UNIVERSITÉ DU QUÉBEC À MONTRÉAL

VALIDATION DE DÉPENDANCES DE PROGRAMME PAR TRACES D'EXÉCUTION

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

MOHAMED AMINE HADDAJI

JANVIER 2021

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

RÉSUMÉ

Une étape importante de la ré-ingénierie consiste à identifier des regroupements d'éléments de code qui implantent une fonctionnalité relativement cohésive et fréquemment utilisée, c'est à dire, des 'services' potentiels. Pour l'identification de services, Mili et al. ont choisi de se contenter de l'analyse du code source de l'application. Pour ce faire, ils ont besoin de construire un graphe de dépendances du programme. La construction du graphe d'appels est lui-même problématique car les applications légataires Java utilisent un certain nombre de mécanismes (e.g. introspection, utilisation de fichiers de configurations) et de technologies (divers services offerts par les « conteneurs » ou serveurs d'applications J2EE) qui cachent certaines relations d'appel entre différentes parties du code de l'application, ce qui peut être en échec l'analyse statique du code, menant à des « graphes d'appels disjoints ». (Hecht *et al.*, 2018) ont développé une technique pour spécifier ces dépendances cachées sous forme de règles, et d'appliquer ces règles sur le graphe d'appel généré par l'analyse statique, pour ajouter les dépendances cachées.

Dans ce mémoire, notre objectif est de valider ces dépendances cachées (relations d'appel implicites) en s'appuyant sur des traces d'exécution de ces mêmes applications. Pour y parvenir, nous devons construire des graphes d'appels statiques et dynamiques et les comparer. Nous présentons les outils que nous avons créés pour construire des graphes d'appels. Nous avons également résolu certains problèmes afin d'avoir des graphes plus précis tels que le polymorphisme, les appels qui se trouvent dans les blocs statiques et d'initialisation, les modifications apportées par le compilateur ce qui peut rendre la comparaison plus difficile.

Mots clés : Graphe d'appels, comparaison des graphes d'appels, validation des dépendances cachées

INTRODUCTION

0.1 Modernisation d'applications

Les entreprises se dotent de systèmes d'information pour soutenir leurs processus d'affaires. Pour ce faire, elles doivent acquérir ou adapter des applications existantes sur le marché, ou alors développer des applications sur mesure. Pour toutes sortes de raisons, les entreprises choisissent souvent de développer des applications sur mesure, dont la spécificité de leurs processus, le coût d'acquisition ou d'adaptation de logiciels commerciaux, l'incompatibilité des logiciels commerciaux avec son environnement TI, etc. Or, ces applications sur mesure deviennent vieilles et obsolètes avec le temps. Comme le montre la figure 0.1, les applications ne satisfont pas toujours les besoins de l'entreprise, en raison de l'évolution de ces derniers. Par conséquent, les applications nécessiteront toujours une maintenance(Comella-Dorda *et al.*, 2000). De plus, plus ces applications vieillissent, plus elles seront difficiles et coûteuses à entretenir(Perin *et al.*, 2010). À un moment donné, on doit donc envisager d'autres options.

Face au coût et à la complexité de la maintenance des applications existantes, les entreprises ont le choix entre réécrire toute l'application ou la moderniser. La réécriture des applications à partir de zéro est peut sembler attrayante car on a l'opportunité d'implanter de non seulement utiliser les dernières technologies, mais on peut profiter de la réécriture pour optimiser les processus d'affaires sous-jacents. En pratique, les entreprises sont souvent préférer la modernisation, et ce, pour plusieurs raisons, dont : 1) le coût de la ré-écriture, 2) l'investissement que les entreprises ont déjà consenti dans la création de ces applications, et 3)

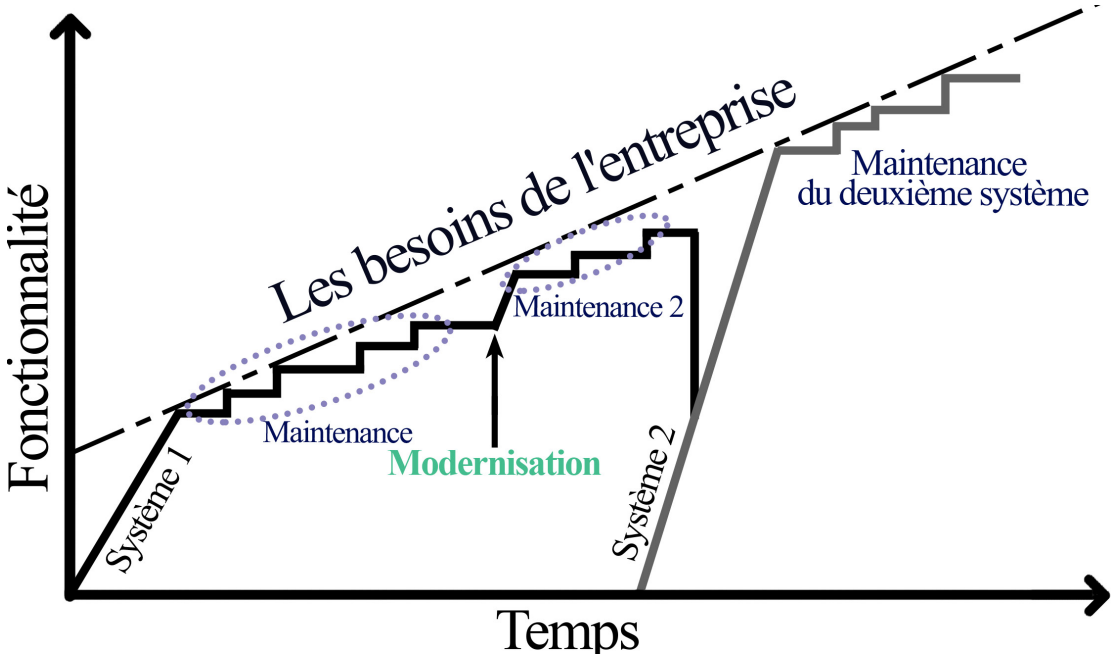


Figure 0.1: Cycle de vie des systèmes informatiques (**?**)

le fait que les applications existantes continuent d'assurer un nombre important de fonctionnalités d'affaires. On cherche alors à dépenser moins d'argent pour les mettre à jour avec le moins d'efforts et de temps possible. Autant pour des raisons de coût que de risques opérationnels, les entreprises vont opter pour la *modernisation*, en évitant autant que possible la ré-écriture. De plus, le code qui est encore présent englobe une grande partie des connaissances et des règles d'affaires accumulées au fil des ans, ce qui est souvent difficile à reproduire.

Quand on parle de modernisation, on pense souvent à des applications Cobol sur des ordinateurs centraux ("mainframe"). Or, les premières applications Java ont déjà 25 ans, qui commencent à montrer des signes de vieillesse. Par exemple, elles peuvent comporter des partie de code qui ne sont plus utilisées car remplacées par des services développés depuis, elles peuvent faire elles-mêmes appel à des services externes qui ne sont plus supportés (par ex. de vieilles bases de données),

elles peuvent comporter de multiples failles de sécurité. Elles peuvent aussi ne pas supporter la mise à l'échelle requise par le volume actuel de transactions qu'elles doivent supporter. Or il existe de nouvelles architectures de développement qui sont plus récentes, plus modulaires, et plus performantes. Parmi ces architectures, on retrouve les Architecture Orientées Services, dont l'architecture micro-services, l'architecture REST, etc. L'objectif principal de ces architectures est de séparer une application en différents services qui peuvent être utilisés et invoqués par les nombreuses autres parties de l'application ou même d'autres applications, contrairement aux applications légataires qui sont développées de manière monolithique. Les applications légataires ont un degré élevé de dépendance entre ses composants, ce qui rend l'ajout des nouvelles fonctions ou la modification de fonctions existantes coûteuse et complexe qui va prendre beaucoup de temps et qui va nécessiter l'intervention d'une personne qui a une connaissance intime du code source de l'application. Or, ces personnes là–les développeurs initiaux de l'application–ne sont pas toujours disponibles. Il importe donc de trouver une façon de moderniser ces applications qui soit peu coûteuse et qui ne nécessite pas d'expertise humaine. Ce mémoire s'inscrit dans le cadre des problématiques liées à la modernisation, et fait partie d'un projet de recherche décrit dans le prochain paragraphe.

0.2 Contexte du projet : Migration orientée-services d'applications légataires Java

Ce travail s'inscrit dans le cadre d'un projet de recherche financé par le Fonds de Recherche du Québec - Nature et technologies (FRQNT) visant la migration d'applications légataires Java vers les architectures orientées services. Le projet en question vise à offrir une solution complète au problème de migration, allant de l'identification des services dans une application légataire, jusqu'à la restructuration complète de cette application. L'approche proposée dans ce projet prend

des applications légataires en entrée et tente de les convertir en une architecture orientée services. L'identification des services passe par l'analyse du code source pour identifier les regroupements fonctionnels qui pourraient correspondre à des "services", selon une typologie fine des services. Par la suite, ces regroupements fonctionnels doivent être "encapsulés" à l'aide d'interfaces/API du type services. Finalement, le code de l'application d'origine qui fait appel à ces fonctionnalités doit être "réusiné" (*refactored*) pour passer, désormais, à travers les nouvelles "interfaces de services".

0.2.1 L'Architecture Orientée Services

Pour mieux comprendre l'objectif du projet de recherche, on doit d'abord expliquer ce qu'est une architecture orientée services (AOS). L'AOS est un ensemble de politiques et de pratiques qui produisent un modèle de conception de logiciel qui décrit comment les choses doivent être conçues(Rosen *et al.*, 2012). L'idée consiste à disposer de différents services inter-opérables, faiblement couplés, qui pourraient tous constituer des composants logiciels. On peut considérer les services comme étant des interfaces avec des méthodes qui offrent des fonctionnalités tout en respectant les règles d'affaires propres au domaine. L'objectif principal de l'utilisation des AOS est de rendre la mise a l'échelle des systèmes d'entreprise plus évolutive (MacKenzie *et al.*, 2006) . L'utilisation des services permettra aussi de réduire les coûts de maintenance. En effet, dans le cas de la maintenance d'un service existant ou de l'introduction d'un nouveau service, il n'est pas nécessaire d'arrêter complètement le système : dépendant de l' interstitiel (*middleware*) utilisé puisque, l'introduction de nouveau services n'exige aucun arrêt, et le remplacement de services peut *souvent* se faire sans arrêt (*hot-swap*). Aussi, le fait que les services utilisent un mode de communication uniforme entre eux permet aux utilisateurs d'appeler, en toute transparence, des services mis en œuvre sur des

plateformes différentes, utilisant des langages et des technologies différents(Bianco *et al.*, 2007).

0.2.2 L'identification des services

Une étape importante de la migration consiste à identifier des services potentiels. Pour ce faire, l'une des approches consiste à identifier des regroupements d'éléments de code qui implantent une fonctionnalité relativement cohésive et fréquemment utilisée. Cependant, pour pouvoir identifier les services, nous devons être capable de savoir quel type de services on cherche.

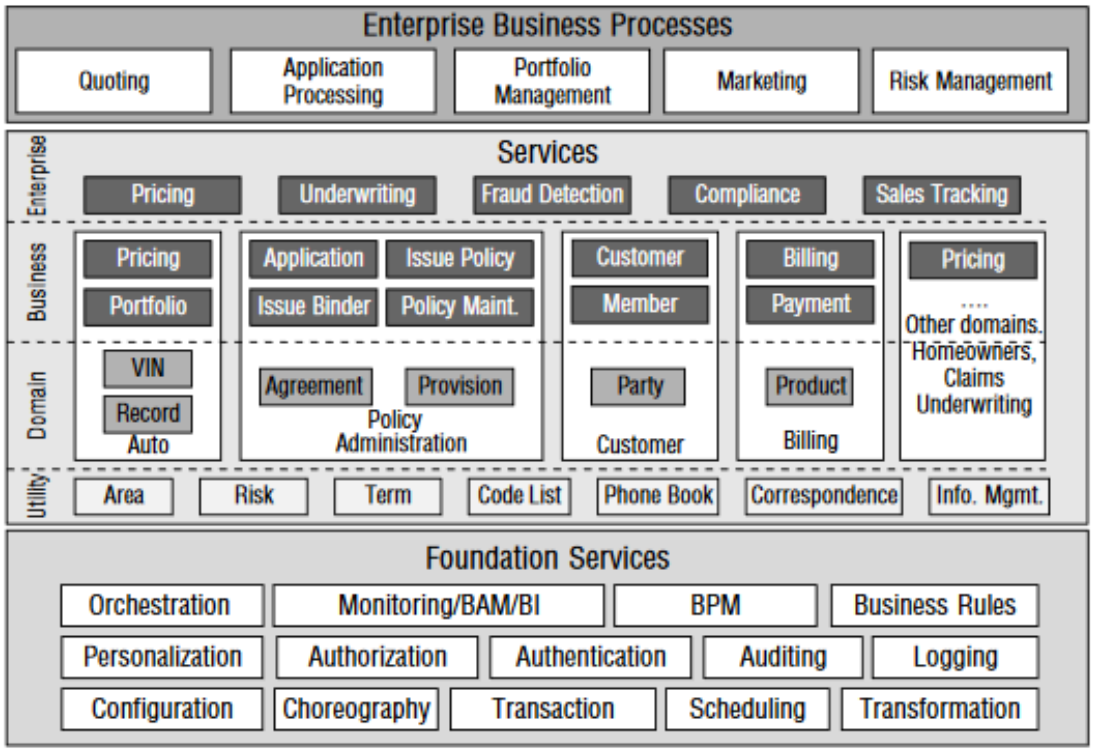


Figure 0.2: Exemple d'un système qui utilise L'AOS (Rosen *et al.*, 2012)

Différentes taxonomies de services ont été proposées dans la littérature, qui utilisent divers aspects clés de services. La taxonomie proposée par le *Open Group*

SOA Reference Architecture est déterminée principalement par leur rôle et leur comportement dans le système. [1] Ainsi, on distingue les services d'entreprise (*enterprise services*), les services d'infrastructure, etc. Ces services peuvent être décomposés en 2 groupes, en fonction de leur *portée* : 1) les services qui exécutent une **fonction métier**, reliée au domaine d'application mais ne peuvent pas être exposés en dehors du domaine (e.g. calcul d'intérêt pour une application bancaire), et 2) d'autres qui effectuent des actions génériques et qui peuvent être utilisés dans différents domaines, tels un service d'authentification, de création de nouveaux utilisateurs, d'expédition de colis, etc. La majorité des taxonomies identifient :

1. Les services qui sont spécifiques à un domaine :
   - Les services de niveau/portée 'entreprise' (*enterprise services*) : ce sont des services spécifiques au domaine d'affaires de l'entretprise (Rosen *et al.*, 2012). Les services "Pricing", "Underwriting" et "Fraud Detection", dans la Figure 0.2, sont des 'services entreprise', puisqu'ils encapsulent les politiques et les règles d'affaires de l'entreprise en question.
   - Services d'entité : Les services d'entité fournissent un accès commun basé sur les services à des entités d'affaires (*business entities*) communes. Par exemple, les services "Record", "VIN", "Agreement", et "Product" dans la figure 0.2 en sont des exemples.
2. Les services qui sont plus orientés techniques. Ça veut dire qu'ils fournissent des services qui pourrait être utilisés par plusieurs domaines d'affaires non reliés :
   - Utility service - Les services d'utilité sont les plus petits ou les moins grossiers. Ils fournissent des services de niveau inférieur qui fournissent

1. `www.opengroup.org/soa/source-book/soa_refarch/p1.htm` (Accès Juin 2020)

des fonctionnalités communes dans toute l'entreprise (par exemple, la fonctionnalité du carnet d'adresses ou la validation des numéros de pièce).

— Services d'infrastructure : ces services contiennent généralement les composants nécessaires pour déployer et faire fonctionner une application SOA. tels que les emplacements des services et comment les accéder(Rosen *et al.*, 2012).

Afin d'identifier les services dans un système, on doit identifier tous les liens reliant les différentes parties du code source. On doit voir ce que les différents morceaux de codes tentent à manipuler dans le système et étudier le comportement des différentes méthodes et classes. l'équipe voit que la bonne approche pour identifier les services doit prendre en compte le type de service spécifique que nous souhaitons trouver puis utiliser les indicateurs pertinents pour ce type (nombre d'appels à cette entité / nombre d'appels sortant de cette entité, l'utilisation des fonctions spécifiques, l' accès à la base de données,etc.).

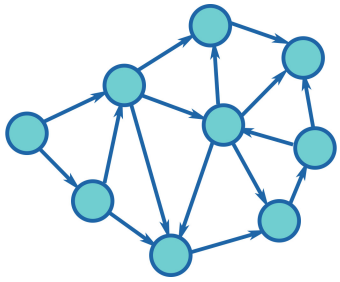


Figure 0.3: Exemple d'un graphe d'appel d'un système simple

On s'appuie donc sur l'analyse du graphe d'appel pour identifier des regroupements fonctionnels. Le graphe d'appel est généralement obtenu grâce à une analyse statique du code. Il décrit les relations entre les différentes méthodes et classes d'un programme (Grove *et al.*, 1997), où chaque nœud représente une méthode

dans le programme (les noeuds bleues dans la figure 0.3), et les flèches reliant les différents nœuds représentent les appels/dépendances (les flèches bleues dans la figure 0.3).

0.3 Les défis de l'analyse statique dans Java et J2EE

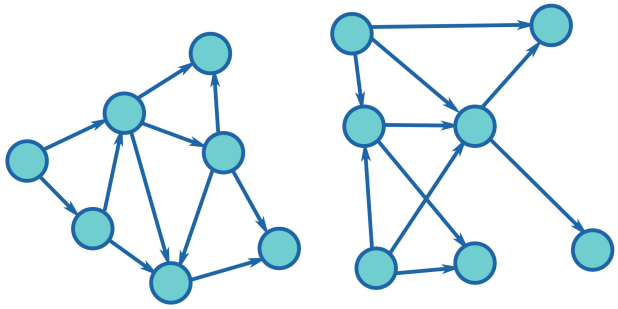


Figure 0.4: Graphe d'appel d'un système à plusieurs niveaux.

Dans le développement de logiciels, les développeurs peuvent choisir de faire une application à un couche ou une application à plusieurs niveaux en fonction des spécifications et des besoins. L'obtention du graphe d'appel d'une application simple à un niveau est facile car la plupart du code source de l'application s'exécute sur la même machine. Mais dans une application multi-tiers(exemple dans la figure 0.4, le code pour réaliser une tâche donnée peut se retrouver dans plusieurs couches, et les appels qui traversent les couches échappent à l'analyse statique car ils sont souvent véhiculés par des services d'infrastructure. Ceci complexifie d'avantage l'analyse statique d'applications Java. Comme dans le cas de JAVA RMI "Remote Method Invocation" (voir section 3.6.2) où un client ou le serveur peut invoquer une méthode située dans une autre machine (machine virtuelle Java indépendante) qui sera exécutée utilisant des interfaces (Architecture dans

la figure 0.5) (Narasimhan *et al.*, 2000).

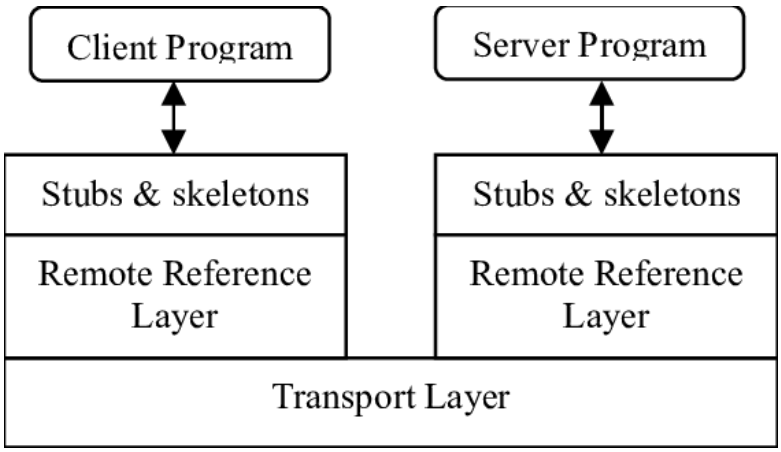


Figure 0.5: L'architecture de JAVA Remote Method Invocation(Singh *et al.*, 2011)

Dans le cadre du projet de migration, l'équipe de M. Mili a développé une technique pour l'identification et la codification de dépendances cachées dans les applications J2EE dont nous parlerons dans la section 3.5. Pour capturer ces dépendances, (Hecht *et al.*, 2018) ont fait ce qui suit :

— Étape 1 : Ils ont identifié les dépendances inhérentes aux services d'infrastructure de la plateforme EJB en étudiant les spécifications EJB2
— Étape 2 : Ils ont établi des règles pour ajouter les dépendances cachées à l'aide des patrons découverts à partir du code source.
— Étape 3 : Ils ont ajouté les dépendances manquantes nouvellement générées en utilisant leurs règles sur le code source des applications J2EE.

Normalement, dans un langage "simple", les traces dynamiques doivent être intégralement incluses dans le graphe d'appel généré statiquement, puisque, pour définition, l'analyse statique explore tous les chemins "possibles" à travers un programme, alors que une trace correspond à une exécution, ou un ensemble nécessairement incomplet de traces d'exécutions. ce qui rend une trace d'exécution

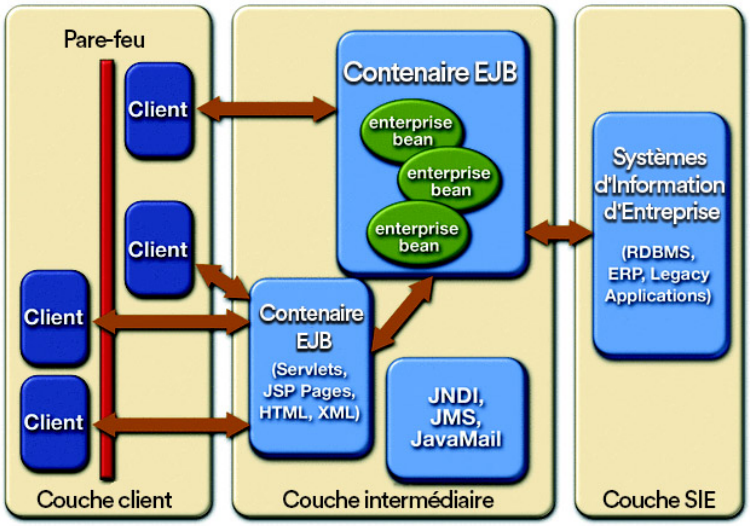


Figure 0.6: Architecture générale et composants de J2EE

incomplète est du code mort (certaines conditions if-then ne seront jamais exécutées) ou dans des cas particuliers (par exemple : une base de données en panne) le système se comporte différemment.
Dans notre cas, plusieurs raisons font que nous n'aurons pas cette inclusion : certains appels observés lors de l'exécution dynamique seront non-détectables lors de l'analyse statique. En effet, outre les dépendances cachées mentionnées précédemment, plusieurs caractéristiques de Java rendent les applications Java/J2EE difficiles à analyser statiquement.

D'abord, les applications J2EE sont construites en utilisant différents langages de programmation qui sont distribués dans les différents niveaux du système tels que Java, HTML, JSP, JavaScript (Shatnawi *et al.*, 2019),(DeSantis, 2009) et XML pour les fichiers de configuration (Sharma *et al.*, 2001).

De même, Hecht et al. (Hecht *et al.*, 2018) mentionnent que les applications J2EE utilisent des techniques de liaison dynamique telles que :

— La Réflection et l'introspection : Les développeurs ont la possibilité d'invoquer des méthodes de manière dynamique en utilisant des chaîne de caractères. (Blondeau *et al.*, 2017)

— Génération de code à l'exécution : en Java, il est possible, lors de l'exécution d'un programme, de générer du code Java, le compiler, le charger, et l'exécuter, et ce, lors de la même exécution. Dépendamment de fichiers de configuration, la JVM(Java Virtual Machine) génère les lignes de code lors du déploiement ou lors de l'exécution. (Fourtounis *et al.*, 2018)

— contrôle basé sur les données : les valeurs des données contrôlent les appels entre clients et serveurs

— Des entrées pendant l'exécution : les valeurs des données de contrôle contrôlent les appels entre les clients et les serveurs.

Généralement, Les appels réflexifs sont impossible à détecter. Car, lorsque la réflexion est utilisée, on peut seulement savoir que l'appel retourne un objet, mais il est difficile de déterminer le type de l'objet (voir chapitre 2, section 1.1.3)(Fourtounis *et al.*, 2018). Par contre, on veut s'assurer de ne pas manquer des dépendances liées aux services d'infrastructure.

Pour ce faire, concernant l'analyse statique, nous allons extraire les relations d'appel détectées par l'analyse du code source utilisant Modisco [2], et y ajouter les relations non détectées par Modisco, mais en utilisant la base de règles(Hecht *et al.*, 2018) pour ajouter les dépendances cachées. Nous allons aussi créer un outil qui extraie un graphe d'appel qui est généré statiquement au format JSON en utilisant le fichier KDM de chaque projet. De même nous allons tester notre outil sur des projets Java et J2EE. Puis, passant à l'analyse dynamique, nous allons utiliser MaintainJ [3] pour capturer des traces d'exécution à partir des projets

2. Modisco : `https://www.eclipse.org/MoDisco/` (Accès Juin 2020)

3. MaintainJ : Outil de traçage dynamique des applications Java/J2EE. `http://www.`

J2EE. Puis, nous créerons un outil qui transforme nos traces d'exécution obtenues vers un fichier JSON. En suite, en passe à l'importation de nos graphes d'appels statiques et dynamiques (les fichiers JSON) dans Neo4j et les préparer pour les analyses. Créer les requêtes Cypher [4] de traitement et de comparaison. Visualiser les graphes et les parties commune ainsi les dépendances cachées validées par nos outils.

0.4 Objectif du mémoire

Le but de ce travail en premier lieu est d'améliorer les graphes d'appels statiques générés par Modisco en incluant : (1) les appels polymorphes.(2) les appels qui se trouvent dans les blocs statiques / d'initialisation qui sont à la base un peu vague au niveau de KDM.(3) les dépendances cachées spécifiques à J2EE découvertes par (Hecht *et al.*, 2018). En dernier lieu, on va valider nos règles de construction du graphe d'appels statiques et les appels ajoutés à l'aide des traces dynamiques. Où nous se basons sur la comparaison entre les graphes d'appels dynamiques et statiques

0.5 Plan du mémoire

Le reste de ce rapport est structuré comme suit :
Le chapitre 2 parlera de l'état de l'art où nous discuterons des résultats obtenus par les autres chercheurs et comment cela motive notre approche de modélisation de notre graphe d'appels et comment nous capturons les données. Dans le chapitre

---

`maintainj.com/` (Accès Juin 2020)

4. Cypher : un langage de requête de graphes `https://neo4j.com/developer/cypher-query-language/` (Accès Juin 2020)

3, Nous décrivons la méthodologie générale. Le chapitre 4 expose comment nous avons ajouté des dépendances non détectées et généré nos graphes d'appels à l'aide de MoDisco et nos propres outils développés pour l'analyse statique. Ensuite, chapitre 4 va se concerner de l'analyse dynamique. À la suite dans le chapitre 6, nous nous révélerons les et examinerons nos résultats. Et finalement on va conclure ce nous avons fait dans le dernier chapitre.

CHAPITRE I

TECHNIQUES D'ANALYSE DE COMPORTEMENT DE PROGRAMMES

Un logiciel est une structure complexe composée de données et de fonctions qui s'appellent (se composent) pour produire des données en sortie, en fonction de données fournies en entrée.

Cette structure est conçue par les concepteurs et développeurs initiaux du logiciel, et doit être comprise par toutes les personnes chargées du test et de la maintenance du logiciel en question, sous la forme de questions que l'on est amené à se poser sur le logiciel du genre :

- Quelle fonction appelle quelle fonction avec quels paramètres.
- Sous quelles condition le logiciel soit le chemin d'exécution X plutôt que Y.
- Quel est l'impact de la modification de la fonction f(...) sur les autres parties du programme.
- Quelle partie du programme dépend de l'infrastructure.
- Quelle partie du logiciel consomme le plus de ressources (mémoire ou CPU), etc.

Pour répondre à ces questions, nous devons "analyser le comportement du programme". Pour cela, il y a deux grandes stratégies.

La première est l'analyse statique du code, où nous nous basons sur le code source

d'un programme pour en savoir plus sur les exécutions du code et les liens entre les méthodes. La deuxième stratégie consiste à exécuter le logiciel de sorte à générer une "trace" de l'exécution moyennant une instrumentation préalable du code (source ou exécutable). Chacune de ces deux techniques a des forces et des faiblesses, et donne de meilleurs résultats pour l'un ou l'autre des questions mentionnées ci-haut.

Généralement parlant, l'analyse statique de code produit un graphe qui s'appelle graphes de dépendance de programme où les noeuds représentent des artefacts du programme/logiciel, et les liens représentent différents types de relations entre les artefacts. Des exemples de telles structures incluent :

Graphe de flot des données :

Ce type de graphe représente et se concentre sur les dépendances des données dans un programme. Une dépendance existe si une instruction a besoin du résultat d'une autre instruction pour s'exécuter(Ghosh *et al.*, 2012). Les nœuds représentent des instructions/lignes de code et les relations représentent les dépendances. Par exemple, supposons qu'on a les deux lignes de code suivantes :

```
Ligne x   :  l1 = 2;
Ligne x+1:  l2 = l1+3;
```

On va avoir 2 nœuds à ce niveau représentant la ligne (x) et (x+1). Afin d'exécuter l2 = l1+3, l'instruction l1 = 2 doit être exécuté avant. Donc (x+1) dépend de (x). Dans ce cas, on va avoir un lien sous la forme d'une flèche allant de x à x+1.
Ces graphes peuvent être utilisés pour détecter les failles de sécurité. (Wüchner *et al.*, 2015) (Wu *et al.*, 2016) et peuvent aider à améliorer la lisibilité de certaines architectures complexes d'apprentissage automatique (Wongsuphasawat *et al.*,

2018)

Graphe de flot de contrôle :

Les graphe de flot de contrôle est une autre manière de représenter les dépendances dans un programme. Ils représentent le chemin d'exécution. La principale chose qui contrôle le chemin et la direction de ces graphe sont les structures de contrôle comme les boucles ou les conditions (Shivers, 1991). Par exemple, On peut voir,dans la figure 1.1, la partie du graphe de flot de contrôle qui correspond à les lignes de code ci-dessous :

```
Ligne 1 :  x=1;
Ligne 2 :  IF(x==1)
Ligne 3 :    *instruction1*;
Ligne 4 :  ELSE *instruction2*;
```

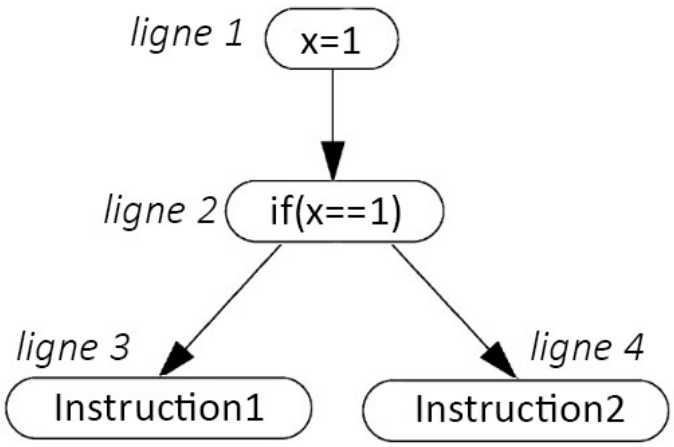


Figure 1.1: Exemple d'un graphe de flot de contrôle

Les graphes aident à calculer la complexité du programme, à produire les ensembles des changements avec l'analyse d'impact. Ils sont également utilisés pour

faire le débogage via une technique appelée découpage "Slicing" et aident aussi à mieux comprendre le comportement des programmes (Krinke et Breu, 2004).

Graphe d'appels :

Dans le cas des graphes d'appels, un nœud peut représenter des méthodes écrites par l'utilisateur et peut également représenter des méthodes externes comme des méthodes de bibliothèque et de système(Kinable et Kostakis, 2011). Chacun de ces nœuds (méthode) est la source de 0 ou plus relation sortant vers d'autres nœuds (**?**). Ces relations montrent les invocateurs possibles de ce nœud. Donc chaque nœud aussi a au moins 1 ou plus appel entrant (sauf la méthode de départ qui a 0). Les graphes d'appels peuvent être trouvés d'une manière statique ou dynamique, la structure ne change pas.

Cette structure de graphe peut aider à tester les logiciels et à réduire la taille des programmes car elle peut mettre en évidence les parties de code qui ne sont pas invoquées et ne le seront jamais.(Bhat et Singh, 2012)

1.1 Techniques et enjeux d'analyse statique de code : cas des graphes d'appel

1.1.1 Exemple

Le graphe d'appel dans la figure 1.3 du petit programme ci-dessus 1.2 montrera les nœuds qui correspond aux méthodes écrites par l'utilisateur sans tenir compte des autres de librairie due à leur grand nombre.

```
public class Exemple {
 public static void main(String[] args){
        int a=Integer.parseInt(args[0]);
        int b=Integer.parseInt(args[1]);
        methode(a,b);
        }
 void methode(int a,int b) {
        int resultat=somme(a,b);
        imprime(resultat);
        }
int somme(int a, int b) {
        imprime(a);
        imprime(b);
        return a+b ;
        }
void imprime(int resultat) {
        System.out.println(resultat);
        }
}
```

Figure 1.2: Un petit programme Java

1.1.2 Enjeu 1 : Application-only call graph

Nous allons d'abord présenter le problème, et ensuite montrer un exemple de solution.

Problème

L'écrasante majorité des programmes dépendent du code de librairies pour s'exécuter. Ces librairies peuvent provenir de kits de développement (comme le JDK) ou d'autres projets, utilisés comme des librairies externes. On peut alors dire que la dépendance de l'application avec des librairies est omniprésente(Reif *et al.*, 2016).

Le problème que posent ces librairies est qu'un graphe d'appels complet va nécessairement couvrir des méthodes et des appels qui font partie de ces librairies. Ce faisant, les graphes d'appel deviennent plus compliqués. Zhang et Ryder ont essayé de construire un graphe d'appel d'une application JAVA "Compress" qui compresse ou décompresse des fichiers (Zhang et Ryder, 2006). Sur les 3468 ap-

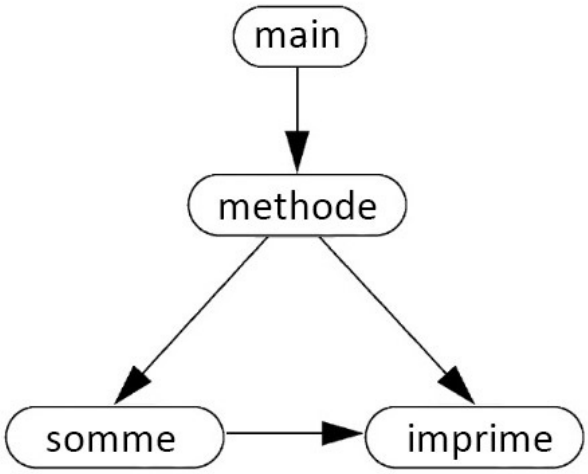


Figure 1.3: Exemple d'un graphe d'appel du programme figure 1.2

pels captures, seuls 60 d'entre eux impliquent des méthodes de Compress ; tous les autres 3408 appels ont des appels internes aux librairies. Même si nous voulons les inclure dans notre graphe d'appels pour les analyser, nous ne pouvons pas toujours trouver le code source, et ce, pour de nombreuses raisons(Law et Rothermel, 2003).

Généralement, lorsque les développeurs importent du code externe, ils disposent uniquement des fichiers de librairies tierces précompilés, comme par exemple des fichiers .jar en java. Le code des librairies importées peut être invoqué tel quel, ou implémenté–comme lorsque le programmeur doit implanter une interface fournie par la librairie–ou hérité, ou encore, redéfini. Ce ne sont pas tous les environnements de développement existants qui permettent l'analyse de bytecode Java précompilés, et on ne peut pas accéder au code source s'ils ne sont pas open source. Dans le cas de l'utilisation des APIs, il est difficile de connaître la chaîne des exécutions qui se produisent dans les composants qui agissent en tant que services (Sugawara et Yamamoto, 2013).

```
public class MainClass {
 public static void main(String[] args){
        ...
        MyHashMap map = new MyHashMap();
        System.out.println(myMap);
        ...
        }
}
```

```
public class MyHashMap extends HashMap {
    public MyHashMap() {
    }

    public String toString() {
        ...
    }
}
```

Figure 1.4: Un petit programme Java

Ali et Lhotak mentionnent que même si les programmes ont de grandes dépendances avec les appels de librairie et que cela complique les études sur les graphes d'appels, ils ne devraient pas être ignorés lors de l'analyse du comportement du logiciel (Ali et Lhoták, 2012). Pourtant, la plupart des outils et méthodes d'analyse ont tendance à ignorer complètement les appels d'application allant vers les méthodes des librairies. Par conséquent, nous nous retrouverons avec un graphe d'appel déconnecté, car les invocations de la librairie vers le code de l'application ne peuvent pas être retracés.

Enjeu 1 : exemple de solution (Ali et Lhoták, 2012)

Ali et Lhotak ont proposé un algorithme qui construit un graphe d'appel "connecté" qui retrace les appels de la librairie vers le code de l'application, mais sans montrer tous les appels internes à la librairie (Ali et Lhoták, 2012). Nous allons illustrer l'algorithme sur l'exemple du programme illustré dans la figure 1.4. Le coté gauche de la figure 1.5 montre le graphe d'appel complet, y compris les appels internes à la librairie. Le coté droite montre le graphe d'appel "réduit" produit par l'algorithme de Ali et Lhotak, qui a été intégré dans leur outil CGC. Comme on peut

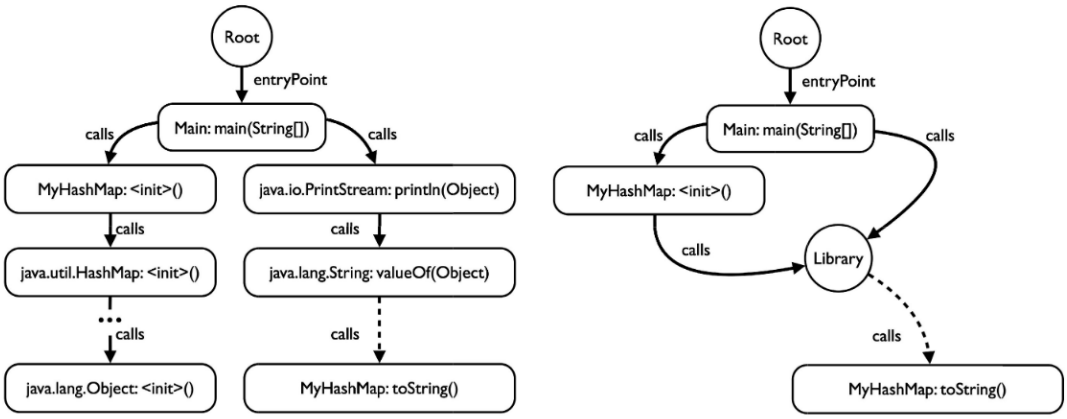


Figure 1.5: Gauche : exemple de graphe d'appel complet. Droite : le graphe d'appel du même exemple en utilisant CGC 1.5

le constater, le graphe de droite ne contient pas les appels internes à la librairie. Par contre, il ne manque pas les appels de la librairie vers le code de l'application. Dans ce graphe d'appel, l'entière librairie est représentée par un seul noeud.

L'hypothèse de compilation séparée : Leur algorithme de construction est basé sur des hypothèses qu'ils ont appelé "l'hypothèse de compilation séparée". Vu que le code de librairie a déjà été compilé sans lien avec le code de l'application, ça minimise beaucoup le nombre des cas où les méthodes de librairies peuvent appeler des méthodes d'application ou instancier des classes. Par exemple, puisque le code des librairies a déjà été compilé séparément, on peut dire que :

— Une classe d'application ne peut pas être héritée ou implémentée par une classe de librairie.

— Une nouvelle allocation dans le code de la librairie ne peut pas instancier un objet de type application. La seule exception dans JAVA est le cas où les développeurs connaissent le nom d'une classe d'application, auquel cas ils peuvent l'instancier en utilisant des méthodes de librairie qui prendraient le nom de la classe comme argument (méthodes du paquetage d'introspection

de Java).

Pour surmonter le fait qu'une méthode de librairie puisse instancier une classe d'application, leur algorithme surestime les classes d'application que les méthodes de bibliothèque peuvent appeler (pour l'instancier) en : 1) observant les objets passés et retournés de et vers les méthodes de bibliothèque, 2) en vérifiant les objets qui se trouvent dans l'application qui peuvent être lus par le code de la librairie, et 3) en vérifiant si le type de l'objet est un sous-type de `java.lang.Throwable` et qu'il correspond à une exception interceptée par des méthodes de librairie. Ces objets sont ensuite placés dans un ensemble appelé `LibraryPointsTo`.

La vérification des appels : Afin de confirmer qu'un lien entre une méthode de librairie et une méthode d'application peut exister. Ils ont vu que un appel potentiel devrait satisfaire l'une de ces conditions :

- Une méthode de librairie peut appeler une méthode d'application `m` d'une classe d'application `C` si et seulement si la méthode `m` n'est pas statique et redéfinit une méthode d'une classe de librairie et la variable de retour dans le site d'appel doit pointer vers un objet de classe `C` ou une sous-classe de `C`. Cela veut dire que l'objet de type `C` ou sous-classe de `C` doit être dans l'ensemble `LibraryPointsTo`.
- Une méthode de librairie peut lire ou modifier une variable `f` d'un objet `o` d'une classe `C` déclarée dans le code de d'application si :
  - La variable `f` est déclarée dans une classe ou sous-classe de librairie.
  - L'objet `o` doit être inclus dans l'ensemble `LibraryPointsTo`.
  - Dans le cas de modification de la variable, l'objet dans cette variable doit être pointé par une variable de librairie.
- Une méthode de librairie peut accéder à n'importe quel objet via son index. Par conséquent, ils peuvent accéder à des objets d'application invoqués par eux. Les objets accédés en lecture ou en écriture doivent être dans

l'ensemble `LibraryPointsTo`..

Performance : Ali et Lhotak ont testé le rappel et la précision de leurs outil par rapport à différents outils existants tels que DOOP et Spark (Ali et Lhoták, 2012). Le rappel évalue la mesure dans laquelle tous les appels observés durant l'exécution ont été relevés par CGC, et pour ce faire, ils sont utilisés le graphe d'appel dynamique généré par l'outil J*. En ce qui concerne le rappel, CGC a montré de bons résultats contre DOOP et Spark par rapport aux appels présents dans le graphe d'appel généré dynamiquement. Pour la précision, CGC a aussi démontré des résultats prometteurs. Il montre parfois des résultats similaires à DOOP et meilleurs que SPARK dans plusieurs cas. Également, la taille des graphes d'appel CGC est inférieure ou égale à celle des graphes générés par SPARK, tout en prenant beaucoup moins de temps pour les générer. Ce résultat peut s'expliquer par l'élagage des appels internes à la librairie, grâce aux hypothèses de 'compilation séparée'.

Finalement, ils ont reconnu que plus le programme analysé implémente d'interfaces déclarées par des librairies, plus la précision du graphe d'appel diminue, et se rapproche de celle des autres outils qui analysent les appels des applications et des librairies en même temps. C'est-à-dire, lorsque le code de l'application commence à dépendre du code de la bibliothèque, les hypothèses faites deviennent inutiles.

1.1.3 Enjeu 2 : complexité du langage

Certaines caractéristiques des langages de programmation tels que Java, C# ou Python, rendent l'analyse statique plus complexe et imprécise. Ces caractéristiques se manifestent autant dans le code source–au moment de la compilation–que dans le *comportement* du programme, i.e. durant l'exécution. Ces caractéristiques rendent le graphe d'appel généré statiquement pollué et peu représentatif du com-

portement du programme (Ali *et al.*, 2019), (Lhoták, 2007) (Fourtounis *et al.*, 2018). Nous allons discuter deux telles caractéristiques : 1) la *réflexion*, et 2) le *polymorphisme*

La réflexion : La *réflexion* est une "capacité" de langages de programmation qui permet à un programme, écrit dans ce langage, d'accéder à sa propre représentation et comportement durant l'exécution. Précisément, la *réflexion* se décline en deux capacités, l'**introspection**, qui permet à un programme d'accéder à sa représentation interne durant l'exécution, et l'**intercession**, qui permet au programme de modifier sa propre interprétation ou son propre comportement durant l'exécution (Kurtev, 2010). La disponibilité d'une représentation, à l'exécution, de "code exécutable" permet alors d'invoquer des fonctionnalités, à l'exécution, sans les nommer explicitement dans le code source, et par conséquent, ces invocations échappent à l'analyse statique. En pratique, seule une inspection manuelle/visuelle du code source–ou alors, une analyse de flux de données poussée–permettent de comprendre ce qui se passe à l'exécution (Bodden *et al.*, 2011).

Nous allons illustrer la réflexion sur un simple exemple en Java où il sera question de créer une instance d'une classe sans que le nom de la classe n'apparaisse comme *type* dans le code (figure 1.6). La première ligne attribue une valeur à la chaîne de caractères `nomQualifieClasse`, représentant le nom qualifié (avec le paquetage) de la classe ; même si dans cet exemple, le chaîne est donnée sous forme littérale, on peut imaginer la valeur retournée par une fonction externe, voire même une entrée de l'usager (user input). La deuxième ligne "charge" la classe ayant pour nom qualifié `nomQualifieClasse` dans le 'dictionnaire' du 'chargeur de classe' (ClassLoader) courant et retourne l'*objet* du type `Class` représentant la classe en question durant l'exécution. Cet objet offre différents services, dont la *création d'instances*, illustrée dans la troisième ligne. Notez qu'en Java, `newInstance` retourne un objet du type `Object` ; dans ce cas-ci, le/la programmeur(se) est

"confiant(e)", à ses risques et périls, que le type instancié est (un sous-type de) `Etudiant`.

```
String nomQualifieClasse = "ca.uqam.info.EtudiantGradue";
Class objetClasse = Class.forName(nomQualifieClasse);
Etudiant nouvelEtudiant = (Etudiant)(objetClasse.newInstance());
```

Figure 1.6: Un exemple qui illustre la réflexion

De la même façon qu'on a invoqué le constructeur de la classe de façon générique, Java permet aussi d'invoquer des méthodes, en allant les chercher d'abord avec leur signatures, qui elles aussi peuvent être construites dynamiquement/à l'exécution. Tout cela échappe à l'analyse statique.

Polymorphisme : Le polymorphisme en programmation orientée-objet est une capacité qui permet de lier un objet déclaré de type T_1 à un autre objet d'un autre type T_2 (Rountev *et al.*, 2004). Dans les langages typés, l'affectation n'est valide que si T_2 est un sous-type de T_1. Prenons l'exemple suivant, où `EtudiantGradue` est un sous-type de `Etudiant`, on peut écrire :

```
Etudiant nouvelEtudiant = new EtudiantGradue("Jean Dupont", "DUPJ31129900");
nouvelEtudiant.setProgramme("Maitrise");
```

Pour que ce code "passe la compilation", il faut que la méthode `setProgramme(String pg)` soit définie au niveau du type `Etudiant`. Par contre, si la classe `EtudiantGradue` redéfinit (*override*) la méthode `setProgramme(String pg)` de `Etudiant`, alors c'est la version de `EtudiantGradue` qui sera appelée à cet endroit, et non celle de `Etudiant`. Dans un graphe d'appel, les deux méthodes seront représentées par des noeuds différents. Encore une fois, ici nous sommes capables de déterminer "visuellement" la version de la méthode qui est appelée parce que la première

ligne montre un appel explicite au constructeur de `EtudiantGradue`. Mais si la variable `nouvelEtudiant` était initialisée par un appel d'une "méthode fabrique" (*factory method*) qui va lire les données dans un fichier :

```
Etudiant nouvelEtudiant = getNewEtudiant(fichierEntree);
```

même l'analyse du code de la méthode `getNewEtudiant(...)` pourrait ne pas suffire [1].

D'une manière générale, le polymorphisme pose des problèmes autant pour les analyses statiques que pour les analyses dynamiques (Xie et Notkin, 2002). Nous allons traiter ce point en détail dans le chapitre 3.

1.1.4 Enjeu 3 : la précision

L'analyse statique des programmes nous permet de découvrir tous les chemins d'exécution imaginables. Comme mentionné dans (), les graphes d'appel ne sont que des graphes de flux de contrôle, mais avec plus d'information et de détails. Ainsi, les graphes d'appel générés par une "simple" analyse statique peuvent comporter des "transitions" qui ne sont pas "traversables" et des états qui ne sont pas atteignables.

Considérons l'exemple suivant :

```
if (A) {
    X
```

1. On peut imaginer que le fichier d'entrée contienne des données sur les différents types d'étudiants, indiqués par un champ de données, et dépendant de la valeur du champ, on instancie une classe ou une autre.

```
} else {
    Y
}
```

Ici, on peut déduire que le chemin d'exécution dans ce cas sera soit X, soit Y, selon la condition A. Par contre, dans l'exemple suivant :

```
B = NOT A;
if (A) {
     X;
}
if (B) {
    Y;
}
```

Ici, une analyse statique "simple" va nous dire qu'une exécution donnée peut exécuter {} (aucun bloc exécuté), {X} (seulement X), {Y} (seulement Y), ou {X,Y}. Or, une inspection visuelle nous permet de constater que X et Y sont mutuellement exclusifs. Pour arriver à cette conclusion automatiquement, il faut une analyse statique beaucoup plus fine, utilisant l'analyse du flux de données ou l'analyse d'accessibilité, pour éliminer les combinaisons impossibles et ne garder que X et Y comme exécutions possibles du programme (Bastos *et al.*, 2016). `à discuter` Donc, nonobstant les difficultés liées à la réflexion et au polymorphisme, l'analyse statique est *par défaut* conservatrice, i.e. veille à ne rien manquer–quitte à en mettre trop–et manque de précision (Lhoták, 2007).

1.2 Analyse dynamique

1.2.1 Principes et exemples

À cause de la différence entre le comportement dynamique des programmes et leur description de comportement obtenue grâce à l'analyse statique du code source (Taniguchi *et al.*, 2005), les développeurs utilisent des techniques simples d'analyse dynamique afin de mieux comprendre le comportement du code dans la machine virtuelle et au moment de l'exécution. Ce type d'analyse se base sur les traces d'exécution qui représentent des méthodes de code qui sont déclenchées par l'exécution et l'interaction avec un programme(Pirzadeh *et al.*, 2013). L'une des techniques les plus courantes que l'on utilise quand on veut diagnostiquer un problème qui se produit lors de l'exécution, consiste à ajouter de nouvelles lignes de code qui produisent une "trace" de l'exécution, que l'on peut par la suite analyser. Ces instructions de "traçage" peuvent être insérées manuellement, de façon ad-hoc (e.g. des instructions `println(...)`), ou en invoquant des fonctions de traçage faisant partie de librairies externes (e.g. Log4J), ou encore différentes variantes de la programmation par aspects, ou alors grâce à des outils qui instrumentent directement la machine virtuelle pour produire une trace des appels de fonctions.

À titre d'exemple, la figure 1.7 montre qu'on peut tout simplement utiliser des messages de sortie au début de chaque appel afin de connaître le chemin d'exécution pris. En suivant les messages de notre sortie, nous conclurons que les méthodes a après c et puis b ont été exécutées.

```
public class Exemple {                    void b() {
 public static void main(String[] args){   System.out.println("b a été appelée");
        a();                                }
        }                                  void c() {
 void a() {                                  System.out.println("c a été appelé");
    System.out.println("a a été appelée"); b();
    c();                                    }
        }                                  }
```

Figure 1.7: Exemple simple de l'utilisation des messages pour avoir les traces d'exécution

1.2.2 Enjeu 1 : Il faut pouvoir exécuter le code

L'analyse dynamique est basée sur l'exécution du logiciel dans un environnement où on peut accéder à la JVM pour générer les traces. L'exécution de programmes récents est assez compliquée en soi, puisqu'il faut trouver et reconstituer l'environnement d'exécution original de l'application en question, dont les bonnes versions du système d'exploitation, des librairies, des logiciels externes, des variables d'environnement, et de diverses autres source de données utilisées par le logiciel. Cela est d'autant plus compliqué pour les "vieux logiciels légataires", pour lesquels certains éléments externes (librairies, logiciels, etc.) peuvent être obsolètes ou simplement non-disponibles. À titre d'exemple, pour notre expérimentation, nous avons utilisé l'application J2EE "Java Pet Store" (JPS) [2]. Cette application, développée par Sun Microsystems (depuis, rachetée par Oracle) au début des années 2000, était supposée être une vitrine ("showcase") de toutes les technologies composant J2EE à l'époque, dont les JavaServer Pages ("JSP"), Servlets, Entre-

2. JPS 1.3.1 : https ://www.oracle.com/technetwork/java/petstore1-3-1-02-139690.html (Accès Juin 2020)

prise JavaBeans ("EJB") et Java Messaging Service ("JMS"), etc. À l'époque, JPS était destinée à être exécutée avec JDK 1.4 et J2EE 1.3.1. Le serveur J2EE correspondant à la version 1.3.1 venait avec la base de données IBM Cloudscape, qui n'est plus supportée depuis belle lurette.

1.2.3 Enjeu 2 : Taille des traces

Les applications appellent généralement certaines méthodes plus fréquemment que d'autres. C'est pourquoi les traces d'exécution capturées à l'aide d'une analyse dynamique ont souvent d'énormes tailles. En effet, ça rend l'analyse et l'étude de ces traces plus difficile et complexe (Pirzadeh *et al.*, 2013) (Feng *et al.*, 2018). La raison derrière les grandes tailles est que certaines pratiques de programmation utilisent des modèles répétés tels que les boucles et les appels récursifs de méthodes (Taniguchi *et al.*, 2005).

Selon le type d'études effectué sur les traces, les développeurs décident s'ils veulent compresser et réduire la taille ou non. Huang et Bond ont utilisé le groupement pour réduire les nœuds similaires qui représentent les mêmes méthodes et économiser de l'espace et du temps (Huang et Bond, 2013), . Dans le cadre de leur étude, qui devait aider à détecter les bogues et identifier les erreurs, ils ont considéré que deux nœuds étaient équivalents s'ils représentaient le même site et s'ils avaient le même nœud parent. L'une des étapes clés de leur approche, qui est utilisée dans leur outil appelé CCU (Huang et Bond, 2013), consiste à créer une sorte d'index pour les nœuds qui utilise, justement, une fonction du site et du parent comme clé. Cela aide à identifier les nœuds équivalents déjà définis, et permet d'y faire référence ultérieurement ; les doublons sont simplement ajoutés à un "Garbage Collector" et les flèches entrantes au noeud jugé équivalent sont redirigées vers le noeud existant ayant le même index.

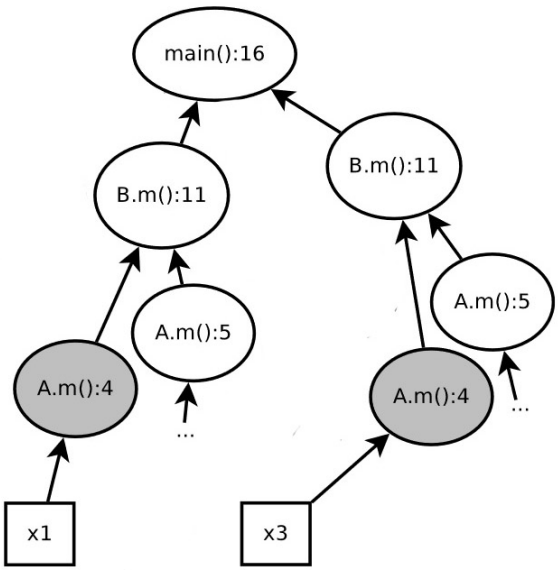


Figure 1.8: un graphe avec des nœuds répétitifs

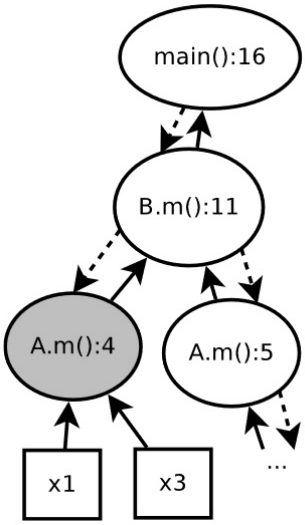


Figure 1.9: un graphe avec le regroupement de nœuds

Comme illustré dans leur exemple (figure 1.8), après avoir identifié les nœuds en double (en gris), leurs liens entrants ont été détournés vers leur représentant correspondant. En conséquence, nous aurons le graphe d'appel dans la figure 1.9.

Similaire à la façon (Huang et Bond, 2013) a essayé de réduire la taille des données, (Law et Rothermel, 2003) partagent la même idée, mais ils ont choisi de manipuler les traces d'exécution avant de passer aux graphes. Leur approche utilise le chemin d'exécution pour améliorer l'analyse d'impact afin d'aider les développeurs à prédire ce qui va changer lorsqu'une instruction donnée est ajoutée ou modifiée. Puisqu'ils capturent le comportement du programme à l'exécution(c'est à dire au niveau dynamique), ça signifie qu'ils n'ont pas besoin d'avoir accès au code source. Dans (Law et Rothermel, 2003), Law et Rothermel ont remarqué que si une méthode X change, cela peut impacter, autant toutes les chaines d'appel menant à X, que les méthodes appelés directement ou 'transitivement' par X. On reconnaît ces méthodes au niveau de la trace d'exécution comme étant : 1) toutes les mé-

thodes précédant l'appel à X mais qui n'ont pas encore terminé leur exécution, i.e. les méthodes qui sont encore sur la pile d'exécution pendant que X est active, et 2) toutes les méthodes qui sont appelées après X, avant que X ne termine son exécution, i.e. toutes les méthodes qui ont été ajoutées à la pile d'exécution "au dessus de X", pendant que X était encore active. Cependant, ils sont également confrontés aux mêmes problèmes de (Huang et Bond, 2013) (boucles, méthodes récurrentes, appels de méthode répétitifs, etc.). L'observation précédente leur a permis de proposer une heuristique pour compresser les traces d'exécution, comme expliqué ci-après.

```
M B r A C D r E r r r r x M B G r r r x M B C F r r r r x
```

Figure 1.10: Exemple des traces d'exécutions (Law et Rothermel, 2003)

Ils ont utilisé une version modifiée d'un outil appelé SEQUITUR pour construire un graphe acyclique dirigé sur lequel ils peuvent appliquer leur algorithme. L'outil est généralement utilisé pour compresser des textes et des fichiers (Nevill-Manning et Witten, 1997). Dans ce cas, ça prend des traces d'exécution en entrée et renvoie en sortie une chaîne plus petite qui représente une grammaire, sans les redondances, permettant la régénération de la chaîne d'entrée. Suivant l'exemple utilisé dans (Law et Rothermel, 2003) (dans la figure 1.10), chaque trace d'exécution, elles sont séparées par x pour signifier la fin d'une exécution, seront transmises une après l'autre au SEQUITUR pour être traitées (commençant par une chaîne vide). À chaque itération, le programme attribue des symboles aux appels répétés. Par exemple, pour la 2e itération la chaîne retournée par l'outil va être comme suit :

$\Gamma \rightarrow$ `M r B r A C D r E 1 1`

$1 \rightarrow$ `r r`

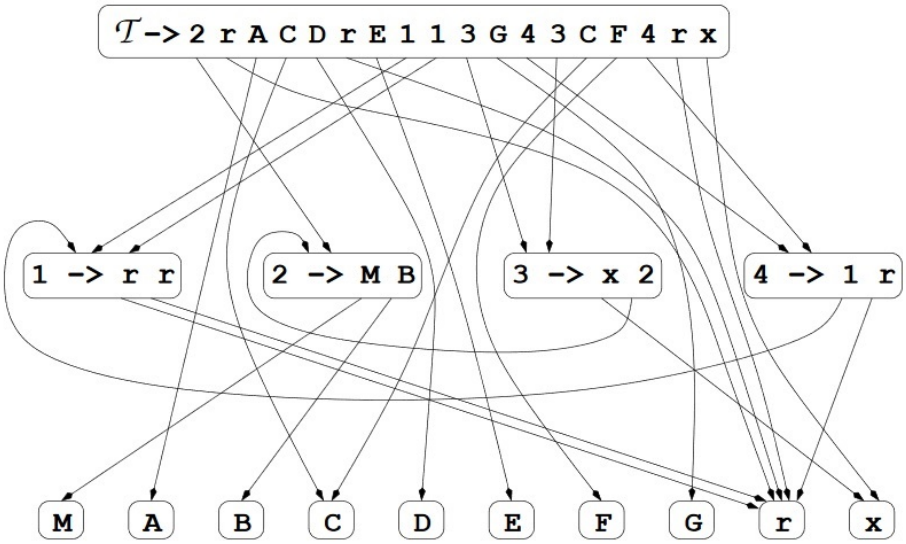


Figure 1.11: Diagramme de chemin complet acyclique dirigé(Law et Rothermel, 2003)

Lorsque l'algorithme de l'outil finira de s'exécuter, nous aurons comme résultat la grammaire suivante :

$\Gamma \rightarrow$ `M 2 r A C D r E 1 1 3 G 4 3 C F 4 r x`

$1 \rightarrow$ `r r` , $2 \rightarrow$ `M B` , $3 \rightarrow$ `x 2` , $4 \rightarrow$ `1 r`

Après les itérations de SEQUITUR, ils sont ensuite passés à la représentation de la grammaire de sortie sous forme d'un graphe (figure 1.11) sur lequel ils peuvent ensuite appliquer leur algorithme d'impact de changement qu'ils ont appelé *PathImpact*. Ils disent qu'en partant de la méthode modifiée `X` puis en remontant et en descendant dans le graphe (en suivant les liens), c'est comme avoir les traces d'exécution d'origine non compressées. Donc, pour détecter ce qui pourrait être affecté lorsqu'une méthode est modifiée, *PathImpact* devrait parcourir le graphe et collecter les noeuds, partant du bas du graphe–le noeud qui représente la méthode– et en naviguant à droite pour capturer les méthodes appelées après la méthode `X`,

et en arrière pour les méthodes qui n'ont pas terminé leur exécution lorsque notre méthode modifiée `X` a été appelée.

Ils ont testé leur outil sur un logiciel développé en C pour l'agence spatiale européenne qui contient 6200 lignes de code. Ils ont utilisé 36 versions du programme pour tester leur approche où chacun a un bogue. Ils les ont comparés avec d'autres techniques d'analyse d'impact telles que la *fermeture transitive* sur les graphes d'appel (Badri *et al.*, 2005), (le découpage statique et le découpage dynamique) (Li *et al.*, 2013). *PathImpact* a montré de meilleurs résultats en donnant de meilleures prédictions des ensembles de changements. Ils ont affirmé que SEQUITUR avait réduit la taille de leurs traces d'exécution à un cinquième et pour (Larus, 1999), il les a compressés de 2 Go à 100 Mo, mais ils n'ont pas effectué d'analyses concernant le temps d'exécution pour voir si l'utilisation de la compression des traces a aidé ou non. Car comme ils l'ont mentionné dans (Law et Rothermel, 2003), cela pourrait mieux fonctionner si SEQUITUR itère simultanément avec la capture des traces d'exécutions.

1.2.4 Enjeu 3 : Chaque trace reflète une exécution

Alors que l'analyse statique explore toutes les combinaisons imaginables (y compris celles qui ne sont pas possibles, telle que l'exemple ci-haut dans la section 1.1.4), une trace reflète un chemin dans le code qui dépend : 1) du point d'entrée, 2) des valeurs des variables fournies en entrée, et 3) des méthodes accessibles (Sereni, 2007). Chaque exécution d'un programme nous donne une séquence de méthodes invoquées et ces scénarios d'exécution représentent un cas d'utilisation ou plus dans ce programme, comme par exemple créer un nouveau compte, effectuer un paiement, ou remplir un formulaire (Feng *et al.*, 2018) (Miranskyy *et al.*, 2007). Donc, pour obtenir un graphe d'appel dynamique, la capture de

plusieurs traces d'exécution est requise (Xie et Notkin, 2002). Certains scénarios ne peuvent pas être invoqués ou peuvent être manqués, surtout si on ne connaît pas le programme ou que l'on n'a pas accès au code source. À titre d'exemple, c'est difficile d'avoir le comportement du programme en cas d'erreur de connexion à la base de données ou à un service, et il est impossible d'appeler du code mort, etc. Par conséquent, les graphes obtenus à l'aide de ces traces représentent une vue incomplète du comportement du programme.

Lorsqu'on utilise les traces d'exécution, on essaie de couvrir autant de scénarios que possible. Cependant, si l'objectif est de faire une analyse d'impact, cela pourrait être problématique puisqu'on ne pourra pas identifier toutes les parties d'un programme impactées par un changement donné.

1.2.5 Enjeu 4 : Quoi faire des librairies

Nous retrouvons ici un peu le même enjeu que dans le cadre de l'analyse statique (section 1.1.2), c-à-d les traces obtenues dynamiquement vont être polluées avec les méthodes des librairies et elles vont 'submerger' les traces d'exécution du programme. Ou pire encore, si le code des librairies est caché, par exemple lorsque des APIs sont utilisées, on n'y voit que les interfaces utilisées pour se connecter avec le programme et rien d'autre.

1.3 Combiner et comparer l'analyse statique avec l'analyse dynamique

Nonobstant les difficultés potentielles inhérentes aux langages (dans la section 1.1.3), l'analyse statique est conservatrice et sécuritaire. Donc, à l'exclusion des dépendances cachées qui ne sont pas détectées par l'analyse de code traditionnelle, et les appels qui utilisent la réflexion, elle surestime souvent les appels qui peuvent être faits lors de l'exécution. Par contre, l'analyse dynamique est précise, mais

incomplète. Donc, dans les meilleurs cas, le graphe d'appel statique devrait être un sur-ensemble du graphe d'appel dynamique (Xie et Notkin, 2002) (Lhoták, 2007). Si ce n'est pas le cas, c-à-d qu'il y a des appels du graphe dynamique qui ne sont pas présents dans le graphe statique, cela voudra dire qu'il y a un problème dans la construction du graphe statique.

On peut donc utiliser le graphe d'appel dynamique pour valider le graphe d'appel statique ; c'est ce que nous faisons dans ce travail. On peut aussi utiliser le graphe d'appel statique pour comparer différentes techniques de génération de graphe d'appel dynamiques, en comparant leur *couverture* du graphe statique. Inversement, on peut comparer différentes techniques de génération de graphes d'appel statiques en comparant leur précisions par rapport à un graphe d'appel dynamique.

Selon Lhoták, la comparaison des graphes d'appels statiques à un graphe d'appels dynamique permet d'exposer et d'identifier les appels erronés et manqués par l'analyse statique (Lhoták, 2007). Mais à cause des différences qu'ont les outils d'analyse de programme dans la représentation de leurs résultats, il est difficile de les comparer les uns aux autres (Lhoták, 2007). Pour ce faire, on doit trouver comment les représenter dans la même structure où on peut après utiliser des métriques de comparaison pour les évaluer.

Lhotak et al. ont proposé un format pour représenter les graphes d'appel, quel que soit l'outil utilisé, pour rendre la comparaison possible (Lhoták, 2007). Le format défini est illustré par le métamodèle de la figure 1.12 [3]. On y voit que chaque méthode est représentée par un noeud qui a un nom et une signature comme attribut. Chaque méthode est connectée à un noeud `Class` qui représente

---

3. `https://plg.uwaterloo.ca/~olhotak/probe/schemas/callgraph.html` (Accès Juin 2020)

la classe où elle a été déclarée. Les noeuds du type `Class` ont des attributs pour représenter de nom et le paquetage de la classe.

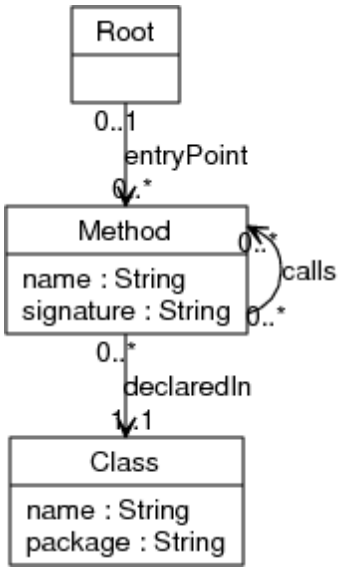


Figure 1.12: Format de graphe d'appel utilisé dans ProBe

Lhoták et al ont également présenté un outil de comparaison de graphes d'appel, nommé *ProBe*, qui utilise le format de fichier ci-dessus. Les autres outils de comparaison commencent généralement par un nœud de départ et parcourent le graphe jusqu'à ce qu'ils atteignent le nœud final. Au contraire, Probe n'a pas besoin d'un nœud d'entrée pour la comparaison. Leur approche vérifie simplement si un nœud est absent du deuxième graphe, puis l'ajoute à l'ensemble de différences. Leur outil fonctionne de manière asymétrique : il compare le premier graphe par rapport au deuxième graphe et renvoie la différence. Mais, on peut inverser les graphes et obtenir la différence du deuxième par rapport au premier.

Dans (Lhoták, 2007), ils se concentrent sur la différence entre deux graphes, plutôt que sur ce qui est commun, pour pouvoir trouver les appels "parasites". Leur objectif est de se débarrasser des appels, qui nuisent à l'analyse et à la comparaison

de graphes d'appel. Un appel parasite et faux pour eux c'est quand une méthode trouvée dans les deux graphes (les noeuds blancs dans la figure 1.13) appelle une méthode qui n'existe que dans le deuxième graphe (les noeuds noirs dans la figure 1.13).

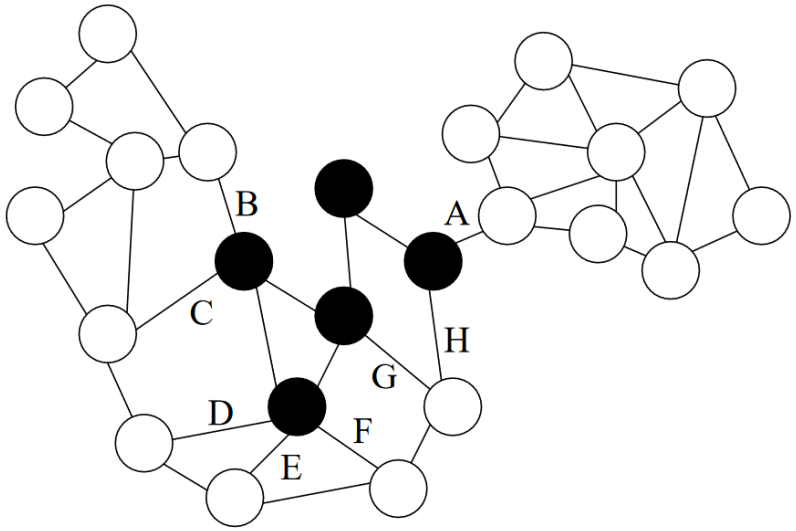


Figure 1.13: Un exemple de graphe d'appels avec ProBe (Lhoták, 2007)

Le lien qui représente un appel et qui relie une méthode commune à une autre visible dans seulement un graphe est identifié comme le début d'un "faux appel". Ces faux appels identifiés (les appels de A à H dans l'exemple) seront ensuite regroupés et classés afin de déterminer le(s)quel(s) est (sont) réellement à l'origine du plus grand nombre de faux appels, causant des différences entre les graphes.

L'algorithme utilisé pour classer les faux appels potentiels prend tous les nœuds qui sont détectés dans un seul graphe (les noeuds noirs dans l'exemple) et navigue dans le sens opposé de la direction d'appel jusqu'à ce qu'il atteigne les méthodes présentes dans les deux graphes d'appel (les noeuds blancs). Par la suite, on compte le nombre de fois qu'un lien a été traversé. Celui qui a été traversé le plus

de fois est classé premier. Le classement sera ensuite envoyé à l'utilisateur pour choisir lequel est le plus susceptible d'être à l'origine de l'erreur présente dans le graphe d'appel et la supprimer après avoir étudié la classe de la méthode et dans quel paquetage elle a été déclarée. Lhoták et al. ont choisi de tester la précision de leur outil sur le logiciel open-source JESS (Java Expert Shell System), un moteur à base de règles qui utilise beaucoup la réflexion [4] (Lhoták, 2007). Après l'obtention des graphes d'appel à l'aide des outils d'analyse statique et dynamique, ils ont passé à l'évaluation de ProBe. Ils étaient capables de trouver les principales raisons de la différence entre les graphes d'appel. Lors de la première exécution de ProBe, les premiers appels classés comme appels incorrects étaient ceux qui utilisaient la réflexion. Ces appels ont été trouvés uniquement dans le graphe dynamique, mais pas dans le graphe statique. Après avoir corrigé le graphe statique en ajoutant ces appels manqués, ils ont exécuté l'outil plusieurs fois. Le classement a retourné des classements différentes à chaque exécution. Par exemple les classes réseau de Java, les appels correspondant au chargeur de classes JVM (class loaders), les appels aux classes de bibliothèque externes et les appels à des méthodes de sécurité. Après chaque exécution, le nombre de méthodes dans le graphe statique est réduit grâce à l'élimination des mauvais appels, notamment, la suppression des méthodes inaccessibles. À la fin, ils ont pu identifier 1357 des 2441 méthodes potentiellement erronées, qui ne sont présentes que dans le graphe statique, sont 100% inaccessibles et il ont dit que leur outil peut être utilisé pour identifier et bien comprendre le reste des appels.

---

4. https ://www.spec.org/jvm98/jvm98/doc/benchmarks/ (Accès Juin 2020)

1.4 Conclusion

Dans ce chapitre nous avons comparé les techniques d'analyse statique et dynamique pour comprendre le comportement de programmes, et les forces et faiblesses de chaque type de technique.

Pour rappel, le but de notre travail est de valider les dépendances cachées inférées statiquement en extrayant d'abord les dépendances statiques puis en définissant un environnement de comparaison pour pouvoir les confirmer en utilisant l'analyse dynamique.

Dans le chapitre 2, nous décrirons la méthodologie que nous allons utiliser pour comparer, d'un côté, le graphe d'appel statique augmenté des dépendances cachées ajoutées selon (Hecht *et al.*, 2018), et le graphe d'appel dynamique. Notre façon de générer le graphe statique est décrite dans le chapitre 3. Le chapitre 4 abordera l'analyse dynamique, dont la capture des traces d'exécution, le regroupement des traces, et la préparation des données pour la comparaison ; la comparaison en tant que telle sera présentée dans le chapitre 5.

CHAPITRE II

MÉTHODOLOGIE

Ce chapitre présente un survol de la méthodologie que nous proposons pour valider les dépendances cachées par traces d'exécution. Pour une application donnée P, l'idée de base est la suivante :

1. Générer un graphe d'appel statique GAS de l'application P, auquel on va ajouter les dépendances cachées qui correspondent aux services d'infrastructure J2EE, tel que décrit dans la section 0.3. Ceci va nous donner un *graphe d'appel statique augmenté*, GAS_aug.
2. Générer un graphe d'appel dynamique GAD en exécutant l'application P.
3. Comparer GAS_aug à GAD pour nous assurer d'avoir identifié, statiquement, *toutes* les dépendances cachées liées aux services d'infrastructure.

La génération de graphe d'appel statique augmenté est décrite brièvement dans la section2.1, et dans le chapitre 4. La génération du graphe d'appel dynamique est décrite brièvement dans la section 2.2, et dans le chapitre 5. La comparaison est décrite brièvement dans la section 2.3 et dans le chapitre 6.

2.1 Les graphes d'appels statique

Notre objectif ici est de construire un graphe d'appels statique qui soit, à la fois *complet*, c'est à dire ne manquant aucun appel potentiel, mais aussi, le plus *précis*

que possible, c'est à dire contenant le moins d'appels improbables que possible. A ce graphe, nous allons ajouter les dépendances cachées qui échappent à l'analyse statique de code, mentionnées dans la section 1.1.3, dans l'espoir d'améliorer la précision et enrichir le graphe d'appels obtenu. Nous allons implémenter certaines règles pour corriger et ajouter des dépendances cachées pour les applications développées dans JAVA, et nous utiliserons un moteur à base de règles (Hecht *et al.*, 2018) pour les dépendances cachées de J2EE. L'analyse statique est décrite dans le chapitre 4. L'ajout de dépendances cachées y est discuté dans la section 3.5.

2.2 Les graphes d'appels dynamique

Comme nous l'avons expliqué dans la section 1.2.4, chaque trace représente une exécution d'un programme. Notre but donc est d'effectuer plusieurs exécutions et de collecter et corréler les traces produites par chaque exécution. L'un des enjeux des traces d'exécution est la *couverture* du comportement du programme à l'étude. Les traces seront d'autant plus utiles qu'elles couvrent le plus d'exécutions possibles, nonobstant certaines limites que nous avons discutées dans la section 1.2.2. Comme nous l'avons vu dans le chapitre 2, les traces doivent être "purgées" d'appels non-pertinents à notre comparaison, dont les appels vers les librairies externes, comme cela a été fait dans (Huang et Bond, 2013). La production des graphes d'appel dynamique est décrite dans le chapitre 5.

2.3 Comparaison

Pour comparer les deux graphes d'appel, nous devons :

1. utiliser la même représentation des graphes d'appel, lesquels sont produits par des outils différents
2. utiliser une représentation qui supporte leur manipulation.

Dans notre cas, nous avons choisi d'utiliser la base de données orientée graphes, *Neo4J*. Par conséquent, avant de commencer à parler de la comparaison, à proprement parler, nous commençons par décrire Neo4J (Section). Les principes qui sous-tendent la comparaison sont décrits dans la section **??**.

2.3.1 Neo4J

Pour représenter, stocker et analyser les graphes d'appels statiques et dynamiques extraits, nous avons choisi d'utiliser Neo4J. C'est une base de données orientée graphe développée en Java (Guia *et al.*, 2017). Neo4j représente des bases de données de graphes sous forme de nœuds, d'arêtes les reliant et de propriétés de nœuds et d'arêtes (Guia *et al.*, 2017). Neo4J est évolutive, performante, et offre deux langages de requête pour la manipulation des données, `Gremlin` et `Cypher` (Holzschuher et Peinl, 2013). `Gremlin` est un langage de bas niveau pas facile à lire, alors que `Cypher`[1] est un SQL déclaratif langage plus lisible et plus facile à comprendre. Nous utiliserons Cypher dans nos manipulations de graphes d'appel, y compris le calcul des différentes métriques de comparaison.

Dans ce qui suit, nous allons présenter une requête pour illustrer le fonctionnement de Cypher. Notre exemple ci-dessous retourne toutes les personnes fréquentant l'UQAM avec lesquelles David est ami.

```
MATCH (p1:Personne)-[r1:ami]->(p2:Personne) where p1.nom="David", p2.frequente="u
Return p2
```

— **Match** : Recherche ce qui vient après le mot-clé (c'est-à-dire nœud, relation, propriété, etc.). L'équivalent de `SELECT` en SQL.

1. Cypher : https ://neo4j.com/developer/cypher/ (Accès Juillet 2020)

— **(<variable :type source>)-[<relation>]-(<variable :type destination>)** : Un patron que neo4j doit trouver dans le graphe.
— **p1,p2,r1** : des variables que nous déclarons pour faciliter l'accès dans de reste de la requête.
— **Where** : Cela indiquera à Neo4J qu'on veut chercher des valeurs spécifiques.
— **p1.nom,p2.frequente** : Des propriétés pour le label(noeud de type) Personne.
— **Return** : Spécifie ce que l'utilisateur doit obtenir en retour.

Le reste des mots-clés et les syntaxes se trouvent dans la fiche de référence [2] fourni par Neo4j.

2.3.2 Métriques de comparaison

Comme nous l'avons dit précédemment, nonobstant les difficultés discutées dans la section 1.1, les graphes d'appels statiques sont supposés être complets, mais manquer de précision : ils contiennent plus d'appels que ne sont *possibles* durant l'exécution. D'un autre coté, les graphes d'appels dynamiques sont précis, puisqu'ils ne montrent que les appels ayant eu lieu lors d'une exécution, mais ils sont forcément incomplets puisqu'ils ne refléteront que les exécutions ayant eu lieu, et non *toutes* les exécutions possibles. Donc, si un graphe d'appel statique est généré correctement, le mieux que l'on puisse espérer est qu'il contiennent intégralement le graphe d'appel dynamique, c'est à dire, *tous les appels qui ont pu être observés durant l'exécution ont été pris en compte dans le graphe statique*. Donc, dans notre cas, pour valider les dépendances cachées qui ont été ajoutées "artificiellement" pour codifier les services d'infrastructure, nous devons nous assurer *minimalement*

2. Refcard : https ://neo4j.com/docs/cypher-refcard/current/ (Accès Juillet 2020)

qu'elles rendent compte des appels observés durant l'exécution, c'est à dire qu'elles aient un *rappel* de 100%. Accessoirement, et idéalement, ce serait bien aussi de vérifier que les règles qui ont été utilisées pour ajouter les dépendances cachées n'ajoutent pas "trop de relations d'appel" improbables ou impossibles, c'est à dire qu'elles aient une bonne *précision*. La précision est beaucoup plus difficile à établir, pour toutes sortes de raisons [3]. Pour les besoins de cette étude, nous nous limiterons au *rappel*.

Puisque nous comparons deux graphes différents, nous devons trouver un moyen de les représenter tous les deux de manière similaire afin de faciliter la comparaison. Nous nous sommes inspirés du format publié et utilisé dans ProBe (Lhoták, 2007) et nous avons créé notre propre version qui satisfait nos objectifs. Donc, nous proposons ce qui suit (dans la figure 2.1).

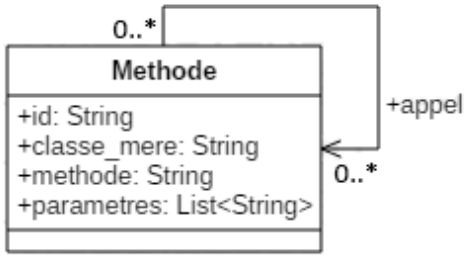


Figure 2.1: La format des graphes d'appels statique et dynamique

La figure 2.2 illustre un exemple d'un graphe d'appel dans Neo4J que nous avons crée utilisant notre format proposé. Contrairement à (Huang et Bond, 2013) (voir discussion dans la section 1.2.3), nous considérons que deux noeuds sont identiques s'ils représentent la même méthode, c'est à dire, même nom, même signature et

3. Par exemple, plusieurs fonctionnalités de gestion du cycle de vie des EJB sont invoquées "au besoin", et non systématiquement, telles `ejbActivate` et `ejbPassivate`

issue de la même classe.Dans le cas de surcharge, c'est à dire lorsque deux méthodes ont le même nom mais des paramètres différents, chacune sera représentée avec son propre nœud. Chaque méthode a un attribut qui fait référence au nom qualifié de la classe [4] où elle a été déclarée, et une liste de paramètres qui ont été utilisés pour l'invoquer. Cette liste est nulle si aucun paramètre n'a été utilisé. Pour faciliter la comparaison entre noeuds, nous avons attribué à chaque méthode un identifiant obtenu par hachage de la chaîne de caractères représentant la méthode, notamment "<nom complet classe> : :<nom de la méthode> : :<liste des paramètres formels>". Dans notre exemple, en bas de la figure 2.2, on trouve les attributs du nœud sélectionné "TaskBean".

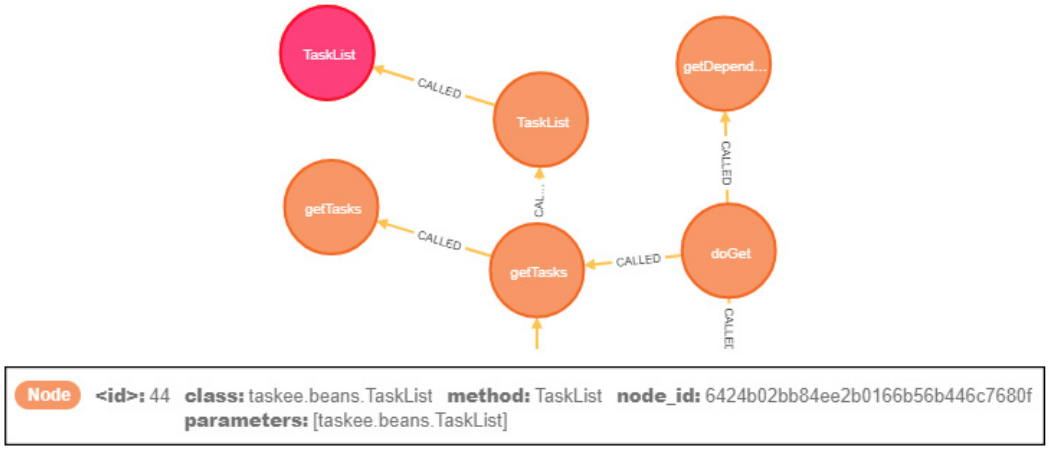


Figure 2.2: Exemple d'un graphe d'appels dans Neo4j

2.4 Conclusion

Dans ce chapitre, nous avons présenté les trois principales étapes de notre méthodologie, et identifié quelques enjeux sur la mise en oeuvre de notre méthodologie.

4. C'est à dire, le nom de la classe préfixé par la hiérarchie de paquetages la contenant.

Les trois prochains chapitres reprennent chacune de ces étapes en expliquant en détail les défis auxquels nous avons fait face pour la mise en oeuvre de ces étapes, et les choix méthodologiques et technologiques que nous avons fait à cet gard.

CHAPITRE III

ANALYSE STATIQUE

Dans ce chapitre, nous présentons notre approche pour l'obtention de graphes d'appels statiques, à partir du code source d'applications Java/J2EE, à des fins de comparaison avec les traces d'exécution (voir section 2.3. Rappelons que notre objectif de recherche est de valider les *dépendances cachées*, qui sont implicites dans les 'contrats de service' de la plateforme J2EE, et qui ont été ajoutées par l'outil développé par l'équipe de M. Mili, et décrit dans (Hecht *et al.*, 2018). Donc, étant donné le code source d'une application Java/J2EE, nous devons effectuer les tâches suivantes, dans l'ordre :

1. Analyser le code source de l'application en utilisant un outil d'analyse statique 'traditionnel' pour obtenir un graphe d'appel–dans notre cas, un fichier XMI représentant un modèle KDM de l'application (voir section 3.1)
2. Ajouter les *dépendances cachées* inhérentes aux services J2EE en utilisant l'outil décrit dans (Hecht *et al.*, 2018) ; cette étape génère un nouveau fichier XMI
3. Transformer le fichier XMI en un fichier JSON pour pour l'importer dans Neo4J, et
4. Charger le fichier JSON dans Neo4J, en préparation pour la comparaison.

Or, un tel graphe va assurément manquer beaucoup d'appels possibles durant

l'exécution, et ce, pour différentes raisons, certaines liées à la sémantique de Java, d'autres aux limitations de l'outil MODISCO, et d'autres, aux limitations de l'outil MaintainJ de génération des traces dynamiques :

— Problèmes liés à la complexité de Java : comme nous l'avons mentionné dans la section 1.1.3, la complexité de Java vont faire que plusieurs appels possibles à l'exécution ne vont pas apparaître dans les graphes d'appel. Outre le cas de la *réflexion*, pour lequel on a peu/pas de recours, il y a le cas de *polymorphisme* où une variable v déclarée du type T est 'occupée' à l'exécution par un objet du type T' qui est un sous-type de T, et donc, si on appelle la méthode f sur v, le graphe d'appel statique montrera un appel à $T :: f$ alors que la trace dynamique montrera un appel à $T' :: f$. Ce cas est facilement traitable, comme expliqué plus bas (Section 3.4.

— Limitations de MODISCO : MODISCO ne sait pas quoi faire des blocs d'initialisation de variables d'instances, et des blocs statiques de code qu'on trouve en début de définition des classes. Les deux peuvent comporter des appels, qui vont apparaître dans la trace dynamique, mais dont MODISCO ne sait que faire. Le problème des blocs d'initialisation et des blocs statiques est décrit dans la section 3.2.

— Problèmes liés à MaintainJ : si l'un des paramètres d'une méthode est une classe C paramétrée par un type T, sa représentation par KDM est $C < T >$, alors que MaintainJ fait abstraction du paramètre T, empêchant la comparaison. Le traitement de la généricité est expliqué dans la section 3.3.

Nous devons donc ajouter ces trois étapes de traitement au processus de base décrit plus haut. Pour des raisons de commodité, certaines étapes ont été effectuées sur le XMI, c-à-d avant le chargement dans Neo4J, et d'autres, à l'intérieur de Neo4J.

Finalement, la chaîne complète de traitement est la suivante (voir aussi Figure

3.1) :

1. Analyser le code source de l'application avec MODISCO (voir section 3.1) pour produire un modèle KDM sous format XMI
2. Ajouter les *dépendances cachées* en utilisant l'outil décrit dans (Hecht *et al.*, 2018) (Section 3.5) ; cette étape génère un nouveau fichier XMI
3. Traiter les blocs d'initialisation et les blocs statiques (Section 3.2)
4. Traiter la généricité (Section 3.3)
5. Transformer le fichier XMI en un fichier JSON pour pour l'importer dans Neo4J, et
6. Charger le fichier JSON dans Neo4J
7. Traiter l'héritage et le polymorphisme dans Neo4J (Sections 3.4.1 et 3.4).

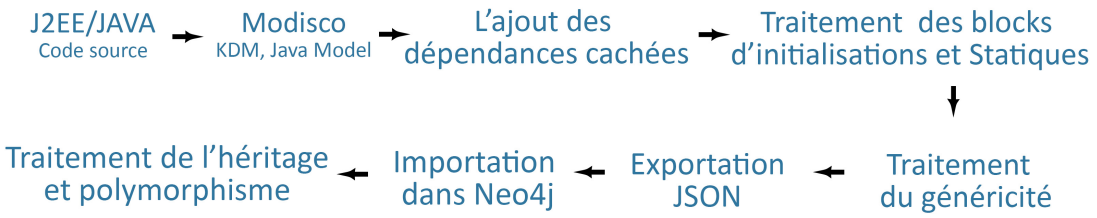


Figure 3.1: L'architecture générale de l'outil qui traite et prépare les graphes statiques

3.1 Environnement technique : MODISCO - KDM

La première étape de notre chaîne de traitement (voir Figure 3.1) consiste à produire une première représentation du graphe d'appels statique sous la forme d'un modèle *KDM*, produite par l'outil *MODISCO*. KDM, pour *Knowledge Discovery Metamodel* est une spécification de l'OMG qui vise a représenter des applications

logicielles (Santibáñez *et al.*, 2015) de sorte à faciliter leur ré-ingénierie [1]. Cette spécification utilise une représentation standard des artéfacts logiciels, quelle que soit la technologie et le langage de programmation en question, en se basant sur un métamodèle commun aux différents langages. Ainsi, il est possible d'utiliser le même métamodèle pour représenter les différentes composantes d'une application, qui peuvent être développées dans différents langages. Par exemple, une "simple" application JEE va contenir, en plus des fichiers Java, de fichiers écrits en HTML, JSP, JSF, différents fichiers en XML, etc.

MODISCO (Bruneliere *et al.*, 2014) est un outil de ré(rétro)-ingénierie de logiciels qui implante la norme KDM. Outre le métamodèle KDM de base, MODISCO comporte un "framework" pour : 1) représenter un nouveau langage, et 2) développer des outils pour lire des fichiers écrits dans ce langage, et construire un modèle KDM correspondant–appelés des *Model Discoverer*. MODISCO vient, d'office, avec des *Model Discoverer*s pour les principaux langages de programmation, dont Java. MODISCO est disponible sous la forme de plug-ins Eclipse. La Figure 3.2 montre une image écran du navigateur de modèles KDM.

3.2 Problématique des blocs statiques et d'initialisation

Les extraits de code suivants montrent trois "bouts de code" Java qui posent problème à MODISCO : 1) les blocs d'initialisation, statique et dynamique, et 2) l'initialisation de variables d'instances.

```
class Etudiant {

    private Collection<Cours> cours;
```

1. Specification de KDM : https ://www.omg.org/spec/KDM/About-KDM/ (Accès Août 2020)

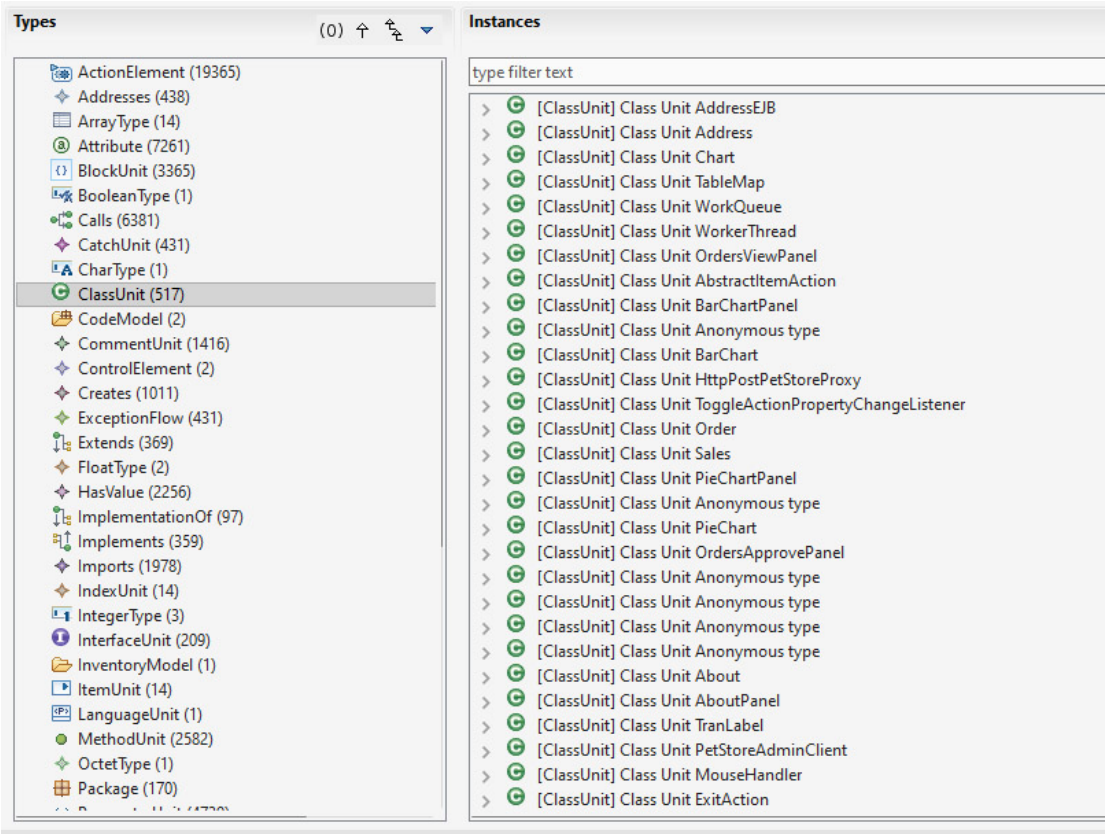


Figure 3.2: Navigateur KDM de Modisco

```
private static Hashtable<String,Etudiant> listeEtudiants;

// bloc statique
static {
    listeEtudiants = new Hashtable<String,Etudiant>();
}

// bloc non-statique
{
    cours = new ArrayList<Cours>();
}
// initialisation de variables d'instance
```

```
    private String nom = "Jean" + "Dupont";
}
```

Les *blocs statiques* sont des blocs de code (compris entre accolades { et })), déclarés `static`, qui sont exécutés au moment de la "naissance de la classe", i.e. au moment de son *chargement par le class loader*. De tels blocs peuvent comporter des appels à des méthodes du code usager de la librairie ; dans notre cas, au constructeur de la classe `Hashtable`. Un bloc non-statique est un bloc de code qui est exécuté *au moment de la création/construction* d'une objet de la classe. L'initialisation d'une variable d'instance (attribut `nom` dans notre exemple) est aussi exécuté au moment de la création/construction de l'objet. Chacune de ces instructions peut comporter des appels à des méthodes qui peuvent apparaître dans les traces dynamiques, et dont il faudra rendre compte au niveau du graphe d'appel statique.

Que dit la spécification Java concernant ces blocs ? la spécification du langage Java (JLS) [2] dit que la machine virtuelle Java commence d'abord par réserve d'abord l'espace mémoire pour toutes les variables d'instance de la classe initialisée et de ses superclasses ; par la suite, elle exécute, en conservant l'ordre d'exécution spécifié par l'utilisateur, les blocs d'initialisation.

Or, au moment de la rédaction de ce rapport, la dernière version du MoDisco (1.5.1) n'affecte pas les appels émanant de ces différents blocs comme ayant pour point de départ la classe en question–classe *Etudiant* dans notre exemple. En effet, MODISCO détecte qu'un appel a été fait à une méthode, mais sans connaître plus de détails sur l'invocateur.

Nous avons donc analyser le modèle KDM généré par MODISCO–sous forme de fixchier XMI–pour reconnaitre ces divers d'initialisation, et traiter les appels qu'ils

2. JLS : https ://docs.oracle.com/javase/specs/jls/se10/html/jls-8.html (Accès Août 2020)

contiennent proprement. Nous expliquons plus en détail comment cela est fait.

Blocs d'initialisation et variables d'instance : D'après la spécification du langage Java, le compilateur Java copie ces blocs d'initialisation au début de chaque constructeur de la classe. Donc, par exemple, pour chaque appel $C :: f()$ émanant d'un bloc d'initialisation de la classe A, ayant un constructeur $A()$, nous ajoutons l'appel $A :: A() -> C :: f()$. Pour l'exemple ci-haut, nous ajouterons, pour le bloc d'initialisation, l'appel suivant : $Etudiant :: Etudiant() \rightarrow ArrayList :: ArrayList()$.

Blocs statiques : Selon la spécification du langage Java (JLS), ces blocs seront appelés une seule fois lorsque une classe est initialisée pour la première fois, c-à-d, au moment de son chargement par le *class loader*. Quand est-ce que le *class loader* charge une classe ? différentes JVM peuvent utiliser différentes stratégies, mais le "plus tard possible" correspond à l'une des deux situations suivantes : 1) la première fois que l'on rencontre une invocation à un constructeur de la classe (`new NomClasse (paremetres)`), ou alors : 2) la première fois qu'on essaye d'accéder ou invoquer à un élément (champ ou méthode) statique de la classe. Donc, la méthode qui va causer le chargement de la classe–et exécuter le bloc statique–va dépendre de la situation. Pour couvrir tous les cas possibles, nous considérons que chaque méthode $f(..)$ qui invoque le constructeur d'une classe ayant un bloc statique, comme "premier invoquateur", et donc, comme "invoquateur" du bloc statique, et nous ajoutons donc une relation d'appel entre $f(...)$, et toutes les méthodes contenues dans le bloc statique.

Nous illustrons le traitement des blocs d'initialisation et variables d'instance, et des blocs statiques, à travers l'exemple qui suit, où la classe `A` a des blocs d'initialisation et un bloc statique qui initialise la classe `B`

```
class main {
 public static void main(String[] args){
  A a = new A(); }
 }
class A {
 static {  // bloc statique
  B b = new B();}
 { // bloc d'initialisation
   Collection<Employee> personnel;
   Employee directeur = new Employee
   ("Annie", "317677607");
   personnel = new ArrayList<Employee>();
   personnel.add(directeur);}
 public A() {
  System.out.println("A a été initia
  lisé");}
}
public class Employee {
private String nom,nas;
public Employee(String nom,String
nas) {
        this.nom=nom;
        this.nas=nas;
          }
   }
```

Comme expliqué précédemment, ces appels sont un peu vague dans le modèle KDM généré. On trouve la partie du graphe statique correspondante à ces lignes de code dans la figure 3.3. MODISCO détecte que les constructeurs `Employee` et `B` ont été appelés mais ne sait pas par quelle méthode, marquant méthode = `null`. Pour compléter le graphe d'appel plus représentatif, nous identifions, pour chaque appel de ce type (méthode source = `null`) le fichier source. Ensuite, nous attribuons les appels contenus dans les blocs d'initialisation *non-statiques* aux constructeurs de la classe (Constructeur `A()` dans notre exemple). Pour l'initialisation de `B` dans le bloc statique, MODISCO l'attribuera à (classe : `A`, méthode : `null`). Or, en étudiant les traces d'exécution, on constate un lien d'appel entre la méthode `main` de `A`, et le constructeur de `B`, car : 1) `main` a instancié la classe `A`, et 2) la classe `A` contient un bloc statique qui, à son tour appelle le constructeur de `B`. Nous devons donc rajouter les liens d'appel correspondants au graphe statique. Donc, après notre intervention, nous passons du graphe d'appel de la figure 3.3 vers le graphe de la figure 3.4.

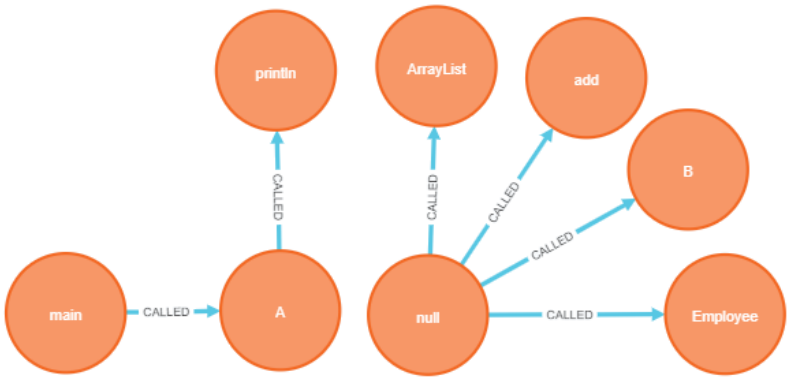


Figure 3.3: Comment MoDisco voit les blocs d'initialisation

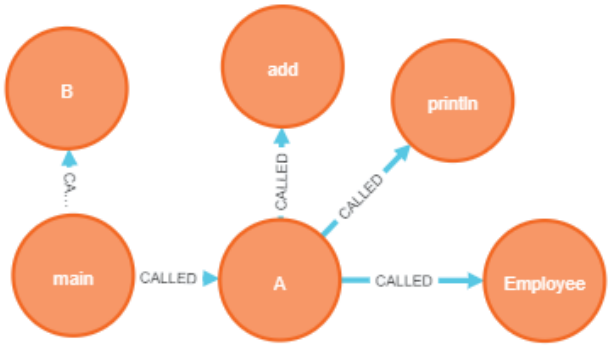


Figure 3.4: Un graphe d'appels avec des blocs d'initialisation après notre intervention

3.3 La problématique de la généricité

La *généricité* en java fait référence à la possibilité de définir des *classes paramétrées par un ou plusieurs types*, c-à-d, des classes qui font référence à des types (comme type de variable d'instance, type de retour ou de paramètre de méthode, etc.) qui sont des *paramètres* pouvant être spécifiés/instanciés au moment de la déclaration. L'exemple type de *classes paramétrées* (ou *génériques*) est les structures de données, telles les *collections*, pour pouvoir y emmagasiner *des objets du même type* (Bracha, 2004). Notons l'exemple de l'interface java.util.List <Type>, ou la classe java.util.HashMap <T1, T2>. Le "Model Discoverer" de Java dans MODISCO traite correctement ces types (interfaces ou classes), c-à-d, en identifiant leurs paramètres. Par contre, l'outil de traçage dynamique perd cette information, et ne montre pas les types paramètres.

Donc, afin d'avoir des graphes d'appels statique et dynamique comparables, nous avons dû éliminer la référence, les modifier et pour ce faire, nous avons utilisé les expressions régulières suivantes :

- `".*\<.*\>$"` : Pour capturer tout ce qui suit le modèle `<nom complet de classe> < Type1, Type2, etc.>` Où .* correspond à n'importe quel caractère, un nombre illimité de fois ; `\<` correspond au caractère `"<"` ; `\>` correspond au caractère `">"` et `$` correspond à la fin du texte.
- `\<.*\>$` : Pour changer tout ce qui est comme `<chaîne de caractères>` en rien, C'est-à-dire les supprimer.

3.4 Toute la problématique du polymorphisme

Dans la section 1.1.3, nous avons montré comment l'utilisation du polymorphisme complexifie l'analyse statique du code, et une analyse naïve risque de créer une représentation inexacte du programme, qui peut se manifester par des discontinuités

dans le graphe d'appel. Comme nous l'avons expliqué dans la section 1.1.3, le polymorphisme permet à une variable déclarée du type T_1 de recevoir, par affectation, des variables ou objets de type T_2, ou T_2 est un sous-type de T_1. Nous rappelons par un petit exemple, les difficultés que cela pose, notamment au niveau de la comparaison entre le graphe d'appel statique et le graphe d'appel dynamique.

```
1  class Main {
2      public static void main(String[] args){
3          Personne p = creerPersonne("Jean","Dupont");
4          System.out.println(p.getNom());
5      }
6      public static Personne creerPersonne(String p, String n) {
7          int rand = random(0,10); // générer un nombre aléatoire entre 0 et 10
8          if (rand > 5) return new Personne(p,n);
9          // sinon
10         return new Etudiant(p,n)
11     }
12 }
13 class Personne {
14     private String prenom, nom;
15     public Personne (String p, String n) { ...}
16     public String getNom() {
17         return prenom + " " + nom
18     }
19     ...
20 }
21
22 class Etudiant extends Personne {
23     ...
24     public Etudiant(String p, String n) { ...}
25     ...
26 }
```

Dans la ligne 3, un compilateur/analyseur ne peut pas savoir a priori si l'objet affecté à la variable *pers* est du type `Personne` ou `Etudiant`, et donc, il a se fier au type de retour déclaré de la méthode `creerPersonne(...)`, et donc, le graphe d'appel statique montrera, pour la ligne 4, un appel de `Main::main()` vers `Personne::getNom(...)`. Par contre, à l'exécution, dépendant de la valeur retournée par la fonction `random(...)`, l'objet retourné par `creerPersonne(...)` sera soit une `Personne` soit un `Etudiant`.

Cet exemple illustre l'un des problèmes posés par le polymorphisme. Pour simplifier la discussion, nous traitons ces difficultés selon trois sous-cas, discutés plus en détail dans ce qui suit.

3.4.1 La problématique des méthodes héritées non redéfinies

Pour reprendre l'exemple précédent, dans le modèle KDM généré par Modisco, si la classe `Etudiant` ne redéfinit pas la méthode `getNom(...)`, nous n'avons pas de moyen de savoir que `Etudiant::getNom(...)` est une cible possible pour les appels. Parce que le graphe d'appel dynamique montre les appels effectifs, nous devons "précalculer" les cibles d'appels possibles, avant de faire les comparaisons. Cette inférence peut se faire à deux endroits et moments différents :

- Directement dans le modèle KDM, avant la génération du fichier JSON et l'importation dans Neo4J, ou
- Directement dans Neo4J.

Nous avons choisir de faire cela à l'intérieur de Neo4J, à cause de la puissance de son langage de requêtes.

Voici donc les deux requêtes Cypher qui rajoutent les méthodes héritées comme cibles (noeuds) potentiels d'appels dans le graphe statique :

```
Match (childclass:Class)-[:Extends*1..]->(superclass:Class),(n1:Node) Where n1.class=superclass.
with childclass,superclass,n1
OPTIONAL MATCH (n3:Node{class:childclass.Class_name,method:n1.method})
WHERE n3 IS NULL with childclass,superclass,n1
Merge (new:Node{class:childclass.Class_name, method:n1.method, parameters:n1.parameters})
return new
```

Cette requête est chargée de trouver si une classe (variable `superclass`) a des méthodes qui ne sont pas définies dans l'une de ces sous-classes (variable `childclass`), et si oui, ajoute les méthodes à la sous-classe. La super-classe doit être une classe ou une classe abstraite, car dans ces deux cas, les méthodes vont être déclarées avec leurs paramètres.

```
Match (n3:Node),p=(n1:Node)-[r1:CALLED]->(n2:Node)
inter=(childclass:Class)-[:Extends*1..]->(superclass:Class) Where
n1.class=superclass.Class_name and n3.class=childclass.Class_name
and n3.method=n1.method and  n3.parameters=n1.parameters
with n3,n2,r1
Merge (n3)-[r:CALLED{parameters:r1.parameters}]->(n2)
return count(r)
```

Cette requête ajoute les appels que les nœuds manquants peuvent effectuer. En effectuant les mêmes appels que la méthode héritée et retourne le nombre d'appels ajoutés.

3.4.2 Problème des interfaces

Les interfaces peuvent introduire des discontinuités dans le graphe d'appel statique, puisqu'une méthode peut avoir comme cible (être appelée sur) une variable déclarée d'un type correspondant à une interface, mais on ne verra pas cette méthode comme *source* d'appel.

Considérons l'exemple suivant : autorisé par le compilateur vu que la classe `Etudiant` implémente l'interface `Personne`.

```
1  public class Main {
2      public static void main(String[] args){
3          ...
4          Personne nouvellePersonne = new Etudiant("Jean Dupont","DUPJ31129900");
5          nouvellePersonne.getCodePer());
6      }
7      ...
8  }
9
10 public interface Personne {
11     public void getCodePer();
12 }
13
14 public class Etudiant implements Personne {
15     private String nom, code_permanent;
16     public void getCodePer() {
17         System.out.println(code_permanent);
18     }
19 }
```

Ici, pour la ligne 5, le model discoverer Java de MODISCO notera une relation d'appel de `Main::main(String[])` vers `Personne::getCodePer()`, où `Personne::getCodePer()` fait référence à la méthode de l'interface `Personne` déclarée dans la ligne 11. Par ailleurs, KDM va représenter la méthode `Etudiant::getCodePer()` définie dans les lignes 16 à 18, par un noeud distinct, et va représenter un appel de `Etudiant::getCodePer()` à `PrintStream::println(...)`. Or, on sait que `Personne::getCodePer()` ne va jamais apparaître dans un graphe d'appel dynamique, puisque `Personne` n'est pas instanciable. Aussi, si `Etudiant` est la seule

classe du présent projet qui implante l'interface `Personne`, on va manquer le fait qu'en réalité, `Main::main(String[])` appelle `Etudiant::getCodePer()` directement, et, par exemple, la méthode `PrintStream::println(...)`, transitivement.

La figure 3.5 montre le graphe d'appel résultant dans Neo4J. Clairement, il nous faut "unifier" les deux noeuds étiquetés `getCodePer`.

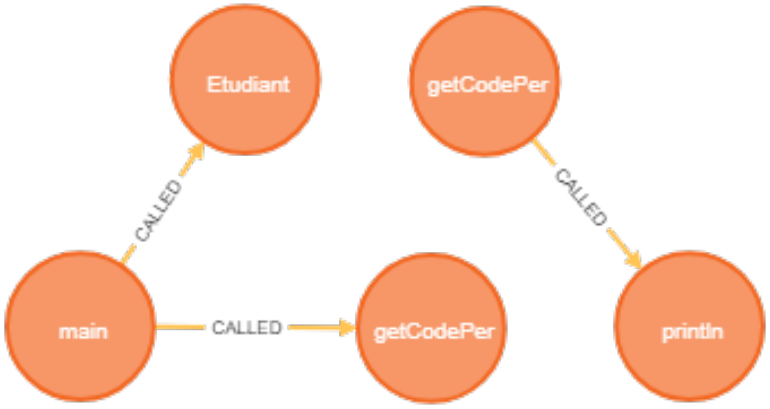


Figure 3.5: Exemple du polymorphisme avec une interface

3.4.3 Problème des classes abstraites

Outre le problème des interfaces et des classes qui les implantent, nous avons un problème similaire avec les classes abstraites et leurs sous-classes, entraînant lui aussi des discontinuités dans le graphe d'appel. Considérons l'exemple suivant, où `Classmain::main(String[])` déclare deux variables du type `Cours` (lignes 4 et 5), qui est une classe abstraite, mais qui reçoivent des objets appartenant à des sous-classes concrètes de `Cours`. Notez aussi la méthode `c()`, qui est définie dans la classe abstraite *Cours* (ligne 14) et dans ces sous-classes (lignes 18-20, et 24-26).

```
1 public class Classemain {
2     public static void main(String[] args){
3         ...
```

```
4           Cous cINF8000 = new C_DepInfo();
5           Cours cINF8882 = new Seminaire_mait_info();
6           cINF8882.c();
7           cINF8000.c();
8           ...
9       }
10  }
11
12  abstract class Cours{
13      private String sigle;
14      public abstract void c();
15  }
16
17  public class C_DepInfo extends Cours{
18      public void c() {
19          System.out.println("Cours informatique");
20      }
21  }
22
23  public class Seminaire_mait_info extends C_DepInfo {
24      public void c() {
25          System.out.println("Seminaire de maitrise en informatique");
26      }
27  }
```

Le graphe d'appel résultant du model discoverer Java de MODISCO est montré dans la Figure 3.6 :

- On note trois noeuds distincts étiquetés *c()*,
- Le noeud *Cours : :c()* n'apparaîtra jamais dans un graphe d'appel dynamique, puisque *Cours* est une classes abstraite (non-instanciable), et surtout
- Une discontinuité au niveau du graphe d'appel, qui ne tient pas compte du

fait que `Classemain::main(String[])` appelle, en réalité, `C_DepInfo::c()` et `Seminaire_mait_info::c()`

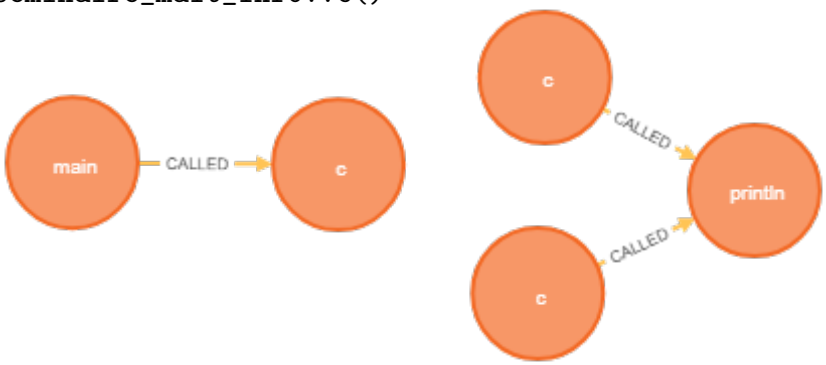


Figure 3.6: Problème avec les classes abstraites

3.4.4 Solution proposée

Comme les graphes d'appels partiels où le polymorphisme a été utilisé montrent, les méthodes déclarées par des interfaces et les méthodes abstraites ont été appelées. Cela ne pourrait jamais arriver. En fait, les méthodes redéfinies sont appelées lors de l'exécution.
Puisque c'est une question de hiérarchie des objets (superclasse et sous-classe / superinterface et sous-interface), nous avons décidé d'introduire une requête Cypher pour résoudre le problème du polymorphisme.

Chaque graphe d'appels statique a un sous-graphe qui lui correspond. Ça représente qui a implémenté/hérité qui. À partir de ce graphe, nous pouvons extraire l'ensemble des classes qui dépendent d'une classe, d'une classe abstraite ou d'une interface. Chaque nœud définit le nom complet de l'objet et son type ;soit `Class`, `Abstract` ou `Interface`. Par exemple (la figure 3.7), la classe A implémente l'interface I, la classe B hérite la classe A et la classe C hérite la classe B. Avec le graphe, nous pouvons conclure que l'interface I ou la classe A peut être utilisée

pour initialiser la classe C (par polymorphisme) et cela nous permettra de couvrir tous les scénarios possibles d'initialisation faite par le polymorphisme.

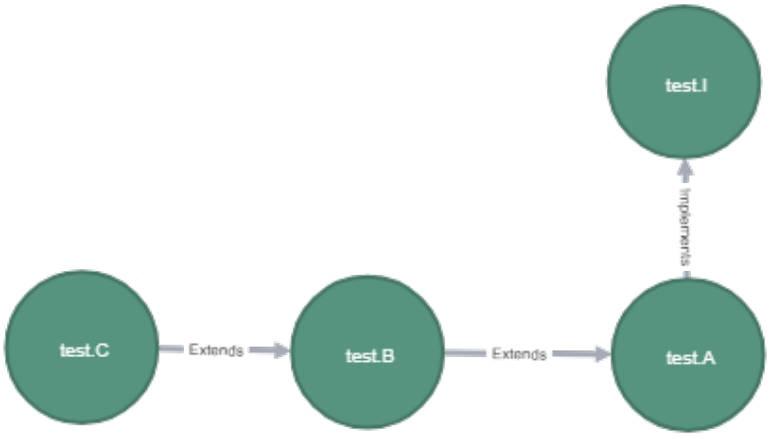


Figure 3.7: Exemple de hiérarchie des classes

Notre requête Cypher suit l'algorithme suivant :

Pour chaque(Classe X qui dépend de Y)

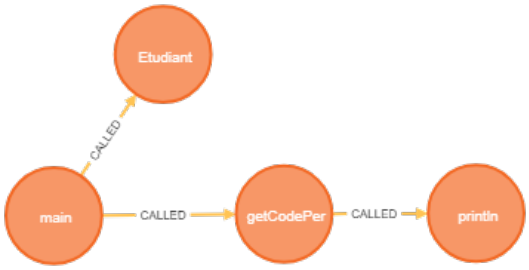
Si (∃ méthode a d'une classe A qui appelle méthode b d'une classe Y & une méthode avec le même nom existe dans la classe X)

Si la méthode b de la classe X et si la méthode n'a pas déjà été appelée par la méthode A.a

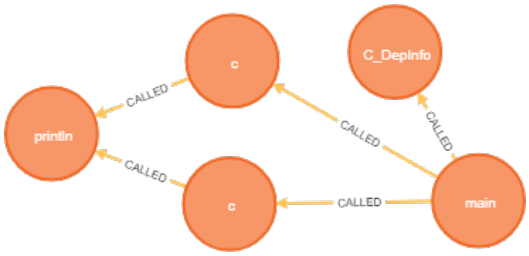
On ajoute l'appel avec les paramètres nécessaires pour cette méthode

```
Match (c3:Class)-[*1..]->(c1:Class),(n1:Node)-[r1:CALLED]->(n2:Node),(n3:Node)
Where n2.class=c1.name and  n3.class=c3.name and n1 <> n3 and
(c1.type="interface" or c1.type="abstract") with c3,c1,n1,n2,n3,r1
FOREACH (y in CASE WHEN (NOT exists((n1)-[:CALLED]->(n3))) THEN [1] ELSE [] END |
MERGE (n1)-[:CALLED{parameters:r1.parameters}]->(n3))
return *
```

Une fois que nous appliquons notre requête sur les graphes d'appels que nous avons dans les figures 3.5 et 3.6, on obtient le suivant :



(a) Interface et classes



(b) Classe abstraite et sous-classes

Figure 3.8: Comment le polymorphisme est présenté après notre intervention

3.5 Problématique des dépendances cachées (Hecht *et al.*, 2018)

Autre que les dépendances cachées des applications java discutée dans la section 1.1.3 et le fait que J2EE utilise plus d'un langage (comme nous l'avons mentionné dans 0.3, J2EE introduit encore plus de techniques de développement. Par conséquent, ils dévoilent plusieurs dépendances cachées. Beaucoup d'entre eux sont des dépendances de contrôle entre les composants du système et ses applications qui ne peuvent pas être facilement découvertes et trouvées (Hecht *et al.*, 2018).
Après avoir examiné les Entreprise Java Beans (EJB2) de J2EE et comment ils fonctionnent, (Hecht *et al.*, 2018) ont découvert que certaines dépendances échappent l'analyse statique à cause de la façon dont les composants J2EE s'appellent. Où ils utilisent l'inversion de contrôle (Johnson, 2005) (Hecht *et al.*,

2018) ; Un EJB exige que pour qu'un client puisse utiliser une de ses méthodes, il doit implémenter 2 de ces interfaces (selon la guide du développeur [3]). Les 2 interfaces sont les suivants :

1) "Home interface" : elle déclare les méthodes qui contrôlent le cycle de vie de l'EJB.

2) "Remote interface" : elle spécifie les méthodes métier disponibles à utiliser via l'EJB. pour que l'utilisateur les utilise, un appel des méthodes à distance doit être utilisé.

La figure 3.9 montre un exemple d'un EJB Customer où ses méthodes métier sont définies dans le côté serveur, la classe EJB `CustomerEJB`. Cet EJB a comme "Home interface"= `CustomerHome` et "Remote interface"= `Customer`

— ***Les services de gestion du cycle de vie des EJBs :*** J2EE gère ses beans de manière optimale pour minimiser les ressources, (Hecht *et al.*, 2018) disent. La figure 3.10 illustre les états d'un "Entity bean". Pour qu'il passe par les différentes étapes de sa vie (Null, Disponible Prêt), un ensemble de méthodes est appelé (**(1)** à **(5)**. Les méthodes en rouge sont appelées par le client et celles en bleu du côté serveur.

À titre d'illustration , si le client dans notre exemple de la figure 3.9 veut appeler la méthode `create()` pour utiliser l'EJB, il appelle le "Home interface" `CustomerHome_proxy` qui implémente `CustomerHome`. Notre graphe d'appel généré statiquement s'arrête ici, mais en réalité, cela a appelé la méthode ejbCreate() du côté serveur utilisant l'inversion de contrôle.

— ***L'appel des méthodes à distance "Remote Method Invocation" :***

3. Guide du dev : https ://docs.oracle.com/cd/A97329_03/web.902/a95881/overview.htm (Accès Août 2020)

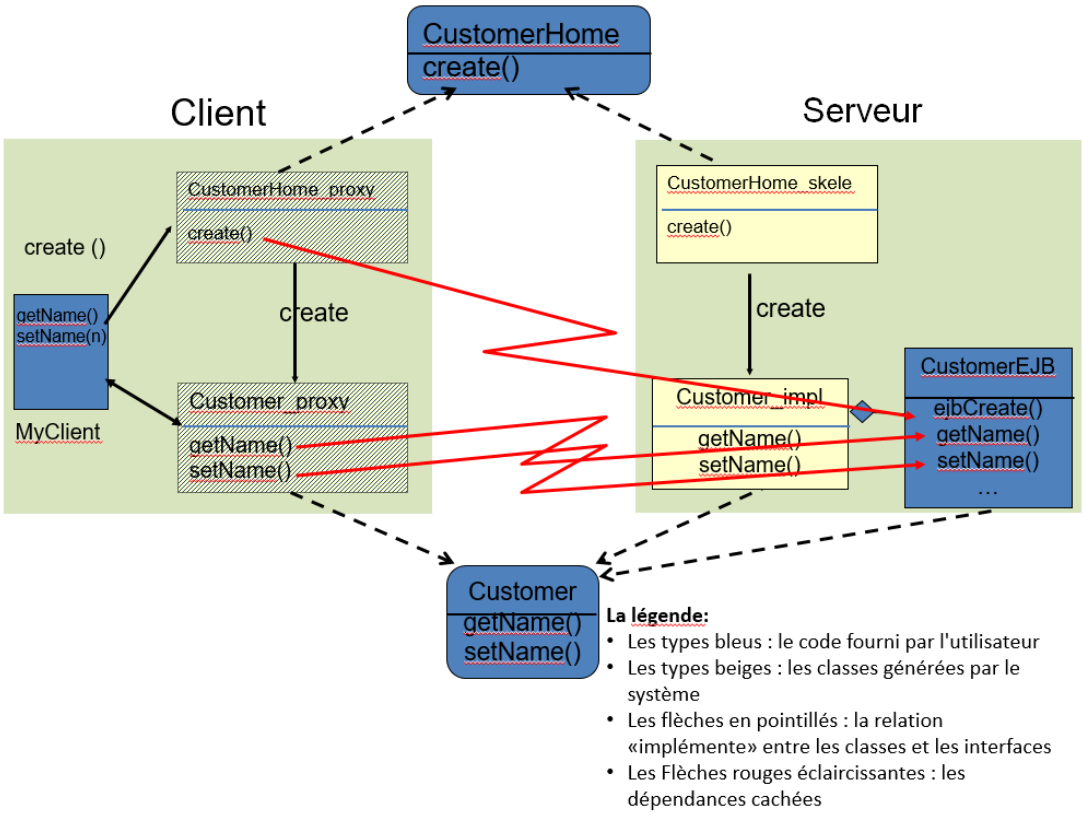


Figure 3.9: Exemple de dépendances cachées entre le client et le serveur d'un EJB.

Pour qu'une methode peut etre appeler, Le client doit Dans l'exemple qui se trouve dans la figure 3.9, pour qu'un client puisse utiliser une des méthodes de `Customer` qui sont déclarées dans `CustomerEJB`, ils doivent utiliser RMI où le code client doit implémenter des interfaces et des méthodes spécifiques qui va servir comme un intermédiaire (comme montre la figure 3.9 par les flèches rouges de getName() et setName()) pour bénéficier des services d'infrastructure. Par contre, ces appels ne sont pas explicites dans le code client(Fink *et al.*, 2004) (comme le montrent les flèches rouges sur la figure 3.9).

L'ajout des dépendances cachées découvertes : Après avoir spécifié les dé-

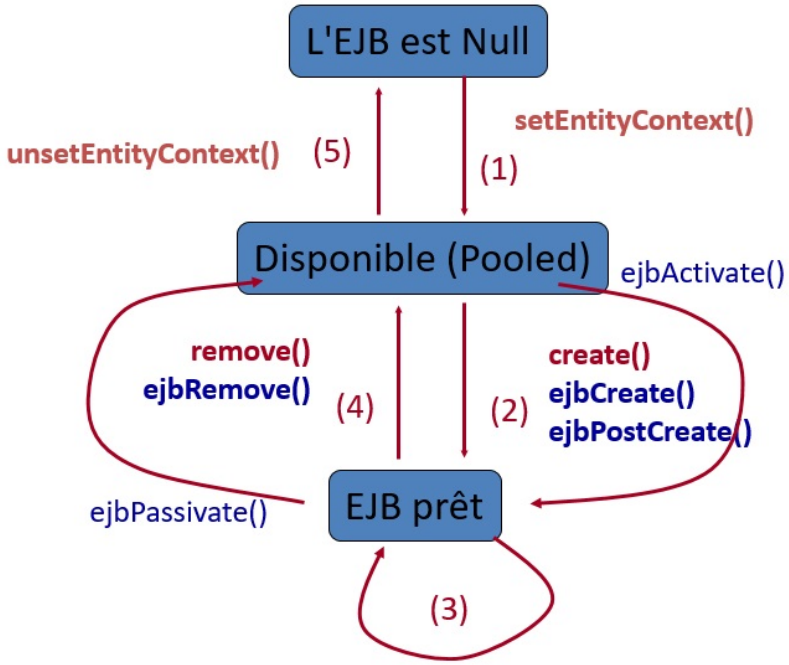


Figure 3.10: Cycle de vie d'un Entity bean

pendances de J2EE non détectés par l'analyse statique qui sont introduit par RMI et cycle de vie des EJB2, ils ont présenté 12 règles (figure 3.11) qui aident à les identifier et les ajouter aux autres dépendances déjà découvertes par KDM à l'aide d'un outil qu'ils ont introduit. Leurs règles fonctionnent avec des conditions, avec lesquelles ils ont pu déterminer où se produisent les dépendances cachées. Puis, ils déclarent le type d'action à prendre en fonction de la dépendance en question. Prenons à titre d'exemple les deux flèches rouges en bas dans la figure 3.9. Ce type d'appel invoqué utilisant RMI est découvert à l'aide du deuxième règle (figure 3.11). Ils disent que **pour chaque** méthode `foo` définit dans l'objet X qui implemente `RemoteInterface`, On ajoute une dépendance de X.foo() vers la même méthode (qui a le même nom et paramètres de la méthode en question) déclarée dans la classe EJB du même EJB de l'objet X. Alors, on va ajouter une dépen-

Rule	Condition: $\forall$ *type* T *and method* m *of* T *such that*	Action: Add call dependency from $T.m$ to
R1	$isHomeInterface(T), \quad m = create(paramlist)$	$EjbClass(EJB(T)).ejbCreate(paramlist)$
R2	$isRemoteInterface(T), \quad m = f(param_1, ..., param_i)$	$EjbClass(EJB(T)).m$
R3	$isHomeInterface(T), \quad m = remove(paramlist)$	$EjbClass(EJB(T)).ejbRemove(paramlist)$
R4	$isHomeInterface(T), \quad m = create(paramlist),$ $\exists\ method\ mpost = ejbPostCreate(paramlist)\ \epsilon\ EjbClass(EJB(T))$	$EjbClass(EJB(T)).ejbPostCreate(paramlist)$
R5	$isHomeInterface(T), \quad isEntity(EjbClass(EJB(T)),$ $m = create(paramlist)$	$EjbClass(EJB(T)).setEntityContext(ctx : EntityContext)$
R6	$isHomeInterface(T), \quad isSession(EjbClass(EJB(T)),$ $m = create(paramlist)$	$EjbClass(EJB(T)).setSessionContext(ctx : EntityContext)$
R7	$isHomeInterface(T), \quad m = remove(paramlist)$	$EjbClass(EJB(T)).unsetEntityContext()$
R8	$isRemoteInterface(T), \quad m = f(param_1, ..., param_i),$ $\forall\ obj\ instance\ of\ T\ s.t. isPassive(EJB(T))$	$EjbClass(EJB(T)).ejbActivate()$
R9	$isHomeInterface(T), \quad m = create(paramlist),$ $\forall\ obj\ instance\ of\ T\ s.t.\ isActive(EJB(T)), isPoolSizeLow(SERVER)$	$EjbClass(EJB(T)).ejbPassivate()$
R10	$isRemoteInterface(T), \quad isEntity(EjbClass(EJB(T)),$ $\forall\ obj\ instance\ of\ T\ s.t.\ isPassive(EJB(T))$	$T.ejbLoad()$
R11	$isRemoteInterface(T), \quad isEntity(EjbClass(EJB(T)),$ $m = create(paramlist), \quad \forall\ obj\ instance\ of\ T\ s.t.\ isActive(EJB(T))$	$T.ejbStore()$

Figure 3.11: Les règles présentées dans (Hecht *et al.*, 2018) qui ajoutent des dépendances cachées de RMI et de la gestion du cycle de vie des beans

dance de `Customer_proxy.setName()` vers `CustomerEJB.setName()`.

3.6 Implémentation

3.6.1 Notre outil

Après avoir générer les méta-modèles des applications désignées avec Modisco (figure 3.2), nous passons à l'extraction des dépendances qui se trouve dans les fichiers KDM. Pour ce faire, nous avons construit un outil basé sur Eclipse Modeling Framework (EMF) [4]. Grâce à EMF , nous avons pu accéder aux librairies qui nous ont permis d'analyser et d'accéder ces méta modèles générés où se trouve les appels des méthodes, à quelle classe ils appartiennent, les paramètres requis pour les invoquers, les dépendances des classes et leurs types, etc. L'utilisation d'un cadre logiciel qui a le support d'une large communauté telle que EMF, améliore la fiabilité et la stabilité des outils qui l'utilise (Bruneliere *et al.*, 2014).

4. EMF : https ://www.eclipse.org/modeling/emf/ (Accès Août 2020)

D'un autre côté, Pour les applications J2EE, nous devons d'abord ajouter certaines des dépendances cachées avant de passer à l'extraction. Dont nous avons parlé dans 3.5 (comme illustré avec "Insertion des dépendances cachées" sur la figure 3.1).

Après l'extraction, l'outil effectue également le filtrage tel que la suppression des appels de librairies et aussi la correction et l'ajout de dépendances manquées par MoDisco qui ont été discuté dans les sections précédentes. Pour l'élimination des appels ne nous servira pas, nous ignorons l'appel dès que nous vérifions si les deux parties (l'invocateur et l'invoqué) n'appartiennent pas à l'application. Similaire à (Ali et Lhoták, 2012), nous garderions toujours l'appel où une méthode d'application appelle une méthode de librairie puisque l'invocateur ici appartient a l'application. Après avoir toutes les données concernant les appels, l'outil passe à la préparation du graphe. L'outil suit le format que nous avons présenté dans la figure 2.1 pour simplifier la comparaison en exportant les dépendances vers des fichiers JSON.

Avant de commencer la comparaison de graphes et après l'importation des graphes, nous corrigeons les problèmes causé par le polymorphisme avec Neo4J qui a été discuté dans la sous section 3.4.4.

3.6.2 Outil de Geoffrey

L'outil introduit dans (Hecht *et al.*, 2018) et utilisé pour découvrir et ajouter les dépendances cachées des applications J2EE, a été employé dans certains des projets J2EE sur lesquels nous avons travaillé. Ça compte également sur les artefacts produits par MoDisco (les fichiers KDM) et leurs implémentations des métamodèles spécifiques à J2EE afin de capturer les artefacts qui ne sont pas liés à JAVA. Donc, il prend comme fichiers d'entrée et paramètres le fichier KDM original généré par MoDisco, le code source de l'application J2EE, les descripteurs

de déploiement des EJBs, etc.

Pour appliquer les règles présentées, ils ont utilisé un moteur basé sur des règles appelé Drools [5]. Drools enregistre les règles comme un ensemble de <Condition, Action>. Il parcourt le modèle KDM en essayant de trouver les méthodes qui remplissent les conditions. S'il en trouve une, l'action consiste à ajouter un appel au métamodèle en suivant la règle. Une fois que tout le KDM est analysé, il génère un fichier KDM modifié qui contient les dépendances découvertes.

3.7 Exemple

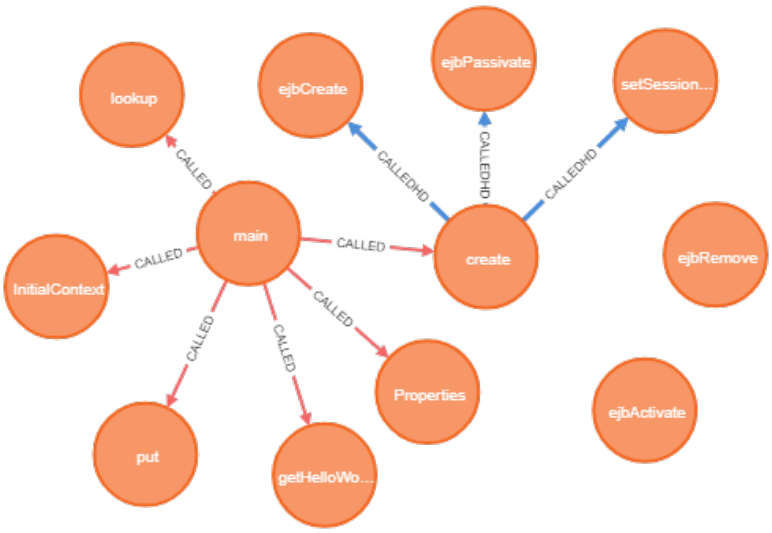


Figure 3.12: Le graphe d'appel statique généré à l'aide de notre outil et de l'outil (Hecht *et al.*, 2018) qui ajoute les dépendances cachées de J2EE

Dans cette section, nous prendrons le code source qui se trouve dans l'appendice A comme un exemple. C'est une simple application J2EE qui fait appel aux services d'un bean *Session* (EJB2). Le graphe d'appel correspondant, que nous avons

5. Drools : https ://docs.jboss.org/drools/release/latest/ (Accès Août 2020)

généré et importé dans Neo4j se trouve dans la figure 3.12. Nous avons utilisé l'outil mentionné dans 3.5 pour ajouter les dépendances spécifiques à J2EE (les appels en bleu dans le graphe) et notre outil pour les graphes d'appels statiques pour nous assurer que nous n'aurons aucun des problèmes abordés dans les sections précédentes.

Dans le chapitre suivant, nous décrirons les étapes et les interventions que nous avons effectuées afin d'avoir un graphe d'appel dynamique prêt à être comparé avec le graphe statique généré en utilisant les démarches dont nous avons parlé ici.

CHAPITRE IV

ANALYSE DYNAMIQUE

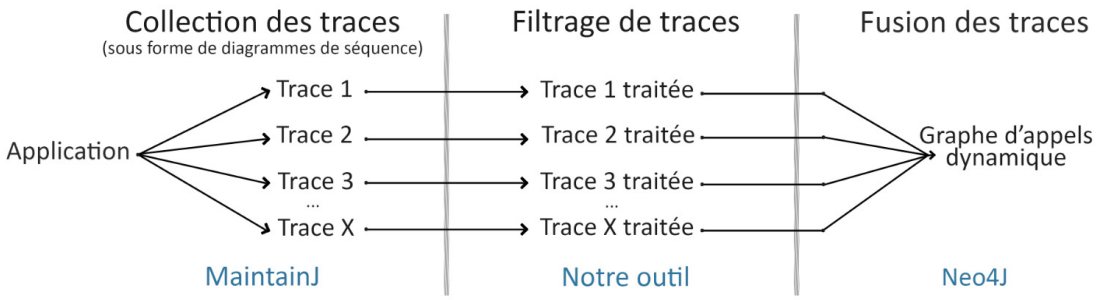


Figure 4.1: Les différentes étapes pour générer un graphe d'appel dynamique

4.1 Environnement technique : MaintainJ

Pour obtenir des graphes d'appels dynamiques, nous avons utilisé MaintainJ [1]. Il s'agit d'un outil de rétro-ingénierie dynamique dont l'objectif est d'aider à comprendre, déboguer et surveiller les applications basées sur Java. Pour qu'il capture les traces des applications java/j2ee durant l'exécution, il doit être installé en tant qu'agent java "Javaagent" pour la JVM. Les agents Java utilisent des services qui leur permettent de surveiller et de modifier les applications exécutées sur la JVM en cours de l'exécution, selon la documentation Java.

1. MainainJ : http ://www.maintainj.com/ (Accès Août 2020)

Comme nous avons discuté dans la section 1.2.4, l'action de traçage générerait plusieurs fichiers où chaque capture de trace représente un cas d'utilisation. Nous devons donc essayer de couvrir autant de scénarios d'utilisation que possible. Les fichiers produits par l'outil de traçage utilisé sont présenté sous la forme de diagramme de séquence UML2 (exemple des traces générées par MaintainJ dans la figure 4.2). Pour avoir les traces prêtes à importer dans Neo4j en suivant notre format pour les graphes d'appels de la figure 2.1, nous avons dû faire de l'ingénierie inverse des diagrammes de séquence UML2 générés. Pour ce faire, nous avons réalisé un outil qui prend en entrée les diagrammes de séquence générés. Il analyse et traite les traces qu'ils contiennent. Puis les exporter vers des fichiers JSON prêts à être importés dans Neo4j. La figure 4.1 illustre les différentes phases et étapes à suivre pour obtenir un graphe d'appels généré dynamiquement.

4.1.1 Système déployé sur plusieurs JVM

En général, les applications Java sont les plus faciles à extraire des traces. Mais comme nous l'avons mentionné précédemment, les composants du système peuvent être dispersés autour de différents niveaux, en particulier dans le cas de J2EE. Notez bien qu'on peut avoir des applications J2EE en phase de développement ou de test qui se trouvent sur une seule machine virtuelle de Java.

MaintainJ permettra non seulement de capturer les traces d'exécution des applications qui s'exécutent dans une seule JVM. Mais aussi les applications qui en utilisent plusieurs. Dans ce cas, MaintainJ devra être installé sur toutes les JVM. Puis, nous devrons spécifier les adresses IP et les ports des machines sur lesquelles nos composants ont été déployés. Ensuite, Maintainj fusionnera automatiquement les traces capturées.

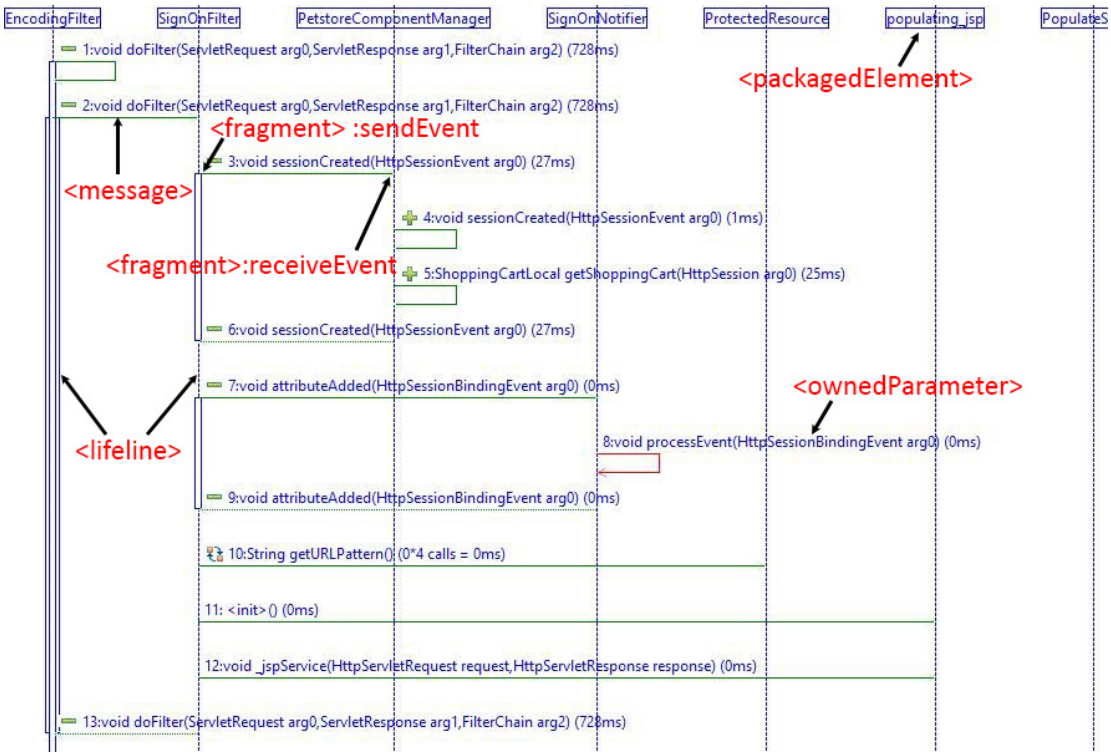


Figure 4.2: un exemple de traces MaintainJ au format de diagramme de séquence UML

4.1.2 Analyse des diagrammes de séquence - les fichiers XMI des traces

Les diagrammes de séquence UML peuvent représenter des interactions complexes entre les composants du système(Kundu *et al.*, 2012). MaintainJ les utilise pour enregister les traces capturées dans des fichiers XML Metadata Interchange 2.1 (XMI[2]).Ces fichiers sont conforme aux normes et spécifications de l'OMG pour UML (figure 4.2). Afin d'avoir les traces dans Neo4j pour faire la comparaison, nous avons construit un outil qui analyse des diagrammes de séquence UML et les transformer en des fichiers JSON par une ingénierie inverse. Les fichiers de sortie sont dans le même format de notre outil présenté dans la section 3.1 qui fait

2. XMI 2.1 (2004) :https ://www.omg.org/spec/XMI/2.1/About-XMI/ (Accès Août 2020)

l'analyse statiques (des fichiers KDM aux fichiers JSON). Notre outil prend en charge les fichiers XMI 2.2 et plus récentes. Mais, puisque MaintainJ génère des fichiers XMI 2.1 datant de 2004 , ils doivent alors être transformés en une version plus récente (vers XMI 2.4.1 [3] dans la figure 4.2) en utilisant des outils tiers (i.e IBM Rational Software Architect [4]).

La figure 4.3, montre un exemple du fichier de diagramme de séquence contenant des traces. En suivant la spécification de XMI, les éléments de modèle dans (Alalfi *et al.*, 2009) qui fait la capture des traces pour les applications web PHP et (Kundu *et al.*, 2012) qui transforme les diagramme de séquences vers des graphes de flot de contrôle, nous avons pu connaître chaque chaque balise `<balise>` XMI représente quoi dans le diagrammes de séquence. Les balises sont illustrée dans la figure 4.2. Elles sont écrites en rouge et elles pointent vers l'élément correspondant dans le diagramme.

4.2 Problématique de l'exclusion de package de l'analyse, et en particulier, les librairies

MaintainJ permet la capture et l'exclusion par package. Cela signifie qu'avant la capture, nous pouvons spécifier les paquets que l'agent java de MaintainJ capturerait et ceux à ignorer. Puisque nous essayons de rapprocher les deux graphes le plus possible l'un de l'autre et que dans l'analyse statique nous avons choisi de ne pas nous concentrer sur les appels des librairies, nous devons faire le même

---

3. Specification de XMI 2.4.1 (2013) : https ://www.omg.org/spec/XMI/2.4.1/About-XMI/ (Accès Août 2020)

4. IBM RSA : https ://www.ibm.com/ca-en/marketplace/rational-software-architect-designer/details (Accès Août 2020)

```
<?xml version="1.0" encoding="UTF-8"?>
<uml:Model xmi:version="2.4.1" xmlns:xmi="http://www.omg.org/spec/XMI/2.4.1" xmlns:xsi="http://www.w3.org/200
  <packagedElement xmi:type="uml:Collaboration" xmi:id="_Qsse3WVEEemrZNJuFG45Pw" name="Collaboration">
    <ownedBehavior xmi:type="uml:Interaction" xmi:id="_Qsse3mVEEemrZNJuFG45Pw" name="Interaction">
      <lifeline xmi:type="uml:Lifeline" xmi:id="_Qsse32VEEemrZNJuFG45Pw" represents="_QstH1GVEEemrZNJuFG45Pw"
       ...
      <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="_Qsse5GVEEemrZNJuFG45Pw" name="doGetsta
       ...
      <message xmi:type="uml:Message" xmi:id="_QstG2WVEEemrZNJuFG45Pw" name="CartHandler" messageSort="reply"
       ...
    </ownedBehavior>
    <ownedAttribute xmi:type="uml:Property" xmi:id="_QstH1GVEEemrZNJuFG45Pw" type="_QstH2WVEEemrZNJuFG45Pw"/>
      ...
  </packagedElement>
  <packagedElement xmi:type="uml:Class" xmi:id="_QstH2WVEEemrZNJuFG45Pw" name="com.sun.j2ee.blueprints
    <ownedOperation xmi:type="uml:Operation" xmi:id="_QstH2mVEEemrZNJuFG45Pw" name="doGet">
      <ownedParameter xmi:type="uml:Parameter" xmi:id="_QstH22VEEemrZNJuFG45Pw" name="arg1" type="_QstH_mVEEe
      <ownedParameter xmi:type="uml:Parameter" xmi:id="_QstH3GVEEemrZNJuFG45Pw" name="arg2" type="_QstH_2VEEe
    </ownedOperation>
    ...
</uml:Model>
```

Figure 4.3: un exemple qui montre les éléments utilisés pour représenter un diagramme de séquence dans un fichier XMI

en analyse dynamique. Donc, nous spécifions que le package de l'application en question.

4.3 Comportement des classes imbriquées dans la JVM

Par défaut, tout le code java qui est écrit dans une classe sera transformé en bytecode qui sera placé dans un fichier spécifique à cette classe (i.e des fichiers `.java` aux fichiers `.class`). Mais à partir de la version 1.1 de Java, les classes imbriquées ont été introduite. Donc, Pour maintenir la compatibilité des anciennes versions de la machine virtuelle Java (JVM), le compilateur les sépare de la classe externe (Arnold *et al.*,) (Dmitriev, 2002). Il enregistre leur bytecode java dans leurs propre fichiers (`.class`) et non pas avec les classes dans lesquelles elles ont été déclarées (Classes externes) en ajoutant un suffixe au nom de la classe externe aux contient

Après cette étape, la JVM modifie les noms des méthodes, noms des classes et

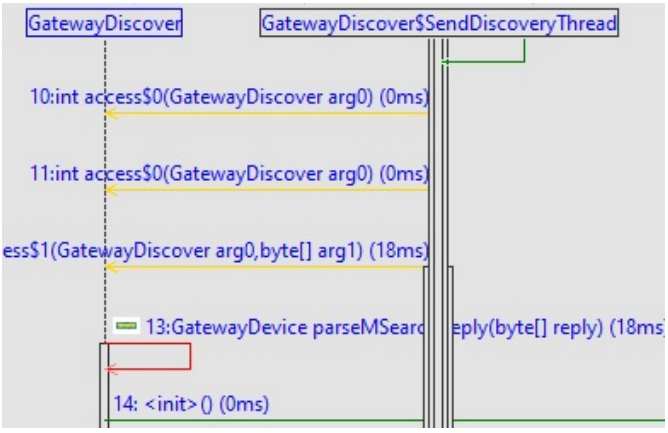


Figure 4.4: un exemple de traces d'une classe utilisant une classe imbriquée

ajoute même de nouvelles méthodes en fonction de ce qui s'est passé (Arnold *et al.*,). Comme illustré par l'exemple dans la figure 4.5, elle ajoute un signe dollar `$` et un entier. Alors, pour procéder à inverser les changements effectués lors de la compilation, nous devons savoir quelle est la signification de chaque modification et comment reconnaître chacune d'elles. Nous avons pu le faire grâce à la documentation de java et les expressions régulières (regex).

— Les classes, classes abstraites et interface : On utilise l'expression régulière `"[a-zA-Z-0-9.]+$"` pour détecter que le nom n'a pas été changé. c'est-à-dire ne contient pas de `$`.

— Les classes imbriquées : On utilise l'expression `".*\$\D+\w*$"` pour trouver le modèle `ClasseExterne$Classeimbriquée`. Elles auront un signe dollar `$` entre la Classe externe et la classe imbriquée. notre outils change le signe `$` par un point. Revenons à notre exemple, `GatewayDevice$SendDiscoveryThread` devient `GatewayDevice.SendDiscoveryThread`.

— Les classes locales : On les détecte utilisant `".*\$\d+\D+\w*$"`. Elles auront un entier aléatoire N entre le signe dollar et le nom de la classe

imbriquée `ClasseExterne$Entier+ClasseLocal`. Notre outil supprime le $+Entier afin d'annuler les modifications.

— Les classes anonymes : On les détecte utilisant `".*\$\d+$"`.Elles auront un entier après la classe externe `ClasseExterne$N`. Pour ces classes, notre outil ne les inclut pas dans le graphe d'appel car elles sont détectées comme "Anonymous Class" dans KDM, donc nous ne pouvons pas les comparer.

— Les méthodes des classes imbriquées : elles auront le nom de la classe imbriquée avant le nom de la méthode `ClasseImbriquée$méthode`. Dans ce cas, nous gardons juste le nom de la méthode.

— les méthodes Access : Si nous avons des variables de la classe externe déclarée de type *private*, les classes internes n'y auront pas accès puisqu'elles sont séparées. Dans ce cas, le compilateur ajoute des méthodes synthétiques nommées `access$Entier` qui ne sont pas écrites par le développeur (Lai, 2008). Ces méthodes permettent aux classes internes d'accéder aux variables privées déclarées dans la classe externe (i.e la méthode access$0 sur la figure 4.5). Notre outil ne tient pas compte de ces méthodes.

4.4 Exemple

Nous appliquons, dans cette section, notre outil présenté dans ce chapitre avec MaintainJ pour vous montrer un exemple de graphe d'appel dynamique. Notre exemple est constitué d'une application java simple qui implémente une classe imbriquée. La figure 4.5 illustre le graphe d'appels dynamique importé dans Neo4j du code source B.

Dans le chapitre suivant, nous discuterons de la façon dont nous comparons nos graphes d'appels après avoir importé les graphes statiques et dynamiques dans Neo4j.

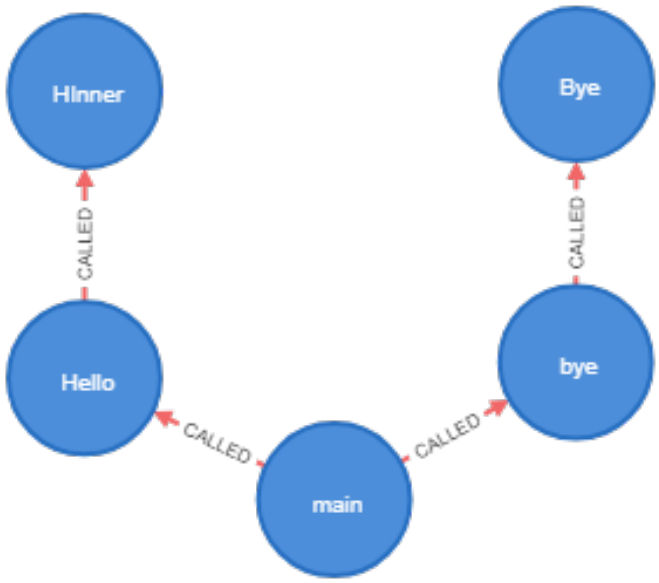


Figure 4.5: Graph d'appels dynamique du code dans l'annexe B généré avec notre outil.

CHAPITRE V

VALIDATION DES DÉPENDANCES CACHÉES PAR COMPARAISONS DES GRAPHES D'APPEL STATIQUES ET TRACES DYNAMIQUES

Le but ultime de ce projet est de valider les dépendances cachées, qui représentent les services d'infrastructure (voir Section 0.3 et Chapitre 3), avec les traces d'exécution. En particulier, nous voulons nous assurer, empiriquement, de ne pas avoir manqué de dépendances inhérentes aux services d'infrastructure de la plateforme cible.

Les graphes d'appel statiques, que l'on construit généralement pour diverses analyses d'impact (voir Section 1.1), sont typiquement construits de façon *conservatrice*, c-à-d, on va s'assurer d'inclure tous les appels que la structure du code permet d'envisager, pour être sur de ne pas en manquer. Ce faisant, on va souvent inclure des appels qui ont peu ou pas de chance de se produire en réalité, par excès de prudence et par manque d'information. Par contre, une trace dynamique va, *par définition*, montrer les appels spécifiques *à une exécution donnée*, pour un état donné de l'environnement, des données spécifiques saisies par l'usager, un ensemble limité de scénarios, etc. Donc, en principe, le graphe d'appel résultant des traces dynamiques devra être *un sous-graphe du graphe d'appel statique*. Or, justement, toute la problématique des dépendances cachées est que ces dépendances échappent à l'analyse statique du "code source de l'usager", et que nous

avons ajoutée, programmatiquement, grâce à des règles qui encodent les services d'infrastructure. Donc, de premier abord, pour valider la complétude des règles encodant les services d'infrastructure, *nous devons nous assurer que les graphes d'appels dynamiques sont inclus dans les graphes d'appels statiques*.

En pratique, outre la problématique des dépendances cachées, comme nous l'avons vu dans les chapitres 2 et 4, la construction des graphes d'appels statiques est problématique pour d'autre raisons.

5.1 Méthodologie pour la validation des dépendances cachées

Notre méthodologie pour comparer les graphes d'appels et valider les dépendances cachées ajoutées peut être représentée en deux objectifs :

- O_1 : Nous validons empiriquement les dépendances cachées en vérifiant que les règles les ont identifiées toutes, à travers les exécutions tentées.
- O_2 : Nous établissons des profils d'applications selon la nature et la prépondérance des dépendances cachées.

Pour l'objectif O_1, à la base, il faut s'assurer que le graphe dynamique est inclus dans le graphe statique. Si ce n'est pas le cas, par exemple des appels qui existent dans le graphe dynamique mais pas dans le graphe statique, on peut avoir deux raisons principale :

- Lorsqu'un nœud (une méthode) présent dans le graphe dynamique et manquant dans le graphe statique.
- Les nœuds sont présents, mais un appel entre les deux n'est pas présent. Cela veut dire que les méthodes sont couvertes par l'analyse statique mais elles ont été invoquées d'autres chemins que le chemin décrit par les traces d'exécution.

Une méthode ne soit pas détectée dans le graphe d'appels statique, mais en réalité

elle a bien été invoquée (par capture de trace), signifie que la méthode en question a été appelée par réflexion et elle appartient à un autre projet. Par contre, comme nous avons mentionné dans les sections 3.6.1 et 4.1, nous excluons tous les appels qui ne sont pas couverts par le code usager. Donc, cela ne devrait pas arriver. Nous devons également mentionner qu'il est possible que d'autres applications utilisent le même nom de package que le code utilisateur. Or dans Eclipse, Modisco fait l'analyse du code par projet. Cela explique l'absence des méthodes de nos graphes statiques alors qu'elles sont présentes dans le graphe d'appels dynamique. Ainsi que dans la JVM, ce code externe qui partage le même nom qualifié est devenu inclus avec notre projet analysé lors de compilation (i.e sous forme des fichiers .jar). Après, quand on spécifie les noms de packages à tracer dans MaintainJ, l'outil ne pourra pas les distinguer et va les inclure dans les traces.

Pour ce qui est de la présence des nœuds (méthodes) et l'absence des liens entre eux, justement, cela est possible dans deux situations :

— Des erreurs dans la construction du graphe statique

— Des dépendances cachées manquées que nous aurions manquées.

Les erreurs dans la construction du graphe statique peuvent provenir de plusieurs sources : 1) reflexion : une classe chargée et invoquée programmatiquement avec le réflexion de package, ou alors 2) des inférences erronées ou incomplètes dans la construction du graphe statique où le polymorphisme a été utilisé. Par exemple, si on a :

```
class A  extends ... implements ... {
 //A est une sous classe d'une classe
 //qui implémente l'interface IEtudiant
    public void f(...) {
        IEtudiant monEtudiant = getMeNewEtudiant() ;
        monEtudiant.setProgramme(pgm) ;
```

```
    }
}
```

Puis, On a deux classes qui implantent IEtudiant ; AEtudiant et BEtudiant. Normalement, la règle qu'on a introduit dans la section 3.4 rajoute les liens :

— A : :f() − > AEtudiant.setProgramme()

— A : :f() − > BEtudiant.setProgramme()

Si On oublie le deuxième lien et qu'a l'execution, getMeNewEtudiant() retourne, justement, un BEtudiant, le lien va être visible à l'exécution, mais pas dans le statique.

Donc, le premier « test » de l'inclusion va permettre de valider notre construction du graphe statique. L'autre situation est, évidemment, les dépendances cachées manquées par nos règles. On ne peut distinguer les deux « simplement », sauf par inspection manuelle.

Donc, Nous proposons un processus itératif :

1. On fait un test d'inclusion
   (a) A chaque fois qu'un lien ou nœud est présent dans le dynamique et absent du statique
   (b) On examine la raison de la contradiction
      i. Soit on corrige la construction du graphe statique s'il y a un problème.
      ii. Ou alors on ajoute ou exclue un package de l'analyse dans MaintainJ.
      iii. Ou alors on modifie les règles d'ajouts de dépendances cachées.
   (c) Répéter.

Une fois que le test d'inclusion est complété, on peut passer à l'objectif O2.

Dans un premier temps, nous allons utiliser ce protocole expérimental pour valider les règles de construction du graphe d'appel statique, décrites dans le chapitre 4.

Pour l'objectif O2, toute comparaison, pour fin de profilage, doit se limiter à la portion du graphe d'appel statique qui est potentiellement accessible par une exécution donnée. Ce que on propose c'est si on peux « profiler » les applications, on doit comparer, non pas tout le graphe statique avec tout le graphe dynamique, mais plutôt « naviguer » le graphe statique à partir de la trace dynamique, et faire la comparaison « site d'appel » par « site d'appel », mais en se limitant aux sites d'appel effectifs (i.e. les méthodes qui ont été appelées dans les traces). Ça nous donne un sous-graphe du graphe statique ∩ le graphe dynamique.

5.2 Valider les règles de construction du graphe d'appel statique

5.2.1 Données expérimentales

Pour nous assurer que nous construisons des graphes d'appels corrects et représentatives, nous avons d'abord commencé à essayer nos outils sur des applications java. Les applications expérimentées sont mentionnées dans le tableau 5.3

Nom d'application	Version Java	Lignes de code
Weupnp	1.6+	1736
LuceneIndexTutorial	1.7+	681
jreversepro	1.8+	8534
JHotDraw	1.8+	32123
apache log4j 1.2.17	1.8+	34345

Tableau 5.1: Les caractéristiques des applications java utilisées

5.2.2 Résultats complets

	NMD	**NMC**	**PMC**	**AD**	**AC**	**PAC**
Weupnp 0.1.2	46	45	97.83	52	46	88.46
LuceneIndexTutorial	18	18	100	17	15	88.23
jreversepro	150	139	92.67	165	131	79.39
JHotDraw 7.0.6	600	547	91.17	1215	523	43.05
apache log4j 1.2.17	480	473	98.54	690	565	82.90

Tableau 5.2: Test d'inclusion du graphe dynamique dans le graphe statique des applications Java et J2EE*.

Avec :

NMD : Nombre de méthodes dans le GAD.

NMC : Nombre de méthodes du GAD qui se trouve dans le GAS.

PMC : Pourcentage de méthodes communes.

AD : Appel capturé dans le GAD.

AC : Appel de méthode commun dans le GAD trouvé dans le GAS.

PAC : Pourcentage des appels communs.

5.3 Valider les dépendances cachées

Dans cette section, nous nous concentrerons sur les dépendances cachées spécifiques à J2EE où nous testerons l'outil présenté dans (Hecht *et al.*, 2018) et vérifierons les dépendances qu'ils ajoutent. nous nous sommes concentrés sur les applications légataires qui utilisent EJB2.x de J2EE.

5.3.1 Données expérimentales

Application	Version JAVA	Version J2EE	LOC	Serveur de déploiement
Petstore 1.3.2	1.4	1.3	25951	jboss-3.2.3/4.2.3
Enroller	1.6	1.5	1312	Glassfish3
SavingsAccount	1.6	1.5	741	Glassfish3

Tableau 5.3: les applications J2EE utilisées dans la phase expérimentale

5.3.2 Résultats préliminaires

Comme nous l'avons mentionné dans la section 3.5, (Hecht *et al.*, 2018) ont introduit des règles qui ajoutent des dépendances cachées si certaines conditions existent. Dans cette sous-section, nous allons utiliser leur outils avec des application J2EE. Nous allons après discuter la précision des les résultats obtenus par rapport à ce qui se passe durant l'exécution.

Petstore 1.3.2 **:**

	Dépendances non détectée par l'outil	Dépendances validées	Dépendances non validée *	Pourcentage de validation
Régle 1 : ejbCreate()	1	5	3	62.5
Régle 3 : ejbRemove()	1	0	0	N/A
Régle 5 : ejbPostCreate()	0	1	0	100
Régle 6 : setSessionContext()	1	2	2	50

Tableau 5.4: Résultats de la validation de l'outil de dépendance cachée spécifique à j2ee

* : Dépendances non validée signifie que la méthode et son invocateur existent

dans le graphe d'appel dynamique mais d'après les traces, l'appel n'existe pas (une autre méthode a fait appel).

5.3.3 Résultats complets

***Petstore 1.3.2* :**

À compléter.

***Enroller* :**

À compléter.

***SavingsAccount* :**

À compléter.

5.4 Analyse des résultats

Figure 5.1: Les nœuds non communs de apache log4j 1.2.17

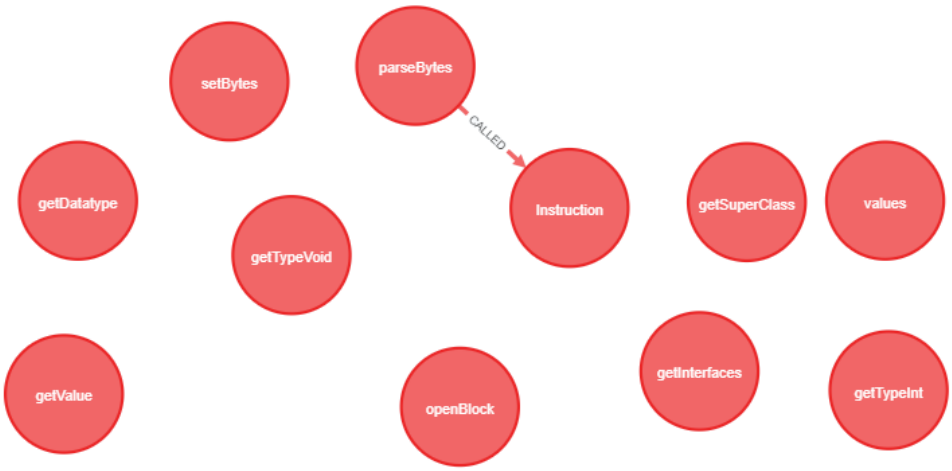


Figure 5.2: Les nœuds non communs de jreversepro

Les figures 5.1 et 5.2 montrent les méthodes du *GAD* qui ne peuvent pas être trouvées dans le *GAS* pour respectivement apache log4j et jreversepro. En regardant le code source de log4j, nous remarquons que ce sont tous des constructeurs. Certains sont sans paramètres (par exemple : `org.apache.log4j.helpers.PatternParser.LiteralPatternConverter LiteralPatternConverter()` et `org.apache.log4j.ConsoleAppender ConsoleAppender()`). Après avoir vérifié les classes correspondantes, nous avons constaté qu'ils n'étaient pas déclarés. En fait, ces méthodes ont été implicitement ajoutées par la JVM pour instancier ces classes sans paramètres.

Pour le reste des méthodes, nous avons remarqué que ces constructeurs n'existaient pas dans le GAS parce qu'ils avaient un paramètre supplémentaire. Selon la JLS [1], ce sont des paramètres synthétiques et implicites, ajoutés par le compilateur aux classes internes afin de représenter les classes externes. par exemple la mé-

1. JLS : https ://docs.oracle.com/javase/tutorial/reflect/member/methodparameterreflection.html (Accès Décembre2020)

thode `Processor. (org.apache.log4j.chainsaw.MyTableModel)` de la classe `org.apache.log4j.chainsaw.MyTableModel.Processor`

À compléter.

CONCLUSION

5.5 Résumé

La migration des applications légataires vers des architectures orientées services peut aider à économiser des ressources en permettant la réutilisation de l'ancien code, qui est entièrement fonctionnel, sous la forme des microservices. L'une des étapes clés consiste à identifier les lignes de codes, les méthodes et les classes qui peuvent servir comme des microservices potentiels. Mais, l'une des plus grandes contraintes qu'on peut rencontrer est de savoir comment les différentes fonctionnalités du système sont couplées, quelles lignes de code font quoi et appartiennent à qui. Une approche pour y parvenir consiste à effectuer une analyse statique des applications où nous pouvons identifier le type de service qu'elles peuvent servir en fonction de la manière dont elles interagissent avec le reste des composants de l'application. Une autre contrainte ici est que certaines techniques et technologies de programmation en java et j2ee modifient le comportement de l'application lors de l'exécution. Ce qui rend le graphe d'appel généré à l'aide de l'analyse statique plus complexe et moins représentatif.

Notre approche consiste à créer des outils qui génèrent des graphes d'appels statiques et dynamiques pour les applications Java et J2EE. Puis, à les analyser et les comparer dans Neo4J. Où pour l'analyse statique, nous avons essayé de rendre le graphe d'appel plus représentatif en ajoutant les méthodes manquées par Modisco, les appels dans les blocs statiques et d'initialisations, les appels modifiés par le polymorphisme et les dépendances cachées spécifiques à J2EE liées aux cycles de vie des EJB et RMI (l'outil présenté dans (Hecht *et al.*, 2018) a été utilisé).

Pour les graphes d'appels générés dynamiquement, nous capturons des traces de programmes sous forme de diagrammes UML2. Ensuite, notre outil les transforme en graphe d'appels grâce à un processus de rétro-ingénierie des diagrammes de séquence. En faisant ça, nous avons essayé d'inverser les changement de nom des classes, méthodes et paramètres effectué par le compilateur afin de permettre une comparaison entre les deux graphes d'appel dans Neo4j.
Pour valider notre méthodologie, nous avons utilisé notre approche avec plusieurs applications Java et J2EE. Nous avons généré leurs graphes d'appels et les importés dans Neo4j pour la comparaison. Ensuite, l'étape suivante était l'implémentation de nos métriques de comparaison dans Neo4j.
Nous avons d'abord essayé de vérifier empiriquement le pourcentage de couverture entre nos graphiques statiques et dynamiques.Nous avons essayé de nous concentrer sur le fait d'avoir un graphe dynamique inclus dans le graphe statique. Pour les applications J2EE et les dépendances cachées, nous avons dans un premier temps essayé de vérifier s'ils étaient trouvés dans le graphe d'appels dynamique. Par la suite, nous avons essayé de les diviser et de les confirmer règle par règle.

5.6 Contributions

À travers ce mémoire, Nous étions capables de :

- Développer une méthodologie pour construire et valider les constructions de graphes d'appels statiques (en utilisant KDM de Modisco) et dynamiques (en utilisant les traces d'exécution de MaintainJ).
- Une méthodologie pour transformer les diagrammes de séquence UML2 qui contiennent des traces en un graphe d'appel. Cet processus de rétro-ingénierie spécifique aux traces d'exécution, est inspiré par une une méthodologie provenant de l'état de l'art.
- Mettre en évidence les difficultés de comparaison des graphes d'appels sta-

tiques et des graphes d'appels dynamiques.

5.7 Directions futures

Cette recherche peut être élargie pour répondre ou aider à répondre multiple autre questions de recherche. Où elle peut être amélioré en visant à obtenir une inclusion complète des graphes d'appels statiques et dynamiques. Nous pouvons commencer par inverser les modifications apportées par les paramètres synthétiques et implicites ajoutés. Nous pouvons également ajouter la possibilité de détecter automatiquement les appels avec réflexion en utilisant le processus de la récupération d'informations (Information Retrieval) sur le code source. Ensuite, en identifiant et en masquant ces appels , nous pouvons isoler le sous-graphe d'appels dynamique qui est entièrement inclus dans le graphe d'appels statique. L'autre étape importante pour aider d'autres chercheurs et même aider l'industrie est d'avoir un nouveau fichier XML généré représentant un nouveau modèle KDM avec les modifications et les améliorations que nous avons apportées dans ce travail.

APPENDICE A

LE CODE SOURCE DE L'APPLICATION J2EE UTILISÉ POUR TESTER LES DÉPENDANCES CACHÉES AJOUTÉES

```
import java.rmi.RemoteException;
import javax.ejb.EJBObject;
public interface Hello extends EJBObject{
 public String getHelloWorld() throws
 RemoteException;}
import javax.ejb.*
public class HelloBean implements
SessionBean {
 public String getHelloWorld(){
   return "Java bean Hello World!!!";
 }
 public void ejbCreate() {
  System.out.println("ejbcreate....");
 }
 public void ejbRemove() {
  System.out.println("ejbremove");
 }
 public void ejbActivate() {
  System.out.println("ejbactivate");
 }
 public void ejbPassivate() {
  System.out.println("ejbpassivate");
 }
public void setSessionContext(Sessi
 onContext sc) {
  System.out.println("session context");
 }
}
public interface HelloHome extends
EJBHome {
 public Hello create() throws
 RemoteException, CreateException;
}
public class HelloWorldClient {
public static void main(String args[]) {
  try {
   Properties p = new Properties();
   p.put(Context.INITIAL_CONTEXT_FA
   CTORY,
   "org.jnp.interfaces.NamingContext
   Factory");
   p.put(Context.URL_PKG_PREFIXES,
```

```
"org.jboss.naming:org.jnp.inter
faces");
p.put(Context.PROVIDER_URL,
"localhost");
InitialContext ctx = new Initial
Context(p);
HelloHome home = (HelloHome)
ctx.lookup("JNDIName_hello");
Hello bean = home.create();
System.out.println(bean
.getHelloWorld());
} catch (Exception e) {
System.out.println(e);
}
}
}
```

Le fichier XML de*deployment descriptor* :

```
\tiny
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC
'-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN'
'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>
<ejb-jar>
<enterprise-beans>
<session>
<ejb-name>hello</ejb-name>
<home>com.testejb.HelloHome</home>
<remote>com.testejb.Hello</remote>
<ejb-class>com.testejb.HelloBean</ejb-class>
<session-type>Stateless</session-type>
<transaction-type>Container</transaction-type>
</session>
</enterprise-beans>
<assembly-descriptor>
<container-transaction>
<method>
<ejb-name>hello</ejb-name>
<method-name>*</method-name>
```

```
</method>
<trans-attribute>Required</trans-attribute>
</container-transaction>
</assembly-descriptor>
</ejb-jar>
```

APPENDICE B

LE CODE SOURCE DE L'APPLICATION J2EE UTILISÉ POUR TESTER MAINTAINJ ET LA CONSTRUCTION DU GRAPHE DYNAMIQUE

```
package test;

public class Main {
  public static void main(Str
  ing[] args){
        Hello newH= new Hello();
        newH.bye();
        }

}
```

```
package test;

public class Hello {

        public Hello() {
        // Init innerclass
        HInner inner= new HInner();
        }
        public void bye() {
        Bye b = new Bye();
        }

  class HInner{

    public HInner() {

     System.out.println("Bonjour!");
        }

        }
}
```

```
package test;
public class Bye {

        public Bye() {
                System.out.println("Bye!");
        }

}
```

RÉFÉRENCES

Alalfi, M. H., Cordy, J. R. et Dean, T. R. (2009). Automated reverse engineering of uml sequence diagrams for dynamic web applications. Dans *2009 International Conference on Software Testing, Verification, and Validation Workshops*, 287–294. IEEE.

Ali, K., Lai, X., Luo, Z., Lhotak, O., Dolby, J. et Tip, F. (2019). A study of call graph construction for jvm-hosted languages. *IEEE Transactions on Software Engineering*.

Ali, K. et Lhoták, O. (2012). Application-only call graph construction. Dans *European Conference on Object-Oriented Programming*, 688–712. Springer.

Arnold, K., Gosling, J. et Holmes, D. *The Java programming language*, volume 2.

Badri, L., Badri, M. et St-Yves, D. (2005). Supporting predictive change impact analysis : a control call graph based technique. Dans *12th Asia-Pacific Software Engineering Conference (APSEC'05)*, 9–pp. IEEE.

Bastos, C., Junior, P. A. et Costa, H. (2016). Detection techniques of dead code : Systematic literature review. Dans *Proceedings of the XII Brazilian Symposium on Information Systems on Brazilian Symposium on Information Systems : Information Systems in the Cloud Computing Era - Volume 1*, SBSI 2016, p. 255–262., Porto Alegre, BRA. Brazilian Computer Society.

Bhat, S. A. et Singh, J. (2012). A practical and comparative study of call graph construction algorithms. *IOSR Journal of Computer Engineering*.

Bianco, P., Kotermanski, R. et Merson, P. F. (2007). Evaluating a service-oriented architecture.

Blondeau, V., Etien, A., Anquetil, N., Cresson, S., Croisy, P. et Ducasse, S. (2017). Test case selection in industry : An analysis of issues related to static approaches. *Software Quality Journal*, *25*(4), 1203–1237. `http://dx.doi.org/10.1007/s11219-016-9328-4`. Récupéré de `https://doi.org/10.1007/s11219-016-9328-4`

Bodden, E., Sewe, A., Sinschek, J., Oueslati, H. et Mezini, M. (2011). Taming reflection : Aiding static analysis in the presence of reflection and custom class loaders. Dans *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, p. 241–250., New York, NY, USA. Association for Computing Machinery. `http://dx.doi.org/10.1145/1985793.1985827`. Récupéré de `https://doi.org/10.1145/1985793.1985827`

Bracha, G. (2004). Generics in the java programming language.

Bruneliere, H., Cabot, J., Dupé, G. et Madiot, F. (2014). Modisco : A model driven reverse engineering framework. *Information and Software Technology*, *56*(8), 1012–1032.

Comella-Dorda, S., Wallnau, K. C., Seacord, R. C. et Robert, J. E. (2000). A survey of black-box modernization approaches for information systems. *Proceedings 2000 International Conference on Software Maintenance*, 173–183.

DeSantis, R. (2009). Automatically creating javascript objects to invoke methods on server-side java beans. US Patent 7,500,223.

Dmitriev, M. (2002). Language-specific make technology for the java programming language. *ACM SIGPLAN Notices*, *37*(11), 373–385.

Feng, Y., Dreef, K., Jones, J. A. et van Deursen, A. (2018). Hierarchical abstraction of execution traces for program comprehension. Dans *Proceedings of the 26th Conference on Program Comprehension*, 86–96.

Fink, S., Dolby, J. et Colby, L. (2004). *Semi-automatic J2EE transaction configuration*. Rapport technique, Technical Report RC23326, IBM TJ Watson Research Center.

Fourtounis, G., Kastrinis, G. et Smaragdakis, Y. (2018). Static analysis of java dynamic proxies. Dans *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 209–220.

Ghosh, P., Yan, Y. et Chapman, B. (2012). Support for dependency driven executions among openmp tasks. Dans *2012 Data-Flow Execution Models for Extreme Scale Computing*, volume 1, 48–54.

Grove, D., DeFouw, G., Dean, J. et Chambers, C. (1997). Call graph construction in object-oriented languages. Dans *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 108–124.

Guia, J., Soares, V. G. et Bernardino, J. (2017). Graph databases : Neo4j analysis. Dans *ICEIS (1)*, 351–356.

Hecht, G., Mili, H., El-Boussaidi, G., Boubaker, A., Abdellatif, M., Guéhéneuc, Y.-G., Shatnawi, A., Privat, J. et Moha, N. (2018). Codifying hidden dependencies in legacy j2ee applications. http://dx.doi.org/10.1109/APSEC.2018.00045

Holzschuher, F. et Peinl, R. (2013). Performance of graph query languages : comparison of cypher, gremlin and native access in neo4j. Dans *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, 195–204.

Huang, J. et Bond, M. D. (2013). Efficient context sensitivity for dynamic analyses via calling context uptrees and customized memory management. Dans *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 53–72.

Johnson, R. (2005). J2ee development frameworks. *Computer*, *38*(1), 107–110.

Kinable, J. et Kostakis, O. (2011). Malware classification based on call graph clustering. *Journal in computer virology*, *7*(4), 233–245.

Krinke, J. et Breu, S. (2004). Control-flow-graph-based aspect mining. Dans *1st Workshop on Aspect Reverse Engineering*.

Kundu, D., Samanta, D. et Mall, R. (2012). An approach to convert xmi representation of uml 2.x interaction diagram into control flow graph. *ISRN Software Engineering*, *2012*. http://dx.doi.org/10.5402/2012/265235

Kurtev, I. (2010). Application of reflection in a model transformation language. *Software & Systems Modeling*, *9*(3), 311–333.

Lai, C. (2008). Java insecurity : Accounting for subtleties that can compromise code. *IEEE software*, *25*(1), 13–19.

Larus, J. R. (1999). Whole program paths. *ACM SIGPLAN Notices*, *34*(5), 259–269.

Law, J. et Rothermel, G. (2003). Whole program path-based dynamic impact analysis. Dans *25th International Conference on Software Engineering, 2003. Proceedings.*, 308–318. IEEE.

Lhoták, O. r. (2007). Comparing call graphs. Dans *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 37–42.

Li, B., Sun, X., Leung, H. et Zhang, S. (2013). A survey of code-based change

impact analysis techniques. *Software Testing, Verification and Reliability*, *23*(8), 613–646.

MacKenzie, C. M., Laskey, K., McCabe, F., Brown, P. F., Metz, R. et Hamilton, B. A. (2006). Reference model for service oriented architecture 1.0. *OASIS standard*, *12*(S 18).

Miranskyy, A. V., Madhavji, N. H., Gittens, M. S., Davison, M., Wilding, M. et Godwin, D. (2007). An iterative, multi-level, and scalable approach to comparing execution traces. Dans *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 537–540.

Narasimhan, N., Moser, L. E. et Melliar-Smith, P. M. (2000). Transparent consistent replication of java rmi objects. Dans *Proceedings DOA'00. International Symposium on Distributed Objects and Applications*, 17–26. IEEE.

Nevill-Manning, C. G. et Witten, I. H. (1997). Linear-time, incremental hierarchy inference for compression. Dans *Proceedings of the Conference on Data Compression*, DCC '97, p. 3., USA. IEEE Computer Society.

Perin, F., Girba, T. et Nierstrasz, O. (2010). Recovery and analysis of transaction scope from scattered information in java enterprise applications. Dans *2010 IEEE International Conference on Software Maintenance*, 1–10. IEEE.

Pirzadeh, H., Shanian, S., Hamou-Lhadj, A., Alawneh, L. et Shafiee, A. (2013). Stratified sampling of execution traces : Execution phases serving as strata. *Science of Computer Programming*, *78*(8), 1099–1118.

Reif, M., Eichberg, M., Hermann, B., Lerch, J. et Mezini, M. (2016). Call graph construction for java libraries. Dans *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 474–486.

Rosen, M., Lublinsky, B., Smith, K. T. et Balcer, M. J. (2012). *Applied SOA : service-oriented architecture and design strategies*. John Wiley & Sons.

Rountev, A., Milanova, A. et Ryder, B. G. (2004). Fragment class analysis for testing of polymorphism in java software. *IEEE Transactions on Software Engineering*, *30*(6), 372–387.

Santibáñez, D. S. M., Durelli, R. S. et de Camargo, V. V. (2015). A combined approach for concern identification in kdm models. *Journal of the Brazilian*

Computer Society, *21*(1), 10.

Sereni, D. (2007). Termination analysis and call graph construction for higher-order functional programs. *SIGPLAN Not.*, *42*(9), 71–84. http://dx.doi.org/10.1145/1291220.1291165. Récupéré de https://doi.org/10.1145/1291220.1291165

Sharma, R., Stearns, B. et Ng, T. (2001). *J2EE connector architecture and enterprise application integration*. Addison-Wesley Professional.

Shatnawi, A., Mili, H., Abdellatif, M., Guéhéneuc, Y.-G., Moha, N., Hecht, G., El-Boussaidi, G. et Privat, J. (2019). Static code analysis of multilanguage software systems. *ArXiv*, *abs/1906.00815*.

Shivers, O. (1991). *Control-flow analysis of higher-order languages*. (Thèse de doctorat). PhD thesis, Carnegie Mellon University.

Singh, I., Salaria, E. et Johal, H. (2011). Comparative analysis of java and aspectj on the basis of various metrics. 270–275. http://dx.doi.org/10.1109/ICIS.2011.50

Sugawara, N. et Yamamoto, T. (2013). Call graph dependency extraction by static source code analysis. US Patent 8,347,272.

Taniguchi, K., Ishio, T., Kamiya, T., Kusumoto, S. et Inoue, K. (2005). Extracting sequence diagram from execution trace of java program. Dans *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, 148–151. IEEE.

Wongsuphasawat, K., Smilkov, D., Wexler, J., Wilson, J., Mané, D., Fritz, D., Krishnan, D., Viégas, F. B. et Wattenberg, M. (2018). Visualizing dataflow graphs of deep learning models in tensorflow. *IEEE Transactions on Visualization and Computer Graphics*, *24*(1), 1–12.

Wu, S., Wang, P., Li, X. et Zhang, Y. (2016). Effective detection of android malware based on the usage of data flow apis and machine learning. *Information and Software Technology*, *75*, 17 – 25. http://dx.doi.org/https://doi.org/10.1016/j.infsof.2016.03.004. Récupéré de http://www.sciencedirect.com/science/article/pii/S0950584916300386

Wüchner, T., Ochoa, M. et Pretschner, A. (2015). Robust and effective malware detection through quantitative data flow graph metrics. Dans *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 98–118. Springer.

Xie, T. et Notkin, D. (2002). An empirical study of java dynamic call graph extractors. *University of Washington CSE Technical Report 02-12*, *3*.

Zhang, W. et Ryder, B. (2006). Constructing accurate application call graphs for java to model library callbacks. Dans *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, 63–74. IEEE.