539	4978353074
		Dif	269	5005	90458
	0.16.1	Occ	316710	38881514	5010343101
		Dif	269	5005	90458
	0.17.5	Occ	564424	91643885	13741073588
		Dif	279	4906	82877

Considérant les trois relations de base (association, agrégation et l'héritage), les quatre relations composées, et l'absence des relations, donc il y a huit connexions possibles entre deux graphes. Cela signifie que pour $n \times n$ paires de sommets (y compris les boucles), nous pouvons avoir (si nous ne prenons pas en considération la symétrie) au maximum $8^{(n \times n-(n-1))} \times 7^{n-1}$ sous-graphes connexes de n sommets. Par exemple, nous pouvons obtenir environ 22×10^{21}

microarchitectures différentes de taille cinq. Si nous considérons l'union des microarchitectures de taille cinq des deux systèmes Rhino et ArgoUml, nous obtenons seulement 32×10^4 différents microarchitectures, ce qui semble être un nombre très élevé, cependant ce nombre est juste une fraction du nombre total des combinaisons possibles.

Les tableaux 5.2 et 5.3 présentent les microarchitectures de taille trois, quatre et cinq, trouvées dans les deux systèmes OO Rhino et ArgoUml. Le tableau 5.4 présente les microarchitectures communes aux deux systèmes OO Rhino et ArgoUml. Les résultats sont représentés sous le format de résumé en 5 chiffres (Min, Q1, Médiane, Q2, Max). Bien que la taille du système Rhino soit petite par rapport au système ArgoUml, Rhino contient plus de combinaisons de graphes connexes pour toutes les tailles des microarchitectures. Le tableau 5.4 montre qu'il existe un nombre important des microarchitectures communes entre les deux systèmes Rhino et ArgoUml, ce qui permettrai d'augmenter les chances de généralisation des résultats.

Tableau 5.2 Microarchitectures existant dans le système Rhino. Chaque ligne indique le résumé en 5 chiffres : Min, Q1, Médiane, Q3, Max

Variables	Microarchitectures de taille		
	trois (373)	quatre (9203)	cinq (190061)
nbRel	2,4,5,6,11	3,6,7,9,20	4,8,10,11,27
nbAssoc	0,2,3,4,6	0,4,5,6,12	0,6,7,9,18
nbAggr	0,0,1,2,4	0,1,1,2,7	0,1,1,2,9
nbInher	0,0,1,1,3	0,0,1,2,5	0,0,1,2,6
nbCycl	0,0,1,1,3	0,0,1,2,6	0,0,1,2,8
nbZones	1,1,1,2,68	1,1,1,1,33	1,1,1,1,20
nbClasses	3,3,5,10,140	4,4,5,9,147	5,5,6,10,156
nbVersions	1,2,5,8,8	1,1,3,6,8	1,1,2,4,8

Concernant les informations de connectivité, les microarchitectures du système Rhino contiennent plus de relations par rapport à celles du système ArgoUml. En regardant le nombre maximum des relations, nous constatons que certaines microarchitectures plus connectées se trouvent dans les deux systèmes Rhino et ArgoUml. La figure 5.1 illustre une de ces microarchitectures. Cette figure représente trois microarchitectures de taille cinq avec le plus grand nombre de relations dans le système Rhino (22 relations), dans le système ArgoUml (21 relations), et dans les deux systèmes (19 relations).

Tableau 5.3 Microarchitectures existant dans le système ArgoUml. Chaque ligne indique le résumé en 5 chiffres : Min, Q1, Médiane, Q3, Max

Variables	Microarchitectures de taille		
	trois (349)	quatre (8224)	cinq (180295)
nbRel	2,4,5,6,10	3,6,7,8,17	4,8,9,10,23
nbAssoc	0,2,3,4,6	0,3,4,6,11	0,5,6,8,15
nbAggr	0,0,1,2,4	0,1,1,2,6	0,1,2,2,8
nbInher	0,0,1,1,3	0,0,1,1,5	0,0,1,2,7
nbCycl	0,0,1,1,3	0,0,1,1,5	0,0,1,2,7
nbZones	1,1,1,3,583	1,1,1,2,315	1,1,1,1,171
nbClasses	3,3,6,19,944	4,4,7,16,940	5,5,8,19,944
nbVersions	1,4,7,8,8	1,2,4,7,8	1,1,3,5,8

Tableau 5.4 Microarchitectures existant dans les deux systèmes Rhino et ArgoUml. Chaque ligne indique le résumé en 5 chiffres : Min, Q1, Médiane, Q3, Max

Variables	Microarchitectures de taille		
	trois (250)	quatre (3993)	cinq (52862)
nbRel	2,4,4,5,8	3,5,6,7,11	4,7,8,9,15
nbAssoc	0,2,3,4,6	0,3,4,5,10	0,5,6,7,14
nbAggr	0,0,1,2,4	0,0,1,2,4	0,0,1,2,6
nbInher	0,0,1,1,3	0,0,1,1,4	0,0,1,1,6
nbCycl	0,0,0,1,3	0,0,1,1,4	0,0,1,1,5
nbZones	1,1,2,5,326	1,1,1,2,174	1,1,1,2,95
nbClasses	3,5,8,26,542	4,6,11,24,544	5,8,14,31,540
nbVersions	1,5,7,8,8	1,3,5,7,8	1,3,4,6,8

5.1.1 Temps de calcul de l'outil SGFinder

Le tableau 5.5 et la figure 5.2 présentent le temps total d'exécution de l'outil SGFinder, sur les huit versions des deux systèmes OO Rhino et ArgoUml. En effet, le temps d'exécution global de l'outil SGFinder pour le recensement des microarchitectures se varie entre 0 secondes et 28 heures. Pour les microarchitectures de taille trois, l'outil SGFinder s'exécute en moins d'une seconde, et pour les microarchitectures de taille quatre, l'outil SGFinder s'exécute entre 0 à 2 secondes. Alors que le temps total d'exécution le plus long est utilisé pour le recensement des microarchitectures de taille cinq. En particulier l'outil SGFinder s'exécute pendant 28 heures pour recenser 82.877 microarchitectures différentes, et 13.741.073.588 occurrences dans le système ArgoUml.

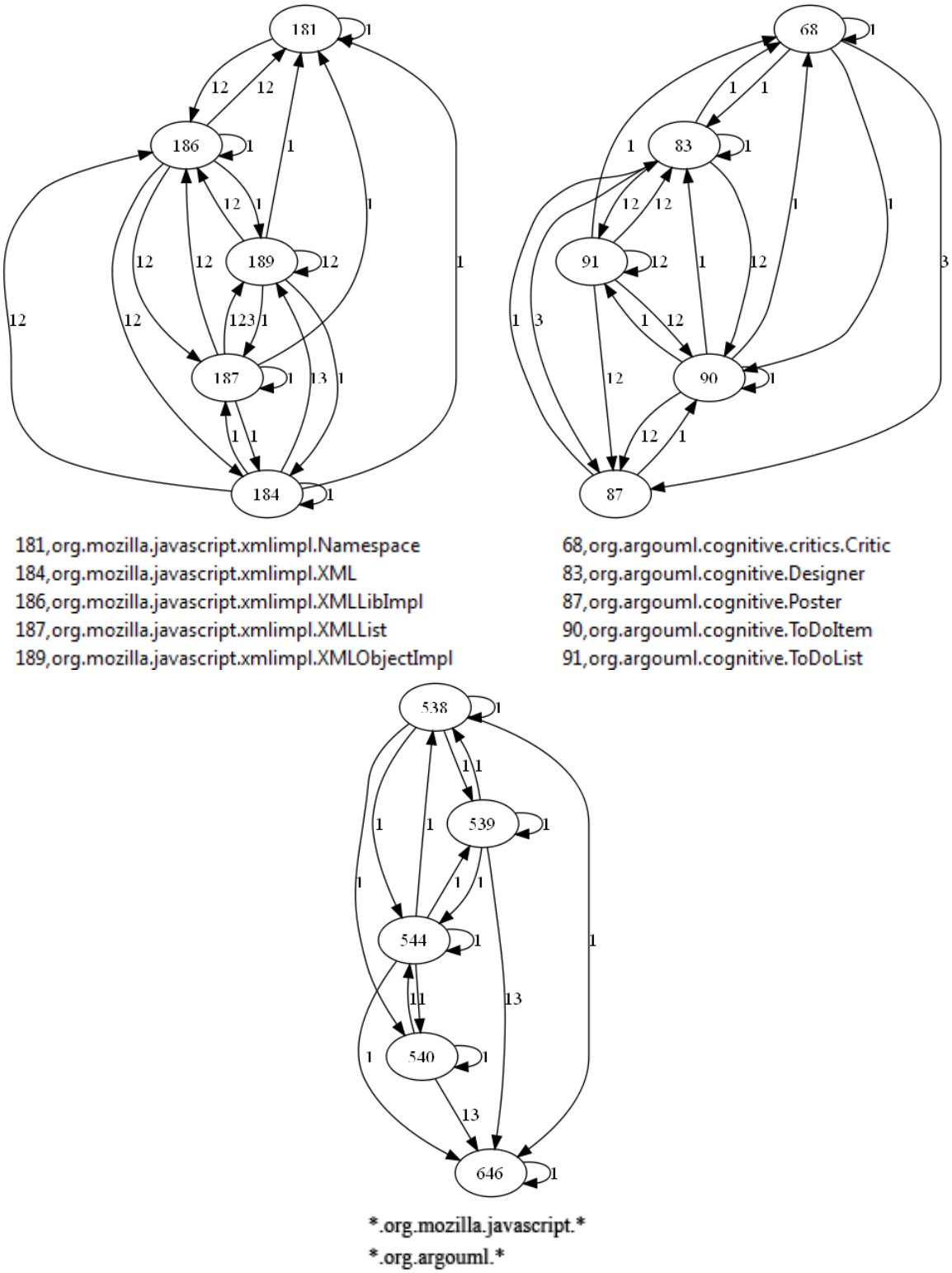


Figure 5.1 Les microarchitectures les plus connectées de taille cinq dans les deux systèmes Rhino et ArgoUml.

En effet, le temps total d'exécution de l'outil SGFinder ne dépend pas des versions d'un système. Cependant, il dépend principalement de la densité du diagramme des classes, c'est-à-dire du nombre de relations existantes entre les classes, et la taille du diagramme des classes. Compte tenu de la présence de certaines classes fortement connectées, le nombre des combinaisons possibles augmente énormément le nombre d'occurrences des microarchitectures. Par conséquent, les classes les plus connectées font augmenter exponentiellement le temps d'exécution global de l'algorithme SGFinder.

Tableau 5.5 Temps d'éxecution de l'algorithme SGFinder sur les deux systèmes Rhino et ArgoUml.

Systèmes	Versions	Temps d'exécution (h mm ss)		
		3	4	5
Rhino	1.4R3	0 00 00	0 00 00	0 00 07
	1.5R1	0 00 00	0 00 01	0 00 20
	1.5R2	0 00 00	0 00 01	0 00 46
	1.5R3	0 00 00	0 00 02	0 00 47
	1.5R4	0 00 00	0 00 02	0 01 06
	1.5R4.1	0 00 00	0 00 01	0 01 06
	1.5R5	0 00 01	0 00 01	0 00 57
	1.6R1	0 00 00	0 00 02	0 01 06
ArgoUml	0.10	0 00 00	0 00 14	0 33 43
	0.10.1	0 00 01	0 00 14	0 34 00
	0.12	0 00 00	0 00 26	1 18 13
	0.14	0 00 01	0 00 49	04 09 11
	0.15.6	0 00 01	0 03 12	11 57 36
	0.16	0 00 01	0 03 11	12 05 49
	0.16.1	0 00 01	0 03 04	11 55 39
	0.17.5	0 00 02	0 06 57	28 02 56

En effet l'outil SGFinder est capable d'explorer les petits et les moyens systèmes OO, pour recenser les microarchitectures de taille allant jusqu'à cinq. Nous pouvons donc répondre positivement à cette première question de recherche QR1.

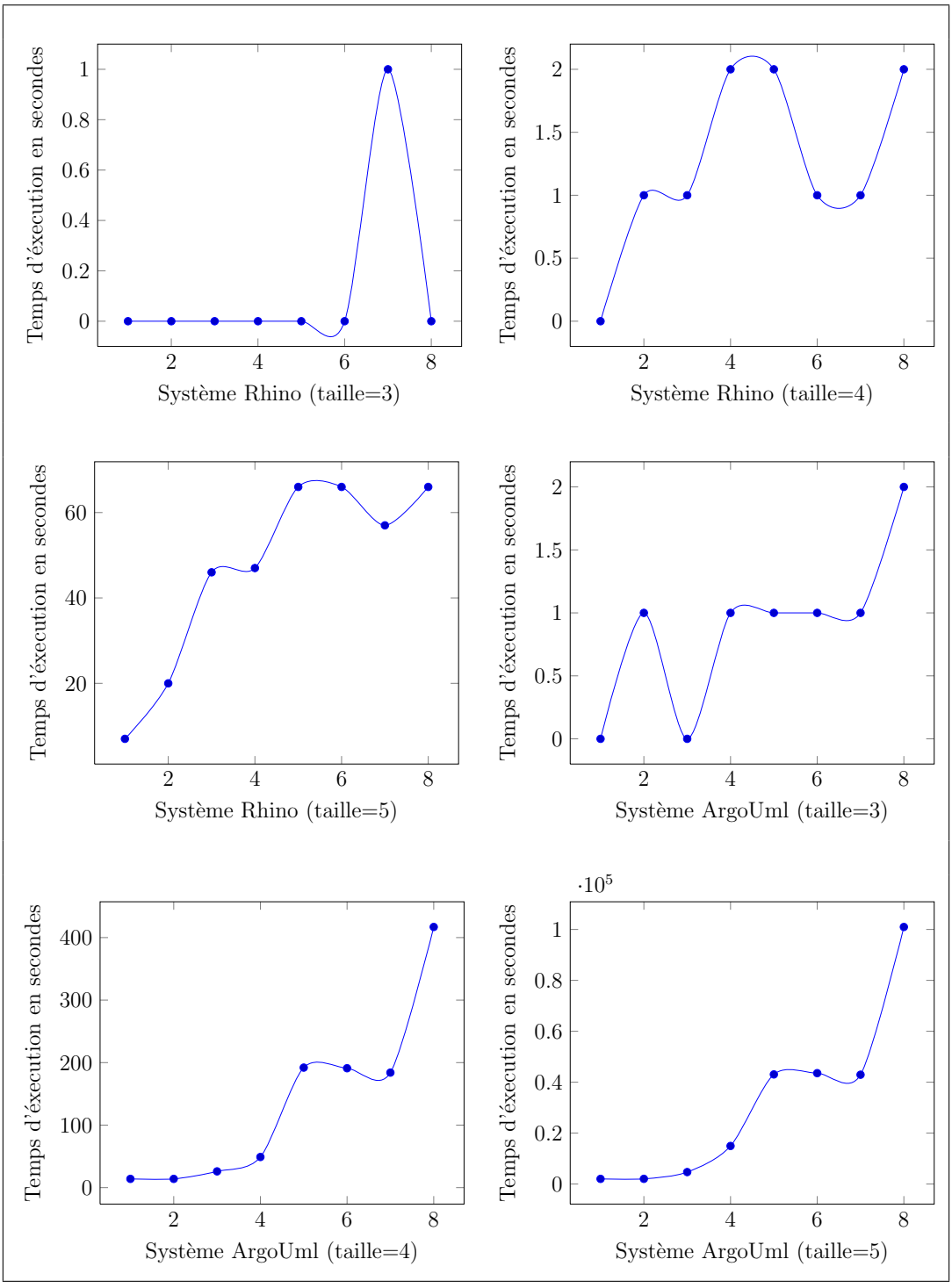


Figure 5.2 Temps d'éxecution de l'algorithme SGFinder sur les deux systèmes Rhino et ArgoUml.

5.2 QR2 : Prédisposition des microarchitectures aux défauts

Le tableau 5.6 présente les microarchitectures de taille trois, quatre et cinq, et qui sont communes entre les deux systèmes Rhino et ArgoUml. En utilisant la précision P, ce tableau 5.6 rapporte les premières 10% des microarchitectures les plus prédisposées aux défauts, ainsi que les dernières 10% qui sont les moins prédisposées aux défauts. Les résultats sont représentés sous le format de résumé en 5 chiffres (Min, Q1, Médiane, Q2, Max). Bien qu'il existe des microarchitectures intéressantes dans chacun des deux systèmes OO séparément, nous nous focalisons uniquement sur les microarchitectures communes pour les raisons suivantes :

- Les deux systèmes sont développés par deux équipes différentes.
- Ils ont des tailles différentes.
- Ils appartiennent à deux domaines d'applications différents.

Tableau 5.6 Les microarchitectures les plus et les moins prédisposées aux défauts des deux systèmes Rhino et ArgoUml. Chaque ligne indique le résumé en 5 chiffres : Min, Q1, médiane, Q3, Max

Variables	Échantillon	Microarchitectures de taille		
		trois (250)	quatre (3994)	cinq (52862)
Precision	Première 10%	41,44,47,52,67	42,45,48,54,83	41,43,46,52,84
	Dernière 10%	0, 5, 9,11,13	0, 8,10,13,14	0, 2, 5, 7,22
F1	Première 10%	5,8,13,17,30	2,10,12,15,40	4,11,14,18,54
	Dernière 10%	0,1, 1, 2, 5	0, 1, 3, 5,16	0, 2, 5, 7,22
nbRel	Première 10%	3,4,6,6,8	4,7,8,8,10	4,8,9,10,14
	Dernière 10%	2,4,4,5,6	3,6,6,7,09	4,7,8,09,13
nbAssoc	Première 10%	2,3,4,5,6	2,5,6,7,10	2,7,8,9,14
	Dernière 10%	0,2,2,3,4	0,3,4,5,08	0,4,5,6,11
nbAggr	Première 10%	0,1,1,2,4	4,7,8,8,10	4,8,9,10,14
	Dernière 10%	0,1,1,2,2	3,6,6,7,09	4,7,8,09,13
nbInher	Première 10%	0,0,0,1,2	0,0,0,1,3	0,0,0,1,4
	Dernière 10%	0,1,1,2,2	0,0,1,1,3	0,0,1,2,5
nbCycl	Première 10%	0,1,1,2,3	0,1,1,2,4	0,1,2,2,5
	Dernière 10%	0,0,0,1,1	0,0,0,1,4	0,0,0,1,5
nbZones	Première 10%	1,1,2,2,9	1,1,1,1,6	1,1,1,1, 7
	Dernière 10%	1,1,1,2,5	1,1,1,1,7	1,1,1,1,14
nbClasses	Première 10%	3,4,5,7,33	4,5,6, 9, 59	1,1,1,1, 7
	Dernière 10%	3,4,5,7,18	4,5,7,12,133	1,1,1,1,14
nbVersions	Première 10%	1,4,5,6,8	1,3,4,5,8	1,2,3,5,8
	Dernière 10%	1,3,4,5,8	1,2,3,5,8	1,2,3,4,8

Donc, le choix des microarchitectures communes pourrait augmenter les chances de généralisation des résultats.

D'après la précision P, le tableau 5.6 montre la présence de certaines microarchitectures particulièrement prédisposées aux défauts, et d'autres microarchitectures sans aucun défaut. Nous constatons d'après la variable $nbVersions$ que les huit versions des deux systèmes Rhino et ArgoUml contiennent certaines microarchitectures qui sont continuellement prédisposées aux défauts.

Prenons comme exemple les microarchitectures de taille cinq. Rappelons que la précision P est la proportion des classes boguées parmi l'ensemble des classes participantes aux microarchitectures. La précision P la plus élevée est $P = 84\%$ et l'unité de mesure la plus élevée est $F1 = 54\%$. Autrement dit, certaines microarchitectures sont particulièrement prédisposées aux défauts, et elles contiennent la majorité des classes boguées (c'est-à-dire il existe un nombre important de classes boguées dans l'ensemble des classes participantes aux microarchitectures). Par contre, il existe des microarchitectures sans aucun défaut (précision $P = 0\%$, c'est-à-dire il n'y a aucune classe boguée dans l'ensemble des classes de ces microarchitectures).

Analysons maintenant les informations de connectivité. Nous remarquons que les microarchitectures les plus prédisposées aux défauts sont les plus connectées par rapport à celles les moins prédisposées aux défauts. En particulier, les microarchitectures ayant des défauts contiennent généralement plusieurs relations d'associations. Lorsque nous évaluons la corrélation [1] entre le nombre des relations d'associations dans les microarchitectures et la précision P, nous trouvons que les deux corrélations de Spearman et Pearson donnent des valeurs supérieures à 0,4 avec un niveau de confiance plus de 99% (les valeurs de corrélation obtenues sont similaires pour les relations cycliques). L'inverse semble être vrai pour les relations d'agrégations et d'héritages qui ont tendance à être moins élevées dans la plupart des microarchitectures prédisposées aux défauts (les valeurs de corrélation sont environ -0,15).

Les informations de la présence et de la répartition des microarchitectures les plus et les moins prédisposées aux défauts, indiquent que la plupart d'entre elles ne sont pas très communes ou bien réparties sur le diagramme des classes. Généralement, elles sont reparties sur une seule zone. Cependant, le nombre maximal des zones contenant les microarchitectures les plus prédisposées aux défauts est égal à sept, ce qui est une exception remarquable. Ceci

1. La corrélation est le degré de liaison qui unit deux ou plusieurs variables.

peut être expliqué par le pourcentage faible des classes boguées dans les deux systèmes. Compte tenu du nombre relativement restreint des classes boguées dans les deux systèmes Rhino et ArgoUml, les précisions P obtenues sont très intéressantes.

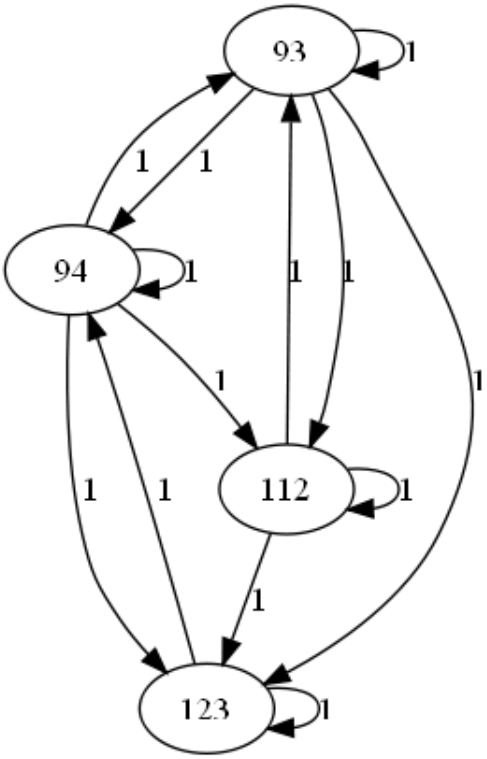


Figure 5.3 Exemple d'une microarchitecture prédisposée aux défauts avec une précision de $P = 81\%$

Nous illustrons dans la figure 5.3 une microarchitecture de taille quatre qui est prédisposée aux défauts. Elle a une précision moyenne $P = 81.25\%$. Cette microarchitecture se trouve dans les deux versions 1.5R4, et 1.5R4.1 du système Rhino et dans les quatre versions 0.14, 0.15.6, 0.16, et 0.16.1 du système ArgoUml. Chaque classe participante à cette microarchitecture communique avec les autres classes via la relation d'association.

En particulier, lorsque nous récupérons l'ensemble des microarchitectures ayant trois relations cycliques et dont chaque classe communique avec les autres, nous obtenons le résumé en 5 chiffres $(28, 44, 56, 67, 81)$. Ce résumé se rapproche de celui des microarchitectures les

plus prédisposée aux défauts (42, 45, 48, 54, 83).

Finalement, nous pouvons donc conclure que la réponse à la question de recherche QR2 est positive, parce qu'il existe des microarchitectures communes entre les deux systèmes OO Rhino et ArgoUml, qui n'ont aucun défaut, et d'autres qui sont fortement prédisposées aux défauts.

5.3 QR3 : Prédisposition des microarchitectures aux changements

Tableau 5.7 Les microarchitectures les plus et les moins prédisposées aux changements qui existent dans les deux systèmes Rhino et ArgoUml. Chaque ligne indique le résumé en 5 chiffres : Min, Q1, médiane, Q3, Max

Variables	Échantillon	Microarchitectures de taille		
		trois (250)	quatre (3994)	cinq (52862)
Precision	Première 10%	83,86,88,91,97	85,88,90,93,100	84,86,89,92,100
	Dernière 10%	20,35,38,39,44	17,38,43,48, 50	10,42,47,50, 53
F1	Première 10%	3,5,6,9,27	4,5,6,8,22	5,8,9,12,37
	Dernière 10%	0,1,1,2,13	0,2,3,5,33	0,3,6,11,55
nbRel	Première 10%	3,5,6,6,7	4,6,7,8,10	4,8,9,10,15
	Dernière 10%	2,4,4,6,7	3,5,6,7,10	4,7,8, 9,13
nbAssoc	Première 10%	1,3,4,5,6	2,5,6,7,10	1,6,8,9,13
	Dernière 10%	0,1,2,3,4	0,3,4,5,08	1,4,5,7,11
nbAggr	Première 10%	0,0,0,1,3	0,0,1,2,4	0,0,1,1,5
	Dernière 10%	0,1,2,2,3	0,1,2,2,4	0,1,2,2,6
nbInher	Première 10%	0,0,1,1,2	0,0,0,1,4	0,0,0,1,5
	Dernière 10%	0,1,1,2,2	0,0,1,1,3	0,0,1,1,4
nbCycl	Première 10%	0,1,1,2,3	0,1,1,2,4	0,0,1,2,5
	Dernière 10%	0,0,1,1,2	0,1,1,2,3	0,0,1,1,4
nbZones	Première 10%	1,1,2,2,5	1,1,1,1, 5	1,1,1,1,10
	Dernière 10%	1,1,1,2,5	1,1,1,1,11	1,1,1,1,10
nbClasses	Première 10%	3,4,5,7,26	4,5,6, 8, 90	5,6, 8,11,158
	Dernière 10%	3,4,5,7,69	4,5,8,14,321	5,8,13,29,404
nbVersions	Première 10%	1,4,5,7,8	1,2,4,5,8	1,2,3,4,8
	Dernière 10%	2,4,5,7,7	1,3,4,5,8	1,2,3,5,8

Le tableau 5.7 présente les microarchitectures communes entre les deux systèmes Rhino et ArgoUml. En utilisant la précision P, ce tableau 5.7 rapporte les premières 10% des microarchitectures les plus prédisposées aux changements, ainsi que les dernières 10% qui

sont les moins prédisposées aux changements. Les résultats sont représentés sous le format de résumé en 5 chiffres (Min, Q1, Médiane, Q2, Max). Pour les mêmes raisons mentionnées dans la section de la question de recherche QR2 (voir la section 5.2), nous choisissons d'analyser les microarchitectures communes entre les deux systèmes. Ce choix pourrait aussi augmenter les chances de généralisation des résultats sur les changements.

Les résultats illustrés dans le tableau 5.7 montrent bien qu'il existe certaines microarchitectures particulièrement stables. Cependant, étant donné que les changements sont très communs dans les deux systèmes choisis, nos données (avec une précision de $P = 100\%$) sont moins concluantes pour les microarchitectures les plus fréquemment changées. Nous constatons d'après la variable $nbVersions$ que les huit versions des deux systèmes Rhino et ArgoUml contiennent certaines microarchitectures qui sont continuellement prédisposées aux changements.

Les informations de la connectivité des microarchitectures donnent des conclusions similaires à la question de recherche RQ2. La majorité des microarchitectures prédisposées aux changements contiennent plus de relations d'associations et moins de relations d'agrégations et d'héritages.

Vu le nombre élevé des classes modifiées dans les deux systèmes Rhino et ArgoUml, les microarchitectures avec des précisions P faibles méritent plus de traitement.

La figure 5.4 illustre une microarchitecture particulièrement stable. Cette microarchitecture se trouve dans la version 1.6R1 du système Rhino, et dans la version 0.17.5 du système ArgoUml. Cette microarchitecture a une précision moyenne $P = 30\%$. Elle est comme un patron avec la forme d'une cascade.

5.4 Limites de validité

Nous avons montré précédemment dans la section 5.1 que l'algorithme SGFinder peut fonctionner sur les petits et les moyens systèmes orientés objets. Afin de chercher les microarchitectures des grands systèmes, nous partitionnons le diagramme des classes de ces systèmes soit en composantes, ou en sous-systèmes, soit nous introduisons des méthodes heuristiques.

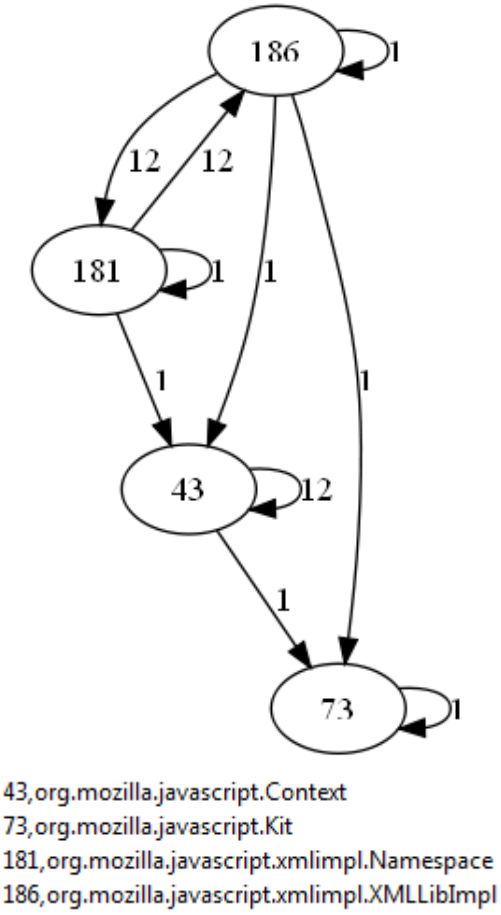


Figure 5.4 Exemple d'une microarchitecture sans défauts avec une précision P de 30%

5.4.1 Limite de validité de construction

Cette limite concerne la relation entre la théorie et l'observation. Il est principalement dû à la nécessité d'une validation manuelle et d'une analyse qualitative. En effet, nous ne pouvons pas donner une interprétation spécifique aux microarchitectures prédisposées aux défauts et aux changements. Nous ne pouvons pas deviner l'intention des développeurs sur les structures des classes crées pendant l'implémentation. Donc, nous supposons que nous connaissons seulement le domaine d'application du système traité, les classes, et les relations entre elles.

5.4.2 Limite de validité interne

Cette limite est principalement liée à toute confusion pouvant influencer les résultats obtenus. En particulier, cette limite peut être due à la distinction entre les relations d'associations et d'agrégations, aussi, elle peut être liée au nombre des défauts et des changements attribués

aux classes du système traité. Nous évitons plusieurs facteurs, en utilisant des outils solides, et en réutilisant les données des défauts fournies soit par d'autres chercheurs, soit par les applications de suivi des bogues et des changements.

5.4.3 Limite de validité externe

Cette limite concerne la généralisation des résultats obtenus. Notre étude est limitée à deux systèmes Rhino et ArgoUml, sachant que notre approche est faisable sur d'autres systèmes de taille équivalentes. En effet, nous ne pouvons pas garantir que nous obtenons les mêmes microarchitectures communes qui sont prédisposées aux défauts et aux changements. Nous croyons que le choix des systèmes répond à cette limite parce que les deux systèmes choisis appartiennent à deux différents domaines d'applications, ils ont des tailles différentes, et ils sont développés par deux équipes différentes.

5.4.4 Limite de validité de la conclusion

Cette limite concerne la relation entre le traitement et les résultats obtenus. Nous ne prétendons aucune relation entre les microarchitectures et les caractéristiques indésirables. En effet, le jugement de l'existence de telle relation peut-être confirmé par l'expérience des développeurs. Dans cette étude, nous mettons en évidence les microarchitectures les plus et les moins prédisposées aux défauts et aux changements, en utilisant la précision P et le rappel R comme unités de mesure.

5.5 Conclusion

Dans ce chapitre, nous avons présenté, et analysé les résultats trouvés par SGFinder. Nous avons montré que notre outil SGFinder permet de recenser les microarchitectures des petits et moyens systèmes OO. Nous avons utilisé les informations de connectivité, et de présence des microarchitectures dans les différentes versions des deux systèmes OO Rhino et ArgoUml, pour chercher la relation entre ces microarchitectures et leurs propriétés telles que les changements et les défauts.

CHAPITRE 6

CONCLUSION

Les microarchitectures existantes dans les systèmes OO sont des modèles (i.e. structures de classes et des relations entre elles) conçus pour réaliser un ou plusieurs composants d'un logiciel. Dans le cadre de ce projet, nous avons présenté un nouvel algorithme efficace, et un outil appelé SGFinder pour énumérer toutes les microarchitectures d'une taille donnée. Le nouvel outil SGFinder utilise une technique d'énumération efficace pour détecter toutes les microarchitectures des petits et moyens systèmes orientés objets. Cet outil est capable de regrouper les microarchitectures identiques dans des catégories, en utilisant la librairie existante nauty (voir McKay, 1981). N. De plus, il nous permet d'étudier les propriétés associées aux microarchitectures telles que leur stabilité, et leur prédisposition aux bogues et aux défauts.

Dans les sections suivantes, nous synthétisons les travaux que nous avons effectué pendant notre recherche, ensuite, nous exposons la limitation de la technique proposée, puis, nous finissons par les orientations et les réalisations éventuelles pour améliorer notre travail.

6.1 Synthèse des travaux

Pour valider la technique de recensement des microarchitectures des systèmes orientés objets, et atteindre les objectifs visés par notre recherche,

- Nous avons testé l'algorithme SGFinder sur :
  i. Huit versions de l'interpréteur JavaScript/ECMAScript (**Rhino** [1]) intégré dans les applications Java de la communauté Mozilla, et
  ii. Huit versions de l'outil d'analyse et de conception en UML des systèmes orientés objets (**ArgoUml** [2])
- Nous avons démontré que notre algorithme SGFinder est capable de recenser toutes les microarchitectures de taille trois, quatre et cinq appartenant aux systèmes Rhino[1] et ArgoUml[2].
- Nous avons effectué de nombreuses opérations pour définir les rôles joués par les classes, identifier les microarchitectures dans les trois côtés du tunnel, et tracer l'évolution des

1. http ://www.mozilla.org/rhino/
2. http ://argouml.tigris.org/

microarchitectures entre les différentes versions. Ces opérations ont été réalisées lors de cette recherche, et peuvent servir pour des éventuelles publications et travaux futures.
- Nous avons concentré notre étude sur les deux propriétés de la stabilité et des défauts (bogues) des microarchitectures.
- Nous avons concentré notre recherche sur les microarchitectures les plus boguées, ainsi que sur celles les moins boguées. De même, cette étude s'est concentrée sur les microarchitectures les plus impliquées dans les changements, et également celles les moins impliquées dans les changements.
- Nous avons rapporté quelques microarchitectures les plus intéressantes relativement à leurs connectivités (relations entre les classes), à leurs présences dans les différentes versions des systèmes traités, et à leurs nombre d'occurrence dans le diagramme des classes.

6.2 Limitations de la solution proposée

A travers ce travail, nous avons pu démontrer que l'algorithme SGFinder est capable de recenser toutes les microarchitectures de taille trois, quatre, et cinq appartenant aux petits et moyens systèmes orientés objets. Par ailleurs, nous avons vu que le problème de recensement de toutes les microarchitectures est un problème exponentiel. Donc, parmi les limitations notables de nos travaux, nous pouvons toutefois distinguer les deux principales limitations suivantes :

- **Une limitation sur la taille des microarchitectures et des systèmes :** La méthode proposée pour recenser toutes les microarchitectures dépend toujours de la taille des microarchitectures à trouver, ainsi que de la taille des systèmes orientés objets à traiter. En effet, notre algorithme SGFinder ne peut pas garantir la recherche de toutes les microarchitectures d'une taille supérieure à cinq des petits et moyens systèmes orientés objets. De plus, l'outil SGFinder ne peut pas garantir le traitement des grands systèmes.
- **Une limitation technique des ressources matérielles :** Nous notons principalement le fait que nous traitons un problème exponentiel, qui a une conséquence reliée directement aux ressources matérielles telles que la taille de la mémoire RAM, et la vitesse du CPU. Cependant, nous ne pouvons ni prévoir le temps total d'exécution que l'algorithme SGFinder peut prendre pour détecter toutes les microarchitectures, ni l'espace mémoire à allouer pour stocker les résultats reliés à la recherche de ces microarchitectures. En effet, ce problème est lié principalement à la densité du graphe représentant le système orienté objet à traiter.

6.3 Améliorations futures

Malgré que nous ayons atteint nos objectifs de recherche, il reste encore d'autres travaux à faire dans l'intention d'optimiser l'algorithme SGFinder, d'effectuer une analyse qualitative pour documenter les microarchitectures stables, et les analyser de la même façon que les précédentes études réalisées sur les patrons de conception. Ces éléments sont détaillés ci-dessous :

i. **Optimisation de l'algorithme :** Lorsqu'un logiciel est mis en œuvre, il rentre dans sa phase de maintenance lui rapportant des changements sur le code comme les corrections des bogues et l'ajout des nouvelles fonctionnalités. De plus, la maintenance a un grand impact sur la génération de plusieurs versions. Les versions contiennent souvent le même diagramme des classes avec quelques modifications mineures sur la structure de ce diagramme des classes. Ces modifications consistent à ajouter ou enlever certaines classes ou méthodes. Donc, dans le cas général, la plupart des changements apportés au code ne touchent pas la structure globale des classes et les relations entre elles.

   Donc, afin d'optimiser le temps global d'exécution de l'algorithme SGFinder et le rendre efficace, il faut recenser les microarchitectures d'une manière incrémentale c'est-à-dire nous ne devons pas énumérer de nouveau les microarchitectures existantes dans le diagramme des classes qui est commun entre les versions. Donc, avec cette technique, nous réduisons le temps d'exécution de l'algorithme par rapport à la technique classique où le recensement des microarchitectures est effectué sur le diagramme des classes de chaque version au complet. En conséquence, nous exploitons les résultats de recensement des microarchitectures d'une version précédente pour trouver les microarchitectures d'une version courante.

ii. **Rôles des classes :** Une éventuelle étude possible serait de traiter et analyser les microarchitectures de la même façon que les patrons de conception c'est-à-dire étudier les microarchitectures en définissant le rôle attribué à chacune des classes qui les compose, et de définir les particularités structurelles associées à chaque rôle. Par analogie aux sous-graphes, il faudrait prendre en compte le type et le nombre d'arcs sortants et entrants au niveau de chaque sommet qui fait partie de l'ensemble des sommets d'un sous-graphe.

iii. **Généraliser l'étude :** Une autre étude possible serait donc d'étendre notre recherche sur plusieurs systèmes orientés objets. Pour les grands systèmes, nous pouvons les décomposer en plusieurs sous-systèmes en utilisant soit les techniques existantes de découpage des graphes en sous-graphes connexes, soit en traitant les composants du logiciel un à la fois indépendamment des autres composants.

iv. **Microarchitectures stables :** Il est possible de chercher les microarchitectures stables qui se trouvent à l'intérieur du tunnel et dont leur nombre d'occurrences augmente toujours entre les différentes versions. Donc, nous pouvons utiliser les résultats trouvées par notre algorithme SGFinder concernant la progression des microarchitectures et leurs présences dans les tunnels pour atteindre cet objectif.

v. **Analyser les microarchitectures :** Un autre domaine de recherche possible serait l'analyse qualitative qui consiste à documenter les microarchitectures intéressantes pour le développement des logiciels. Ces microarchitectures donc, peuvent exploiter par les concepteurs et les architectes logiciels pour créer le diagramme des classes d'un système orienté objet. Cette analyse exige une compréhension approfondie de l'historique du projet, de l'évolution du développement du logiciel, ainsi que de l'interaction avec les différents groupes de développement.

RÉFÉRENCES

ALBIN-AMIOT, H. et GUÉHÉNEUC, Y.-G. (2001). Meta-modeling design patterns : Application to pattern detection and code synthesis. P. van den Broek, P. Hruby, M. Saeki, G. Sunyé et B. Tekinerdogan, éditeurs, *Proceedings of the* 1^{st} *ECOOP workshop on Automating Object-Oriented Software Development Methods*. Centre for Telematics and Information Technology, University of Twente.

AUDIBERT, L. (2009). *UML 2 : De l'apprentissage à la pratique*. Ellipses Marketing.

BATAGELJ, V. et MRVAR, A. (2003). Pajek - analysis and visualization of large networks. *Graph Drawing Software*. Springer, 77–103.

BROWN, W. J., MALVEAU, R. C., BROWN, W. H., MCCORMICK III, H. W. et MOWBRAY, T. J. (1998). *Anti Patterns : Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1^{st} édition.

CHEN, J., HSU, W., LEE, M. L. et NG, S.-K. (2006). Nemofinder : dissecting genome-wide protein-protein interactions with meso-scale network motifs. *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, New York, NY, USA, KDD '06, 106–115.

COFFEY, N. (2011). A strong 64-bit hash function in java (ctd). `http://www.javamex.com/tutorials/collections/strong_hash_code_implementation.shtml`.

DHAMBRI., K., SAHRAOUI., H., et POULIN, P. (2008). Visual detection of design anomalies. *In Proceedings of the 12th European Conference on Software Maintenance and Reengineering, Tampere, Finland*. 279—283.

EADDY, M., ZIMMERMANN, T., SHERWOOD, K. D., GARG, V., MURPHY, G. C., NAGAPPAN, N. et AHO, A. V. (2008). Do crosscutting concerns cause defects ? *IEEE Transaction on Software Engineering*, 34, 497–515.

EASTWOOD, A. (1993). Firm fires shots at legacy systems. Computing Canada, vol. 19 (2), p. 17.

ECMA (2007). *ECMAScript Standard - ECMA-262 v3*. ISO/IEC 16262.

ERLIKH, L. (2000). Leveraging legacy system dollars for e-business. *(IEEE) IT Pro*, pp. 17–23.

GAMMA, E., HELM, R., JOHNSON, R. et VLISSIDES, J. (1994). *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1^{st} édition.

GAREY, M. R. et JOHNSON, D. S. (1990). *Computers and Intractability ; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

GUÉHÉNEUC, Y.-G. (2004). A reverse engineering tool for precise class diagrams. J. Singer et H. Lutfiyya, éditeurs, *Proceedings of the* 14^{th} *IBM Centers for Advanced Studies Conference (CASCON)*. ACM Press, 28–41.

GUÉHÉNEUC, Y.-G. et ANTONIOL, G. (2008). DeMIMA : A multi-layered framework for design pattern identification. *Transactions on Software Engineering (TSE)*, 34, 667–684.

HIMSOLT, M. (1997). Gml : A portable graph file format.

HUFF, S. (1990). Information systems maintenance. The Business Quarterly, vol. 55, 30–32.

JAHNKE, J. H. et ZÜNDORF, A. (1997). Rewriting poor design patterns by good design patterns. S. Demeyer et H. C. Gall, éditeurs, *Proceedings the* 1^{st} *ESEC/FSE workshop on Object-Oriented Reengineering*. Distributed Systems Group, Technical University of Vienna.

KELLER, R. K., SCHAUER, R., ROBITAILLE, S. et PAGÉ, P. (1999). Pattern-based reverse-engineering of design components. D. Garlan et J. Kramer, éditeurs, *Proceedings of the* 21^{st} *International Conference on Software Engineering*. ACM Press, 226–235.

KPODJEDO, S., RICCA, F., GALINIER, P. et ANTONIOL, G. (2009). Recovering the evolution stable part using an ecgm algorithm : Is there a tunnel in mozilla ? *CSMR*. 179–188.

KRÄMER, C. et PRECHELT, L. (1996). Design recovery by automated search for structural design patterns in object-oriented software. L. M. Wills et I. Baxter, éditeurs, *Proceedings of the* 3^{rd} *Working Conference on Reverse Engineering*. IEEE Computer Society Press, 208–215.

KREHER, D. L. et STINSON, D. R. (1998). *Combinatorial algorithms : generation, enumeration, and search*. CRC.

KULLBACH, B. et WINTER, A. (1999). Querying as an enabling technology in software reengineering. P. Nesi et C. Verhoef, éditeurs, *Proceedings of the* 3^{rd} *Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, 42–50.

KURAMOCHI, M. et KARYPIS, G. (2004). An efficient algorithm for discovering frequent subgraphs. *IEEE Trans. on Knowl. and Data Eng.*, 16, 1038–1051.

LANZA., M. et MARINESCU., R. (2006). *Object-Oriented Metrics in Practice*. Springer.

LIENTZ, B. et SWANSON, E. (1981). Problems in application software maintenance. Communications of the ACM, vol. 24 (11), 763–769.

MARINESCU, R. (2004). Detection strategies : Metrics-based rules for detecting design flaws. *In Proc. IEEE International Conference on Software Maintenance*.

MASLOV, S. et SNEPPEN, K. (2002). Specificity and stability in topology of protein networks. *Science*. vol. 296 (5569), 910–913.

MCKAY, B. D. (1981). Practical graph isomorphism. Congressus Numerantium, vol. 30, 45–87.

MCKAY, B. D. (2009). *nauty User's Guide (Version 2.4)*. Department of Computer Science, Australian National University Canberra ACT 0200, Australia.

MCKAY, J. (1984). Maintenance as a function of design. Proceedings of the AFIPS National Computer Conference, 187–193.

MILO, R., SHEN-ORR, S., ITZKOVITZ, S., KASHTAN, N., CHKLOVSKII, D. et ALON, U. (2002). Network motifs : simple building blocks of complex networks. *Science*. vol. 298, 824–827.

MOAD, J. (1990). Maintaining the competitive edge. Datamation, 61–62, 64, 66.

NIERE, J., SCHÄFER, W., WADSACK, J. P., WENDEHALS, L. et WELSH, J. (2002). Towards pattern-based design recovery. M. Young et J. Magee, éditeurs, *Proceedings of the* 24^{th} *International Conference on Software Engineering*. ACM Press, 338–348.

PETTERSSON, N. et LÖWE, W. (2006). Efficient and accurate software pattern detection. P. Jalote, éditeur, *Proceedings of the* 13^{th} *Asia Pacific Software Engineering Conference*. IEEE Computer Society Press, 317–326.

PORT, O. (1988). The software trap automate or else. Business Week, vol. 3051 (9), 142–154.

RASCHE, F. et WERNICKE, S. (2006). *fast network motif detection*.

RAZAGHI, Z. et KASHANI, M. (2009). Kavosh : a new algorithm for finding network motifs. *Bioinformatics*. vol. 10, 0–0.

RICH, C. et WATERS, R. C. (1990). *The Programmer's Apprentice*. ACM Press Frontier Series and Addison-Wesley, New York, NY, USA, première édition.

RUMBAUGH, J. R., BLAHA, M. R., LORENSEN, W., EDDY, F. et PREMERLANI, W. (1990). *Object-Oriented Modeling and Design*. Prentice-Hall.

SCHREIBER, F. et SCHWÖBBERMEYER, H. (2005). Mavisto : a tool for the exploration of network motifs. *Bioinformatics*. vol. 21, 3572–3574.

SEBASTIAN, W. (2006). A tool for fast network motif detection. *Bioinformatics*. vol. 22, 1152–1153.

SEEMANN, J. et VON GUDENBERG, J. W. (1998). Pattern-based design recovery of software. B. Scherlis, éditeur, *Proceedings of* 5^{th} *international symposium on Foundations of Software Engineering*. ACM Press, 10–16.

TONELLA, P. et ANTONIOL, G. (2001). Inference of object oriented design patterns. *Journal of Software Maintenance - Research and Practice*, 13, 309–330.

TRAVASSOS, G. H. (1999). Detecting defects in object oriented designs : Using reading techniques to increase software quality. *In Conference on Object-oriented Programming Systems, Languages Applications (OOPSLA*. 47–56.

TSANTALIS, N., CHATZIGEORGIOU, A., STEPHANIDES, G. et HALKIDIS, S. (2006). Design pattern detection using similarity scoring. *Transactions on Software Engineering*, 32.

ANNEXE A

Concepts de fréquences

Définitions

La fréquence d'un sous-graphe G' dans un graphe G est définie par le nombre des sous-graphes identiques à G' dans G. Il y a trois concepts raisonnables pour déterminer la fréquence d'un sous-graphe. Ces concepts sont basés sur des restrictions de partage des éléments du graphe G (sommets et arcs) (voir la figure A.1). Dans le premier concept C_1, il y a aucune restriction c'est-à-dire les éléments du graphe peuvent être utilisés plusieurs fois. Ce concept C_1 est utilisé pour chercher le nombre total des occurrences des sous-graphes. Le deuxième concept C_2 permet de partager les sommets, mais pas les arcs. Donc, les sous-graphes identiques ne peuvent pas partager les arcs entre elles. Nous avons utilisé ce concept C_2 pour définir des zones. Dans le troisième concept C_3, les sous-graphes identiques ne partage ni les sommets ni les arcs.

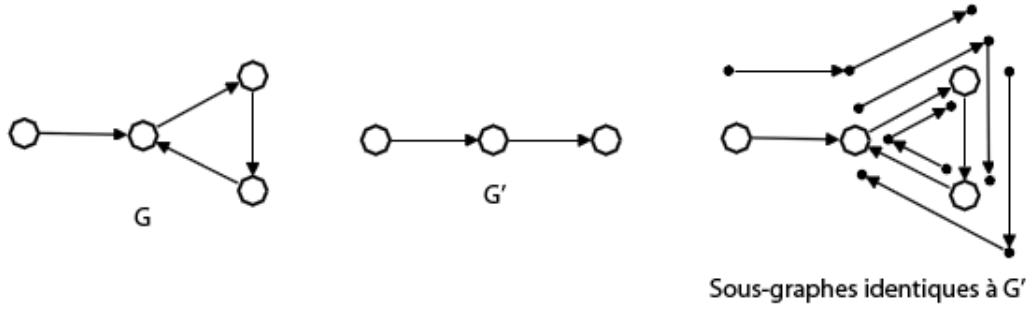


Figure A.1 Concepts de fréquences. Pour le concept C_1, il y a quatre occurrences, pour le concept C_2, il y a deux occurrences, et pour le concept C_3, il y a une seule occurrence.

ANNEXE B

Algorithme nauty

Utilisation de l'algorithme nauty

Pour chercher la matrice d'adjacence représentant la forme canonique, nous avons utilisé la fonction nauty suivante :

$$nauty(g, lab, ptn, active, orbits, options, stats, workspace, worksize, m, n, canong)$$

Voici la description des paramètres importants de la fonction nauty.
- g : Le sous-graphe duquel nous voulons chercher la forme canonique.
- lab : Un vecteur contenant les indices des sommets de g.
- ptn : Un vecteur indiquant les groupes des sommets ayant la même étiquette.
- $options$: Définit certaines caractéristiques comme le type du sous-graphe à traiter s'il est orienté ($options.digraph = TRUE$) ou non, et s'il faut chercher la forme canonique ($options.getcanon = TRUE$) ou non, etc.
- $canong$: Retourne la matrice d'adjacence qui représente la forme canonique.

L'idée principale de l'algorithme nauty est de chercher la forme canonique d'un graphe ayant des sommets coloriés (i.e. ayant des étiquètes sur les sommets). Donc, pour chercher la forme canonique d'un sous-graphe G' étiqueté (i.e. ayant des étiquettes sur les arcs), nous devons transformer le sous-graphe G' en un autre sous-graphe G'' dont l'ensemble des sommets est composé par tous les sommets de G' plus des sommets additionnels remplaçant les arcs ayant des étiquettes différents de "1" (voir McKay, 2009).

Comme l'algorithme nauty traite l'isomorphisme des graphes coloriés, chaque groupe de sommets est représenté par une seule couleur tel que :

i. Le premier groupe contient tous les sommets de G'.

ii. Le deuxième groupe contient les nouveaux sommets ajoutés pour remplacer les arcs ayant des étiquettes identiques.

iii. Et ainsi de suite.

De plus, le nombre d'arcs du nouveau sous-graphe G'' est égale au nombre d'arcs de G' plus le nombre des sommets additionnels.

Exemple : Considérons l'exemple illustré par la figure B.1. Cet exemple montre bien comment élargir le sous-graphe G' induit par l'ensemble des sommets $\{1, 2, 6, 7\}$ (tous les sommets sont coloriés par une seule couleur). Premièrement, nous constatons que certaines étiquettes associées aux arcs de G' sont différents de "1". Il s'agit donc ici les trois arcs (1,2), (1,6), et (7,1). Donc, il faut élargir le sous-graphe G' pour pouvoir utiliser nauty correctement. Dans ce cas, nous procédons aux étapes suivantes :

i. En premier lieu, les arcs (1,2) et (7,1) du graphe G' ont la même étiquette. Nous devons donc les remplacer respectivement par deux nouveaux sommets A et B.

ii. En deuxième lieu, nous ajoutons aussi un autre sommet C entre les sommets de G' reliés par l'arc (1,6) qui a l'étiquette 13 .

Donc, l'ensemble des sommets du nouveau sous-graphe G'' est défini par $\{1, 2, 6, 7, A, B, C\}$, et qui contient trois groupes de couleurs $\{1, 2, 6, 7\}$, $\{A, B\}$, et $\{C\}$.

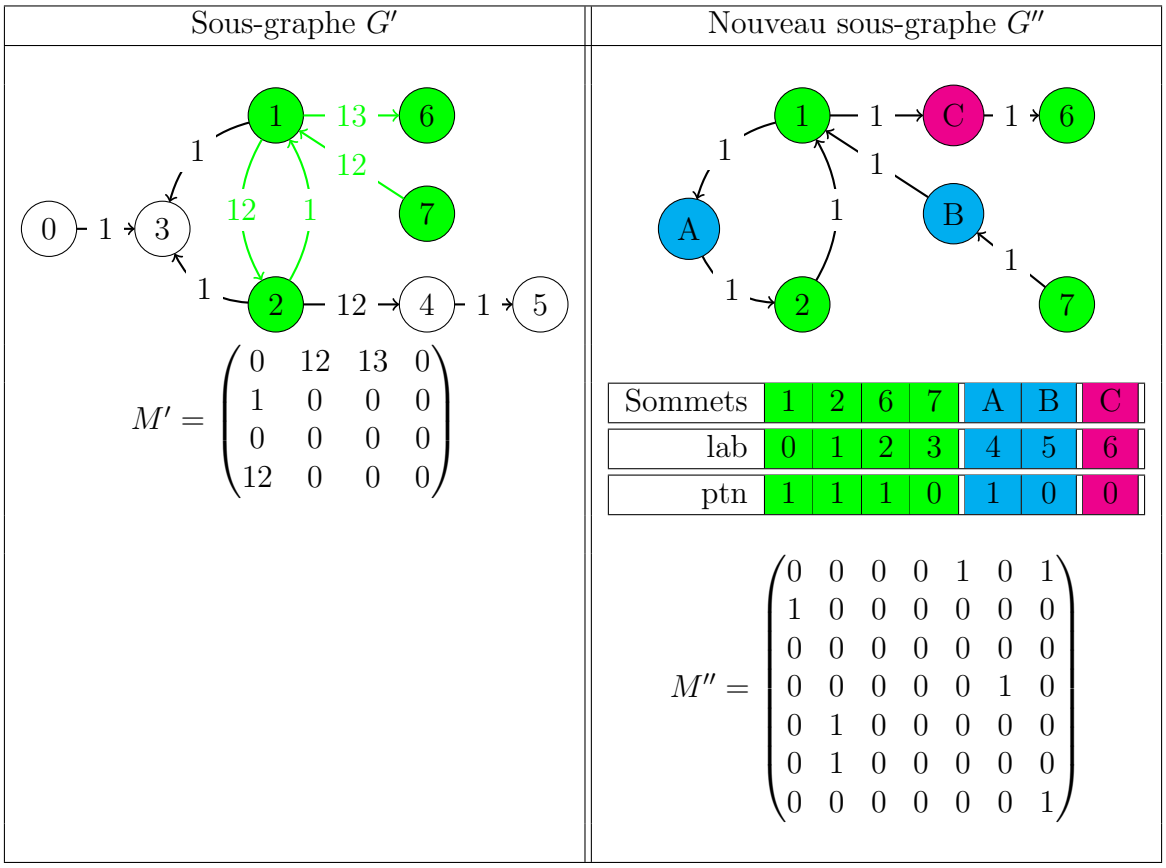


Figure B.1 Élargissement d'un sous-graphe à un autre sous-graphe

La figure B.1 montre aussi les valeurs des paramètres de la fonction nauty (B) ($g = M''$, *lab* et *ptn*) pour chercher la forme canonique de G'.

En résumé, la matrice d'adjacence M'' de chaque sous-graphe G'' doit être fournie à la fonction nauty pour obtenir la matrice d'adjacence M''_{new} qui représente la forme canonique $cl(M''_{new})$ de G''. Pour générer la forme canonique $cl(G')$ du sous-graphe G', nous procédons à la relation inverse c'est-à-dire nous utilisons la forme canonique de G'', puis nous remplaçons les étiquettes des sommets additionnelles par les étiquettes des arcs de G' correspondants.

ANNEXE C

Guide d'utilisation de SGFinder (Version 1.0)

Introduction

L'objectif de cette section est de fournir un guide complet permettant aux utilisateurs de pouvoir utiliser notre outil SGFinder.

Qu'est-ce l'outil SGFinder ?

SGFinder est une application console destinée à l'énumération des microarchitectures d'un système orienté objet. L'outil SGFinder est capable d'explorer des systèmes ayant jusqu'à 1716 classes et 10287 relations entre elles pour chercher des microarchitectures de taille allant jusqu'à cinq sans avoir de problème de débordement de mémoire. SGFinder est développé en C++ sous l'IDE Qt de Nokia. De plus, il est conçu pour fonctionner sur plusieurs plateformes.

Qu'offre l'outil SGFinder ?

- Il permet de recenser toutes les occurrences des microarchitectures d'un système orienté objet et compter également leurs présences dans les différentes régions (zones).
- Il est capable de regrouper les microarchitectures identiques.
- Il fournit les rôles joués par les classes.
- Il identifie les microarchitectures dans les trois côté du tunnel (intra-tunnel, extra-tunnel, et inter-tunnel).

Démarrage de l'outil SGFinder

Cette section fournit les paramètres disponibles pour utiliser l'outil SGFinder de la recherche des microarchitectures. L'utilisation de SGFinder est assez simple. Il suffit donc de passer tous les paramètres nécessaires pour l'exécuter. Le message d'aide suivant est affiché si les paramètres obligatoires n'ont pas été passés à l'outil SGFinder.

Usage : SGFinder options[inputfile...]

-h –help "Display this usage information"

-i –input graphfile "Input Graph File Name"

-o –output folder "Output Folder"

-s –size subgraphsize ”Subgraph size”
-b –start vertex ”Start vertex”
-e –end vertex ”End Vertex”
-t –tunnel classesfile ”Tunnel Classes File”

Donc, pour lancer l'outil SGFinder, vous devez taper la commande suivante :

SGFinder –i inputfile –o outputfolder –s size –b vertexn –e vertexm –t tunnelfile

Paramètres de l'outil SGFinder

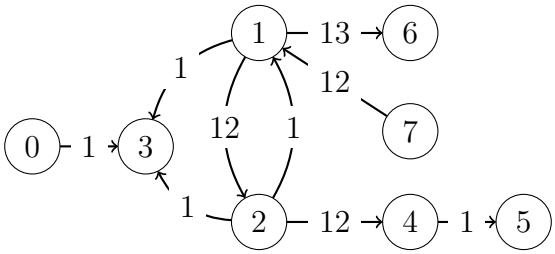


Figure C.1 Un sous sous-graphe extrait de la version 1.7R1 de l'application Rhino.

- **Graphe du système en entrée (-i inputfile) (paramètre obligatoire) :** La première étape est de choisir le fichier en entrée contenant le graphe représentant le système à analyser. Le fichier d'entrée doit contenir une ligne pour chaque arc du graphe à l'exception de la première ligne qui contient le nombre de sommets du graphe. Les lignes suivantes doivent contenir trois entiers séparés par un espace, et doit être ressemblé à la ligne suivante :

  $$int1\ int2\ int3$$

  Les deux entiers $int1$ et $int2$ représentent respectivement l'extrémité initiale et l'extrémité finale de l'arc, et $int3$ représente l'étiquette associée à cet arc. Pour le graphe de la figure C.1, le contenu du fichier en entrée est indiqué dans la figure C.
- **-s size (paramètre obligatoire) :** Le paramètre size représente la taille des microarchitectures à trouver c'est-à-dire le nombre des classes présentes dans les microarchitectures.
- **-o outputfile (paramètre optionnel) :** Après l'exécution de l'outil SGFinder, les résultats seront stockés dans le répertoire outputfolder. Si le répertoire de stockage n'est pas indiqué, l'outil SGFinder sauvegarde les résultats dans le répertoire courant. Pour plus de détails sur le format et le contenu des fichiers contenant les résultats, il faut consulter la section C.

```
8
0  3  1
1  3  1
1  2  12
1  6  13
2  3  1
2  1  1
2  4  12
4  5  1
7  1  12
```

Figure C.2 Fichier d'entrée de l'outil SGFinder basés sur le sous sous-graphe de la figure C.1.

- **-b vertexn et –e vertexm (paramètres optionnels) :** Les deux paramètres consistent à énumérer toutes les microarchitectures contenant la classe de départ numérotée par **vertexn**, puis la classe numérotée par **vertexn+1**, et ainsi de suite jusqu'à la classe numérotée par **vertexm**. Par défaut, le paramètre **vertexn=0** et le paramètre **vertexm** égale l'ordre du graphe en entrée.

La consultation des résultats

À la fin de l'exécution de l'outil SGFinder, les résultats de recherche des microarchitectures seront stockés dans quatre fichiers au format CSV.

i. **Fichier des fréquences :** Ce fichier est nommé avec l'extension ".frq". Il contient un en-tête comprenant différentes informations comme le temps d'exécution de l'outil SGFinder, le nombre total des microarchitectures trouvées par SGFinder, le nombre des microarchitectures différentes, et la taille des microarchitectures. Le reste du fichier contient des lignes sous le format suivant :

HashCode,Int1,Int2

HashCode est un code représentant la forme canonique d'une microarchitecture donnée.

Int1 est le nombre d'occurrence d'une miroarchitecture dans le système orienté objet donné en entrée à l'outil SGFinder.

Int2 représente le nombre des zones contenant cette microarchitecture.

ii. **Fichier des structures :** Ce fichier est nommé avec l'extension ".sgr". Chaque ligne de ce fichier représente une catégorie c'est-à-dire une seule microarchitecture identique.

Ce fichier est exploité par l'outil SGViewer (section D) pour visualiser graphiquement les microarchitectures. Donc, le format d'une ligne définissant une microarchitecture est la suivante :

HashCode,Int1,Int2,...,Intk

Le **HasheCode** est un entier qui représente la forme canonique d'une structure quelconque. Le reste des entiers (**Int1**, **Int2**,...,**Intk**) représente les numéros des classes qui font partie d'une microarchitecture donnée.

iii. **Fichier des rôles :** Ce fichier est nommé avec l'extension ".rol". Les lignes de ce fichier définissent les rôles joués par les classes dans les microarchitectures trouvées par l'outil SGFinder. Chaque ligne prend le format suivant :

HashCode,Int1,Int2,Int3

HashCode est un code représentant la forme canonique d'une microarchitecture donnée.

Int1 détermine le rôle joué par la classe numérotée par $Int2$. Cette valeur est toujours dans l'intervalle $[0..k-1]$ où k est la taille des microarchitectures.

In2 identifie le numéro de l'une des classes d'une microarchitecture.

Int3 est le nombre d'occurrences des classes numérotées par $Int2$ dans toutes les microarchitectures identiques, et qui sont identifiées par le code unique $HashCode$.

iv. **Fichier des tunnels :** Ce fichier est nommé avec l'extension ".tnl". Les lignes de ce fichier identifient les microarchitectures dans les trois côtés du tunnel (intra-tunnel, extra-tunnel, et inter-tunnel). Elles sont sous le format suivant :

HashCode,Int1,Int2,Int3

HashCode est la forme canonique d'une microarchitecture donnée.

Int1 est le nombre des microarchitectures qui sont à l'intérieur du tunnel.

In2 est le nombre des microarchitectures qui ont des classes à l'intérieur et d'autres classes à l'extérieur du tunnel.

Int3 est le nombre des microarchitectures qui sont à l'extérieur du tunnel.

License

- SGFinder is a free tool, it may be used, modified and distributed under the same terms as Qt Nokia. See the file COPYING in the Qt distribution. Note that SGFinder uses the nauty program version 2.4 by Brendan McKay (voir McKay, 2009) ; hence nauty's license restrictions also apply to your use of SGFinder.
- Redistributions in binary form must reproduce this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

ABSOLUTELY NO GUARANTEES OR WARRANTIES ARE MADE CONCERNING THE SUITABILITY, CORRECTNESS, OR ANY OTHER ASPECT OF THE DISTRIBUTED FILES. ANY USE IS AT YOUR OWN RISK (ANY WARRANTY YOU MIGHT DREAM OF).

ANNEXE D

Guide d'utilisation de SGViewer

Qu'est-ce l'outil SGViewer ?

SGViewer est une application Windows permettant de visualiser en mode graphique les microarchitectures trouvées par l'outil SGFinder vu précédemment dans la section C. L'outil SGViewer utilise la librairie graphviz [1] pour engendrer les sous-graphes représentants les microarchitectures existant dans le diagramme des classes des systèmes OO. Il est entièrement développé en C++ sous l'IDE Qt de Nokia. De plus, il peut fonctionner sur plusieurs plateformes.

Démarrage rapide

Cette section décrit les différentes sections de l'interface graphique de l'outil SGViewer sans rentrer trop dans les détails. La figure D.1 montre la fenêtre principale de l'outil SGViewer.

Section 1 : La première étape consiste à choisir le fichier contenant le graphe (fichier avec l'extension ".grp") représentant le diagramme des classes d'un système orienté objet. Les quatre fichiers indiqués dans la section C ayant les extensions ".frq", ".grp", ".sgr", et ".rol" doivent être placés dans le même répertoire.

Une fois que le fichier contenant le graphe est sélectionné, l'outil SGViewer affiche

- La taille des microarchitectures énumérées par l'outil SGFinder dans le champ "***Size***".
- La liste des microarchitectures récurrentes dans la section 3 de la figure D.1.
- Le nombre total des microarchitectures récurrentes dans le champ "***Frequents***".
- Le nombre total des occurrences des microarchitectures dans le champ "***Instances***".

Section 2 : Cette section permet de chercher une microarchitecture bien spécifique, en utilisant son "***Hash Code***" pour visualiser le sous-graphe représentant sa structure.

Section 3 : Cette partie affiche la liste des microarchitectures identiques, le nombre d'occurrences de chaque structure, et le nombre des zones contenant les microarchitectures identiques.

Section 4 : Cette section affiche le rôle joué par les classes participantes à la microarchitecture sélectionnée. Le rôle est défini par l'ordre des classes dans la matrice d'adjacence

1. http ://www.graphviz.org/

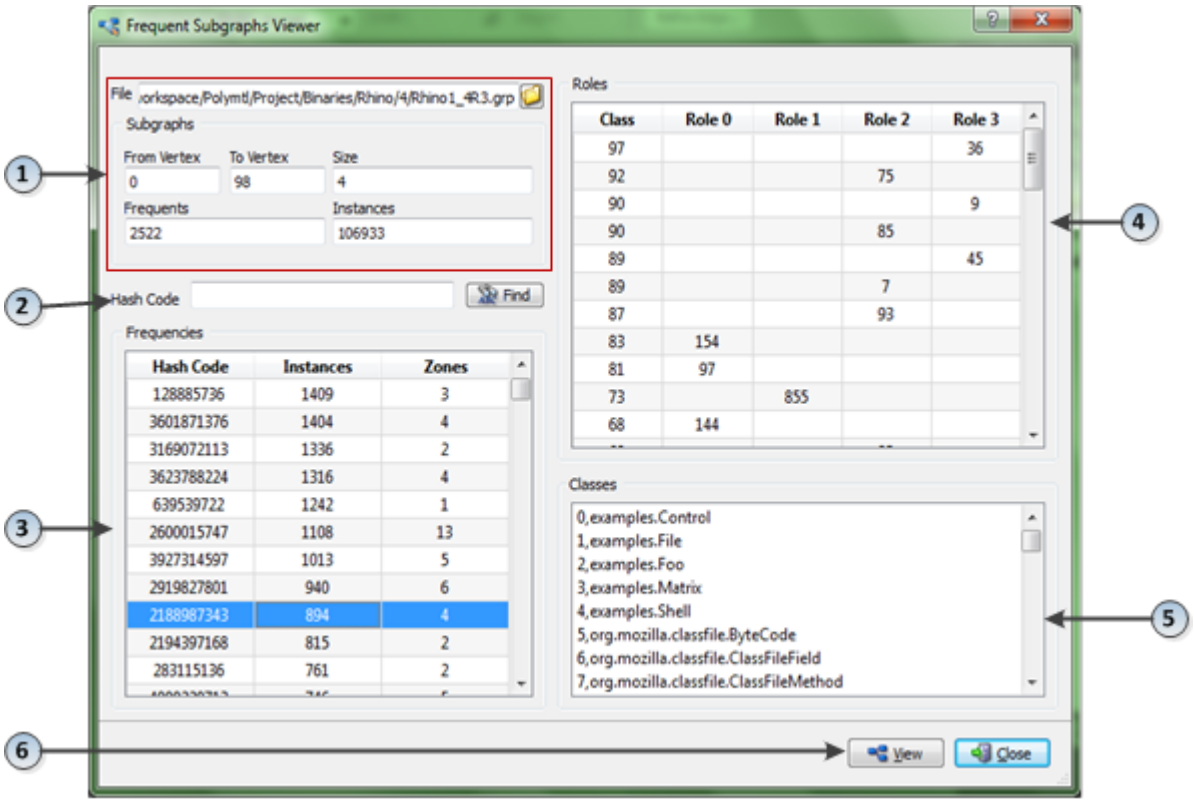


Figure D.1 Aperçu de la fenêtre principale de SGViewer

retournée par l'algorithme nauty (voir McKay, 1981). En effet, pour afficher les rôles joués par les classes, il faut sélectionner une microarchitecture de la liste des microarchitectures identiques (voir la section 3).

Section 5 : Cette section affiche l'ensemble des classes participantes aux microarchitectures identiques.

Section 6 : Le bouton "***View***" permet d'afficher la fenêtre de visualisation graphique du sous-graphe représentant la microarchitecture sélectionnée. Cette fenêtre est illustrée par la figure D.2. Elle contient les sections suivantes :

Section a : Cette section affiche les rôles joués par les classes participantes à la microarchitecture sélectionnée.

Section b : Cette section affiche la liste des classes participantes à la microarchitecture sélectionnée.

Section c : Cette section affiche le sous-graphe représentant la structure des classes et les relations entre elles. Le sous-graphe est généré à l'aide de la librairie graphviz, et stocké dans le répertoire contenant l'exécutable de l'outil SGViewer.

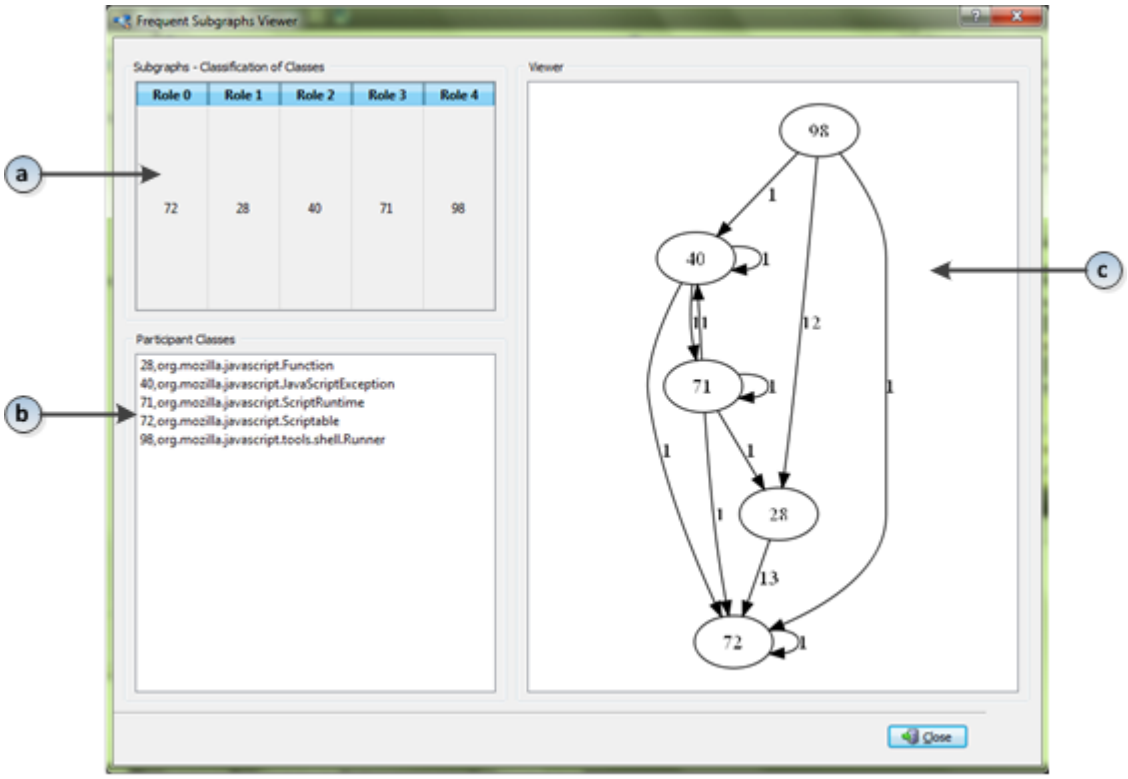


Figure D.2 Représentation graphique des microarchitectures