Compréhension de systèmes orientés-objet par l'utilisation d'informations dynamiques condensées

DEA
École Supérieure en Sciences Informatiques
Sophia-Antipolis, France

Roland Bertuli

Juin 2003

Encadreurs :

Prof. Dr. Stéphane Ducasse
Dr. Michele Lanza
Prof. Dr. Oscar Nierstrasz

Software Composition Group
Université de Berne, Suisse

Remerciements

Ce travail a été réalisé dans l'équipe de recherche *Software Composition Group* de l'université de Berne, Suisse, dirigée par Prof. Dr. Oscar Nierstrasz.

En premier lieu, je souhaiterais remercier très sincèrement Stéphane Ducasse, l'encadreur de ce projet. Merci de m'avoir donner l'opportunité de faire ce projet à l'Université de Berne, merci également de tous tes nombreux conseils et du temps que tu m'as consacré (je sais qu'il est précieux pour toi). Tout en étant extrêmement enrichissant, cela fut un réel plaisir de travailler avec toi.

Je voudrais particulièrement remercier Oscar Nierstrasz pour m'avoir accueilli dans son équipe, ainsi que Michele Lanza avec qui j'ai eu plaisir à travailler durant ce stage. Je remercie également tous les autres membres du *Software Composition Group* pour leur gentillesse et tout ce qu'ils ont pu m'apporter de leur contact : Gabriela Arévalo, Juan Carlos Cruz, Markus Gaelli, Tudor Girba, Laura Ponisio, Matthias Rieger, Nathanael Schaerli et Roel Wuyts.

Je remercie Alexandre Bergel, mon ami, membre du *Software Composition Group*, qui m'a hébergé durant ces quelques mois passés à Berne. Même si nous avons beaucoup travaillé, nous nous sommes bien amusé durant mon séjour.

Je remercie également Mireille Blay-Fornarino dirigeant l'équipe *Raimbow* de l'I3S de Sophia-Antipolis, sans qui je n'aurais pu effectuer ce projet dans une Université suisse.

Je remercie enfin, ma famille et mes amis que j'ai laissé le temps de ce travail. Même si travailler loin d'eux à parfois été difficile, je sais qu'ils seront là à mon retour.

Roland Bertuli
23 juin 2003

Table des matières

Chapitre 1

Introduction

Dans l'industrie, le coût de maintenance d'une application informatique est établie entre 50% et 75% de son coût total [26] [5]. On estime également que plus de la moitié de cette maintenance est passée dans la compréhension de l'application elle-même [4]. Ce constat démontre que la compréhension des applications informatiques est un enjeu crucial, présent tout au long de leur durée de vie.

Le paradigme objet exacerbe la difficulté de cette tâche par la présence de la liaison tardive et du polymorphisme. De plus, l'envoi de messages par le biais des héritages rend ardu une identification statique précise du rôle des objets lors de l'exécution d'une application.

Il existe deux approches du problème de l'analyse d'une application informatique. Il est possible de se baser soit sur son code (analyse statique), soit sur son comportement (analyse dynamique). L'analyse des informations dynamiques issues de l'exécution d'un programme informatique, s'oppose donc à l'analyse d'informations statiques extraites de son code. Dans ce travail nous nous focaliserons uniquement sur l'analyse d'informations dynamiques par le biais de leur visualisation.

Dans le but d'analyser les informations dynamiques d'un programme, il est courant d'instrumenter le code (enrobage de méthodes, extension de machines virtuelles, etc...) puis de lancer l'exécution du système à analyser. Une fois instrumenté, ce code pourra ainsi générer une *trace* reflétant les informations sur le comportement du système. Typiquement, cette trace contient des informations décrivant quelles méthodes appellent quelles autres méthodes, quels objets ont été créés durant l'exécution, etc...

Un problème majeur de cette approche réside dans les traces issues des exécutions. Lors d'une exécution, un système informatique de taille conséquent génère le plus souvent une trace d'informations très volumineuse [25]. L'expérience montre que collecter la trace d'une grosse application pendant quelques secondes peut générer des dizaines de milliers d'évènements. De plus, le bas niveau d'abstraction des informations contenues par ce genre de trace rend, par conséquent, difficile l'abstraction de l'analyse à un niveau supérieur. À grande échelle, en plus de se répercuter sur la clarté des représentations graphiques obtenues, ce problème engendre des problèmes techniques de stockage des informations obtenues.

Ce travail propose une approche basée sur la *condensation* des informations collectées et visualisées. La trace entière d'exécution ne sera ni collectée, ni analysée, des valeurs calculées durant l'exécution, représentant le nombre d'apparitions de certains évènements, lui seront préférées. Ces valeurs sont utilisées pour décorer des schémas, fournissant ainsi des représentations graphiques simples et de haut niveau du comportement d'une application, résistant bien mieux à la montée dans l'échelle.

Le travail, exposé dans ce rapport, apporte différentes contributions au champ de l'analyse dynamique d'applications orientées-objet. Une nouvelle approche de synthétisation et de visualisation d'informations dynamiques a été élaborée. Cette approche propose l'idée de la condensation des informations. Une implantation de l'idée a été réalisée afin d'expérimenter l'approche et d'appliquer des vues sur des cas d'étude concrets.

Dans ce rapport, nous présentons, tout d'abord, les problèmes et les challenges posés par l'analyse d'exécutions d'applications orientées-objet, ainsi qu'un état de l'art de ce domaine (Chapitre 2). Nous décrivons en détail notre approche, en montrant en quoi elle se démarque des travaux existants (Chapitre 3). Afin de valider ses avantages sur ses concurrents nous confrontons quatre exemples issus de cette approche à une étude de cas réel (Chapitre 4). Nous concluons enfin par les perspectives qu'ouvre ce nouveau mode de visualisation d'informations dynamiques (Chapitre 5).

Chapitre 2

Visualisation d'informations dynamiques

Depuis l'avènement du paradigme objet, un pan de la recherche informatique s'est orienté vers l'aide à la compréhension d'applications orientées-objet. Parmi les différentes approches proposées dans la littérature, aidant à l'appréhension du comportement de systèmes informatiques, la représentation graphique de programmes est unanimement reconnue.

2.1 Problèmes de l'analyse d'exécution

Wilde et Huitt [28] estime que la compréhension des d'applications orientées-objet est une tâche difficile qui se doit de surmonter les particularités de ce paradigme. En effet, même si elles sont les points forts de ce type de programmation, elles élèvent l'analyse et la compréhension des applications orientées-objet à un autre niveau de difficulté.

Le polymorphisme et la liaison tardive rendent les traditionnels outils d'analyse comme la découpe de programmes, inadéquates. La réalisation d'analyseurs de flots de données est rendue plus ardue, principalement en raison de la présence de langages dynamiquement typés. L'utilisation simultanée de l'héritage et de l'incrémentation des définitions de classes ajoutée à la sémantique du *this*, de certains langages, rend bien souvent la compréhension des applications bien plus difficiles. Ainsi, le modèle conceptuel d'une application est dispersé à travers ses classes résidant dans différentes hiérarchies et sous-systèmes. Dans ces conditions, il devient complexe de situer précisément certaines fonctionnalités.

L'appréhension d'une application écrite dans un langage de programmation orienté-objet étant une tâche ardue, l'utilisation d'informations dynamiques est une des voies à prendre pour aider aux processus de compréhension. L'analyse d'informations issues de l'exécution d'un système informatique, nous amène à répondre aux types de questions suivantes :

- Quelles sont les classes les plus instanciées ?
- Quelles sont les classes ayant des objets attitrés ? D'un point de vue architecturel, la détection de *singletons* [11] peut s'avérer une information importante.
- Quelles sont les classes créant des objets ?
- Quels sont les interactions entre les classes ?
- Quel est le pourcentage d'utilisation des méthodes définies dans une classe ?

2.2 Challenges et contraintes

Quelque soit le type d'information recherché lors de l'analyse de l'exécution d'un système informatique orienté-objet, deux problèmes récurant à ce domaine ressortent fréquemment, la quantité et la granularité des données récoltées.

Quantité et densité. Les traces d'exécution de l'analyse dynamique emmagasinent de très importantes quantités d'informations de bas niveau. De plus, les sommes de données à analyser engendrent l'utilisation de techniques visant à réduire leur complexité, comme des filtres, des analyses de concepts ou des regroupements [25] [15].

Granularité. L'information contenue dans une trace d'exécution traduit chaque pas du comportement de l'application. Sur un plan conceptuel, elle est donc de très bas niveau, *e.g.*, telle méthode appelle telle méthode, telle méthode accède tel attribut, tel objet est créé par telle méthode, etc... Il est donc difficile d'accéder à une compréhension de haut niveau, en utilisant ce type d'information.

2.3 État de l'art

Si la visualisation d'informations statiques, issues du code d'une application, a été la base d'un très grand nombre de recherches, son homologue dynamique a beaucoup moins été abordé dans la littérature. Cette différence n'est en rien la conséquence d'une maîtrise du domaine ni même de l'inefficacité de cette approche. Elle résulte des graves problèmes de montée dans l'échelle de ce type d'approche. En effet, les quantités d'informations dynamiques extraites de l'exécution d'un programme informatique sont généralement beaucoup plus importantes que les informations statiques relatives au code de l'application. Ceci expliquerait que l'analyse statique, ayant donné de très bons résultats, encourage la multiplication des recherches, à l'inverse de son homologue dynamique. Il existe cependant, plusieurs équipes de recherche spécialisées dans ce domaine, proposant des approches intéressantes du problème.

À travers leur outil AVID, Murphy *et al.* ont développé ce qui permet au ingénieurs informatiques de spécifier différents niveaux d'abstraction d'un systèmes afin d'en étudier les entités désirées [27]. L'exécution d'une application est visualisée par le biais de ces modèles d'abstractions, ciblant leur durée de vie et leur nombre. De part la nature de cette approche, la quantité d'entités visualisables simultanément est assez restreinte. Elle est donc difficilement utilisable dans le cadre d'un premier contact avec un système.

Lange *et al.*, avec leur Program Explorer, se focalisent sur la représentation des classes et des objets [18] [17]. Les auteurs ont développé un système traquant les invocations de fonctions, les instanciations d'objets et l'accès aux attributs. Les vues proposées par l'approche montrent les relations entre classes et instances (généralement focalisées sur une instance ou une classe particulière du système), et un court historique des invocations de méthodes. Dans ces travaux, la solution n'est toujours pas applicable à l'ensemble d'un système. L'utilisateur doit connaître à l'avance les entités dont il désire suivre le comportement durant l'exécution, ce qui est à l'opposé de la philosophie de notre approche.

Jerding *et al.* ont créé leur propre diagramme d'interactions pour visualiser le comportement d'une application durant son exécution [12] [13]. Le but principal de leur outil ISVis est de visualiser l'ensemble des appels de méthodes d'une exécution. Il peut extraire et reconnaître des schémas comportementaux, mais le manque de flexibilité dans l'analyse représente le principal défaut de l'approche. Cependant, les diagrammes obtenus résistent bien à la montée en échelle du nombre de messages, durant l'exécution, mais en présence d'un grand nombre de classes dans le système, il devient très vite inutilisable.

De Pauw *et al.* ont travaillé sur deux approches différentes du problème. Comme dans l'approche précédente, au travers de Jinsight, ils se proposent d'utiliser leurs propre diagrammes d'interactions [24]. En ce sens, on retrouve donc les avantages et les inconvénients précédents. En présence de traces d'exécution issues de très gros systèmes, cette technique perd son efficacité à appréhender le rôle des classes. De par leur simplicité, les travaux antérieurs de De Pauw se rapprochent le plus de l'approche proposée dans ce rapport. Dans ses *class call clusters* et autres *class call matrix* [22] [23], il fut le seul à proposer une approche

basée sur la condensation des informations dynamiques d'un système informatique. Les représentations obtenues sont simples, elles résistent bien à la montée dans l'échelle, mais ne sont malheureusement orientées uniquement vers des aspects restreints du comportement d'une application orientée-objet.

À l'exception de la dernière approche citée, celles qui ont pris le parti de visualiser le comportement dans son ensemble, collectent et restituent la trace d'exécution entière. Ce choix ne peut qu'accentuer les difficultés liées à la montée dans l'échelle du système. Afin, de palier à ce problème certaines équipes ont tenter de relever automatiquement les informations pertinentes d'une trace d'exécution. Une tranche de trace d'exécution d'un programme (*slice*) [15] est une partie exécutable de ce programme dont le comportement est identique, pour les même entrées du programme, qu'un comportement donné du programme dans son ensemble, composé de tous ces comportements. Dans la littérature, le tranchage de trace d'exécution a déjà été proposé dans le but de soutenir la compréhension du comportement d'une application [16]. Dans la même optique, CodeSurfer [1] supporte la compréhension par l'utilisation d'hypertexte, mais n'autorise pas une représentation visuelle de l'étude.

Chapitre 3

Approche proposée

Lors de l'exécution, de nombreuses classes, des hiérarchies d'héritage et des schémas d'interactions dynamiques contribuent à la complexification de la compréhension de larges systèmes orientés-objet. Beaucoup de ces difficultés sont dues à la quantité et à la qualité des données extraites de la trace d'exécution. Les informations pertinentes sont souvent cachées derrière des sommes de données colossales, qu'il faudra filtrer et analyser avant de pouvoir enfin les visualiser. L'approche proposée dans ce rapport évite ces énormes accumulations de données de bas niveau en ne récupérant pas les traces d'exécution. Elle se borne à les synthétiser au travers d'un nombre de mesures relativement restreint. Ces mesures viendront décorer des graphes polymétriques simples, chacun pouvant traduire un aspect particulier du comportement de l'application.

3.1 Graphes polymétriques

Le point de départ de ce travail se situe sur l'approche implémentée dans l'outil de visualisation d'information statique, CodeCrawler [6] [21]. Dans le but de comprendre l'architecture d'applications informatiques, M. Lanza utilise des graphes polymétriques enrichis de métriques extraites du code source du système à analyser. Dans l'approche dynamique proposée ici, l'idée est reprise pour décorer des graphes d'informations reflétant le comportement d'une application et non plus sa structure.

La notion de visualisation implique intrinsèquement une notion dimensionnelle. Ce rapport se restreint à l'étude des algorithmes de placement à deux dimensions, monochromatiques, le type de graphes engendrés offrant un bon compromis entre nombre d'informations représentées et complexité de compréhension. Les graphes étudiés disposent d'une sémantique intuitive. Ils comportent des nœuds représentant des entités du système étudié (classes, méthodes, instances, etc...), pouvant être relier par des arêtes en fonction de relations entre ces nœuds (héritage, communication, création, etc...). L'idée principale de l'approche est de décorer ces graphes. De par leur nature, ils pourront être enrichis de six mesures numériques, reflétant des informations tirées de l'exécution d'une application.

Taille du nœud. La hauteur et la largeur d'un nœud peut représenter deux mesures. La convention suivie dans cette approche traduit ces deux dimensions proportionnellement aux mesures qui leur sont attribuées. Plus ces mesures seront importantes, plus le nœud sera imposant.

Couleur du nœud. La palette de gris entre le blanc et le noir peut traduire une mesure. Le blanc représente la valeur minimale tandis que le noir représente la valeur maximale de la mesure associée, les gris intermédiaires étant plus ou moins foncés proportionnellement à la valeur du nœud en question.

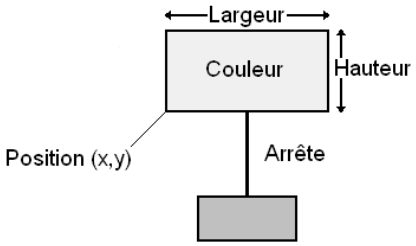

FIG. 3.1 – Vue polymétrique

Position du nœud. Positionné dans un graphe à deux dimensions, un nœud, à l'aide de ses coordonnées, traduit deux mesures. La présence d'une origine absolue fixée dans un système de coordonnées fixé est alors nécessaire. Cependant, certains types de graphes ne peuvent pas exploiter ces dimensions, la position des nœuds étant dépendante de la nature de ces graphes (graphes d'héritage).

Largeur de l'arrête. Dans certains graphes, des arrêtes relient les nœuds entre eux. L'épaisseur de ces arrêtes peut être utilisée pour traduire une mesure entre les deux entités connectées, *e.g.*, une épaisse arrête d'invocation entre deux classes représentera un grand nombre de communication entre elles.

Les informations dynamiques tirées d'une exécution sont très souvent non-linéaires, comportant des différences colossales entre elles. Une méthode peut, par exemple, être invoquée 10 fois lors d'une exécution, tandis qu'une autre méthode le sera 50.000 fois. Ces différences de valeur entre les mesures imposent d'utiliser des échelles logarithmiques dans certains types de vues, afin d'aplanir ces écarts.

3.2 Collection d'informations dynamiques

La collection d'information dynamique est un riche domaine, allant de l'enrobage de méthodes [2] au contrôle d'objets par l'instrumentalisation de machines virtuelles [3] [9].

La majorité des travaux réalisés, dans le domaine de la visualisation d'information dynamique, utilisent une trace d'évènements collectée durant l'exécution. Ce type de visualisation fournit grand nombre d'informations, cependant, il consume un espace mémoire important et requiert de nombreuses abstractions et manipulations pour extraire les informations désirées. Notre approche se démarque donc radicalement de cette voie, nous ne manipulons plus les évènements un à un, mais des mesures tirées de l'exécution (le nombre d'invocations, le nombre d'objets créés, le nombres de classes utilisées lors de l'exécution, etc...). Les mesures possibles sont infinies, allant des plus triviales aux plus complexes. Le maître mot de nos travaux étant de fournir une solution simple, nous nous contentons de combiner des mesures relativement basiques entre elles, afin de fournir des informations de haut niveau sur l'exécution d'applications.

L'idée de l'approche ne restreint pas le type d'entités visualisées, les entités de base de la programmation orientée-objet (classes et méthodes) seront les seules utilisées dans ce rapport. Cependant, comme pour les mesures, il n'existe pas de limites aux entités choisies, nous pourrions visualiser des groupes de classes, de méthodes, des patrons de conceptions ou même des tests unitaires d'applications. Ce rapport propose l'approche en elle-même, ses avantages et les analyses possibles. Il ne fournit en aucun cas une liste exhaustive de schémas, la modularité étant précisément l'un des fondements de notre solution.

Nom	Description
Mesures de Classes	
NCM	Nombre de méthodes utilisées (invoquées au moins une fois)
RCM	Taux de méthodes utilisées contre méthodes non-utilisées
NMI	Nombre d'invocations de méthodes de la classe
NIMI	Nombre d'invocations internes de méthodes de la classe
NEMI	Nombre d'invocations externes de méthodes de la classe
NCCM	Nombre de méthodes de classes (*static*) utilisées
NCMI	Nombre d'invocations de méthodes de classe (*static*)
NCI	Nombre d'instances créées de la classe
NCO	Nombre d'objets créés par les instances de la classe
Mesures de Méthodes	
TI	Nombre total d'invocations
ITI	Nombre d'invocations internes (l'objet receveur est l'émetteur de l'invocation)
ETI	Nombre d'invocations externes (l'objet receveur est différent de l'émetteur de l'invocation)

Tab. 3.1 – Listes des mesures extraites d'une exécution

Les mesures utilisées dans ce rapport sont énumérées dans le tableau ci-dessus. À première vue, les différence entre NCM, NMI et RCM peuvent être délicates à cerner. NCM représente le nombre de méthodes utilisées d'une classe lors d'une exécution, tandis que NMI reflète le nombre invocations sur les méthodes de la classe. La mesure RCM de cette classe représente le taux de ses méthodes utilisées. Pour éviter les ambiguïtés, prenons l'exemple d'une classe dans laquelle sont définies 5 méthodes d'instances. Si durant l'exécution d'un système, 3 des méthodes de cette classe ont été invoquées en tout 500 fois, les deux autres méthodes n'étant pas été utilisées, alors pour cette classe, NCM sera égal à 3, NMI vaudra 500, tandis que RCM quant à lui sera égal a 0,6.

Chapitre 4

Expérimentations

Après avoir décrit les grandes lignes de notre approche, dans ce chapitre quatre vues polymorphiques possibles sont proposées. Par leur intermédiaire, l'implantation de notre approche (Annexe B) est confrontée à l'étude d'un cas concret. Le résultat de l'application des quatre vues choisies est exhibé et une possible analyse en sera proposée.

4.1 Étude de cas

Pour l'expérience, un scénario couvrant les différents aspects d'une application de taille raisonnable est lancé. L'application informatique choisie est l'environnement de re-ingénierie Moose (Annexe A), développé par le *Software Composition Group* [10]. Ce système sert de fondation à plusieurs outils d'ingénierie inverse écrit en Smalltalk [19] [14]. Moose ne fut pas le seul terrain d'expérimentation, plusieurs autres applications ont été analysées afin de valider l'approche de ce travail. Le choix de Moose comme cas d'étude, proposé dans ce rapport, fut édicté par deux raisons. La première de ces raisons découle de la connaissance de cette application. En effet, la proximité avec ses concepteurs a rendu aisé l'analyse et l'interaction lors de son analyse. Une autre raison est à l'origine de ce choix. L'implantation de l'approche s'appuie sur Moose, il est donc pertinent de montrer que l'outil d'analyse résultant a la capacité de s'auto analyser. Cette épreuve constitue une bonne validation sur la stabilité de l'implantation.

Moose fournit une représentation indépendante de langages orientés-objet à partir de code source écrit en C++ , Java, Cobol et Smalltalk. Cette indépendance de langage est basée sur le méta-modèle FAMIX [8], décrivant la représentation des éléments de base d'un code source (attributs, méthodes, classes et paquetages) [7].

Afin de manipuler le code source d'applications écrites en Java ou C++, Moose interprète des fichiers CDIF ou XMI. L'extraction d'applications Smalltalk utilise le propre analyseur syntaxique de Moose suivie d'une analyse de l'arbre de syntaxe abstraite résultant pour générer les modèles. Moose est un cas d'étude de taille moyenne, il est composé de 137 classes et de 2093 méthodes de code Smalltalk.

Dans les sections suivantes, quatre vues différentes issues de notre approche sont appliquées à l'exécution de Moose. Le *Instance Usage Overview*, le *Communication Interaction*, le *Creation Interaction* et le *Method Call Origin* ont été choisies pour représenter le plus simplement possible un éventail de possibilités offertes par l'approche. Afin d'obtenir une visualisation et une analyse représentative du système, le scénario utilisé est un cas d'application de base couvrant les différentes facettes de Moose.

4.2 Instance Usage Overview

Description du **Instance Usage Overview**	
Représentation	Arbre d'héritage, sans tris
Nœuds	Classes
Arrêtes	Héritage
Portée	Système entier
Echelle	Logarithmique
Largeur des Nœuds	NCI (*Nombre d'instances créées*)
Hauteur des Nœuds	NCM (*Nombre de méthodes utilisées*)
Couleur des Nœuds	NMI (*Nombre d'invocations de méthodes*)
Figure	Figure 4.1

4.2.1 Intention de la vue

La vue *Instance Usage Overview* montre comment les classes du système ont été instanciées et utilisées durant l'exécution. Comme le montre la description de la vue ci-dessus, la largeur des nœuds est utilisée pour représenter le nombre d'instances créées, leur hauteur traduit le nombre d'objets créés par les instances de la classe, tandis que leur couleur reflète le nombre d'invocations de méthodes pendant le fonctionnement de l'application.

4.2.2 Symptômes

Il est important de noter que cette vue ne prend en considération que les invocations sur les méthodes d'instances, les appels sur les méthodes de classes (*static*) n'étant pas pris en compte. Alors que la vue donne un aperçu du comportement d'une application dans sa globalité, elle offre également l'avantage d'exhiber des informations détaillées. Ci-dessous, sont énoncées les formes visuelles pouvant être dégagées de son utilisation.

- Nœuds petits carrés blancs représentent des classes ni instanciées ni utilisées.
- Nœuds étroits et pales représentent des classes peu ou pas été instanciés dont des méthodes ont été utilisées. Cela peut être le cas de singletons ou de classes abstraites non instanciées dont les méthodes ont été utilisées par héritage.
- Nœuds plats et pales représentent des classes très instanciées mais qui n'ont que peu été utilisées.
- Nœuds plats et foncés représentent des classes lourdement instanciées dont un faible nombre à été énormément invoquées durant l'exécution.
- Nœuds larges et sombres représentent des classes lourdement instanciées et utilisées.

4.2.3 Cas d'étude

La Figure 4.1 montre une partie du *Instance Usage Overview* appliquée à notre cas d'étude. Le large nœud sombre annoté *A*, dans notre vue, représente le scanner CDIF de MOOSE, cette classe parse les fichiers écrits dans le format CDIF, un format d'échange industriel. Une instance du scanner est créée chaque fois que le modèle est chargé en mémoire. Elle est largement invoquée, l'analyse lexicale étant un processus dense faisant appel à de nombreuses petites méthodes spécifiques.

Le large nœud sombre annoté *B* représente le méta-méta-modèle de Moose, la classe *AttributeDescription* instanciée un très grand nombre de fois. Cette classe du méta-méta-modèle est instanciée pour représenter le méta-modèle Moose courant. Comme Moose est un environnement dynamique et les méta-modèles peuvent être étendus, la représentation du méta-modèle courant est créée chaque fois qu'un modèle est chargé, ce qui est fréquemment

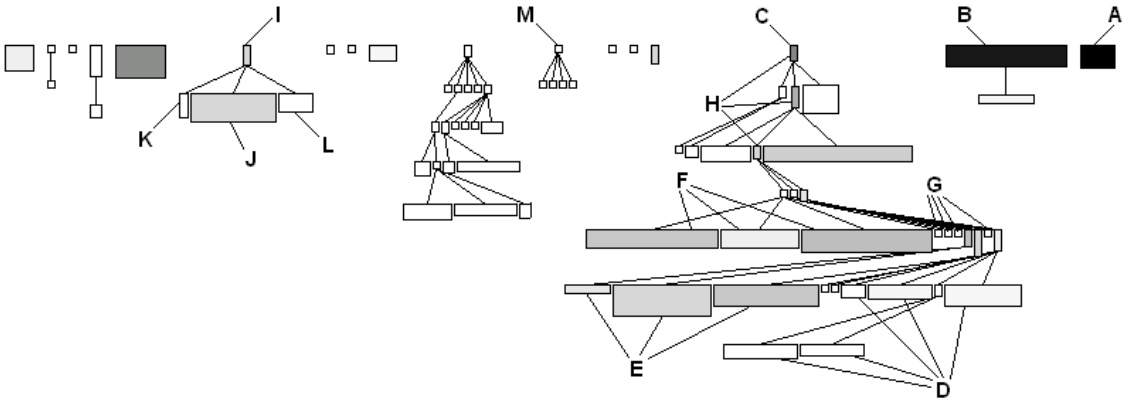

FIG. 4.1 – Application du Instance Usage Overview

le cas lors du scénario de notre cas d'étude. Ceci explique donc le nombre élevé de création d'instance lors de l'exécution. Il est cependant intriguant de constater l'importance des valeurs d'instanciation et d'utilisation (350.000 appels sur 3.500 instances). Une investigation plus poussée pourrait amener à déboucher sur certaines optimisations.

Les classes décrivant le méta-modèle FAMIX (représentées dans la hiérarchie d'héritage dont la racine est nommée *C* sur la figure), traduisant les entités du code source, sont représentées par des nœuds plats et de couleurs claires. Le scénario exécuté ne charge que de simples modèles dans Moose contenant quelques classes seulement. Ainsi, ces classes n'ont pas été les plus instanciées, comme elles auraient pu l'être dans le cas du chargement d'un imposant modèle. Cette hiérarchie d'héritage contient trois types de formes :

1. Les nœuds plats traduisent les informations extraites du code Smalltalk (Classes, Methodes, Attributes, Inheritances, etc...), ils se situent toujours sur les feuilles de l'arbre. Les classes blanches (*D*), représentant les instances du modèle et les variables locales, sont moins instanciées et utilisées que les accès de variables et les invocations de méthodes (*E, F*).
2. Les petites feuilles carrées (*G*) représentent des classes définies dans le méta-modèle indépendant du langage, mais qui ne sont pas pertinentes en Smalltalk (Includes, SourceFile, Function). Ces classes n'ont donc pas été instanciées.
3. Les nœuds étroits dans le milieu hiérarchie (*H*) représentent des classes abstraites qui n'ont pas été instanciées mais dont les méthodes ont été invoquées par des instances de sous-classes.

La petite hiérarchie, sous la classe annotée *I*, représente l'arbre de parsage Visiteur [11] qui extrait le méta-modèle FAMIX du code source Smalltalk. La classe *VWParseTreeEnumerator*[1] (*J*) est invoquée à chaque création de modèle à partir d'un code source Smalltalk, alors que les deux autres Visiteurs que sont *VWParseTreeMetricCalculator* (*K*) et *VWParseAnnotator* (*L*) sont dédiés à des analyses explicitement demandées.

Enfin, la petite hiérarchie *M* n'est pas couverte du tout par notre exécution. En fait, ces classes représentent l'interface graphique de Moose, qui n'est pas invoquée lors du déroulement de notre scénario.

[1]VW représente l'abréviation de *VisualWorks*, une distribution de Smalltalk

4.2.4 Résumé

Le *Instance Usage Overview* donne un aperçu du comportement d'un système dans son ensemble. Il fournit des indications sur l'utilisations des classes d'une application dans un contexte de réutilisation du code par héritage. Cette vue offre le double avantage de combiner informations statiques (hiérarchie d'héritage, nombre de classes) et dynamiques de chaque classe (nombre d'instances créées, nombre de méthodes utilisées et nombre d'invocations). Elle aide, ainsi, à identifier les classes largement instanciées, non-instanciées, très utilisées ou inutilisées.

4.3 Communication Interaction

Description du **Communication Interaction**	
Représentation	Spring Layout
Nœuds	Classes
Arrêtes	Invocations
Portée	Système entier
Echelle	Linéaire
Largeur des Nœuds	NCM (*Nombre de méthodes utilisées*)
Hauteur des Nœuds	NCM (*Nombre de méthodes utilisées*)
Couleur des Nœuds	NMI (*Nombre d'invocations de méthodes*)
Largeur des Arrêtes	Nombre d'invocations entre les deux classes
Figure	Figure 4.2

4.3.1 Intention de la vue

La vue *Communication Interaction* présente les communications entre les classes d'un système durant son exécution. Comme le décrit le tableau ci-dessus, la taille des nœuds est proportionnelle au nombre de méthodes utilisées tandis que leur couleur représente le nombre d'invocation sur les méthodes de la classe. Cette vue reprend les avantages du *Spring Layout* [2], il pondère les classes communicatives entre elles, en les agrégeant ensemble. De plus, la largeur des arrêtes entre les nœuds traduit la communication *classe à classe*.

4.3.2 Symptômes

Il est nécessaire de préciser que, par l'intermédiaire de cette vue, seules les communications entre les instances sont prises en comptes. Les méthodes de classes ne sont pas étudiées ici. Cette vue peut contenir :

- Petits nœuds déconnectés représentent des classes dont les méthodes n'ont pas communiqué avec d'autres méthodes. Ces classes n'ont soit pas été instanciées, soit pas été utilisées.
- Petits nœuds clairs connectés représentent des classes dont peu de méthodes ont été invoquées peu de fois. Ces classes n'ont que peu été utilisées.
- Petits nœuds foncés connectés représentent des classes dont un nombre restreint de méthodes a été invoqué très souvent.
- Larges nœuds clairs représentent des classes ayant un nombre important de méthodes utilisées, qui n'ont été que peu de fois invoquées durant l'exécution.
- Larges nœuds foncés représentent des classes comportant un grand nombre de méthodes utilisées, très invoquées durant le déroulement du scénario.
- Ilots de classes connectées au noyau de la vue représentent des classes communicant par un entonnoir avec le reste du système.

4.3.3 Cas d'étude

La Figure 4.2 montre l'application de la vue *Communication Interaction* au cas d'étude. Un groupe de classe est clairement déconnecté du reste du noyau de la vue, il est cependant rattaché celui-ci par l'intermédiaire de la classe annotée *A*. Dans notre cas d'étude, ce groupe de classes implémente la production de fichiers XMI basée sur l'interfaçage d'un MOF. La formation de cet îlot de classes résulte de la nature du producteur XMI/MOF, qui est un

[2] Le *Spring Layout* est un algorithme de placement reposant sur un principe de répulsions et de ressorts

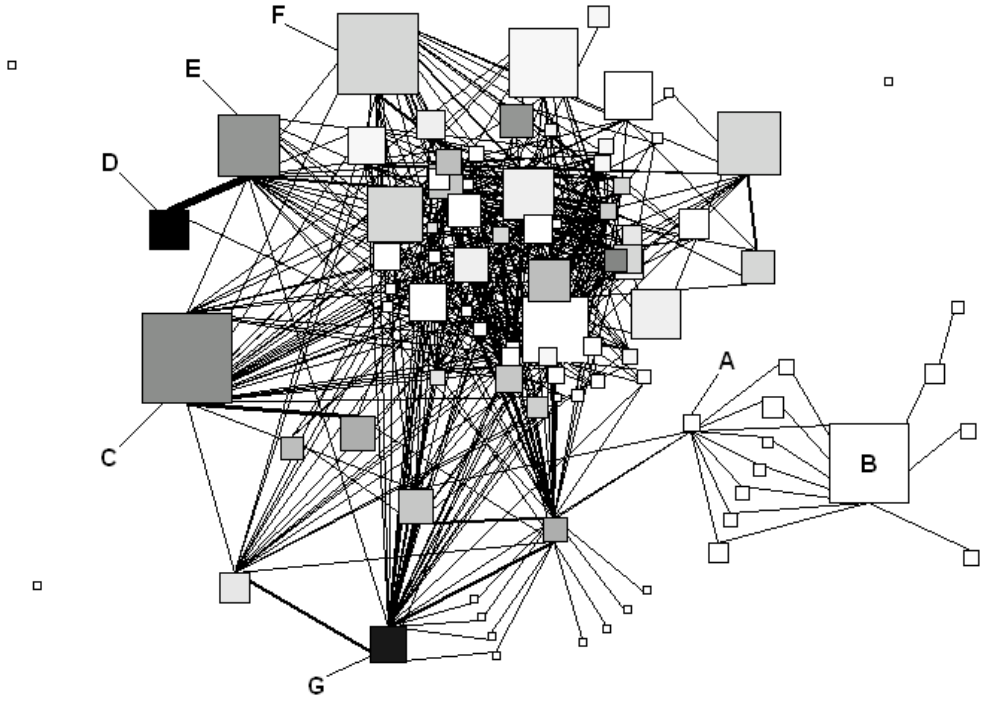

FIG. 4.2 – Application du Communication Interaction

paquetage indépendant de Moose. L'imposante classe *B* est le producteur XMI, il utilise l'interface MOF communicant via la *classe-pont A* avec le méta-modèle utilisé, représenté dans le noyau de la vue. Le producteur XMI est rarement utilisé dans notre exemple, car Moose privilégie le format d'échange CDIF, ce qui expliquerait la teinte pale de ces classes.

La grosse classe *C* est le dépôt central stockant l'ensemble des modèles analysés. De plus, elle joue le rôle de point d'entrée des requêtes sur les modèles. Pour ces raisons, cette classe communique avec toutes les classes modélisant le code source d'une application. La classe sombre de taille moyenne *D* est le scanner CDIF, elle est fréquemment invoquée par la classe importeur *E* chargeant les modèles en mémoire. Au sein de l'architecture de Moose, l'importeur a la responsabilité de décorer un modèle en transformant une représentation textuel d'un code source (à partir d'un fichier CDIF) en objets. Il est intéressant de noter que les développeurs de Moose ont appris que cette classe était également invoquée par une autre classe comme le montre la Figure 4.2. La grosse classe *F* représente la classe *MSEClass* modélisant les classes dans Moose. La classe *G* est la représentation du méta-méta-modèle de Moose qui est instanciée pour représenter le méta-modèle FAMIX. Elle est utilisée par l'ensemble des classes FAMIX ainsi que par les outils d'entrée/sortie fournissant des fonctionnalités indépendantes du méta-modèle.

4.3.4 Résumé

La vue *Communication Interaction* identifie le flot de communication au sein d'un système lors de son exécution. De par sa nature, il offre une moindre résistance à la montée dans l'échelle que le *Instance Usage Overview*. En effet, en présence d'un grand nombre de classes communiquant intensément, un *Spring Layout* naïf rencontre des difficultés pour offrir des groupes de classes bien différenciés. Nous pourrons remarquer que dans notre approche, nous prenons en compte les communications par héritage. Une voie pour réduire le nombre de communication serait de grouper les classes dans des hiérarchies communes.

4.4 Creation Interaction

Description du **Creation Interaction**	
Représentation	Spring Layout
Nœuds	Classes
Arrêtes	Instantiations
Portée	Système entier
Echelle	Logarithmique
Largeur des Nœuds	NCO (*Nombre d'objets créés par les instances*)
Hauteur des Nœuds	NCI (*Nombre d'instances créées*)
Couleur des Nœuds	NCI (*Nombre d'instances créées*)
Largeur des Arrêtes	Nombre de créations entre les deux classes
Figure	Figure 4.3

4.4.1 Intention de la vue

La vue *Creation Interaction* montre les créations d'instances entre les classes durant l'exécution du système. Comme le tableau ci-dessus le décrit, la largeur et la couleur des nœuds représentent le nombre d'objets créés par la classe, alors que la hauteur est proportionnelle au nombre d'instances créées de la classe durant le déroulement du scénario. La vue *Creation Interaction* utilise également un *Spring Layout* agrégeant les instanciations de classes ensemble. Il est décoré, en donnant aux arrêtes une largeur proportionnelle aux nombres d'instanciations *classe à classe*.

4.4.2 Symptômes

Cette vue considère uniquement le niveau des créations d'objets, les formes extraites sont :

- Petits nœuds déconnectés représentent des classes n'ayant pas été instanciées durant l'exécution, et donc non utilisées.
- Les nœuds plats et clairs représentent des classes ayant créé un grand nombre d'instances, mais qui n'ont pas été souvent instanciées elles-mêmes. Un petit nombre d'instances de ces classes créent une multitude d'instances d'autres classes. Il est intéressant de constater qu'il est possible d'avoir une classe non-instanciées (le nœud étant complètement plat) créant un grand nombre d'objets. Ceci traduirait une classe abstraite dont les méthodes, invoquées par des sous-classes, créent des objets.
- Etroits nœuds foncés représentent des classes instanciées un très grand nombre de fois, mais dont les instances n'ont créé qu'un nombre restreints d'autres objets.
- Larges nœuds foncés représentent des classes lourdement instanciées dont les instances ont créé beaucoup d'objets durant l'exécution du système.

4.4.3 Cas d'étude

La Figure 4.3 montre l'application de la vue *Creation Interaction* sur l'exécution de notre cas d'étude. Quatre grandes tendances de classes ressortent :

1. Le groupe de classes de la partie supérieure de la vue, composé du sombre nœud étroit A et des nœuds plats A, offre une intéressante forme. Le nœud étroit représente la classe *AttributeDescription* une entité du méta-méta-modèle, instanciée tout au long de l'exécution par l'ensemble des entités du méta-modèle FAMIX.

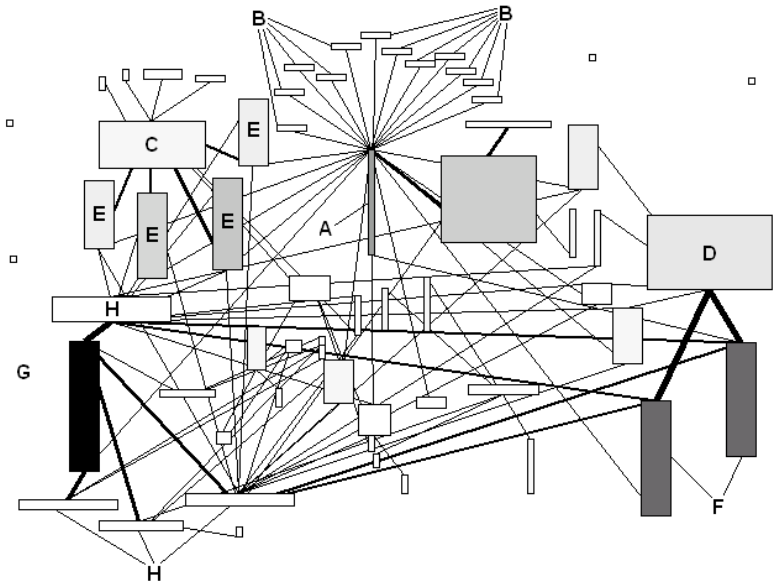

FIG. 4.3 – Application du Creation Interaction

2. Dans Moose, l'extraction du modèle d'un source code Smalltalk s'effectue en deux différentes étapes. Ces étapes sont effectuées par deux entités différentes au sein de l'architecture. Dans un premier temps, la classe *VWImporter* (C) utilise l'interface réflexive de Smalltalk afin de collecter des informations structurelles simples (classes, méthodes, attributs, etc...). La seconde étape de cette extraction est réalisée par la classe *VWParseTreeEnumerator* (D), décrivant un *Visiteur* extrayant de l'arbre de syntaxe abstraite des informations plus fines.
3. Le groupe de nœuds de la partie gauche de la vue représente la première phase d'extraction, dans laquelle nous remarquons que la grosse classe C crée un grand nombre d'instances des classes environnantes (E). À l'opposé de la vue, un autre groupe de nœuds décrit la seconde phase, lorsque le *VWParseTreeEnumerator* crée un important nombre de classe *Access* et *Invocation* (F) qui sont les entités les plus nombreuses de notre méta-modèle.
4. Finalement le groupe, dans le coin inférieur gauche de notre vue, révèle un aspect intéressant du système. Le gros nœud sombre G représente la classe *Measurement*, ces mesures traduisent les métriques du code source qui sont les entités les plus créées durant l'analyse d'un modèle. Le nombre de création est si grand, que ces entités ne sont pas présentées en mémoire mais stockées dans un fichier. Les instances de *Measurement* sont donc créées lors du chargement d'un fichier comme les autres entités de notre modèle. Cependant dans un second temps, elles seront effacées de la mémoire par le ramasse-miettes. La vue montre que, durant les chargements et les enregistrements du modèle du code source, des instances de *Measurement* sont créées. Les classes avoisinantes sont des classes responsables de ces entrées/sorties (H).

4.4.4 Résumé

La vue *Creation Interaction* résiste nettement mieux à la monter dans l'échelle que la vue *Communication Interaction*, bien qu'elles soient toutes deux basées sur l'enrichissement d'un *Spring Layout*. La probabilité de création d'une instance est moindre que l'invocation d'une méthode pour une classe. Cette différence explique le caractère moins chargé de la vue *Creation Interaction*, par rapport à la vue précédente.

4.5 Method Call Origin

Description du **Method Call Origin**	
Représentation	Repère normé
Nœuds	Méthodes
Portée	Système entier, sous système, classe unique
Echelle	Logarithmique
Coordonnée X	ETI (*Nombre d'invocations externes*)
Coordonnée Y	ITI (*Nombre d'invocations internes*)
Couleur des Nœuds	TI (*Nombre d'invocations totales*)
Figure	Figure 4.4

4.5.1 Intention de la vue

Les vues proposées précédemment sont orientées vers l'analyse du comportement des classes. L'avantage de notre approche réside dans sa flexibilité. Le *Method Call Origin* propose d'étudier l'utilisation des méthodes, les entités d'étude ne seront plus les classes du système mais les méthodes. Dans le but d'appréhender l'origine des invocations sur les méthodes durant l'exécution, la vue peut être appliquée, sans distinction, sur l'ensemble du système étudié ou sur des sous parties. Les nœuds de la vue, représentant les méthodes analysées, sont répartis sur un repère normé à échelle logarithmique. La coordonnée X représente le nombre d'invocations *interne* sur la méthode (une méthode de l'objet contenant la méthode est à l'origine de l'invocation), tandis que la coordonnée Y traduit le nombre d'invocations venant de l'*extérieur*. La couleur des nœuds, quant à elle, est fonction du nombre total d'invocations sur les méthodes correspondantes.

4.5.2 Symptômes

La vue *Method Call Origin* propose une compréhension de l'utilisation des méthodes analysées. Elle offre moins de facettes que les précédentes, ses signes graphiques et leurs interprétations sont moins complexes. Ci-dessous, sont représentés les informations visuelles issues de l'application de cette vue :

- Les nœuds, proches du coin supérieur gauche du graphe, représentent des méthodes très peu utilisées durant l'exécution.
- Les nœuds, proches de l'axe des abscisses, représentent les méthodes fréquemment invoquées par des objets étrangers. Elles jouent le rôle d'interface au sein de leur classe.
- Les nœuds, proches de l'axe des ordonnées, représentent les méthodes fréquemment invoquées par leur propriétaire. Leur rôle au sein de leur classe est un comportement interne.
- Les nœuds, proches de la première bissectrice du plan, représentent des méthodes hybrides servant d'interface avec l'extérieur et de comportement interne aux objets propriétaires.
- Les nœuds sombres représentent les méthodes les plus fréquemment utilisées du domaine d'étude.

4.5.3 Cas d'étude

Appliquée à l'ensemble de notre cas d'étude, la vue représentée par la Figure 4.4, montre différents aspects de l'utilisation des méthodes. Outre la possibilité d'analyser la fréquence d'invocation des méthodes étudiées, elle fournit l'origine de leurs utilisations. La première

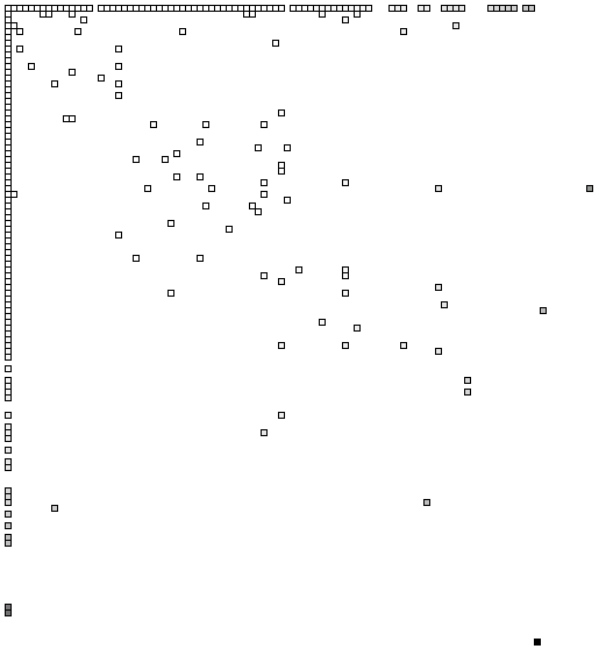

FIG. 4.4 – Application du Method Call Origin

constatation pouvant être tirée de la vue obtenue, réside dans la particularité du langage d'implémentation de notre cas d'étude. Le système étant écrit en Smalltalk, où l'ensemble des méthodes est public, il est intéressant de constater que nombre d'entre elles sont pourtant utilisées de manière interne. Elles pourraient donc être considérées comme privées.

Excepté la reconnaissance des méthodes les plus utilisées durant l'exécution. Nous constaterons dans cette analyse que le nuage de nœuds, situé aux alentours de la première bissectrice du plan, représente des accesseurs. À l'exception de quelques rares cas particuliers, cet état de fait traduit une bonne utilisation des méthodes de notre système durant son exécution.

4.5.4 Résumé

Un simple repère normé est un bon outil pour pressentir la distribution des éléments d'un système suivant deux mesures. En adaptant échelle logarithmique au plan, cette vue offre une montée dans l'échelle extrêmement robuste. Ainsi, l'analyse d'un système dans son entier ou d'une classe particulière sont possible, promulguant au *Method Call Origin* divers usages.

4.6 Discussion

Les quatre exemples exposés précédemment montrent qu'à partir de mesures calculées pendant l'exécution, il est possible d'effectuer une analyse fine des aspects comportementaux d'une application. Par cette méthode, nous évitons la collecte, le traitement et la visualisation des quantités d'informations colossales générées. Outre les avantages de stockage et de lisibilité, la technique est incrémentale. Il pourrait donc être possible de regarder s'additionner les informations durant le déroulement d'une application, voyant ainsi les évolutions comportementales au travers des modifications des vues. Dans la même idée, l'approche autorise d'analyser le déroulement d'exécution sur des programmes tournant en permanence, ce qui est impossible dans les solutions proposées dans la littérature.

Cependant, comme dans la plupart des domaines, il n'existe malheureusement pas de solutions miracles résolvant la totalité des problèmes. Indéniablement le *Spring Layout* souffre d'une perte de visibilité sur l'application à un gros système intensément communicatif. L'inconvénient majeur de l'approche réside dans la finesse de l'analyse résultante. En recherchant une condensation dans notre approche, nous perdons la vision bas niveau des séquences d'interaction qu'offrent les diagrammes de séquences.

Chapitre 5

Conclusion

L'approche présentée dans ce travail, appuyée sur un ensemble d'informations dynamiques condensées, propose de nombreux avantages dans l'aperçu du comportement d'une application. Les vues étudiées précédemment sont riches en interprétation. Elles disposent de multiples facettes, révélant chacune différents types d'informations sur le déroulement d'un système, à différents niveaux. Ces exemples ne sont cependant pas les seules vues intéressantes pouvant être issues d'une approche basée sur l'enrichissement de vues polymétriques. L'atout principal de cette solution est justement sa flexibilité. Un travail de recherche peut amener à faire ressortir de nouveaux types de schémas pertinents, traduisant des aspects comportementaux particuliers des applications. Dans ce but, la porte reste ouverte à de nombreuses expérimentations orientées vers la recherche de nouvelles vues possibles. Elles seront facilitées par l'utilisation de Divoor sur des cas concrets.

Dans ce rapport nous n'avons proposé d'étudier que deux types d'entités d'un système, les classes et les méthodes. Il serait intéressant de poursuivre ces investigations dans différentes voies, comme l'étude des utilisations des variables au sein des classes. Une autre voie que nous avons commencer à entrevoir et qui semblerait prometteuse, serait la visualisation de l'impact des tests unitaires d'une application, afin de cerner leur points forts et leurs faiblesses. Le regroupement des entités constitue une autre amélioration intéressante à étudier. Pour certaines vues, cela permettrait de mieux résister à la montée dans l'échelle.

Annexe A

Moose

Dans ce travail, Moose apparaît sous deux aspects. Il est utilisé pour l'implantation de notre approche ainsi que comme cas d'étude pour la valider. Dans cette annexe nous irons un peu plus loin qu'auparavant dans la description de l'architecture et des fonctionnalités de ce système.

Moose est un environnement d'ingénierie inverse et de re-ingénierie de systèmes décrits par un langage orienté-objet. Implanté avec VisualWorks, une implantation de Smalltalk, par le *Software Composition Group* [10], cet outil sert de fondation à plusieurs outils d'ingénierie inverse [19] [14]. Par le biais de modèles, il fournit une représentation indépendante de langages orientés-objet issue du méta-modèle FAMIX [8], décrivant les différents artefacts d'un système objet (attributs, méthodes, classes, paquetages, etc...) [7]. La figure ci-dessous représente les entités de bases et leurs relations employées par Moose, telles que le propose le méta-modèle FAMIX.

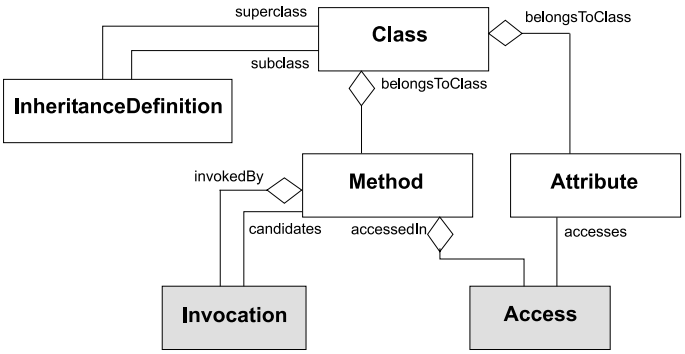

FIG. A.1 – Le modèle FAMIX

Moose est une plateforme d'ingénierie inverse indépendante des différents langages de programmation orientés-objet. La version actuelle supporte la modélisation d'applications écrites en Ada, C++ , Cobol, Java et Smalltalk. Elle permet également la génération de modèles à partir de l'interprétation de fichiers CDIF ou XMI, des fichiers d'échanges industriels. Le principal but de Moose est de pouvoir manipuler une représentation abstraite du code d'une application afin de fournit la brique de base à de nombreux outils d'ingénierie inverse ou de re-ingénierie. La Figure A.2 présente les parties majeures de son architecture et de ses fonctionnalités.

La trame d'importation et d'exportation de Moose joue le rôle de point d'entrée et de sortie des informations, entre les modèles générés par Moose et les code sources ou les fichiers d'échange extérieurs. Par son biais, il peut importer des systèmes de différentes sources. Pour le cas d'une application écrite en Smalltalk, sous VisualWorks, le code peut être chargé

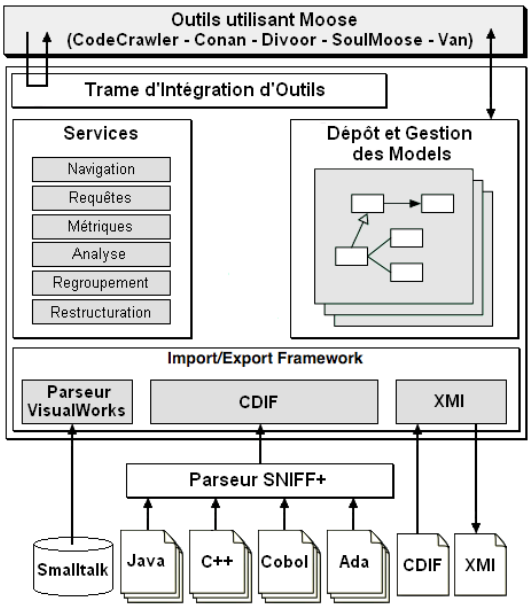

FIG. A.2 – L'architecture de Moose

directement à partir de l'image dans laquelle Moose s'exécute. Dans le cas d'un programme implémenté dans un autre langage à objet, l'utilisation de Snif+ permet de parser le code Ada, C++, Cobol et Java et d'en générer une représentation dans un format CDIF. À partir de fichiers à ce format, Moose est capable de générer le modèle du code source. Par ce biais, tous les langages, à partir desquels une représentation CDIF est possible, peuvent être utilisés avec Moose. L'utilisation direct de fichiers CDIF est évidemment une troisième option, pour la génération du modèle d'une application.

Dans Moose les systèmes informatiques sont représentés par des modèles Moose, suivant le méta-modèle FAMIX. Ces modèles peuvent être extériorisés dans des fichiers aux formats CDIF ou XMI utilisant la trame décrites précédemment dans le sens inverse. Le dépôt et la gestion de ces modèles sont les fonctionnalités qui seront utilisées par les applications construites *au-dessus* de Moose. Dans cette optique il propose de nombreux services utiles à la manipulation de représentations de code source :

- Navigation à l'intérieur des entités et des relations FAMIX
- Requêtes recherchant des entités adhérant à certains critères
- Représentation de métriques dans les modèles
- Regroupement d'entités au sein d'un groupe manipulable
- Restructuration du code source par l'intermédiaire du modèle

Le but principal de Moose est de fournir un noyau manipulant des représentations abstraites du code source d'applications pour faciliter la construction d'outils de re-ingénierie. Dans cette optique, il propose une trame d'intégration d'outils, fournissant une abstraction pour les applications voulant utiliser ses modèles. À l'heure actuelle, il existe plusieurs outils de re-ingénierie ou d'ingénierie inverse de haut niveau qui s'appuient sur Moose. À cette liste nous pouvant dorénavant rajouter Divoor, notre outil de visualisation et d'analyse d'informations dynamiques.

Annexe B

Divoor - Une implantation

De par la nature de notre approche, sa confrontation avec un cas d'étude dans le Chapitre 4 ne pouvait avoir lieu sans une implantation concrète. Divoor est l'outil de visualisation, implanté durant ce travail, dont sont issues les vues présentées précédemment. Développé en VisualWorks une distribution de Smalltalk, son fonctionnement est relativement simple. Dans un premier temps, Divoor génère une représentation interne (un modèle) du code source de l'application. À partir des évènements générés lors de l'exécution d'une application sous VisualWorks, il calcule des mesures dont il en décorera son modèle. L'utilisateur pourra ensuite utiliser différents types de graphes et les enrichir des mesures souhaitées, pour enfin visualiser la représentation du comportement voulue.

Le choix de Smalltalk comme langage d'implémentation de Divoor fut motivé par deux raisons. D'une part, cela m'a permis de découvrir ce langage controversé de l'informatique. D'un point de vue plus pratique, son homogénéité dans le concept objet offre de grandes facilités et souplesses d'utilisation. Divoor utilise trois applications développées sous VisualWorks. Pour la première étape de son fonctionnement, notre outil nécessite la construction de modèles représentant les entités des systèmes à analyser. Moose, le cas d'étude étudié dans ce rapport est un outil d'ingénierie inverse générant des modélisations d'applications orientées-objet (Annexe A). Il semble donc naturel de l'avoir choisi pour générer les modèles décorés par Divoor avant leur visualisation.

Durant l'exécution des systèmes auscultés, notre outil a besoin de capturer les évènements engendrés afin de calculer des mesures représentatives. Il utilise les Method Wrappers [2] autorisant un contrôle sûr et dynamique des méthodes d'une application sous VisualWorks. Grâce à cette librairie, Divoor est capable d'*intercepter* les messages au sein d'un système informatique. Les calculs de mesures prédéfinies résultant de combinaisons de messages deviennent alors triviales. Cependant, il est pertinent d'expliciter plus en détail la façon dont Divoor capture les créations d'instances au sein du système. La technique employée, également basée sur l'enrobage de méthodes fourni par les Method Wrappers, est cependant plus subtile. L'homogénéité de Smalltalk offre de très nombreux avantages, dont le fait que la création d'instance n'est pas délégué à un opérateur (comme en Java) mais par la méthode *#new* de la classe *Behavior* dont hérite toutes classes instanciables du système. La création d'instance peut donc être réifier aisément. Comme précédemment, cette méthode est enrobée afin de récupérer les appels et donc les créations du système. En pratique cette partie n'est pas aussi triviale, si on veut connaître les classes créatrices d'instance. Sans entrer dans les détails, cette difficulté résulte de l'attribution de la création à une classe. Par exemple, il est légitime de se demander quelle classe est responsable de la création d'une instance lorsqu'une classe *A*, ayant une classe mère *B* redéfinissant la méthode *#new*, appelle crée une instance durant une exécution. Dans une implémentation naïve de notre solution, lorsque la méthode *#new* de la classe *A* est appelée par une classe *C*. Le message remontera par héritage à la redéfinition du *#new* par la classe *B*. Dans la suite du processus ce sera cette dernière

classe qui renvoiera un message à la méthode *#new* de la classe *Behavior* responsable de la création effective d'une nouvelle instance. Du point de vue de l'enrobage placé sur cette dernière méthode, l'envoi de message venant de la classe B, elle sera désignée comme la créatrice de sa classe fille A. Cependant, à un niveau conceptuel, il nous intéressait de voir que la classe C était bien l'instigatrice de la création. Il existe des cas encore plus complexes qui sont détectés dans l'implémentation actuelle de Divoor. Il serait, tout de même, judicieux de poursuivre des investigations dans ce sens dans le but d'éviter les falsifications de créateurs d'instances.

Une fois ce travail d'extraction d'informations terminé, notre outil est capable de leur attribuer des algorithmes de placement, afin de représenter des vues telles que nous avons pu voir dans ce rapport. Dans ce but, Divoor utilise en partie les fonctionnalités de visualisation offerte par CodeCrawler [19] [20]. Il utilise toute la machinerie de visualisation et de manipulation inhérente à cette outil de re-ingénierie.

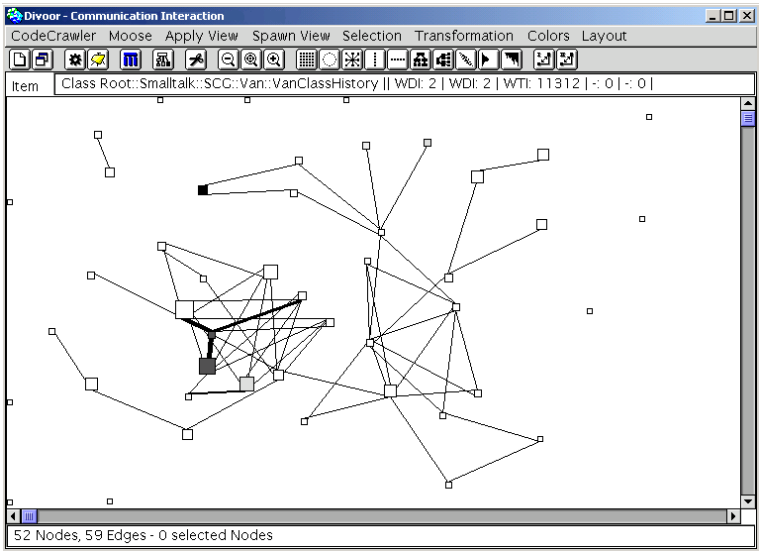

FIG. B.1 – Divoor au travail

L'image ci-dessus représente Divoor traduisant le comportement d'une application, par l'application d'une vue. L'outil permet aisément la définition de nouvelles vues. Les informations sont récoltées durant l'exécution, la visualisation est possible uniquement à la fin du déroulement du scénario. L'approche utilisée autorise parfaitement d'incrémenter les vues tout au long de l'exécution. La version actuelle de Divoor n'interdit en rien une extension de l'application vers cette fonctionnalité. Sa mise en œuvre ne devrait poser aucun problème majeur d'implantation.

Bibliographie

[1] P. Anderson and T. Teitelbaum. Software inspection using codesurfer. In *Proceedings of WISE'01 (International Workshop on Inspection in Software Engineering*, 2001.

[2] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the Rescue. In *Proceedings ECOOP '98*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.

[3] S. Chiba and T. Masuda. Designing an extensible distributed language with a meta-level architecture. In O. Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 483–502, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[4] T. Corbi. Program understanding : Challenge for the 1990's. *IBM Systems Journal*, 28(2) :294–306, 1989.

[5] A. M. Davis. *201 Principles of Software Development*. McGraw-Hill, 1995.

[6] S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In F. Balmas, M. Blaha, and S. Rugaber, editors, *Proceedings WCRE '99 (6th Working Conference on Reverse Engineering)*. IEEE, Oct. 1999.

[7] S. Demeyer, S. Ducasse, and S. Tichelaar. Why unified is not universal. UML shortcomings for coping with round-trip engineering. In B. Rumpe, editor, *Proceedings UML '99 (The Second International Conference on The Unified Modeling Language)*, volume 1723 of *LNCS*, Kaiserslautern, Germany, Oct. 1999. Springer-Verlag.

[8] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 – the FAMOOS information exchange model. Technical report, University of Bern, 2001.

[9] S. Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6) :39–44, June 1999.

[10] S. Ducasse, M. Lanza, and S. Tichelaar. Moose : an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, June 2000.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, Mass., 1995.

[12] D. Jerding and S. Rugaber. Using Visualization for Architectural Localization and Extraction. In *Proceedings WCRE*, pages 56 – 65. IEEE, 1997.

[13] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing Message Patterns in Object-Oriented Program Executions. Technical Report GIT-GVU-96-15, Georgia Institute of Technology, may 1996.

[14] G. G. Koni-N'sapu. A scenario based approach for refactoring duplicated code in object oriented systems. Diploma thesis, University of Bern, June 2001.

[15] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3) :155–163, 1988.

[16] B. Korel and J. Rilling. Dynamic program slicing in understanding of program execution. In *5th International Workshop on Program Comprehension (WPC '97)*, pages 80–85, 1997.

[17] D. Lange and Y. Nakamura. Program explorer : A program visualizer for C++. In *Proceedings of Usenix Conference on Object-Oriented Technologies*, pages 39–54, 1995.

[18] D. B. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings of OOPSLA '95*, pages 342–357. ACM Press, 1995.

[19] M. Lanza. Codecrawler - lessons learned in building a software visualization tool. In *Proceedings of CSMR 2003*. IEEE Press, 2003.

[20] M. Lanza. *Object-Oriented Reverse Engineering - Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, may 2003.

[21] M. Lanza and S. Ducasse. Polymetric views - a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 2003.

[22] W. D. Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings OOPSLA '93*, pages 326–337, Oct. 1993.

[23] W. D. Pauw, D. Kimelman, and J. Vlissides. Modeling object-oriented program execution. In M. Tokoro and R. Pareschi, editors, *Proceedings ECOOP '94*, LNCS 821, pages 163–182, Bologna, Italy, July 1994. Springer-Verlag.

[24] W. D. Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proceedings Conference on Object-Oriented Technologies and Systems (COOTS '98)*, pages 219–234. USENIX, 1998.

[25] R. Smith and B. Korel. Slicing event traces of large software systems. In *Automated and Algorithmic Debugging*, 2000.

[26] I. Sommerville. *Software Engineering*. Addison Wesley, sixth edition, 2000.

[27] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *Proceedings OOPSLA '98*, ACM SIGPLAN, pages 271–283. ACM, Oct. 1998.

[28] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12) :1038–1044, Dec. 1992.