Université de Montréal

Analyse de dépendance des programmes à objet en utilisant les modèles probabilistes des entrées

par

Arbi Bouchoucha

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences - Université de Montréal

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

Juin, 2011

© Arbi Bouchoucha, 2011

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé :

Analyse de dépendance des programmes à objet en utilisant les modèles probabilistes des entrées

Présenté par :

Arbi Bouchoucha

a été évalué par un jury composé des personnes suivantes :

Fabian Bastin, président-rapporteur

Houari A. Sahraoui, directeur de recherche

Pierre l'Écuyer, codirecteur

Bruno Dufour, membre du jury

Mémoire accepté le : …………………..

RÉSUMÉ

La tâche de maintenance ainsi que la compréhension des programmes orientés objet (OO) deviennent de plus en plus coûteuses. L'analyse des liens de dépendance peut être une solution pour faciliter ces tâches d'ingénierie. Cependant, analyser les liens de dépendance est une tâche à la fois importante et difficile. Nous proposons une approche pour l'étude des liens de dépendance internes pour des programmes OO, dans un cadre probabiliste, où les entrées du programme peuvent être modélisées comme un vecteur aléatoire, ou comme une chaîne de Markov. Dans ce cadre, les métriques de couplage deviennent des variables aléatoires dont les distributions de probabilité peuvent être étudiées en utilisant les techniques de simulation Monte-Carlo. Les distributions obtenues constituent un point d'entrée pour comprendre les liens de dépendance internes entre les éléments du programme, ainsi que leur comportement général. Ce travail est valable dans le cas où les valeurs prises par la métrique dépendent des entrées du programme et que ces entrées ne sont pas fixées à priori. Nous illustrons notre approche par deux études de cas.

Mots-clés : **Dépendance interne, comportement du programme, simulation Monte-Carlo, modèle probabiliste.**

ABSTRACT

The task of maintenance and understanding of object-oriented programs is becoming increasingly costly. Dependency analysis can be a solution to facilitate this engineering task. However, dependency analysis is a task both important and difficult. We propose a framework for studying program internal dependencies in a probabilistic setting, where the program inputs are modeled either as a random vector, or as a Markov chain. In that setting, coupling metrics become random variables whose probability distributions can be studied via Monte-Carlo simulation. The obtained distributions provide an entry point for understanding the internal dependencies of program elements, as well as their general behaviour. This framework is appropriate for the (common) situation where the value taken by the metric does depend on the program inputs and where those inputs are not fixed a priori. We provide a concrete illustration with two case studies.

Keywords: **Internal dependency, program behaviour, Monte-Carlo simulation, probabilistic model.**

TABLE DES MATIERES

LISTE DES TABLEAUX ... **vi**
LISTE DES FIGURES ... **vii**
Remerciements ... **ix**
Chapitre 1 ... **1**
INTRODUCTION ... **1**
1.1. Contexte général ... 1
1.1.1. Problématique générale ... 1
1.1.2. Besoins d'analyser les dépendances ... 3
1.2. Problème spécifique ... 4
1.3. Approche et contributions ... 5
1.4. Plan du mémoire ... 6
Chapitre 2 ... **7**
ÉTAT DE L'ART ... **7**
2.1. Introduction ... 7
2.2. Représentation des dépendances par les métriques de couplage ... 7
2.2.1. Les métriques en tant qu'indicateur de qualité ... 7
2.2.2. Définition et utilisation des métriques statiques de couplage ... 10
2.2.3. Définition et utilisation des métriques dynamiques de couplage ... 12
2.3. Analyse de dépendance ... 15
2.4. Utilisation des modèles probabilistes et des techniques de simulation en génie logiciel ... 19
2.4.1. Utilisation des modèles probabilistes en génie logiciel ... 19
2.4.2. Utilisation des techniques de simulation en génie logiciel ... 22
2.5. Synthèse ... 27
2.6. Conclusion ... 28
Chapitre 3 ... **30**
CARACTÉRISATION DES DÉPENDANCES ... **30**

3.1. Introduction 30
3.2. Schéma global de l'approche 30
3.3. Les modèles probabilistes des entrées 32
3.3.1. Entrées définies par un vecteur aléatoire 33
3.3.2. Entrées définies par une chaîne de Markov 37
3.4. Utilité des histogrammes 40
3.5. Conclusion 40
Chapitre 4 41
ÉTUDE DES DÉPENDANCES 41
4.1. Introduction 41
4.2. Rôle d'une classe 41
4.3. Liens entre les dépendances et le rôle d'une classe 43
4.4. Similarité entre exécutions et entre groupes d'exécutions 45
4.4.1. Définitions 45
4.4.2. Similarité intra-bloc (ou similarité interne) 46
4.4.3. Similarité inter-blocs (ou similarité externe) 47
4.4.4. Exemple 49
4.5. Conclusion 52
Chapitre 5 53
ÉTUDES DE CAS 53
5.1. Introduction 53
5.2. Implémentation 53
5.3. Cadre expérimental 57
5.3.1. Description des programmes 57
5.3.2. Les métriques de couplage dynamiques 60
5.3.3. Les modèles probabilistes des entrées 61
5.4. Observations 63
5.4.1. Patron de dépendance 1 : Chaîne de montage *(Assembly-Chain Worker)* 64
5.4.2. Patron de dépendance 2 : Employé *(Clerk)* 64

5.4.3. Patron de dépendance 3 : Soldat *(Soldier)* 65
5.4.4. Patron de dépendance 4 : Secrétaire *(Secretary)* 65
5.5. Résultats et interprétations 67
5.5.1. Cas 1: Cas d'une chaîne de Markov (système d'ascenseurs) 67
5.5.2. Cas 2: Cas d'un vecteur aléatoire (générateur de grilles de Sudoku) 77
5.6. Discussion 82
5.7. Conclusion 84
Chapitre 6 85
CONCLUSION 85
6.1. Rétrospective 85
6.2. Contributions 87
6.2.1. Cadre global pour lier les dépendances au rôle d'une classe 87
6.2.2. Définition des modèles probabilistes des entrées 87
6.2.3. Introduction des patrons de dépendance 87
6.2.4. Analyse et compréhension de dépendance 88
6.3. Perspectives futures 88
BIBLIOGRAPHIE 90
Annexe I : Contexte et notions de base en simulation i
Annexe II : Distributions du système d'ascenseurs iii
Annexe III : Distributions du générateur de Sudoku v

LISTE DES TABLEAUX

Tableau 1. Exemple d'un bloc d'exécutions B_2 correspondant à la valeur de couplage 2... 50

Tableau 2. Exemple d'un bloc d'exécutions B_3 correspondant à la valeur de couplage 3... 50

Tableau 3. Exemple d'un bloc d'exécutions B_4 correspondant à la valeur de couplage 4... 50

Tableau 4. Quelques statistiques sur les classes du système des ascenseurs. 72

Tableau 5. Quelques statistiques sur la classe *Elevator*. 73

Tableau 6. Quelques statistiques sur les classes du générateur de grilles de Sudoku. 81

Tableau 7. Quelques statistiques sur la classe *Solver*. 82

LISTE DES FIGURES

Figure 1. Synthèse sur les principaux travaux reliés à notre domaine de recherche............ 29
Figure 2. Schéma global de notre approche. 31
Figure 3. Exemple de deux distributions ayant la même moyenne 100, mais avec des allures différentes.. 36
Figure 4. Fréquence des variantes d'un service pour une classe... 44
Figure 6. Architecture générale de l'implémentation de notre approche. 54
Figure 7. Exemple d'utilisation de *J-Tracert* sur un programme Java. 56
Figure 8. Interface du système des ascenseurs.. 58
Figure 9. Exemple d'une grille de Sudoku initialisée. 59
Figure 10. La métrique de couplage dynamique *IC_CM* (cl_1). 60
Figure 11. La métrique de couplage dynamique *IC_OM* (cl_1).. 61
Figure 12. Les régions d'exécutions pour la classe *Elevator*.. 75
Figure 13. Proportions et intervalles des valeurs pour une distribution normale................ 76
Figure 14. Situation de la région R_2 par rapport à la plage de valeurs [μ' – σ', μ' + σ'] pour la classe *ElevatorGroup*. 77

À mes parents, pour tous les sacrifices
qu'ils ont consentis à faire pour que
je puisse être là aujourd'hui,
pour leurs entraides et leur soutien moral
à chaque fois que j'en ai besoin, et surtout
pour leurs conseils utiles pour m'encourager
à aller toujours en avant.

À ma sœur, pour sa sympathie, son amour
et pour tous les petits caprices qu'on a
partagés durant notre enfance.

À ma fiancée, pour ses entraides
et son amour inconditionnel,
et pour sa présence morale et ses conseils
pour me motiver à aller toujours en avant.

Aucun hommage ne peut être à la hauteur de l'amour
et de l'affection dont ils ne cessent de me combler,
qu'ils trouvent dans ce travail un modeste témoignage
de mon profond amour.

Et enfin, à tous ceux qui me sont chers
et qui m'ont aidé de proche ou de loin
pour la réalisation de ce travail,
je dis ***MERCI****.*

Remerciements

C'est avec un grand plaisir que je réserve ces lignes en signe de gratitude et de reconnaissance à tous ceux qui ont contribué de près ou de loin à l'élaboration de ce travail.

Je tiens tout d’abord à adresser mes sincères remerciements à mon directeur de recherche Dr Houari Sahraoui qui m’a soutenu et prodigué par ses conseils enrichissants tout au long de ma maitrise. Je tiens également à lui exprimer ma profonde reconnaissance pour sa disponibilité, sa confiance et son assistance technique et morale.

Je tiens ensuite à remercier Dr Pierre L’Écuyer qui a consacré de son temps pour m’aider et me diriger tout au long de la réalisation de mon projet de maitrise.

Je remercie également Dr Fabian Bastin et Dr Bruno Dufour d’avoir accepté la charge d’évaluer ce travail.

J'exprime aussi ma gratitude envers les membres de l'unité de recherche GEODES, Omar, Aymen, Houssem, Martin, … pour l'atmosphère amicale et l'ambiance chaleureuse qui règne au sein de l'équipe.

J’adresse une pensée toute particulière et je témoigne toute mon affection à mes chers parents qui ont su m’apporter tout leur soutien, leurs pensées, leurs prières et leur amour à distance, mais à tout instant.

J’exprime également ma gratitude à tous mes enseignants, qui ont contribué chacun dans son domaine, à ma formation universitaire, sans laquelle je ne serai jamais arrivé à réaliser ce travail, ainsi que le gouvernement Tunisien qui a participé au financement de ce mémoire de maîtrise.

Finalement, je dis MERCI à tous ceux et celles qui ont contribué de près ou de loin à la réalisation et la réussite de ce travail modeste.

Chapitre 1

INTRODUCTION

Nous entamons ce chapitre introductif par la présentation de la problématique générale de nos travaux. Celle-ci est liée principalement au coût élevé de la maintenance et la difficulté de comprendre les liens de dépendance dans les programmes orientés-objet (OO). Ensuite, nous discutons du besoin d'analyser ces dépendances, ainsi que leurs rôles pour faciliter la compréhension du comportement global des programmes. Enfin, avant de donner un aperçu des différents chapitres de notre mémoire, nous résumons nos principales contributions.

1.1. Contexte général

1.1.1. Problématique générale

De nos jours, les produits logiciels deviennent de plus en plus complexes et leur maintenance devient plus coûteuse en termes de temps et de ressources. De plus, les systèmes industriels actuels nécessitent le développement de logiciels de grande taille et de très bonne qualité du point de vue de la maintenabilité et de la réutilisabilité. Pour ces raisons, la maintenance est devenue l'une des phases les plus importantes, mais aussi l'une des plus coûteuses du cycle de vie du logiciel. Ceci peut s'expliquer en partiel par le fait qu'elle est la phase la plus longue qui ne s'achève qu'avec la fin de vie du logiciel.

Dans l'industrie, le coût de maintenance d'un logiciel (par rapport à son budget total) est passé de 40 à 60% dans les années 80, à plus de 75% ou même 80% au début de l'an 2000 [45][58]. On estime également que plus de la moitié de cette maintenance est consacrée à la compréhension du programme lui-même [15]. Ce coût élevé de la maintenance et surtout de la compréhension de programmes, pousse les chercheurs et les industriels à se focaliser sur cette phase du cycle de vie d'un système afin d'essayer de comprendre les facteurs qui influent ce coût.

Durant la phase de maintenance, les développeurs sont censés modifier le code du programme (par ajout, modification, et/ou suppression de classes, de méthodes, etc.). Cette tâche n'est plus simple. En effet, pour pouvoir introduire les bonnes modifications au niveau du programme, le développeur doit comprendre en détail le programme à maintenir et plus précisément les différents liens qui existent entre ses classes et ses méthodes. Ceci n'est pas facile, surtout pour de grands programmes contenant des centaines, voire même des milliers de classes. Ainsi, la maîtrise de la compréhension du programme est indispensable.

On distingue deux façons d'agir sur la compréhension de programmes. La première consiste à rendre le code compréhensible grâce aux bonnes pratiques et le « refactoring » qui consiste à restructurer le code pour améliorer sa lisibilité sans changer sa fonctionnalité [7][61][67]. Quant à la deuxième façon, elle consiste à utiliser des outils et des techniques existantes qui permettent d'explorer le programme en vue de le comprendre, telle que la visualisation qui est une méthode de représentation graphique servant à afficher des données complexes de manière simple [32][34].

D'autre part, la programmation OO a été définie de manière à modéliser des entités du monde réel, c'est-à-dire, la définition d'objets correspondants à des entités (contrairement à la programmation procédurale qui définit une suite de fonctions censées représenter des traitements sur des données). Chaque objet est considéré comme une entité qui possède ses propres caractéristiques, et qui a un comportement spécifique décrit par la liste des méthodes qu'il utilise. Par conséquent, il est facile de comprendre le comportement individuel de chaque objet. Cependant, ces objets ont été également conçus pour qu'ils interagissent entre eux par envoi et réception de messages. Cette communication est primordiale et permet de définir le comportement du programme et de réaliser ses différentes fonctionnalités. Il est donc important d'examiner et de comprendre les interactions entre ces objets. Néanmoins, bien que cette tâche soit importante, elle n'est pas triviale. Ainsi, on a besoin d'analyser et de comprendre les liens de dépendance entre les différents objets d'un programme afin de faciliter la tâche de maintenance et de prédire son comportement général.

1.1.2. Besoins d'analyser les dépendances

Les liens de dépendance contiennent des informations concernant les relations qui existent entre les différents composants d'un programme, telles que les interactions entre les modules (les fichiers par exemple), l'utilisation de variables et leurs types, ainsi que les appels entre les méthodes [35]. Par exemple, une classe *X dépend* d'une autre classe *Y* au cas où *X* instancie un objet de *Y*, ou une méthode de *X* utilise le résultat d'une méthode de *Y*, ou *X* hérite de *Y*, etc. Analyser les liens de dépendance permet d'extraire ces informations et essayer de les comprendre. Plus globalement, cette analyse consiste à faire le lien entre les dépendances et le comportement du programme. Plus précisément, il s'agit de comprendre la participation d'une classe dans le comportement global d'un programme.

L'importance d'analyser les liens de dépendance réside dans le fait que cette tâche d'ingénierie est utile pour mener plusieurs objectifs, par exemple :

- ✓ Faciliter la modification du code [53].
- ✓ Établir des contraintes sur l'évolution de l'architecture du logiciel [17].
- ✓ Diagnostiquer des fautes (test) au niveau du code [5].
- ✓ Réaliser des exécutions symboliques du programme [54].

Nous allons détailler tous ces aspects et d'autres dans le chapitre 2.

Jusqu'à présent, la plupart des travaux utilisent une représentation *explicite* des liens de dépendance, c'est-à-dire, en utilisant des graphes qui représentent les différents composants d'un programme et leurs liens d'interaction, comme celui illustré dans [5]. Bien évidemment, les liens de dépendance peuvent être également représentés de façon *implicite* à l'aide des métriques de couplage. Ces métriques permettent entre autres de résumer ces liens de dépendance. En effet, d'après [14] et [48], le couplage est défini par le nombre des interconnections entre les activités d'un programme ou d'un modèle de programme. Le degré de couplage dépend de la façon avec laquelle ces connexions sont compliquées, et aussi du type de ces connexions entre les différentes activités.

Les métriques de couplage ont été utilisées dans plusieurs domaines en génie logiciel pour résoudre différents problèmes, par exemple, étudier la propension de changements au niveau des codes OO [3], comprendre le comportement dynamique des programmes OO

[1], faciliter la tâche de maintenance d'un logiciel pour empêcher la détérioration de l'architecture du code lors du cycle d'évolution [17], analyser les liens de dépendance entre les composants d'un logiciel [53], etc.

1.2. Problème spécifique

L'analyse de dépendance peut se faire de manière statique, c'est-à-dire, en extrayant des liens de dépendance à partir du code source du programme sans même avoir besoin de l'exécuter. On appelle ce type de dépendances : *dépendances statiques* [35]. Ces dépendances statiques ne prennent pas en compte les mécanismes dynamiques des langages modernes, par exemple le polymorphisme et le chargement dynamique sont des tâches difficiles à réaliser en utilisant juste une analyse statique. Les dépendances statiques donnent donc une vue globale, mais incomplète du comportement du programme.

Par ailleurs, *les dépendances dynamiques* ont été proposées pour analyser les interactions entre les éléments du programme. Ces dépendances sont extraites en observant l'exécution du programme [35]. Cette analyse dynamique de dépendances permet, entre autres, de résoudre le problème des dépendances statiques par la prise en compte des scénarios spécifiques du programme, c'est-à-dire, les variations de son comportement. De plus, l'analyse dynamique donne plus de précision pour faire le polymorphisme et le chargement dynamique, par rapport aux techniques d'analyse statique. Néanmoins, l'analyse dynamique des liens de dépendance est spécifique à une exécution particulière. Ainsi, il n'est pas possible de généraliser le comportement d'un programme juste en analysant dynamiquement ses liens de dépendance pour une seule exécution.

Le problème que nous traitons dans ce mémoire est celui de la généralisation de l'analyse dynamique des dépendances. Pour y parvenir, nous proposons une approche qui consiste à déterminer dans un premier temps un échantillon représentatif des exécutions d'un programme. Par la suite, nous analysons les dépendances extraites de ces exécutions.

1.3. Approche et contributions

Nous proposons une approche semi-automatique pour simplifier l'analyse et la compréhension des liens de dépendance en nous basant sur l'utilisation des métriques de couplage dynamiques et des modèles probabilistes des entrées. Ces modèles permettent de générer un échantillon représentatif des possibilités d'exécutions et peuvent être appliqués à l'aide des techniques de simulation stochastique Monte-Carlo. Nos modèles probabilistes couvrent tout type de programme ayant un nombre limité d'entrées ou pas. Pour cela, nous modélisons les entrées du programme par un vecteur aléatoire au cas où le nombre des entrées est fixé d'avance, et par une chaîne de Markov dans le cas contraire. Ces modèles servent à générer un échantillon parmi toutes les possibilités des exécutions du programme.

Lors de chaque nouvelle exécution, notre simulateur génère les entrées au programme et une valeur de métrique de couplage est calculée dynamiquement pour chaque classe du programme. Cette valeur est calculée depuis une liste de traces d'exécutions obtenues en utilisant un outil de collecte de traces. Après un échantillon de n exécutions, on obtient n valeurs de métriques pour chaque classe. Ces valeurs nous permettent de générer automatiquement des histogrammes. Ces derniers décrivent les distributions des métriques de couplage de chaque classe. Ces distributions sont générées grâce à notre programme de simulation.

Lors de l'analyse de nos histogrammes, nous avons observé l'existence de certaines distributions de métriques qui apparaissent souvent. Ces observations nous ont menées à introduire des patrons de dépendance qui nous permettent d'analyser le comportement global d'une classe et son rôle dans le système, en utilisant sa distribution de probabilité de métriques. Ces patrons constituent un point d'entrée pour explorer les liens de dépendance internes. Afin d'établir le lien entre les dépendances et le rôle d'une classe dans un programme, nous avons défini de nouvelles mesures pour calculer des facteurs de similarité internes et externes entre les possibilités des exécutions.

Les principales contributions de ce travail peuvent être résumées comme suit :

- ✓ Décrire un cadre général pour lier les dépendances avec le comportement global d'un programme (basé sur le rôle des classes qui forment le système).
- ✓ Définir des modèles probabilistes pour simuler les entrées d'un programme par un vecteur aléatoire ou bien par une chaîne de Markov.
- ✓ Introduire de nouveaux patrons de dépendance qui, à partir de la distribution d'une classe, permettent de comprendre le rôle de cette classe dans le programme, ainsi que ses liens de dépendance internes avec le reste du système.
- ✓ Étudier et comprendre les dépendances en utilisant de nouvelles mesures de similarité permettant d'identifier les exécutions dans lesquelles une classe joue un rôle typique, ainsi que de déterminer des régions d'exécutions qui correspondent à un comportement spécifique du système.

1.4. Plan du mémoire

Le reste de ce mémoire est organisé comme suit. Le deuxième chapitre présente un survol des principaux travaux en relation avec l'analyse de dépendance, l'utilisation des métriques de couplage (statiques et dynamiques) pour mesurer et comprendre les dépendances, et l'exploitation des techniques de simulation et des modèles probabilistes en génie logiciel. Dans le troisième chapitre, nous allons donner les détails de nos modèles probabilistes pour le cas du vecteur aléatoire et le cas de la chaîne de Markov. L'étude des liens de dépendances est présentée dans le cadre du quatrième chapitre dans lequel nous allons présenter les liens entre les dépendances et le rôle d'une classe dans un programme, ainsi que les mesures de similarité (internes et externes) permettant d'identifier les exécutions typiques dans un programme. Les résultats de l'évaluation de notre approche sont présentés et discutés dans le cinquième chapitre à l'aide de deux études de cas. Enfin, le dernier chapitre est consacré à la récapitulation des idées principales introduites dans ce mémoire, ainsi qu'à la description de quelques perspectives liées à nos futurs travaux de recherche.

Chapitre 2

ÉTAT DE L'ART

2.1. Introduction

Dans ce chapitre, nous nous intéressons aux travaux reliés à notre domaine de recherche. Notre projet couvre trois domaines de recherche différents : la mesure des dépendances par les métriques de couplage statiques et dynamiques, l'analyse des dépendances, et l'utilisation des techniques de simulation et des modèles probabilistes pour résoudre des problèmes en génie logiciel. Nous avons donc articulé le présent chapitre autour de trois grandes parties. Dans chacune de ces trois parties, nous allons citer certains problèmes traités dans l'un de ces trois axes de recherche ainsi que les approches et les techniques proposées pour les résoudre.

2.2. Représentation des dépendances par les métriques de couplage

2.2.1. Les métriques en tant qu'indicateur de qualité

De nos jours, la structure des logiciels devient de plus en plus complexe, ce qui rend difficile à l'être humain de pouvoir comprendre les codes de ces programmes. Ces derniers sont compris par les développeurs du logiciel seulement. La complexité des programmes est due généralement à la difficulté de comprendre les liens d'interactions qui existent entre ses différents composants. Pour cela, les chercheurs ont décidé d'utiliser des *métriques*. Ce sont des indicateurs pour les caractéristiques de qualité définies à un niveau supérieur. Les métriques sont en fait des mesures [18] qu'on prend sur le code ou sur d'autres artefacts résultant du processus de développement du logiciel. Quelques exemples de mesures sont le nombre de lignes de code, d'attributs ou encore de méthodes dans une classe, le nombre des appels qui existent entre les méthodes, les classes et les objets d'un programme.

Plusieurs métriques ont été définies. Initialement, on trouve celles proposées dans les travaux de Chidamber et Kemerer [12] [13] qui ont introduit six métriques candidates pour la mesure de qualité. Par exemple, ils ont défini la métrique de couplage *CBO* (*Coupling Between Objects*) pour mesurer le degré de couplage, *WMC* (*Weighted Method per Class*) pour mesurer la taille et la complexité, *LCOM* (*Lack of COhesion in Method*) pour la mesure de cohésion entre les méthodes, *DIT* (*Depth in Inheritance Tree*) pour mesurer la profondeur dans l'arbre d'héritage, etc. Dans ce mémoire, nous nous intéressons seulement aux *métriques de couplage*. Ces dernières indiquent le niveau d'interaction entre deux ou plusieurs composants d'un programme. C'est pour cela que la plupart des études qui analysent les liens de dépendance se basent sur les métriques de couplage.

Il existe des études qui montrent que les métriques de couplage sont efficaces et de bons indicateurs de certains axes de la qualité d'un logiciel. Poshyvanyk *et al.* [48] ont pu montrer qu'il y a une relation entre les métriques de couplage et l'analyse de l'impact de changements lors de l'évolution d'un logiciel. Leur étude utilise plutôt les techniques de recherche documentaire pour calculer les valeurs de couplage. Il s'agit en fait d'analyser le code source du programme (vu sous forme de texte) et calculer ensuite la fréquence d'un mot donné dans ce texte. Ils ont validé leur approche sur le système Mozilla et ont montré une grande liaison entre les valeurs de couplage obtenues et l'aptitude d'une classe à changer d'une version à une autre.

Dans ce même contexte, Briand *et al.* [10] ont fait une étude sur un système commercial dont ils possèdent plusieurs versions et une liste de changements collectées depuis des années. L'objectif majeur étant de déterminer à quel point les métriques de couplage sont capables de détecter les classes susceptibles de changer d'une version à une autre. Les résultats obtenus montrent que les valeurs de couplage générées renseignent avec une grande probabilité sur les listes de changements appliqués d'une version à une autre. Ainsi, on déduit de cette étude que les métriques de couplage sont un bon indicateur de l'impact de changement. Les auteurs ont conclu qu'un tel modèle à base de métriques de couplage peut aider à analyser les dépendances, ce qui minimise l'effort consacré pour détecter et comprendre les impacts de changements.

D'autre côté, Binkley et Schach [6] ont défini une nouvelle métrique de couplage qu'ils ont nommée *CDM* (*Coupling Dependency Metric*). Ils ont ensuite appliqué 16 métriques différentes (incluant *CDM*) sur quatre programmes différents et ont montré que les métriques de couplage peuvent être un bon indicateur pour détecter les exécutions qui génèrent des erreurs (« bug »). Ces métriques de couplage interviennent également dans la tâche de maintenance. En effet, les auteurs montrent via plusieurs études que s'il existe un obstacle au niveau de la tâche de maintenance, alors, avec grande probabilité, il sera dû à un problème d'interaction ou de couplage entre les différents modules qui constituent le programme. Les modules qui ont un niveau de couplage faible présentent plus de fautes et sont donc plus difficiles à maintenir.

On distingue deux grands types de métriques de couplage, à savoir les métriques de couplage *statiques* et les métriques de couplage *dynamiques*. Les métriques de couplage statiques sont extraites à partir du code source du programme et sont calculées sur plusieurs exécutions, alors que les métriques de couplage dynamiques requièrent que le programme s'exécute pour être comptabilisées. Chacun de ces deux types de métriques a ses avantages et ses inconvénients. En effet, les métriques statiques ne prennent pas en compte ce qui se passe *au moment de l'exécution* du programme ni les différentes variations de son comportement (c'est-à-dire, les valeurs décidées dynamiquement dans le programme, le polymorphisme et le chargement dynamique), tandis que les métriques de couplage dynamiques prennent en compte toutes ces variations [3]. D'autre côté, les métriques de couplage dynamiques sont calculées sur une seule exécution, et donc ne reflètent pas le comportement général du programme, contrairement aux métriques de couplage statiques qui sont extraites à partir du code source du programme et valides pour toutes les exécutions.

Dans ce travail, nous allons combiner ces deux types de métriques dans le sens où nous allons chercher un compromis entre la précision (grâce aux métriques dynamiques) et la généralisation (grâce aux métriques statiques), ce qui permet donc de rendre les métriques de couplage dynamiques plus généralisables (ou complètes).

2.2.2. Définition et utilisation des métriques statiques de couplage

Plusieurs travaux ont étudié les métriques logicielles de couplage de manière statique et les ont exploitées dans divers domaines. Entre autres, ils ont montré leur importance pour évaluer et améliorer la qualité des produits logiciels. Dans cette sous-section, nous allons débuter par citer quelques études qui ont défini et bénéficié des métriques statiques de couplage, ensuite, nous mettons l'accent sur deux autres travaux qui ont utilisé ces métriques de couplage statiques pour l'analyse des liens de dépendance.

Poels [1] a proposé quatre nouvelles métriques de couplage statiques qui mesurent des dépendances au niveau de la classe, à savoir *IIC* (*Inbound Inheritance Coupling*), *OIC* (*Outbound Inheritance Coupling*), *IAC* (*Inbound Abstraction Coupling*) et *OAC* (*Outbound Abstraction Coupling*). L'auteur a validé ces métriques de façon analytique via la vérification de certaines propriétés connues des métriques de couplage. Les résultats obtenus montrent que ces métriques ne sont pas contradictoires par rapport aux autres métriques existantes (comme *CBO*), mais plutôt complémentaires.

Briand *et al.* [9] ont également défini un ensemble de métriques de couplage statiques (au niveau classe) pour des programmes C++. Ils ont pris en compte trois modalités ou facettes de couplage, à savoir :

- ✓ La relation entre les classes (amitié, héritage, agrégation, …).
- ✓ Le type des interactions entre les classes d'un programme (classe-méthode, méthode-méthode, …).
- ✓ La direction de couplage (entrante/sortante) et son impact sur les classes.

Les métriques proposées par les auteurs ne sont qu'une combinaison de l'une ou l'autre de ces trois différentes modalités. Les auteurs ont validé empiriquement leurs métriques. Les résultats obtenus montrent que la majorité de ces métriques peuvent être, avec une grande probabilité, de bons indicateurs pour mesurer la propension des erreurs au niveau des programmes. Ils ont également conclu que leurs métriques statiques proposées sont différentes de celles de Chidamber et Kemerer [12][13], mais aussi complémentaires en termes de mesure de qualité.

Hitz et Montazeri [23] ont fait la distinction entre le couplage niveau classe (*CLC – Class Level Coupling*) et le couplage niveau objet (*OLC – Object Level Coupling*). Ils ont défini le *CLC* comme des liens de dépendance qui existent entre les classes lors du cycle de développement, et l'*OLC* comme la dépendance entre les objets (c'est-à-dire, les instances des classes) au moment de l'exécution du programme. Les auteurs ont présenté un cadre de travail pour la classification des mesures de couplage basé sur des facteurs qui affectent le couplage entre les objets. Cependant, leur travail manque de justification quantitative, car ils ont considéré juste les niveaux et les degrés de couplage (fort, faible, etc.) sur une échelle ordinale. Cette échelle n'est pas compatible avec les données statistiques et ne peut être éprouvée par des techniques quantitatives et par des supports mathématiques bien justifiés.

En outre, Briand *et al.* [8] ont introduit un ensemble de métriques de couplage statiques. Ils ont proposé un nouveau formalisme leur permettant de définir leurs métriques. Pour ce faire, ils ont défini des terminologies telles que l'invocation de méthodes, des attributs, des prédicats, etc. Dans ce travail, les auteurs ont fait une comparaison entre les différentes mesures existantes, afin de faciliter leur évaluation et leur étude empirique. Ils ont pu conclure qu'il y a une multitude de métriques statiques de couplage qui ont été introduites, sauf que certaines de ces métriques ne sont pas basées sur des modèles empiriques explicites, ce qui rend difficile l'utilisation de telles métriques.

Dans le même cadre, Harrison *et al.* [21] ont testé deux métriques de couplage statiques, à savoir, la métrique *CBO* (*Coupling Between Objects*) de Chidamber et Kemerer [12][13] avec la métrique *NAS* (*Number of Associations between classes*). Ils ont validé empiriquement ces deux métriques sur cinq systèmes à codes accessibles, et ont pu montrer une forte relation entre ces deux métriques. Via leurs expérimentations, les auteurs ont mis en évidence le rôle de ces métriques pour faciliter la compréhension des classes d'un programme, et de prédire le nombre des erreurs qu'il contient.

Il existe d'autres études qui ont exploité les métriques statiques de couplage pour l'analyse de dépendance. D'Ambros *et al.* [16] ont utilisé une approche par visualisation interactive pour comprendre les liens de dépendance entre les différents modules d'un

programme. Pour y parvenir, ils ont créé un nouvel outil nommé *Evolution Radar*. En effet, en se basant sur un historique de valeurs de couplage, il est possible de découvrir les liens de dépendance implicites (ou cachées) entre les artefacts d'un logiciel qui changent fréquemment d'une version à une autre. Les auteurs ont utilisé le cercle pour observer l'évolution du couplage logique entre les fichiers et les modules; plus un module s'approche du centre, plus il est fortement couplé avec le module central, et donc, fortement dépendant des autres modules du même programme. L'avantage de leur outil réside dans sa simplicité et son aptitude à fonctionner sur les grands systèmes. Les auteurs ont validé leur approche pour le cas du système *ArgoUML*. Les résultats obtenus sont satisfaisants et montrent clairement l'importance des métriques de couplage pour détecter les liens de dépendance au sein d'un programme.

2.2.3. Définition et utilisation des métriques dynamiques de couplage

Depuis les dernières années, les métriques dynamiques demeurent fréquemment utilisées dans plusieurs domaines en génie logiciel. Vanderfeesten *et al.* [66] ont défini une nouvelle métrique de couplage dynamique *CP* qui compte les paires des activités entrantes et/ou sortantes de différents types. Chaque paire d'activités reçoit un poids (un coefficient) en fonction du type de liaison (AND / OR / XOR) qui peut exister entre ces différentes activités.

Mitchell et Power [41] ont défini un ensemble de métriques de couplage dynamiques nommées *RCBO* (*Run-time Coupling Between Objects*) qui déterminent le nombre de classes accessibles par une autre classe dans un programme. Ces métriques ont été inspirées de la métrique *CBO* de Chidamber et Kemerer [12][13] sauf qu'elles sont calculées au moment de l'exécution du programme. L'objectif de cette étude est de savoir si les objets d'une même classe se comportent de la même façon ou différemment au moment de l'exécution. En utilisant des techniques de statistique et d'analyse de « clusters », les auteurs séparent les objets d'une même classe et calculent ensuite des mesures de similarité basées sur le coefficient de Pearson et de la distance euclidienne. Pour mesurer la variation de comportements des différents objets d'une même classe, les auteurs utilisent un

coefficient de variance pour savoir à quel point les valeurs de *RCBO* varient d'un objet à un autre pour une même classe. Ils ont validé empiriquement leur approche sur un banc d'essai (ou « benchmark » en anglais) de « JOlden ». Les résultats obtenus ont donné un coefficient de variance strictement positif, ce qui signifie que les objets d'une même classe se comportent différemment au moment de l'exécution (de point de vue couplage). Certes, ce travail est proche du nôtre, sauf que l'objectif n'est plus le même, car nous visons à analyser les liens entre les dépendances et le rôle d'une classe dans un programme lors de son exécution.

Dans le même cadre du travail de Mitchell et Power [41], Kaur *et al.* [28] ont utilisé presque la même technique. En effet, les auteurs ont également exploité les métriques de couplage dynamiques *RCBO* calculées au moment de l'exécution du programme. Les résultats obtenus montrent par conséquent que les objets d'une même classe jouent des rôles différents lors de l'exécution du programme.

Arisholm *et al.* [3] ont proposé un ensemble de métriques dynamiques de couplage définies pour une seule exécution. Ils ont utilisé trois critères de classifications différents, à savoir, l'entité de mesure (niveau classe ou objet), la granularité (les niveaux d'agrégation) et la portée des mesures. Les auteurs se sont basés sur un modèle d'analyse dynamique permettant de définir un ensemble de concepts de base (classes, objets, méthodes, lignes de code), des relations mathématiques entre ces ensembles (invocations de méthodes, relation d'héritage entre classes, …), ainsi que la description de règles consistantes telles que l'identification des dépendances entre deux ou plusieurs relations et le développement d'algorithmes consistants. Les auteurs ont également défini un nouvel outil, qui s'appelle *JDissect Dynamic Coupling Measures* servant à la collecte des données à partir des exécutions, ainsi que l'analyse de ces données via le calcul des mesures de couplages dynamiques. Ils ont validé empiriquement leurs mesures sur un système à code accessible nommé *Velocity*. Ils ont montré que les métriques de couplage dynamiques peuvent être un bon estimateur pour détecter la propension de changements au niveau des codes, et qu'elles peuvent détecter jusqu'à 87% de changements (pour le cas du système *Velocity*), contrairement aux métriques statiques qui n'ont pu détecter que 79% des changements, ce

qui reflète l'aptitude des métriques dynamiques de capter des dimensions supplémentaires par rapport aux métriques statiques. En plus, les auteurs ont montré que leurs métriques dynamiques de couplage ne sont pas contradictoires, mais plutôt complémentaires aux métriques de couplages statiques existantes. À partir de ce travail, nous avons exploité deux métriques de couplage dynamiques pour valider notre approche, à savoir, *IC_CM* (*Import Coupling between Classes and Methods*) et *IC_OM* (*Import Coupling between Objects and Methods*). Les définitions de ces deux métriques seront détaillées dans le chapitre 5.

Liu et Milanova [37] ont mené une étude permettant de calculer des métriques dynamiques de couplage à l'aide d'une analyse statique. Les auteurs pensent que l'analyse statique peut pallier aux inconvénients de l'analyse dynamique. En particulier, l'analyse statique peut fonctionner même sur des programmes incomplets, ce qui permet donc d'avoir une analyse séparée de chaque composant du programme. Elle est pratique et produit des résultats valables sur toutes les exécutions du programme. Pour y parvenir, les auteurs ont utilisé trois métriques dynamiques de couplage extraites des travaux d'Arisholm *et al.* [3], à savoir, *IC_OD*, *IC_OM* et *IC_OC*. Ces trois métriques s'intéressent à la mesure de couplage entre les objets, ce qui permet de détecter les interactions polymorphes entre les classes du programme. Les auteurs ont validé empiriquement leur approche. Les résultats obtenus ont montré que l'analyse statique des métriques dynamiques de couplage peut aider les développeurs à étudier des aspects importants de qualité des programmes tels que la propension d'erreurs et le changement des programmes.

Yacoub *et al.* [68] ont également proposé un ensemble de métriques dynamiques pour mesurer le couplage et la complexité pour les codes OO. Plus précisément, ils ont traité le problème de mesure de la qualité d'un logiciel au début de sa phase de développement, via des métriques dynamiques. Certaines de ces métriques ont été définies pour une seule exécution, d'autres, ont été définies pour un certain nombre d'exécutions, relatives à des scénarios spécifiques, de telle sorte qu'à chaque scénario on attribue une valeur de probabilité (ou *poids*) en fonction de sa fréquence d'apparition. Ensuite, il s'agit de calculer une moyenne pondérée qui reflète la valeur de la métrique en question. Les auteurs ont validé leur approche en effectuant une étude comparative entre les métriques statiques

usuelles et leurs métriques dynamiques (telles qu'ils les ont définies). Ils ont conclu que les métriques dynamiques (contrairement aux métriques statiques) donnent une meilleure image sur le comportement réel du programme. Néanmoins, les auteurs ont calculé une moyenne pondérée sur les valeurs de couplage dynamiques et ont supposé que cette valeur moyenne est valable pour toutes les exécutions. Ceci n'est pas réaliste, car une simple valeur ne peut montrer les différentes variations de comportements du programme au moment de son exécution. Dans notre approche, nous examinons ce problème et nous pensons qu'une simple moyenne n'est pas suffisante pour donner une idée précise sur le comportement interne d'un programme, ainsi que ses liens de dépendance.

2.3. Analyse de dépendance

Analyser les liens de dépendance entre les composants d'un logiciel ou d'un programme peut avoir plusieurs objectifs en génie logiciel. Linos et Courtois [35] ont proposé un nouvel outil qui s'appelle *OO!CARE* (*Object-Oriented Computer- Aided Re-Engineering*) pour la compréhension et la réingénierie des programmes. Cet outil permet d'extraire et de visualiser des liens de dépendance à partir des programmes C++. Dans leur outil, les auteurs ont fait la distinction entre les dépendances statiques (extraites à partir du code source des programmes) et les dépendances dynamiques (qui sont déterminées à partir des exécutions du programme), ainsi que les dépendances explicites (qui apparaissent directement dans le code source) et les dépendances implicites (qui nécessitent une analyse profonde pour pouvoir les déduire). L'objectif de ce travail est de montrer la complexité de la tâche d'analyse et de compréhension de dépendance. En effet, les auteurs ont montré via des utilisations de *OO!CARE* sur plusieurs programmes C++ que l'héritage et le polymorphisme peuvent compliquer cette tâche, en plus, le fait d'augmenter légèrement la taille des programmes, engendre des liens de dépendance compliqués, voire même illisibles. Ils ont conclu qu'il fallait concevoir un « bon » outil (ou de modifier *OO!CARE*) ce qui permet de faciliter la visualisation graphique des liens de dépendance en prenant en compte les dépendances implicites et surtout les propriétés des langages OO, à savoir l'héritage et le polymorphisme.

En fait, il arrive souvent qu'un développeur examine de façon détaillée un code source en vue de modifier une ou plusieurs de ses fonctionnalités. Dans ce cas, il cherche les dépendances structurelles au niveau du code comme un point d'entrée pour pouvoir comprendre où et comment faire ses modifications de la façon la plus efficace. Néanmoins, cette tâche n'est pas facile et requiert du temps et de l'effort. Elle dépend surtout de l'intuition, du niveau du développeur et même de la chance, vu que le nombre de dépendances est généralement grand et ne peut être couvert par le développeur. Robillard [53] a traité cette problématique en utilisant des graphes pour définir une topologie de dépendance entre les composants d'un programme. L'auteur a effectué une analyse statique de dépendance en partant de l'idée qu'une telle topologie peut contenir des indices qui peuvent servir pour sélectionner les composants qui sont plus importants que d'autres au sein du programme. Il a défini ensuite un algorithme prenant comme entrée un ensemble de composants (classes, méthodes) choisis par un développeur et qui sont importants pour une tâche spécifique dans le code. L'algorithme analyse les dépendances structurelles entre ces composants et le programme associé, et affecte ensuite des valeurs pour chacun de ces composants selon le degré de son importance pour la réalisation d'une tâche. Robillard a également défini un nouvel outil qui s'appelle *Suade* et qui réalise automatiquement le processus préalablement décrit. Cet outil fournit de plus des suggestions au développeur (par exemple suggérer d'autres composants intéressants et que le développeur n'a pas sélectionnés), et il lui assure une interaction vivante avec le programme pour faciliter la compréhension du code. L'auteur a effectué deux types de validations : une validation qualitative dans laquelle il a appliqué son outil *Suade* sur deux logiciels connus (*jEdit* et *Azureus*), et une validation quantitative dans laquelle il a appliqué son outil sur plusieurs programmes en vue de généraliser ses résultats de façon empirique. Les résultats obtenus ont montré que cette approche est utile et peut aider les développeurs pour la réalisation rapide et efficace de leurs tâches, même sur des codes complexes. De ce fait, l'auteur a pu conclure que la topologie d'analyse de dépendance peut faciliter les activités des développeurs et simplifier la compréhension des codes.

L'évolution des logiciels est également un autre domaine en génie logiciel où l'analyse de dépendance peut être utile. En effet, lors du cycle d'évolution d'un logiciel, son architecture, sa conception et la structure de son code ont tendance à changer et même à détériorer ce qui rend de plus en plus délicate l'utilisation du code. Eichberg *et al.* [17] ont tenté d'étudier ce domaine de recherche et ont proposé une solution basée sur le contrôle des dépendances structurelles lors du cycle d'évolution d'un logiciel. En effet, les auteurs ont défini un nouveau langage de spécification logique à base de requêtes extraites du « Datalog » et intitulé *LogEn* (*Logical Ensembles*) afin de définir un ensemble de contraintes sur trois niveaux d'abstraction différents : architecture, conception et implémentation. Ces contraintes sont exprimées à l'aide des requêtes déclaratives en vue de définir des ensembles de dépendance. Ensuite, des requêtes logiques sont déclenchées pour spécifier les relations de dépendance entre ces différents ensembles. Les auteurs ont également défini un autre langage visuel intitulé *VisEn* (*Visual Ensembles*) permettant la visualisation graphique des ensembles de dépendance générés. L'objectif majeur de cette approche étant de contrôler ces dépendances et de les imposer tout au long du processus de développement du logiciel ce qui garantit que toutes les prochaines versions auront la même architecture, même modèle de conception et même implémentation. Pour y parvenir, les auteurs ont fait des mises à jour sur les versions du même logiciel, ainsi que du « refactoring » pour les garder conformes aux versions initiales. Ils ont intégré leur approche dans le processus de développement incrémental d'Éclipse et l'ont validée sur trois systèmes de tailles différentes (petite, moyenne et grande). Les résultats obtenus confirment l'utilité de leur approche dans le sens où l'injection de ces dépendances facilite et rendre plus rapide les tâches de maintenance, de compréhension de code, ainsi que sa réutilisation.

Baah *et al.* [5] ont également exploité l'analyse de dépendance, mais dans le but d'étudier les scénarios d'exécutions qui sont sources de fautes dans le programme. Ils ont défini un nouveau modèle pour l'étude du comportement interne du programme à partir de la liste de ses entrées. Pour cela, les auteurs ont défini une nouvelle technique intitulée *PPDG* (*Probabilistic Program Dependency Graph*) en se basant sur une analyse statique

des liens de dépendance. Pour y parvenir, les auteurs ont spécifié cinq étapes, à savoir, la génération du graphe de dépendance en fonction des entrées du programme, le transformer, l'instrumenter, l'exécuter et enfin, la phase d'apprentissage du modèle obtenu. *PPDG* a été conçu pour des activités de débogage du code; il sert à la fois pour localiser et comprendre les erreurs pouvant survenir dans un programme. Pour cela, les auteurs ont présenté deux algorithmes distincts : *RankCP* pour localiser les erreurs dans un programme et *FaultComp* pour la compréhension de ces erreurs. Ils ont validé empiriquement leur approche en comparant *PPDG* avec d'autres techniques existantes comme *SOBER*, *Tarantula* et *CT*. Ils ont considéré sept programmes de tailles différentes pour la prise en compte du critère de *mise à l'échelle* (ou « scalability » en anglais) et ont essayé d'y appliquer ces techniques et de les tester en termes d'efficacité (temps de traitement/parties du code contenant les erreurs). Leurs expériences ont été basées sur le calcul de *scores* en utilisant des métriques afin de prédire le pourcentage du code qui doit être examiné par le développeur pour pouvoir localiser l'erreur. Les résultats obtenus confirment généralement l'importance et l'utilité de leur technique par rapport à d'autres en termes de rapidité pour les deux tâches d'ingénierie et surtout pour la localisation d'erreurs.

L'analyse de dépendance a été également exploitée dans le cadre de l'exécution symbolique des programmes OO. Il s'agit de définir un arbre formé par des chemins et des sous-chemins indiquant les différents scénarios d'exécutions pouvant survenir pour un programme donné. Comme le souligne Santelices et Harrold [54], le problème majeur de la plupart des approches qui adoptent cette technique est la difficulté de *passage à l'échelle*, c'est-à-dire, lorsque la taille du programme augmente, le nombre de chemins et des sous-chemins d'exécutions augmente de façon exponentielle. Ce phénomène est appelé problème du « path-explosion ». Santelices et Harrold [54] ont aussi étudié cet axe de recherche pour la résolution de ce problème en se basant sur l'analyse de dépendance entre les différents composants d'un programme. Ils ont défini une nouvelle technique intitulée *SPD* (*Symbolic Program Decomposition*); au lieu d'analyser un seul chemin d'exécution à la fois, *SPD* considère une famille de chemins d'exécutions voire même des sous-familles de chemins d'exécutions qui sont regroupés ensemble. Ces regroupements ne sont pas

arbitraires, mais ils sont basés sur la compréhension des liens de dépendance entre les différents éléments du programme. Les auteurs ont défini un outil qui s'appelle *JSPD* (*Java Symbolic Program Decomposition*) qui est une implémentation de leur algorithme *SPD*. Ils ont validé empiriquement leur approche sur six programmes de tailles différentes en comparant leur outil par rapport à d'autres outils existants pour faire des exécutions symboliques traditionnelles (*TRAD_SE – TRADitional Symbolic Execution*). Pour chacun de ces deux outils, ils déterminent le nombre de chemins explorés tout au long du processus d'exécutions. Les résultats obtenus montrent l'utilité de *JSPD* en termes de mise en échelle, dans le sens où leur outil fonctionne en dépit du grand nombre de chemins qui augmente exponentiellement en fonction de la taille des programmes. Ceci s'explique par le fait qu'au lieu d'analyser une dizaine ou une centaine de chemins d'exécutions, ils analysent une famille considérée comme un seul chemin à la fois. Chaque famille est un ensemble de chemins regroupés ensemble parce qu'ils partagent des liens de dépendance communs. Cependant, l'inconvénient de leur approche est la perte de précision, car certaines informations relatives aux exécutions restent cachées quand elles sont regroupées en familles. Malgré cet inconvénient, ce travail montre en effet l'importance de l'analyse des liens de dépendance dans le cadre des exécutions symboliques des programmes OO.

2.4. Utilisation des modèles probabilistes et des techniques de simulation en génie logiciel

Dans cette section, nous allons débuter par mentionner quelques travaux qui utilisent des approches probabilistes pour faire avancer la recherche dans certains domaines en génie logiciel. Ensuite, nous allons présenter certains travaux qui exploitent les techniques de simulation et les méthodes Monte-Carlo pour résoudre des problèmes en génie logiciel.

2.4.1. Utilisation des modèles probabilistes en génie logiciel

Plusieurs travaux en génie logiciel ont utilisé les modèles probabilistes pour divers objectifs. En général, ces modèles implémentent un processus incertain ou aléatoire. Les travaux sont assez nombreux, mais nous allons juste mentionner quelques uns.

Lock et Kotonya [38] ont proposé une approche probabiliste pour l'analyse de l'impact de changements relatif aux besoins des utilisateurs. Pour y parvenir, ils ont utilisé une approche basée sur la traçabilité afin de générer les différents besoins des utilisateurs, et prédire par la suite les probabilités de changements entre les modules d'un programme au cas où on applique les différents besoins des utilisateurs.

Abdi *et al.* [1] ont présenté une approche probabiliste basée sur les réseaux bayésiens afin de comprendre les facteurs réels responsables de l'impact de changement et de son évolution entre les versions d'un logiciel.

Mirarab [40] a également construit les réseaux bayésiens pour l'analyse de l'impact de changements. Il a utilisé des métriques de dépendance statiques qui expriment le degré de couplage entre les différents modules d'un programme (en termes d'appels de méthodes et d'utilisation de variables), ainsi que des données historiques qui correspondent à des changements effectués dans le passé sur le programme. Dans le même objectif de l'étude de l'impact de changement à l'aide des réseaux bayésiens, on trouve d'autres travaux, comme celui de Tang *et al.* [62] et Zhou *et al.* [71].

Malak *et al.* [39] ont étudié la qualité des applications web en utilisant une approche probabiliste par le moyen des réseaux bayésiens. Vu l'apparition et l'évolution continue des nouvelles technologies, les applications web sont également en perpétuelle évolution ce qui rend difficile de pouvoir caractériser ou mesurer la qualité d'une application web. En effet, pour assurer la qualité, il faut garantir celle de tous les facteurs (critères) et les sous-facteurs (sous-critères) qui lui touchent. Pour cela, les auteurs ont représenté les différentes relations qui existent entre ces facteurs intuitivement dans le graphe du réseau bayésien. Ils ont défini un modèle probabiliste pour modéliser la navigabilité dans les sites web. Leur modèle est articulé autour de quatre phases qui consistent à collecter les critères de qualité, les raffiner, définir la structure du modèle probabiliste et dériver les paramètres du modèle. Cette dernière étape est basée sur le calcul des valeurs de probabilité. Les auteurs ont fait une étude expérimentale contrôlée de leur modèle probabiliste. Cette expérimentation a été faite sur 20 sujets et 40 pages web (aléatoirement choisies). Ils ont également implanté un environnement qui supporte leur modèle. Les résultats obtenus basés sur les observations

des sujets montrent l'utilité de leur modèle probabiliste et qu'il est un bon estimateur pour analyser la qualité des applications web. En particulier, les scores donnés par le modèle probabiliste sont fortement liés à la navigabilité des sites web. Ces résultats montrent l'utilité d'un modèle probabiliste pour étudier la navigabilité des applications web.

Dans le contexte de la modélisation des interactions dans les applications web, le travail de Zhou *et al.* [70] est plus proche du nôtre. Les auteurs ont défini un modèle de chaîne de Markov qu'ils ont appelé *MNav* pour mesurer la navigabilité dans les sites web. En effet, ils modélisent un site web par un graphe où les nœuds représentent les pages du site et les arcs sont les liens internes qui permettent de passer d'une page à une autre dans le même site. Les auteurs définissent une suite d'actions élémentaires que l'internaute pourra effectuer lorsqu'il visite un site web, comme *quitter*, *aller à*, *retour en arrière*, … et des probabilités qui décrivent les possibilités de transition d'une page à une autre. En fait, *MNav* décrit le comportement de l'internaute face à une application web en utilisant un modèle de chaîne de Markov. Modéliser les entrées de la chaîne de Markov semble naturel ici. Les entrées des sites web sont différentes de celles d'un programme classique. En effet, dans ce travail, juste les clics de souris sur les liens hypertextes sont pris en compte et permettent de caractériser entièrement les états de la chaîne de Markov. Pour valider *MNav*, les auteurs ont effectué deux expérimentations. Dans la première, ils ont utilisé plusieurs sites web avec des nombres de pages illimités. Les scores de navigabilité de ces sites web sont connus d'avance. Ils ont comparé empiriquement *MNav* avec d'autres mesures. Ils ont déduit que leur modèle de chaîne de Markov est conforme avec les mesures existantes et peut différencier de façon efficace les sites dont le score de navigabilité est élevé, c'est-à-dire, les sites les plus fréquemment visités (contrairement aux autres sites web les moins visités et donc ayant un score de navigabilité bas). La seconde expérimentation consiste à appliquer l'analyse en composantes principales pour déterminer si *MNav* peut capter des dimensions supplémentaires à partir du site qui n'ont pas été capturées par les mesures de navigabilité existantes. Les résultats obtenus sont prometteurs dans le sens où leur modèle de chaîne de Markov a pu capter de nouveaux aspects sur la navigabilité et qui n'ont pas été détectés par les autres mesures. Ces résultats ont été également confirmés par les modèles

de régression linéaire *MLR* (*Multiple Linear Regression* en anglais) d'analyse. Ainsi, on en déduit que les modèles de chaîne de Markov sont efficaces pour l'étude de la navigabilité des sites web et donnent par conséquent de meilleures mesures quantitatives.

Les modèles probabilistes ont également été utilisés dans d'autres domaines de recherches en génie logiciel, par exemple pour la détection d'anti-patrons de conception [31], pour définir des modèles de qualité [11], pour la détection des défauts de conception [30], etc.

2.4.2. Utilisation des techniques de simulation en génie logiciel

Dans la partie Annexe I, nous fournissons des généralités sur des notions de base en simulation. Le lecteur est invité à consulter l'annexe chaque fois pour se rappeler de ces notions de base. Dans ce qui suit, nous citons quelques approches existantes qui ont utilisé les techniques de simulation en génie logiciel.

En effet, la majorité de ces travaux peut être regroupée en deux grandes catégories : la première catégorie s'intéresse aux rôles de la simulation dans le processus de développement logiciel et la seconde met l'accent sur le rôle de la simulation dans l'évolution des logiciels.

Parmi les travaux qui ont exploité la simulation pour étudier le processus de développement des logiciels, nous citons les travaux de Setamanit *et al.* [55] qui ont défini un nouveau modèle de simulation hybride pour les projets de développement logiciel. Ce modèle est basé sur des méthodes de simulation à évènements discrets. Ces dernières servent entre autres à capturer tous les détails et les propriétés dynamiques lors de la phase de développement du logiciel. La simulation intervient pour faire de la gestion de projet et gérer des évènements discrets, c'est-à-dire, les activités de développement, ce qui permet de comprendre explicitement la structure et les mécanismes qui interviennent pour coordonner ces activités. Les auteurs ont tenté de combiner le processus global de développement logiciel (ou *GSD – Global Software Development* en anglais) avec un modèle de simulation (*SPSM – Software Process Simulation Modeling* en anglais). Ce modèle pourra servir plus tard pour capturer dynamiquement tout ce qui se passe lors du

processus de développement d'un logiciel, et de donner tous les détails de chaque niveau de ce processus. Pour y parvenir, les auteurs ont mentionné des facteurs qu'ils répartissent en facteurs fondamentaux, facteurs stratégiques et facteurs organisationnels pouvant influencer sur la performance des projets. Lors de la description de l'approche, ils ont recours à une liste de questions-réponses afin de montrer comment on peut planifier le déroulement de n'importe quel projet et l'améliorer en termes de qualité et coût de maintenance. Les auteurs ont validé leurs travaux et ont pu montrer que la simulation peut améliorer le processus de développement des logiciels.

Dans le cadre de ces mêmes travaux, Raffo et Setamanit [50] ont discuté du rôle d'un modèle de simulation (*SPSM*) pour supporter des études empiriques et expérimentales pour le processus global de développement logiciel (*GSD*). Les auteurs ont insisté encore sur l'importance d'un modèle de simulation pour évaluer et enrichir le processus de développement. Ils ont fait un survol des différents types de modèles de simulation existants tels que les modèles dynamiques, continus, hybrides, à évènements discrets, ainsi que les différents composants à inclure pour enrichir de tel modèle. Les auteurs ont conclu que le meilleur des cas serait de combiner un modèle de simulation hybride avec un modèle de simulation à événements continus et discrets, ce qui permet de construire une bonne plateforme pour le processus de développement logiciel.

Kellner *et al.* [29] ont également étudié cet axe de recherche à savoir le rôle de la simulation dans le processus de développement logiciel. Les auteurs commencent par mentionner certains problèmes qui expliquent la nécessité de redéfinir le processus de développement. En effet, la plupart des organisations de développement logiciel délivrent des produits de mauvaise qualité. D'autres côtés, on a besoin de créer des logiciels qui satisfont les besoins des clients, c'est-à-dire, qui soient de bonne qualité, rapides et pas chers. Plusieurs outils, techniques et langages existent pour atteindre ces défis, mais ceux-ci ne sont pas suffisants pour gérer ces problèmes; il fallait bien évidemment changer le processus de développement logiciel ou le redéfinir. Pour pallier les problèmes précédemment mentionnés, les auteurs proposent de définir un modèle de simulation pour le processus de développement logiciel. Ce modèle s'intéresse en particulier aux phases de

développement et de maintenance. Plus précisément, pour des systèmes complexes, il n'est pas possible d'y appliquer les techniques et les modèles statiques existants pour représenter la totalité du système. Il fallait en revanche avoir recours à un modèle pour *simuler* ce système. Pour y parvenir, les auteurs commencent par identifier les besoins de simuler un système, à titre d'exemple :

- ✓ Planifier le processus de développement de point de vue temps, efforts et risques.
- ✓ Faire le contrôle et le suivi du projet en utilisant les techniques de simulation Monte-Carlo qui permettent de générer des distributions de probabilité et servent à faciliter la tâche de conception, les tests d'intégration, etc.
- ✓ Faciliter la compréhension du processus de développement par les développeurs et les chefs de projets et prévoir les futurs résultats du projet (surtout à l'aide des techniques Monte-Carlo).
- ✓ Essayer plusieurs idées sur le projet sans payer les vrais coûts de nos erreurs, mieux que d'assumer les dégâts coûteux si on travaille sur le vrai système.

Les auteurs ont ensuite donné les détails de leur modèle de simulation. En effet, partant des problèmes précédemment mentionnés, et que le modèle est censé résoudre, il faut définir le champ d'application du modèle, les informations qu'il doit fournir pour résoudre ces problèmes, les différents éléments qu'il doit contenir, ainsi que leurs liens de dépendance, leurs comportements et les entrées nécessaires du modèle qui lui permettent éventuellement de répondre à ses objectifs. Éventuellement, les auteurs ont spécifié les techniques et les moyens utilisés pour la définition de leur modèle de simulation pour le processus de développement logiciel. Parmi ces techniques, on trouve les langages de simulation, les techniques de simulation et principalement celles de Monte-Carlo, ainsi que les techniques de mesure pour valider empiriquement le modèle sur des données réelles (valeurs de métriques par exemple). Le modèle de simulation proposé est complémentaire aux travaux existants pour améliorer le processus de développement logiciel, et donc, produire les logiciels de bonne qualité qui satisfont aux exigences des clients.

L'évaluation de ces modèles de simulation est décrite en détail dans [49] où les auteurs présentent une validation empirique sur les entrées/sorties de ces modèles. Ils ont utilisé

plusieurs données statistiques et des techniques analytiques pour les évaluer. Les auteurs se sont focalisés sur les modèles de simulation stochastiques, et plus précisément ceux qui utilisent les techniques connues de Monte-Carlo. Les résultats obtenus prouvent l'importance de la simulation stochastique en général et des méthodes Monte-Carlo en particulier pour faciliter la prise de décisions lors du processus de développement, ce qui permet par conséquent de mieux comprendre les impacts de ces décisions sur le projet.

La seconde catégorie de travaux a exploité les techniques de simulation pour étudier l'évolution des logiciels. En effet, lors du cycle d'évolution d'un logiciel, sa qualité commence à dégrader en passant d'une version à une autre. Plusieurs travaux ont étudié les causes de cette dégradation et de proposer des solutions pour garantir la même qualité tout au long du processus d'évolution. Parmi ces approches, on cite les travaux de Stopford et Counsell [60]. Ces travaux naissent à cause d'un manque de cadre d'application théorique pour décrire clairement le processus d'évolution logiciel. Bien évidemment, il existe déjà des outils et des techniques dont l'objectif est de garder une meilleure qualité pour les différentes versions d'un même logiciel. Principalement, les auteurs ont cité les études empiriques et le « refactoring ». Sauf que ces derniers sont généralement complexes, coûteux en temps et nécessitent cependant plusieurs données statistiques. Pour cela, les auteurs proposent une autre technique qui consiste à définir *un modèle de simulation* afin de suivre les différentes versions d'un logiciel durant son cycle d'évolution. Ce modèle est construit de façon incrémentale par ajout progressif de plusieurs modules et est basé sur la définition d'une politique d'évolution qui doit être respectée pour toute nouvelle version, ce qui garantit entre autres que toutes les versions obtenues restent conformes aux spécifications initiales, et par conséquent, garder la même architecture logicielle. La définition de ce modèle passe par quatre phases importantes :

- ✓ *Analyse des besoins* qui consiste à définir une liste exhaustive de tâches de manière stochastique. Ces tâches constituent l'ensemble des exigences à prendre en compte par la politique d'évolution.

- ✓ *Définition de la politique d'évolution* qui, en se basant sur la liste des besoins déjà spécifiés, permet de définir des règles à prendre en compte pour montrer comment structurer les nouvelles parties du code.
- ✓ *Mesure des impacts de changements* qui déterminent le coût nécessaire pour appliquer la politique d'évolution sur les prochaines versions. Cette phase est basée sur le calcul de métriques pour juger l'acceptation ou le refus de la politique obtenue.
- ✓ *Codification* qui consiste à implanter les propriétés précédemment définies sous forme de classes, méthodes, objets, et forcer l'injection de la politique d'évolution.

L'importance de ce modèle de simulation réside dans le fait que le choix des *critères* à prendre en compte dans la politique d'évolution est dirigé par l'utilisateur lui-même ce qui lui donne toute la liberté de choisir la future architecture de son logiciel. Les auteurs ont validé empiriquement leur approche sur trois systèmes logiciels complexes. Les résultats obtenus justifient clairement le rôle important que peut jouer la simulation en termes d'efficacité, gain de temps et de coût (par rapport aux autres techniques existantes). En outre, les résultats obtenus ont prouvé qu'un simple modèle de simulation peut agir de façon efficace sur le comportement général du logiciel durant son cycle d'évolution.

Un autre travail important et relié à cet axe de recherche est celui de Ramil et Smith [51]. Les auteurs utilisent un raisonnement qualitatif basé sur les techniques de simulation qualitative pour modéliser l'évolution des logiciels à long terme. En effet, le raisonnement qualitatif consiste à définir et construire des modèles dans le cas où il y a un manque ou une incomplétude d'information. La simulation qualitative est l'une des techniques de raisonnement qualitatif que les auteurs ont implémentées sur leur outil *QSIM*. Contrairement aux autres outils existants qui modélisent les systèmes par des équations différentielles ordinaires, *QSIM* utilise plutôt des équations différentielles qualitatives pour représenter les différentes fonctions du système. Le résultat du modèle de simulation correspond généralement au comportement réel et attendu du système. En utilisant d'autres modèles de simulation qualitatifs existants, les auteurs proposent leur propre modèle (*modèle de Smith & Ramil*). Dans la partie validation, les auteurs ont comparé le résultat de leur modèle de simulation avec des données empiriques collectées auprès de cinq systèmes

industriels. L'objectif de ces expérimentations est de voir à quel point leur modèle de simulation prédit correctement le vrai comportement du système, ainsi que celui des futures versions. Les résultats obtenus montrent que la simulation qualitative est complémentaire aux résultats quantitatifs obtenus à partir des données empiriques, et que les prédictions du modèle de simulation qualitatif sont presque conformes avec les attentes du processus d'évolution logiciel.

Les mêmes auteurs (en plus de « Capiluppi ») ont étendu ce travail dans le contexte de la simulation qualitative et l'ont validé sur d'autres systèmes [57]. Ils ont effectué une analyse qualitative du processus d'évolution en termes de croissance. Cette analyse a été réalisée sur 25 systèmes à codes accessibles. Les résultats obtenus montrent que les modèles de simulation existants montrent que la taille et la complexité sont deux facteurs reliés tout au long du cycle d'évolution du système. Par conséquent, les auteurs ont déduit que la simulation qualitative aide à comprendre le cycle d'évolution et constitue un moyen efficace pour pouvoir gérer et contrôler le processus d'évolution logiciel.

2.5. Synthèse

Dans les sections précédentes de ce chapitre, nous avons fait un survol des principaux travaux liés à notre domaine de recherche. La figure 1 donne un aperçu général de ces travaux.

En effet, Robillard [53] a défini une topologie d'analyse de dépendance. À partir de ses études empiriques, il a pu montrer qu'il est possible de comprendre le comportement d'un programme en se basant sur l'analyse de ses liens de dépendance. Nous allons donc adopter cette idée, dans le sens où nous cherchons à trouver un lien entre les dépendances et le rôle d'une classe dans un programme. D'autre côté, Zhou *et al.* [70] ont défini un modèle de chaîne de Markov permettant de modéliser la navigabilité d'un internaute sur une application web. Un site web peut être vu comme un programme ayant un nombre illimité d'entrées, telle que chaque entrée dépend de celles qui la précèdent. Pour cela, nous avons pensé à utiliser un modèle probabiliste basé sur les chaînes de Markov pour le cas des programmes ayant un nombre d'entrées illimité (ou non connu d'avance). En outre, dans

[68], Yacoub *et al.* ont calculé des métriques dynamiques de couplage et ont affecté des poids (valeurs de probabilité) en fonction des scénarios d'exécutions du programme. Sauf que les auteurs ont calculé une moyenne pondérée pour les valeurs de métriques, et ont supposé par la suite que cette moyenne est valable pour toutes les exécutions du programme. Cependant, nous pensons qu'une simple moyenne n'est pas suffisante pour donner une idée précise et globale sur tous les scénarios spécifiques qui se peuvent survenir au moment de l'exécution du programme, ce qui permet donc de cacher l'information. En revanche, nous pensons qu'il sera peut-être plus judicieux de considérer toutes les valeurs de métriques de couplage et les analyser sous forme d'une distribution, afin de prendre en compte tous les scénarios auxquels intervient notre programme. Éventuellement, nous avons utilisé deux métriques de couplage dynamiques proposées et testées dans le travail d'Arisholm *et al.* [3]. Les auteurs ont évalué empiriquement ces métriques et ont montré qu'elles prennent en compte les valeurs décidées dynamiquement au moment de l'exécution du programme, ce qui coïncide avec les exigences de notre approche.

2.6. Conclusion

Dans ce chapitre, nous avons fait un survol de certains travaux qui touchent aux trois domaines de recherche auxquels nous nous sommes intéressés dans ce mémoire. Nous avons commencé par citer des études qui ont exploité les métriques de couplage statiques et dynamiques pour résoudre des problèmes en génie logiciel. Nous avons ensuite cité quelques travaux qui ont analysé les liens de dépendance et ont contribué pour les comprendre. Enfin, nous avons présenté quelques études qui ont exploité les techniques de simulation (et les méthodes Monte-Carlo), ainsi que les modèles probabilistes pour résoudre des problèmes en génie logiciel. Éventuellement, nous avons synthétisé ces études et nous avons montré leurs rapports avec notre approche.

Nous présentons dans les prochains chapitres de ce mémoire les détails de notre approche.

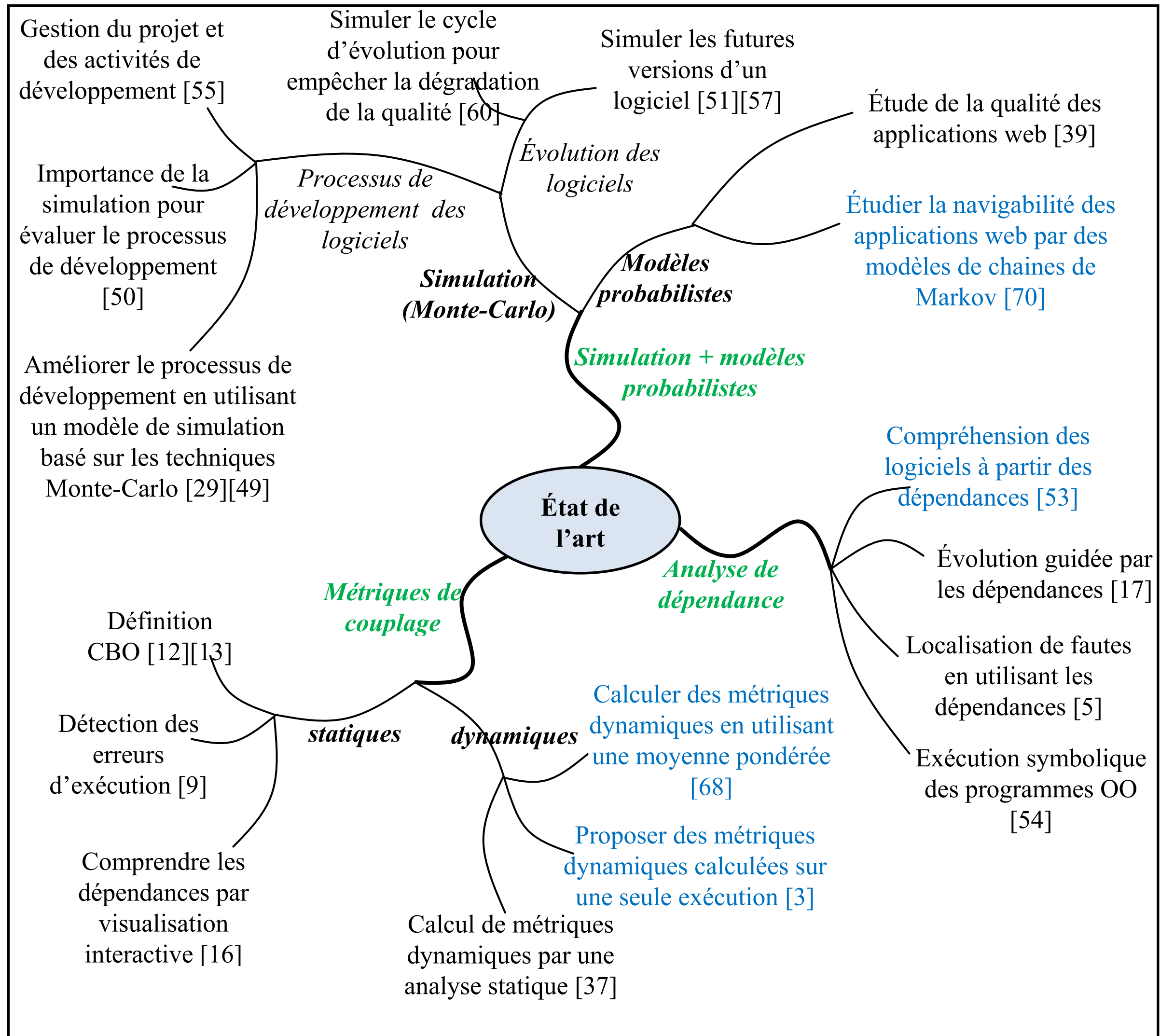


Figure 1. Synthèse sur les principaux travaux reliés à notre domaine de recherche.

Chapitre 3

CARACTÉRISATION DES DÉPENDANCES

3.1. Introduction

Nous proposons une approche semi-automatique qui permet de faciliter la compréhension des liens de dépendance pour les classes d'un programme en utilisant des modèles probabilistes d'entrées. Dans ce chapitre, nous débutons par présenter une vue d'ensemble de notre approche, ensuite, nous définissons nos modèles probabilistes pour le cas d'un vecteur aléatoire et le cas d'une chaîne de Markov. Ces deux cas englobent tout type de programme dont le nombre d'entrées est connu d'avance ou pas. Avant de conclure, nous allons discuter de l'utilité des histogrammes pour la représentation des données.

3.2. Schéma global de l'approche

Dans notre approche, nous nous intéressons à réaliser un processus qui simule les entrées d'un programme selon des lois de probabilité spécifiées d'avance. Ce processus nous permet de calculer des métriques de couplage dynamiques obtenues sur plusieurs exécutions du programme. Chaque exécution correspond à un ensemble de valeurs d'entrées. Le programme, ainsi que la procédure d'extraction de métriques sont définis à priori. Par conséquent, nous nous intéressons dans le présent mémoire à la méthode de génération de l'ensemble de valeurs d'entrées. Ce processus constitue la base pour comprendre les liens de dépendance et le comportement global du programme.

Quand on utilise les algorithmes déterministes, les programmes seront dirigés par un ensemble de données externes (entrées) qui déterminent toute la séquence d'exécutions. En d'autres termes, un algorithme déterministe exécute toujours la même suite d'opérations, c'est-à-dire, en adoptant un processus prédéfini pour résoudre un problème, contrairement à

un algorithme probabiliste (ou non-déterministe) qui permet de choisir entre deux ou plusieurs instructions au moment de l'exécution du programme. De ce fait, deux exécutions différentes d'un algorithme non-déterministe peuvent réaliser des choix différents.

Dans notre approche, nous considérons un programme OO qui peut être déterministe ou non-déterministe, constitué par plusieurs classes (en Java par exemple) et une métrique de couplage dynamique dont la valeur dépend directement des valeurs que prennent les entrées du programme comme le montre la figure suivante.

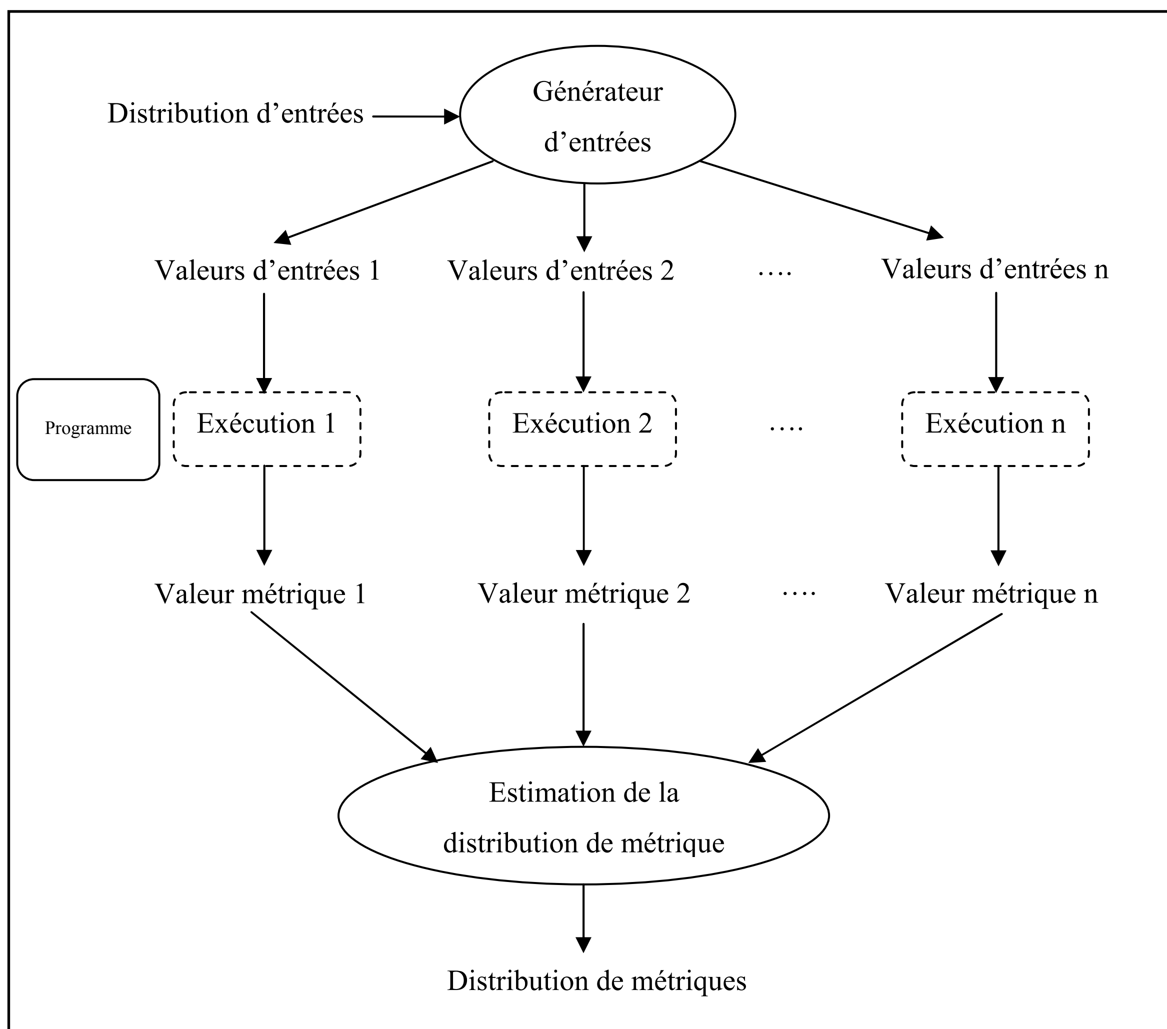


Figure 2. Schéma global de notre approche.

Néanmoins, la taille de la population qui définit toutes les possibilités des exécutions est généralement grande, parfois même infinie pour des programmes de grande taille. Pour cela, nous proposons de considérer *un échantillon d'exécutions* (de taille n) qui soit représentatif de toute la population. Certes, le choix de ces n exécutions est aléatoire, mais il est aussi guidé par un modèle probabiliste préalablement défini et basé sur l'utilisation des lois de probabilité pour caractériser les entrées du programme. Une fois que ces lois de probabilité ont été bien déterminées, il est possible de simuler les entrées du programme, ce qui permet ensuite d'obtenir des valeurs de métriques de couplage. Ces dernières constituent la base de l'estimation de la distribution de métriques pour chaque classe du programme.

3.3. Les modèles probabilistes des entrées

En regardant la nature des programmes OO existants, on peut distinguer deux cas, en fonction de comment les entrées sont spécifiées pour le programme :

(a) Le nombre des entrées est fixé d'avance à d où d est un entier strictement positif. Dans ce cas, les entrées du programme peuvent être représentées par *un vecteur aléatoire* $\mathbf{X} = (X_1,\ldots,X_d)$. Ce cas se trouve généralement dans les programmes à entrées par ligne de commande dont le nombre de paramètres est spécifié d'avance lors de chaque nouvelle exécution.

(b) Le nombre des entrées est variable (aléatoire). Dans ce cas, les entrées successives peuvent être considérées comme *une chaîne de Markov* dont les états sont fonction des entrées. C'est le cas des programmes à interaction avec l'utilisateur ou à environnements extérieurs de telle sorte que la distribution de probabilité de la prochaine entrée (au cas où elle existe) peut dépendre des valeurs de toutes les entrées qui la précèdent et qui ont été données au programme dans l'exécution courante.

Nous allons détailler ces deux cas dans les deux sous-sections qui suivent.

3.3.1. Entrées définies par un vecteur aléatoire

Dans le cas du vecteur aléatoire $\mathbf{X} = (X_1,\ldots,X_d) \in \mathbf{R}^d$ [1], les valeurs des entrées peuvent être données au programme avant même de lancer son exécution. Dans notre modèle, nous allons supposer que $\mathbf{X}$ est un vecteur aléatoire avec une distribution multi-variée arbitraire dans $\mathbf{R}^d$. Cette distribution peut être discrète, continue, ou mixte, dans le sens où des coordonnées de $\mathbf{X}$ peuvent avoir une distribution continue (exponentielle ou normale par exemple) alors que d'autres prennent juste des valeurs entières (par exemple des nombres binaires 0 ou 1). Les entrées, c'est-à-dire les coordonnées du vecteur $\mathbf{X}$ ne sont pas nécessairement indépendantes les unes des autres, mais elles peuvent l'être.

Le vecteur aléatoire $\mathbf{X}$ a une distribution F si et seulement si pour tout $\mathbf{x}=(x_1,\ldots,x_d)\in \mathbf{R}^d$, on a : $F(\mathbf{x}) = \mathbf{P}[\mathbf{X}\leq\mathbf{x}] = \mathbf{P}[X_1 \leq x_1,\ldots, X_d \leq x_d]$ où $\mathbf{P}$ désigne la *probabilité*. La j-ème fonction de distribution marginale (ou simplement la distribution de X_j) est définie par $F_j(x_j) = P[X_j \leq x_j]$. Les variables aléatoires $X_1,\ldots, X_d$ sont considérés *indépendantes* si et seulement si $F(X_1,\ldots, X_d) = F_1(x_1)\ldots F_d(x_d)$ pour tout $\mathbf{x} \in \mathbf{R}^d$. Dans le cas contraire, c'est-à-dire, $X_1,\ldots, X_d$ ne sont pas indépendantes, alors, la manière générale de spécifier leur distribution multi-variée est d'utiliser les *copules* [4]. En effet, il s'agit de spécifier une distribution de dimension d dont les lois marginales sont uniformes dans l'intervalle [0, 1], mais généralement non indépendantes. C'est ce qu'on appelle *copule* ou *fonction de dépendance*. Si $\mathbf{U} = (U_1,\ldots,U_d)$ est une variable aléatoire ayant cette distribution, alors, pour chaque j on définit : $X_j = F_j^{-1}(U_j) = \inf \{x : F_j(x)\geq U_j\}$. De ce fait, notre vecteur $\mathbf{X} = (X_1,\ldots,X_d)$ a une distribution multi-variée de lois marginales de F_j et sa structure de dépendance est complètement déterminée par le choix de la copule. Il est connu que n'importe quelle distribution multi-variée peut être spécifiée de cette façon. Les techniques spécifiques pour sélectionner des copules et générer des vecteurs aléatoires à partir de ces copules sont expliquées en détail dans [4] et [44]. Pour plus de détails, le lecteur est invité à conférer la section Annexe I.

---

[1] $\mathbf{R}^d$ regroupe l'ensemble des d-uplets tel que chaque élément est un nombre réel.

Une *métrique de couplage dynamique* Φ_1 peut être vue comme étant une fonction qui affecte un nombre réel pour chaque possibilité d'exécution du programme. Et puisque le résultat de ces exécutions dépend uniquement du vecteur aléatoire **X** (préalablement défini), alors, notre métrique peut s'exprimer en fonction de **X** telle que Φ_1: $\mathbf{R}^d \rightarrow \mathbf{R}$ et Y= $\Phi_1(\mathbf{X})$ est une variable aléatoire dont la distribution dépend directement de celle de **X** (peut être d'une façon compliquée).

Souvent, la distribution de Y ne sera pas connue explicitement. Dans ce cas, nous proposons d'utiliser les techniques de simulation Monte-Carlo pour estimer cette distribution. Pour y parvenir, on génère *n* réalisations *indépendantes* de **X**, à savoir $\mathbf{X_1}$, …, $\mathbf{X}_n$, et on calcule ensuite les *n* réalisations correspondantes de Y, à savoir $Y_1,\dots,Y_n$. Enfin, la distribution empirique de $Y_1,\dots,Y_n$ est utilisée pour estimer la vraie distribution de Y. Grâce aux techniques de Monte-Carlo, il est également possible d'estimer d'autres paramètres statistiques pour caractériser la distribution de Y, tels que la moyenne, la variance, le coefficient d'asymétrie, etc. et de calculer *un intervalle de confiance* sur ces nombres [56] [36]. Par exemple, on peut estimer la moyenne de Y : $\mu = \mathbf{E}(Y)$ juste en calculant une moyenne sur les *n* réalisations de Y, c'est-à-dire, $\overline{Y}_n = (1/n) \sum_{i=1}^{n} Y_i$. Pour examiner le degré de précision de cet estimateur, on peut penser à calculer un intervalle de confiance pour μ; c'est un intervalle à bornes aléatoires $[I_1, I_2]$ de telle sorte que $\mathbf{P}[I_1 \leq \mu \leq I_2] \approx (1-\alpha)$ où $(1-\alpha)$ nous renseigne sur le niveau de confiance de cet intervalle. Par exemple, si on suppose que $\overline{Y}_n$ a une distribution normale (ce qui n'est pas vrai en pratique, mais peut être une bonne approximation quand *n* est grand, grâce au théorème de la limite centrale), ainsi, l'intervalle de confiance sera de la forme :

$$[I_1, I_2] = [\, \overline{Y}_n - {}_{z\,(1-\alpha/2)}\, S_n / \sqrt{n},\ \ \overline{Y}_n + {}_{z\,(1-\alpha/2)}\, S_n / \sqrt{n}\,]$$

où S_n est l'écart-type de $Y_1,\dots,Y_n$ et $z(1-\alpha/2)$ satisfait la condition $\mathbf{P}[Z \leq z.(1-\alpha/2)] = (1-\alpha/2)$ telle que Z est une variable aléatoire qui suit la loi normale standard $\mathcal{N}(0, 1)$. Pour plus de détails, on invite le lecteur à conférer la partie Annexe I.

D'autres techniques, tels que les méthodes de *bootstrap* (ou *bootstrapping*) [42] peuvent être utilisées ici lorsque la distribution de $\overline{Y}_n$ n'est pas très proche d'une loi normale [42].

De plus, l'intervalle de confiance peut être estimé sur d'autres mesures (autre qu'une simple moyenne) telle que la variance de Y, et peut être donc calculé d'une façon similaire.

Bien évidemment, une distribution empirique donne toujours plus d'information qu'une simple valeur (moyenne, variance,…). Pour cette raison, nous pensons qu'il est généralement meilleur d'utiliser toute la distribution de probabilité et la présenter sous forme d'histogramme par exemple, que d'utiliser une simple moyenne (ou en plus de la moyenne) de la forme $\overline{Y_n} = (Y_1 + \ldots + Y_n)/n$. La figure 3 donne deux exemples de ces histogrammes produits avec un échantillon de *n* observations. Ces deux histogrammes sont issus de deux distributions différentes, mais ayant la même moyenne $\mu = 100$. Malgré que ces deux histogrammes possèdent presque la même moyenne, la variable aléatoire Y se comporte différemment entre ces deux exemples. Autrement dit, les observations issues de la variable aléatoire Y dans les deux exemples, engendrent la même moyenne empirique, sauf que l'allure de ces observations ainsi que leurs variances ne sont plus les mêmes. La distribution à gauche correspond à une allure exponentielle, alors que celle de droite correspond à une distribution mixte formée par deux distributions ayant des allures normales. À partir de cet exemple, on peut penser qu'une simple moyenne n'est pas vraiment suffisante pour refléter le vrai comportement d'une variable aléatoire, ni de donner une idée précise de son allure. Par contre, considérer toute la distribution de Y nous permet d'avoir une idée globale et précise du comportement de Y. À partir de cette idée, nous avons décidé de considérer dans notre approche toute la distribution de probabilité au lieu de prendre en compte seulement la moyenne de *n* observations.

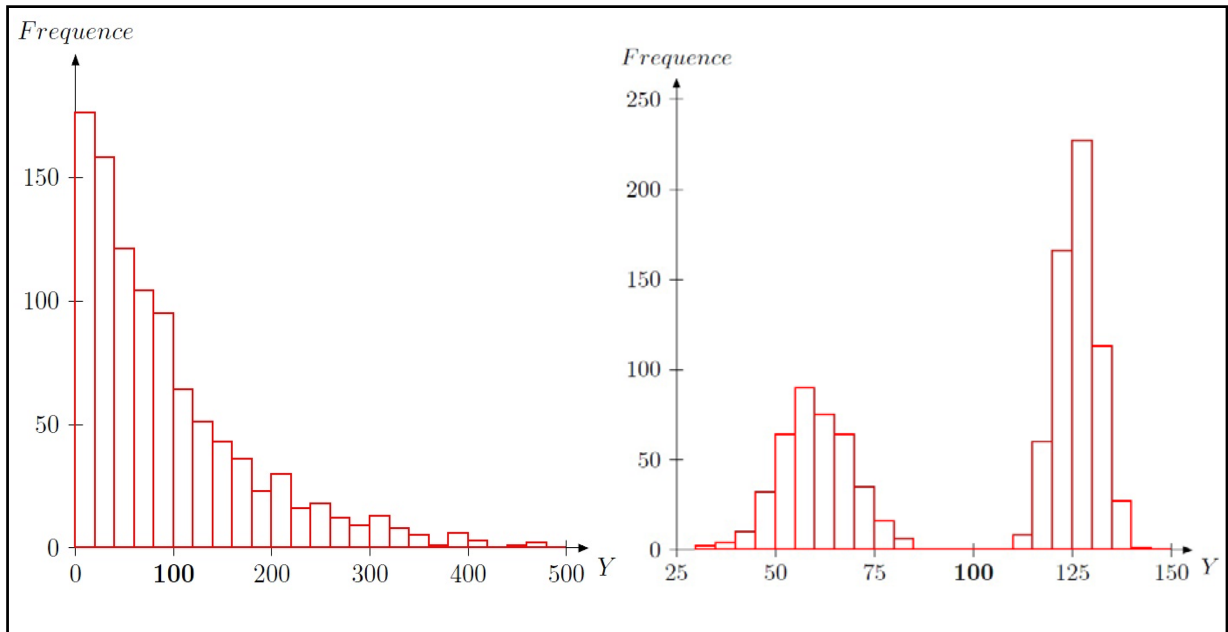


Figure 3. Exemple de deux distributions ayant la même moyenne 100, mais avec des allures différentes.

Le cas des vecteurs aléatoires se trouve généralement dans les programmes par lots (c'est-à-dire, des programmes de types *batch*) ou les programmes à entrées par ligne de commande où un ensemble de paramètres doit être spécifié d'avance avant de lancer l'exécution du programme. Par exemple, lorsqu'on lance une commande de type *lpr* sous Linux, on doit spécifier le « printer-name », le « username », le nombre de copies, etc. Dans le cas où l'un de ces paramètres manquait, par exemple, le « username », ceci n'implique pas qu'une entrée au programme est manquante, ceci signifie tout simplement que la valeur par défaut va être prise en compte (dans ce cas le « username » par défaut). Ainsi, le nombre d'entrées reste inchangé. Par conséquent, la taille de notre vecteur aléatoire est toujours la même.

Considérons un programme *P1* qu'on exécute n fois. Le nombre d'entrées de *P1* est fixé d'avance (par exemple d). À chaque possibilité d'exécution de *P1*, nous allons construire le vecteur aléatoire $\mathbf{X}_k$ correspondant à la k ème exécution de *P1* et qui contient d valeurs

d'entrées. On applique à chaque répétition k la fonction Φ_1 afin de calculer la valeur de métrique sur le vecteur $\mathbf{X}_k$ construit. Ce processus engendre alors n valeurs par la fonction Φ_1. Sachant donc la liste des entrées du programme à chaque possibilité d'exécution (qui est complètement décrite par les coordonnées du vecteur $\mathbf{X}$), ainsi que les valeurs de métriques correspondantes (Φ_1 ($\mathbf{X}$)), il est possible de construire un histogramme qui décrit la distribution de probabilité de la métrique de couplage pour une classe du programme. La fonction Φ_1 décrit le même processus qui correspond exactement à la définition de la métrique de couplage. Cette fonction génère en sortie les valeurs de couplage permettant de construire une distribution. Dans cette distribution, on a toutes les informations nécessaires pour caractériser le comportement de *P1* (ou même une classe de *P1*), à savoir, les valeurs des entrées du programme (c'est-à-dire, les coordonnées du vecteur $\mathbf{X}$), ainsi que les valeurs de métriques obtenues (valeurs-résultats de Φ_1).

3.3.2. Entrées définies par une chaîne de Markov

La majorité des programmes actuels sont interactifs et n'ont pas de nombre d'entrées fixé d'avance, mais ces entrées, aussi bien que leurs types, sont toutes des variables aléatoires. Considérons par exemple le cas d'un programme constitué par m fonctions, chacune d'elles est effectuée en plusieurs étapes. Dans chaque étape, on a besoin d'un certain nombre de paramètres à spécifier. Le processus général de l'exécution du programme (c'est-à-dire, la liste des fonctions à réaliser par le programme) est affecté par la probabilité qu'un paramètre particulier soit demandé au programme, et dans ce cas, la probabilité que de tel paramètre prenne une valeur spécifique. En d'autres termes, le choix des paramètres en entrées au programme, aussi bien que les valeurs à affecter pour ces paramètres, va influencer le processus d'exécution du programme, ainsi que son comportement général face à ces entrées. Par exemple, on suppose que la i-ème fonction du programme ne peut être exécutée qu'après la réception du résultat de la (i-1) ème fonction (ou même de toutes les fonctions qui la précédent). Le résultat de l'exécution de la i-ème fonction dépend directement des résultats de l'exécution de toutes les fonctions qui la précèdent. Cette situation correspond exactement au cas des programmes qui interagissent

avec un utilisateur et/ou avec des environnements externes. Dans ce cas, la distribution de probabilité de la prochaine entrée (si elle existe), dépend généralement des valeurs des entrées précédentes données au programme lors de la même exécution. Dans ce cas de situations, le processus qui décrit les entrées du programme peut être modélisé par une chaîne de Markov à temps discret.

Une façon naïve pour spécifier ce modèle de chaîne de Markov à temps discret est de considérer que $\{X_j, j \geq 0\}$ est une chaîne de Markov sur l'ensemble des nombres réels (ou même un sous-ensemble de celui-ci), tel que X_j représente la j-ème entrée au programme. Ceci n'est pas vraiment réaliste car généralement, la prochaine entrée X_{j+1} dépend non seulement de X_j mais, aussi de toutes les valeurs des entrées qui la précèdent. Par conséquent, le processus $\{X_j, j \geq 0\}$ précédemment défini n'est pas un bon modèle de chaîne de Markov.

Une façon plus propre pour contourner le problème est de modéliser le processus des entrées par une chaîne de Markov $\{S_j, j \geq 0\}$ dont l'état S_j à l'étape j, contient plus d'information que celles de X_j. En effet, notre chaîne de Markov $\{S_j, j \geq 0\}$ peut être définie d'une façon récursive à l'aide d'une récurrence stochastique de la forme $S_j = \gamma_j (S_{j-1}, X_j)$, où γ_j est une fonction de transition (définie au préalable). Dans ce cas, la j-ème entrée X_j est censée avoir une distribution de probabilité qui dépend de S_{j-1} mais elle est également indépendante de $X_0, \ldots, X_{j-1}$ conditionnellement à S_{j-1}. Cette hypothèse nous garantit que $\{S_j, j \geq 0\}$ est bien un modèle de chaîne de Markov, ce qui signifie que chaque fois qu'on connait l'état S_{j-1}, alors, le fait de connaitre les états qui le précèdent (à savoir, $S_0, \ldots, S_{j-1}$) n'apportera plus d'informations supplémentaires utiles pour prédire le comportement de n'importe quelle entrée X_l ou S_l telle que $l > j$.

Nous avons également considéré que notre chaîne de Markov possède un temps d'arrêt aléatoire τ défini comme étant le premier moment où l'état de la chaîne $\{S_j, j \geq 0\}$ est dans l'ensemble Δ, où Δ désigne l'ensemble des états d'arrêts possibles du programme :

$$\tau = \inf \{j \geq 0 : S_j \in \Delta\}.$$

Ce τ représente le nombre (aléatoire) des entrées dont le programme a besoin.

Avec ces hypothèses et ces définitions, nous pouvons définir une *métrique dynamique de couplage* comme une fonction Φ_2 qui associe pour chaque réalisation (S_0, S_1, …, S_τ, τ) le nombre réel Y = Φ_2 (S_0, S_1, …, S_τ, τ). Encore une fois, si notre modèle de chaîne de Markov est complètement spécifié, alors, on peut le simuler et estimer la distribution de Y à l'aide des distributions empiriques des *n* réalisations indépendantes Y_1,…, Y_n de la même façon que nous l'avons faite dans le cas du vecteur aléatoire.

Un autre exemple intéressant où les modèles de chaîne de Markov sont utilisées pour modéliser les entrées des programmes est décrit dans [70] et a été présenté en détail dans le chapitre 2. Dans ce travail, les auteurs ont proposé un modèle de chaîne de Markov pour modéliser la navigabilité dans les sites web. En effet, l'interaction entre le site web et l'utilisateur (l'internaute) est définie par l'ensemble des actions élémentaires que l'utilisateur peut faire dans le site, tels que les clics de souris sur les liens qui correspondent aux « URL » des pages du site. La probabilité que l'utilisateur accède à une page particulière dépend des pages précédentes qu'il a déjà visitées (c'est-à-dire, les entrées précédentes). Le modèle de chaîne de Markov proposé par les auteurs a été utilisé pour mesurer la navigabilité des sites web.

Considérons un programme *P2* qu'on exécute *n* fois. À chaque possibilité d'exécution, on doit entrer un nombre différent de valeurs, jusqu'à ce qu'une condition d'arrêt de *P2* sera satisfaite. Sachant dans chaque répétition *i* les valeurs des entrées de *P2*, ainsi que le temps d'arrêt τ_i, il est possible de calculer une valeur de métrique par application directe de la fonction Φ_2. Après *n* répétitions, on collecte les statistiques de *n* valeurs de métriques de couplage, ainsi que les chaînes de Markov associées ; lors de chaque exécution de *P2*, on lance une nouvelle chaîne de Markov dont on construit les états au fur et à mesure que le programme s'exécute. Par conséquent, il est possible de construire un histogramme qui décrit le comportement de la métrique (sa distribution) lors des *n* exécutions de *P2*. Dans cette distribution, on a toutes les informations nécessaires pour caractériser le comportement de n'importe quelle classe de *P2*, à savoir, les valeurs des entrées du programme (c'est-à-dire, les états de la chaîne de Markov), ainsi, que les valeurs de métriques obtenues (valeurs-résultats de Φ_2).

3.4. Utilité des histogrammes

Un histogramme est une représentation graphique permettant de montrer la distribution de probabilité d'une variable aléatoire. Dans ce travail, un histogramme nous permet d'observer la répartition des valeurs de couplage afin de distinguer les valeurs les plus fréquentes (ou les moins fréquentes), ce qui nous permet entre autres de cibler les exécutions les plus fréquentes (ou les moins fréquentes).

À partir d'un histogramme, il est aussi possible d'observer les différentes variations des valeurs de couplage, ce qui nous permet de distinguer des compartiments dans l'histogramme qui peuvent correspondre à un rôle particulier de la classe.

Éventuellement, à partir des distributions de métriques des classes d'un programme, il est possible de distinguer les classes qui donnent les plus grandes valeurs de couplage observées (respectivement les plus petites valeurs de couplage) à partir des scénarios d'exécutions. Lors des tâches de maintenance, ces classes vont donc être considérées plus que d'autres classes, par exemple lors du « refactoring ».

3.5. Conclusion

Dans ce chapitre, nous avons débuté par présenter le schéma global de notre approche. Nous avons ensuite donné les détails de nos modèles probabilistes. Dans ces modèles, nous avons fait la distinction entre deux types de programmes : ceux dont le nombre des entrées est fixé d'avance et qu'on peut donc les modéliser par un vecteur aléatoire de taille fixe, et les programmes dont le nombre des entrées est inconnu d'avance, et dans ce cas, nous avons décidé de modéliser leurs entrées par une chaîne de Markov à temps discret. Les modèles que nous avons proposés vont nous servir pour caractériser les dépendances en générant des distributions de probabilités qui correspondent aux métriques de couplage relatives à chaque classe du programme.

Une fois que nous avons caractérisé les dépendances, il est important maintenant de comprendre ces dépendances en analysant de façon détaillée les distributions de probabilités de métriques obtenues. C'est l'objectif du chapitre suivant.

Chapitre 4

ÉTUDE DES DÉPENDANCES

4.1. Introduction

Dans le chapitre précédent, nous avons montré comment caractériser les liens de dépendance à l'aide des modèles probabilistes des entrées pour le cas d'un vecteur aléatoire (programme à nombre limité d'entrées) ainsi que le cas d'une chaîne de Markov (le nombre d'entrées du programme est aléatoire et non connu d'avance). Mais, jusqu'à présent, nous n'avons pas encore expliqué comment on peut étudier ces liens de dépendance.

Dans ce chapitre, nous proposons une réflexion sur le lien entre les mesures de dépendance qu'a une classe avec les autres classes du programme et son rôle dans les fonctionnalités du système. Dans ce cadre, nous rappelons d'abord le rôle que peut jouer une classe dans le comportement d'un système. Par la suite, nous établissons un lien entre les variantes de ce rôle et la distribution du couplage. Pour ce faire, nous proposons des mesures pour calculer des degrés de similarité du comportement entre les exécutions générant les mêmes valeurs de couplage (similarité interne) ou des valeurs de couplage voisines (similarité externe). Ces mesures nous permettent d'identifier les exécutions dans lesquelles une classe joue un rôle typique, ainsi que de déterminer des régions d'exécutions qui correspondent à un comportement spécifique du système. Avant de conclure, nous donnons un petit exemple illustratif de notre cadre de travail.

4.2. Rôle d'une classe

La programmation OO a été définie de façon à ce que les objets d'un programme interagissent entre eux pour réaliser les différentes fonctionnalités de ce programme. De ce fait, chaque classe peut offrir un ou plusieurs services lors de l'exécution du programme.

Un service offert par une classe peut se décliner avec une ou plusieurs variantes qui, en général, découlent des variantes des cas d'utilisation du système. En effet, un cas d'utilisation d'un système est réalisé à travers plusieurs scénarios, dont un principal (« main flow ») et les autres des variantes (« alternative flows ») [1]. Par exemple, lors de l'emprunt d'un livre à la bibliothèque, le scénario principal consiste à balayer la carte du lecteur et balayer le code du livre ce qui a pour effet d'enregistrer l'emprunt et de décrémenter le nombre d'exemplaires disponibles du livre. Ce scénario est exécuté dans la majorité des emprunts. Comme conséquence, un service offert par une classe a une variante principale (correspondante au scénario principal) et des variantes secondaires. Le scénario principal étant le plus fréquemment exécuté, il est naturel de penser que la variante principale d'un service est aussi la plus fréquemment exécutée.

Une autre propriété des cas d'utilisation est que ces derniers sont souvent étendus (relation <<extend>>) pour tenir compte de situations exceptionnelles [59]. Par exemple, lors de l'emprunt, la carte du lecteur peut être expirée. Il faut donc renouveler la carte puis effectuer l'emprunt. Ce type d'extensions est rare et nécessite en plus de l'exécution normale du scénario principal, l'exécution d'un comportement additionnel. Là également, la variante principale d'un service peut être étendue en accord avec l'extension du scénario principal correspondant.

Une dernière propriété qui nous intéresse et qui découle aussi des cas d'utilisation a trait aux pré-conditions. En général, pour qu'un scénario s'exécute, il faut que certaines conditions soient réunies. Par exemple, un livre mis en réserve ne peut pas être emprunté ou encore un lecteur ne peut emprunter un livre s'il a déjà atteint le maximum de livres permis. Dans ce cas, une variante minimale du cas d'utilisation est exécutée. La notion de pré-conditions peut se transposer aussi sur les services offerts par les classes. Un service possède des pré-conditions et ne peut s'exécuter que si celles-ci sont vérifiées. Pour le service, comme pour le cas d'utilisation, la variante correspondante aux violations des pré-conditions est moins fréquente et nécessite une exécution plus restreinte que l'exécution principale.

4.3. Liens entre les dépendances et le rôle d'une classe

Chaque fois qu'on exécute un programme, des classes [2] interviennent pour réaliser un ou plusieurs services. Ces services nécessitent des interactions avec les autres classes et donc engendrent une valeur de couplage pour une exécution donnée. D'autre part, lorsqu'on exécute un programme plusieurs fois, une même valeur de couplage peut être obtenue plusieurs fois pour une classe donnée. On appelle *la fréquence* d'un couplage le nombre de fois où une valeur de couplage apparait. Dans ce qui suit, nous allons essayer d'établir le lien entre le comportement d'une classe et les valeurs de couplage obtenues lors des exécutions. Pour ce faire, nous proposons un ensemble d'hypothèses qui seront étudiées empiriquement dans le chapitre 5.

- Hypothèse 1 : Une classe qui exécute un ou plusieurs services dans des exécutions différentes avec la même variante va produire des valeurs de couplage identiques ou tout au moins très proches. Nous pensons qu'il est peu probable que deux exécutions avec des services différents ou des variantes différentes produisent la même valeur de couplage. Nous conjecturons donc que si une classe a la même valeur de couplage pour deux exécutions, alors il est fort probable que ce sont les mêmes services avec les mêmes variantes qui se sont exécutés.

- Hypothèse 2 : Une valeur de couplage produite par une exécution avec l'extension d'une variante principale d'un service est souvent supérieure ou égale à celle produite par une exécution avec la variante principale. Cette hypothèse découle du fait que le comportement additionnel peut susciter des interactions supplémentaires.

- Hypothèse 3 : Une valeur de couplage produite par une exécution avec une violation des pré-conditions d'un service est souvent inférieure ou égale à celle produite par une exécution avec la variante principale. Cette hypothèse découle du fait qu'une partie du comportement n'est pas exécutée et donc les interactions associées à cette partie ne sont pas comptabilisées.

---

[2] En réalité, ce sont les objets qui s'exécutent et qui interagissent. Cependant, comme la classe factorise le comportement de ces objets, nous utiliserons le terme « classe » par extension.

Si les trois hypothèses précédentes sont avérées pour une classe donnée, il est possible d'interpréter sa distribution pratique du couplage (par rapport aux différentes exécutions) en termes de son rôle fonctionnel. Considérons la Figure 4 qui montre une distribution idéalisée du couplage pour une classe donnée. Cette distribution montre les fréquences f_1, f_2 et f_3 respectivement observées pour les valeurs de couplage c_1, c_2 et c_3.

Étant donné que le couplage c_2 est plus fréquent, on peut penser qu'il représente la variante principale des services offerts par la classe. Le couplage c_3 est plus élevé que c_2 et est beaucoup moins fréquent. L'explication qui vient à l'esprit est que c_3 représente une certaine extension des services principaux relative à un traitement exceptionnel. Finalement, le couplage c_1, moins élevé et moins fréquent que c_2, fait naturellement penser à des exécutions tronquées en raison de la non-satisfaction de pré-conditions.

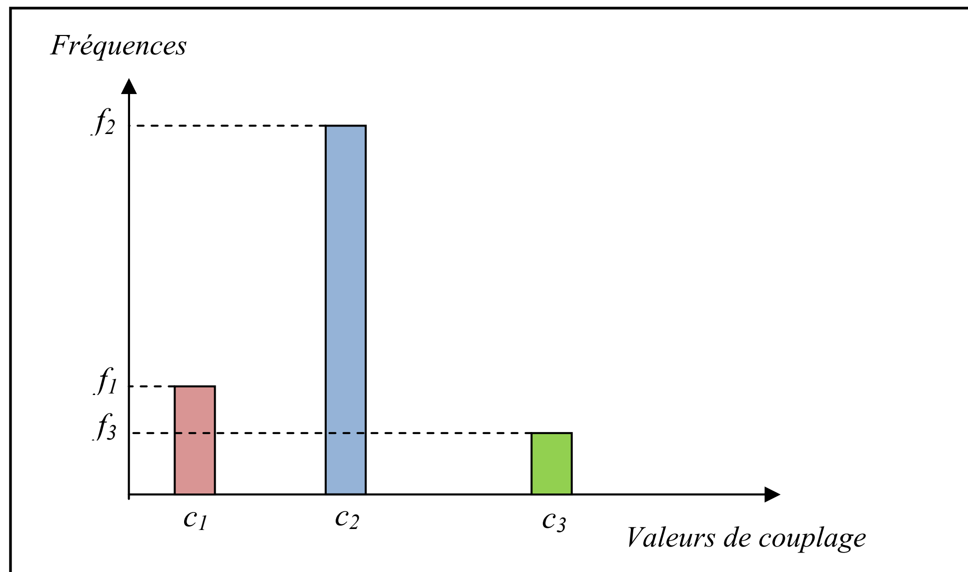


Figure 4. Fréquence des variantes d'un service pour une classe.

Dans le reste de ce chapitre, nous allons proposer un cadre pour étudier empiriquement ces hypothèses. Pour y parvenir, nous proposons des mesures de similarité permettant d'évaluer l'homogénéité d'un groupe d'exécutions ayant la même valeur de couplage, ainsi que la similarité entre groupes d'exécutions ayant des valeurs différentes.

4.4. Similarité entre exécutions et entre groupes d'exécutions

Dans la section précédente, nous avons présenté un ensemble d'hypothèses sur le lien entre le comportement d'une classe et les valeurs de couplage obtenues lors des exécutions d'un programme. Jusqu'à présent, nous n'avons pas encore vérifié ces hypothèses. Pour cela, nous allons proposer dans le cadre de cette section, un ensemble de mesures qui calculent des similarités entre des exécutions et des groupes d'exécutions d'un programme. Nous allons montrer par la suite, comment ces mesures peuvent nous renseigner sur le lien entre le comportement d'une classe et les valeurs de couplage obtenues. Pour cela, nous commençons par présenter quelques définitions sur lesquelles nous allons baser nos mesures. Les définitions que nous proposons ici s'appliquent à un échantillon donné d'exécutions, et non à toute la population définie par le modèle probabiliste.

4.4.1. Définitions

Définition 1 : ***Appel***

Lors d'une exécution d'un programme, un appel de méthode est défini par un quadruplet (*classeAppelante, méthodeAppelante, classeAppelée, méthodeAppelée*) où *classeAppelante* est la classe de l'objet qui exécute la *méthodeAppelante* dans laquelle un message est envoyé à l'objet de *classeAppelée* pour qu'il exécute la *méthodeAppelée*.

Définition 2 : ***Trace d'exécution***

Une trace d'exécution est la séquence d'appels effectués pendant une exécution d'un programme. Pour chaque exécution du programme, une trace d'exécution est obtenue.

Définition 3 : ***Métrique de couplage dynamique***

Une métrique de couplage dynamique comptabilise un volume d'interactions (ou type d'interactions) pour une classe donnée à partir des appels répertoriés dans une trace d'exécution. Une mesure simple consiste à compter le nombre d'appels (quadruplets) différents dans la trace d'exécution. Nous utiliserons cette mesure dans la suite de ce chapitre.

Définition 4 : ***Bloc d'exécutions***

Un bloc d'exécutions B est un ensemble de traces d'exécutions qui engendrent la même valeur de couplage pour une classe donnée. Étant donnée une classe d'un programme, on associe pour chaque valeur de couplage dynamique c un bloc B. On représente B par $B(c,f_c)$ où f_c désigne la fréquence d'apparition de la valeur de couplage c dans l'échantillon d'exécutions.

4.4.2. Similarité intra-bloc (ou similarité interne)

La *similarité intra-bloc* ou *similarité interne* (IS) est une mesure qui représente la diversité des appels dans les exécutions au sein d'un même bloc. Formellement, IS pour un bloc B est défini comme suit :

$$IS(B) = \frac{c_B}{n_B}$$

où n_B est le nombre d'appels différents observés dans les exécutions du bloc B et c_B est la valeur de couplage du bloc B.

Le cas idéal est lorsque toutes les exécutions à l'intérieur d'un bloc possèdent exactement les mêmes appels. Dans ce cas, la similarité interne est maximale. En effet, l'ensemble d'appels différents est égal à l'ensemble des appels de chaque exécution (c_B=n_B), et donc la similarité est égale à 1. La similarité maximale Max_B peut être différente de 1 si la métrique de couplage ne compte pas le nombre d'appels différents. Ici, on fait l'hypothèse que la métrique de couplage considérée calcule le nombre d'appels différents.

Le pire cas est lorsque les ensembles d'appels différents des exécutions de B sont disjoints. Dans ce cas, n_B est égale à $c_B\, f_{c_B}$ où f_{c_B}représente le nombre d'exécutions dans le bloc B. La similarité minimale est donc

$$Min_B = \frac{c_B}{n_B} = \frac{c_B}{c_B\, f_{c_B}} = \frac{1}{f_{c_B}} \ll 1$$

Pour garantir que nos mesures de similarité aient la même interprétation, nous les normalisons par rapport aux valeurs minimale et maximale de chaque bloc. Ainsi, la forme normalisée de *IS* est définie comme suit :

$$NIS(B) = \frac{IS(B) - Min_B}{Max_B - Min_B}$$

Dans le reste de ce mémoire, nous utilisons cette forme normalisée de *IS*. Quand *NIS*(*B*) est proche de 1, on considère que le bloc *B* contient une majorité d'exécutions similaires représentant une exécution typique pour la valeur de couplage associée. Plus nous nous éloignons du 1, plus nous considérons que les exécutions sont hétérogènes et ne représentent pas un service particulier.

4.4.3. Similarité inter-blocs (ou similarité externe)

Une fois que nous avons vérifié grâce à la mesure *NIS* si un bloc représente une exécution typique et donc conforme à l'hypothèse 1, la prochaine étape est de déterminer si des blocs contigus forment une région correspondante à une même variante d'un service. Pour y parvenir, nous proposons une deuxième mesure qui évalue des similarités externes entre blocs. Avant de décrire cette mesure, nous allons d'abord introduire la notion d'*appels fréquents* dans un bloc.

Comme nous l'avons montré dans la section précédente, la similarité interne d'un bloc peut prendre des valeurs différentes. Ces valeurs sont fonction de la présence simultanée des appels dans les exécutions du bloc. Plus un appel est présent simultanément dans le bloc, plus il est dit *fréquent*. Nous définissons *la fréquence relative FR*(*a, B*) d'un appel *a* dans un bloc *B* comme le pourcentage des exécutions de *B* dans lesquelles *a* apparaît. On définit une valeur seuil qu'on appelle *Relative_Frequency_Threshold*, et nous considérons qu'un appel est fréquent au cas où sa fréquence relative est au-delà de notre valeur seuil, et donc participe à l'exécution typique de ce bloc. Pour comparer des blocs successifs, il est donc judicieux de tenir compte uniquement des appels fréquents respectivement dans chaque bloc.

La *similarité inter-blocs* ou *similarité externe* (*ES*) est aussi une mesure, comprise entre 0 et 1, qui évalue la variabilité des appels entre un bloc est un ensemble de blocs qui le précèdent (une région). Le premier bloc B_0 est considéré comme faisant partie d'une région. Par la suite, nous calculons la similarité entre B_0 et B_1. Si celle-ci est supérieure à un certain seuil (qu'on appelle *External_Threshold*), on considère que B_1 fait également partie de la région courante. Par la suite, on calcule la similarité de B_2 avec B_1 mais en tenant compte des blocs déjà inclus dans la région courante et ainsi de suite. Dès que la similarité entre le dernier bloc d'une région et un nouveau bloc est en dessous du seuil, la région est complétée et une nouvelle région est créée. Dans ce cas, la liste des appels communs sera initialisée et elle va contenir les différents appels fréquents du prochain bloc. On répète le processus jusqu'au bloc ayant la valeur de couplage la plus élevée. À la fin, un ensemble de régions contiguës est créé. Formellement, la similarité entre un bloc B_i d'une région R_k et un bloc B_j qui le suit est définie par :

$$ES\,(B_i\,, B_j\,) = \frac{\sum_{a \in cl_k \cap l_i \cap\, l_j} \left(FR\,(a{,}B_i) + FR\,\left(a{,}B_j\right)\right) / \, 2}{|cl_k|}$$

où cl_k est la liste des appels différents communs à tous les blocs déjà inclus dans R_i et, l_i et l_j sont les listes respectives des appels fréquents des blocs B_i et B_j. En effet, en considérant tous les appels communs et fréquents qui apparaissent dans la région courante (R_k) et le bloc suivant (B_j), on calcule la moyenne des fréquences relatives de chaque appel *a* dans le bloc B_i et le bloc B_j (c'est-à-dire, (1/2) [$FR(a,\ B_i) + FR(a,\ B_j)$]). On divise la somme de toutes ces moyennes relatives par le nombre des appels communs observés entre la région courante (R_k) et le bloc B_j (c'est-à-dire, $|cl_k|$). Dans le cas où tous les appels observés ont une fréquence relative maximale (FR=1), alors, $ES\,(B_i\,, B_j\,) = \frac{|cl_k|}{|cl_k|} = 1$. Il est à noter que pour le calcul de la similarité externe, seuls les blocs ayant une similarité interne supérieure à un certain seuil (qu'on appelle *Internal_Threshold*) sont considérés.

4.4.4. Exemple

Afin d'illustrer les détails de notre approche et plus particulièrement ceux que nous avons présentés dans le présent chapitre, nous proposons un exemple que nous allons appliquer sur l'histogramme de la figure ci-dessous obtenu sur un échantillon de 100 exécutions. Cet histogramme donne la distribution de métriques de couplage pour une classe donnée d'un programme. Il montre 8 blocs d'exécutions tels que chaque bloc B_i correspond à la valeur de couplage i où $i \in \{2, \ldots, 9\}$. À partir de l'histogramme, on peut déterminer le nombre d'observations qui ont engendré la valeur de couplage de chacun de ces 8 blocs. Pour cet exemple, nous allons juste nous intéresser aux trois valeurs de couplage 2, 3 et 4. Par conséquent, nous allons nous focaliser sur les trois blocs d'exécutions B_2 (2, 7), B_3 (3, 10) et B_4 (4, 6).

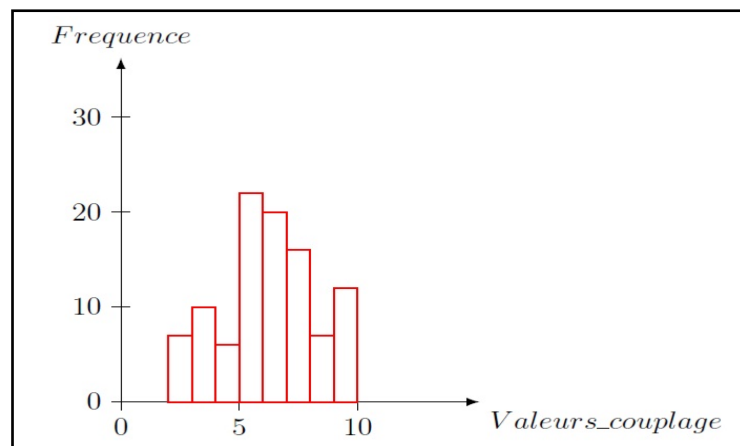


Figure 5. Exemple d'un histogramme avec n =100 réalisations.

Les tableaux 1, 2 et 3 suivants montrent la liste des blocs générés et qui correspondent respectivement aux trois valeurs de couplage 2, 3 et 4. Nous désignons par a_i le i-ème appel observé dans un bloc donné. La première ligne de chaque tableau désigne les numéros des exécutions, alors que les autres lignes correspondent aux différents appels observés.

1	6	14	27	28	46	98
a_1	a_4	a_3	a_3	a_4	a_1	a_4
a_3	a_2	a_1	a_2	a_3	a_2	a_2

Tableau 1. Exemple d'un bloc d'exécutions B_2 correspondant à la valeur de couplage 2

2	3	19	26	37	41	63	71	75	100
a_1	a_3	a_2	a_2	a_2	a_2	a_1	a_8	a_2	a_2
a_8	a_2	a_1	a_8	a_1	a_8	a_2	a_1	a_1	a_1
a_2	a_8	a_8	a_1	a_8	a_3	a_8	a_2	a_8	a_3

Tableau 2. Exemple d'un bloc d'exécutions B_3 correspondant à la valeur de couplage 3.

4	22	34	69	83	94
a_3	a_9	a_8	a_5	a_9	a_6
a_5	a_6	a_5	a_3	a_5	a_8
a_6	a_3	a_6	a_8	a_6	a_5
a_8	a_5	a_3	a_6	a_8	a_3

Tableau 3. Exemple d'un bloc d'exécutions B_4 correspondant à la valeur de couplage 4.

- *Similarité intra-bloc ou similarité interne (IS) :*

Pour chacun des 3 blocs B_2, B_3 et B_4, nous allons associer respectivement les listes $list_2$, $list_3$ et $list_4$ qui contiennent l'ensemble d'appels différents observés dans les exécutions de ces 3 blocs. $list_2 = \{a_1, a_2, a_3, a_4\}$; $list_3 = \{a_1, a_2, a_3, a_8\}$; $list_4 = \{a_3, a_5, a_6, a_8, a_9\}$.
f_2, f_3 et f_4 dénotent respectivement le nombre (ou la fréquence) des exécutions des blocs B_2, B_3 et B_4 tels que $f_2 = 7$, $f_3 = 10$ et $f_4 = 6$.

On a : $n_2 = |list_2| = 4$, $n_3 = |list_3| = 4$ et $n_4 = |list_4| = 5$.

$IS(B_2) = 2 / n_2 = 2/4 = 0.5$; $IS(B_3) = 3 / n_3 = 3/4 = 0.75$; $IS(B_4) = 4 / n_4 = 4/5 = 0.8$.

$Min_2 = 1 / f_2 = 1/7$; $Min_3 = 1 / f_3 = 1/10$; $Min_4 = 1 / f_4 = 1/6$.

$Max_2 = Max_3 = Max_4 = 1$ (car on suppose dans cet exemple que notre métrique de couplage compte le nombre d'appels différents). Ainsi, les mesures normalisées de *IS* seront :

$NIS(B_2) = \frac{IS(B_2) - Min_2}{Max_2 - Min_2} = 5/12$; $NIS(B_3) = \frac{IS(B_3) - Min_3}{Max_3 - Min_3} = 7/9$;

$NIS(B_4) = \frac{IS(B_4) - Min_4}{Max_4 - Min_4} = 19/25$.

Si on considère par exemple que notre valeur seuil est égale à 0.4, alors, aucun bloc ne sera éliminé et les 3 blocs d'exécutions (B_2, B_3 et B_4) seront pris en compte puisque leur facteur de similarité interne est au-delà de 0.4.

- *Similarité inter-blocs* ou *similarité externe* (*ES*) :

Avant de commencer à calculer les coefficients de similarité externes entre les 3 blocs B_2, B_3 et B_4, nous allons examiner la liste des appels que contient chacun de ces 3 blocs, en vue de *supprimer* (c'est-à-dire, ne pas prendre en compte) les appels qui apparaissent moins souvent ou même rarement. Pour cela, nous allons calculer la fréquence relative de chacun de ces appels à l'intérieur de leurs blocs d'exécutions.

- Bloc B_2 : $FR(a_1, B_2) = 3/7$; $FR(a_2, B_2) = 4/7$; $FR(a_3, B_2) = 4/7$; $FR(a_4, B_2) = 3/7$.
- Bloc B_3 : $FR(a_1, B_3) = 8/10$; $FR(a_2, B_3) = 10/10$; $FR(a_3, B_3) = 3/10$; $FR(a_8, B_3) = 9/10$
- Bloc B_4 : $FR(a_3, B_4) = 5/6$; $FR(a_5, B_4) = 6/6$; $FR(a_6, B_4) = 6/6$; $FR(a_8, B_4) = 5/6$; $FR(a_9, B_4) = 2/6$.

A titre d'exemple, si on fixe notre valeur seuil à 0.4, alors tous les appels sont fréquents sauf l'appel a_3 du bloc B_3 et l'appel a_9 du bloc B_4. De ce fait, on doit éliminer l'appel a_3 de $list_3$ et l'appel a_9 de $list_4$. Le reste des appels vont être pris en compte. Ainsi, on définit les nouvelles listes l_2, l_3 et l_4 qui contiennent les appels fréquents respectivement des blocs B_2, B_3 et B_4, de telle sorte que $l_2 = list_2$, $l_3 = list_3 \setminus \{a_3\}$ et $l_4 = list_4 \setminus \{a_9\}$.

Une fois que nous avons sélectionné les appels les plus pertinents, il est possible maintenant de calculer le coefficient de similarité externe entre les deux premiers blocs B_2

et B_3. Pour cela, on crée une liste commune cl_1 associée à une région R_1. cl_1 est censée contenir les appels différents communs entre les deux blocs B_2 et B_3 ou plus précisément, entre leurs listes d'appels respectives l_2 et l_3.

Initialement, $cl_1 = l_2 = \{a_1, a_2, a_3, a_4\}$ et donc $cl_1 \cap l_2 \cap l_3 = \{a_1, a_2\}$.

$$ES\,(B_2, B_3) = \frac{FR(a_1, B_2) + FR(a_1, B_3) + FR(a_2, B_2) + FR(a_2, B_3)}{2\,.\,|cl_1|} = 7/10.$$

Si on prend par exemple une valeur seuil égale à 0.6 alors, B_2 et B_3 doivent être dans une même région d'exécutions (R_1) puisqu'ils partagent presque le même comportement.

$ES\,(B_3, B_4) = \frac{0}{2\ |cl_1|} = 0$ car $cl_1 \cap l_4 = \emptyset$. Et comme $ES(B_3,B_4)<0.6$, alors, B_3 et B_4 ne peuvent être dans la même région d'exécution R_1 puisqu'ils ne partagent aucun comportement commun. Par conséquent, on doit regrouper B_2 et B_3 dans la même région R_1 et B_4 doit appartenir à une autre région (R_2). Bien sûr, il ne faut pas oublier de mettre à jour notre liste d'exécutions communes pour l'itération suivante, c'est-à-dire, $cl_1 = l_4$.

4.5. Conclusion

Dans ce chapitre, nous avons commencé par présenter un descriptif qui résume le rôle que peut jouer une classe dans un programme en se basant sur les cas d'utilisation. Nous avons ensuite fait le lien entre les dépendances et le comportement d'une classe dans un système. Pour y parvenir, nous avons proposé deux mesures qui calculent des degrés de similarité entre les exécutions d'un programme et au sein d'une même classe. Ces deux mesures que nous avons introduites servent non seulement à identifier les exécutions typiques à l'intérieur d'une classe (similarité interne), mais aussi pour définir des régions dans la classe (similarité externe). Ces régions définissent elles-mêmes un comportement spécifique et commun dans tout le programme. Éventuellement, pour expliquer les détails de calcul de notre approche, nous avons présenté un petit exemple.

Une fois que nous avons donné une explication détaillée de notre approche via les deux chapitres 3 et 4, il reste alors de la valider. Dans le prochain chapitre, nous allons présenter deux études de cas pour montrer les résultats de notre approche.

Chapitre 5

ÉTUDES DE CAS

5.1. Introduction

Dans les deux chapitres précédents, nous avons donné les détails de notre approche. La partie étude de cas de nos travaux sera présentée dans ce chapitre, dans lequel nous allons donner l'architecture générale de l'implémentation de notre approche, ainsi que deux études de cas détaillés : une pour le cas d'un vecteur aléatoire et l'autre pour le cas d'une chaîne de Markov. Dans chaque étude de cas, nous allons donner une description du programme, du modèle probabiliste des entrées, ainsi que le choix de la métrique de couplage dynamique. Nous allons ensuite donner les résultats et les observations que nous avons pu obtenir, à savoir, les types de patrons de dépendance générés par notre approche et nous spécifions par la suite comment ces patrons peuvent nous aider à faciliter la compréhension des liens de dépendance et l'analyse du comportement du programme. Avant de conclure, nous allons discuter ces résultats que nous joindrons par quelques critiques dont l'objectif est d'améliorer notre approche dans des futurs travaux.

5.2. Implémentation

Dans nos études de cas, nous avons considéré deux programmes de petite taille :

- ✓ *Un système d'ascenseurs* (8 classes) pour le cas d'une chaîne de Markov où le nombre des entrées du programme n'est pas connu d'avance.
- ✓ *Un générateur de grilles de Sudoku* (13 classes) pour le cas d'un vecteur aléatoire où le nombre des entrées du programme est fixé d'avance.

Nous allons détailler chacun de ces deux cas dans les prochaines sections de ce chapitre. Pour chacun de ces deux types de programme, nous avons construit un modèle probabiliste d'entrées comme nous l'avons déjà expliqué dans le chapitre 3. L'architecture générale de l'implémentation de notre approche est donnée par la figure 6 ci-dessous. Nous avons implémenté pour chaque programme un simulateur qui génère la liste des entrées en respectant le modèle mathématique qui lui est spécifié pour les entrées du programme.

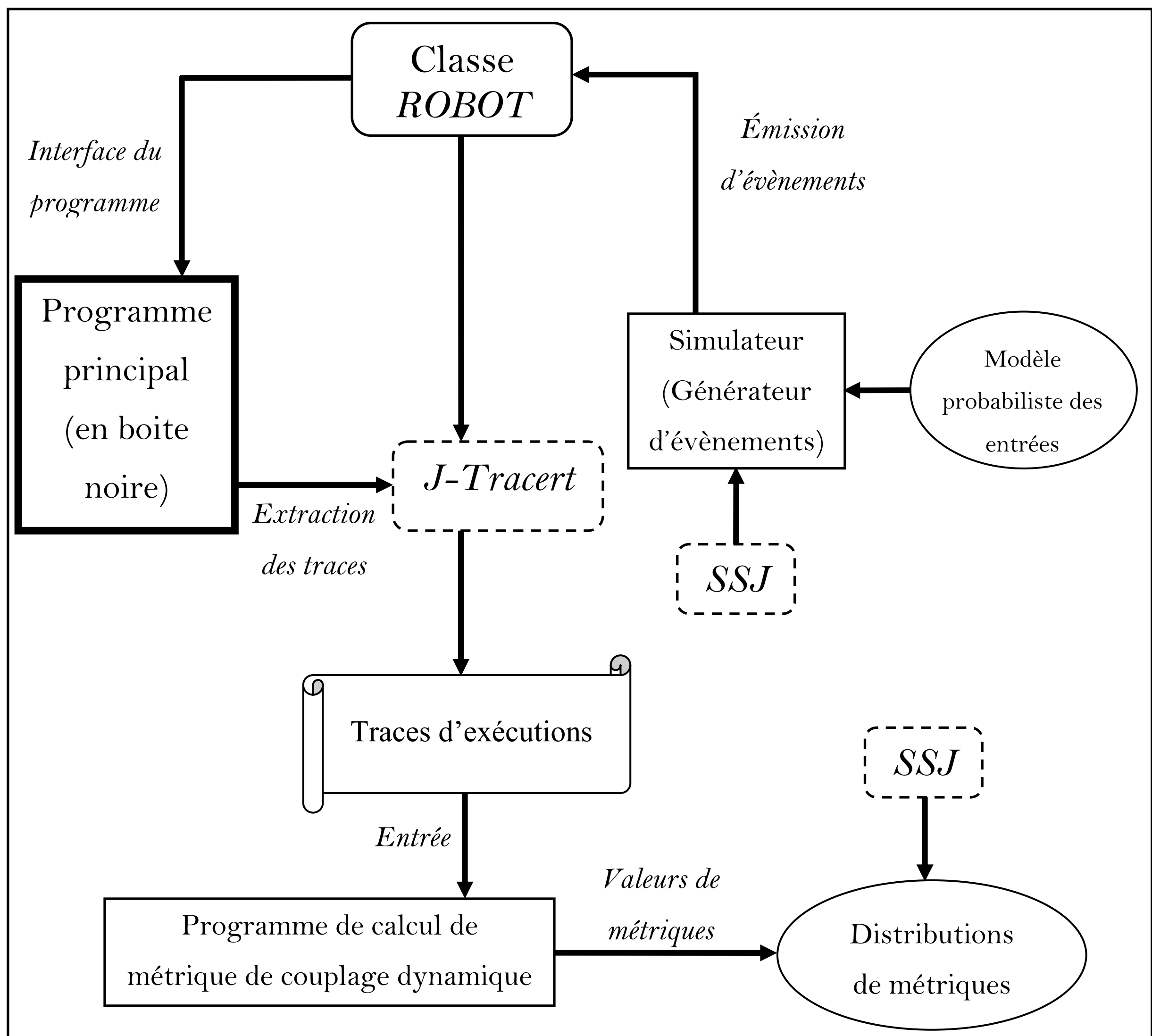


Figure 6. Architecture générale de l'implémentation de notre approche.

Pour simuler les entrées du programme, nous avons utilisé la librairie *SSJ* (*Stochastic Simulation in Java*) [33] qui a été implémentée au sein du département d'informatique et de recherche opérationnelle de l'Université de Montréal. *SSJ* fournit des facilités pour utiliser les techniques de simulation stochastiques en général et les méthodes Monte-Carlo en particulier. Elle contient différents paquetages qui offrent des moyens pour générer des valeurs aléatoires uniformes et non uniformes et pour donner différentes statistiques liées à des lois de probabilité, effectuer des tests d'ajustement (*goodness-of-fit* en anglais), appliquer des méthodes de type quasi-Monte-Carlo, programmer des simulations à évènements discrets, etc.

Pour chaque exécution du programme, on génère une liste d'évènements basée sur les lois de probabilité spécifiées dans notre modèle mathématique. Ces évènements vont être introduits à une classe *ROBOT* qui sert à gérer les interactions avec l'interface graphique du programme. Entre temps que la classe *ROBOT* manipule l'interface du programme, un graphe d'appel dynamique sera produit au fur et à mesure que le programme s'exécute. Pour y parvenir, nous avons utilisé l'outil *J-Tracert* (sous sa version 0.1.0) [24] qui, couplé avec l'environnement de programmation de Java (*Éclipse*), permet de collecter des traces lors de l'exécution du code. En effet, *J-Tracert* a été conçu pour extraire des diagrammes de séquences lors de l'exécution d'un programme, ce qui facilite la compréhension et la lisibilité du code (un apprentissage rapide). En particulier, *J-Tracert* permet de déterminer les appels de méthodes, identifier quels sont les objets instanciés qui communiquent ensemble par envoi et/ou réception de messages, les appels entre les classes d'un programme, etc. ce qui permet d'étudier le comportement du programme *lors de son exécution.* Nous avons modifié le code de *J-Tracert* afin de visualiser uniquement les méthodes et les classes appelantes, ainsi que les méthodes et les classes appelées. La figure suivante montre un exemple d'utilisation de *J-Tracert* en parallèle avec l'exécution d'un programme.

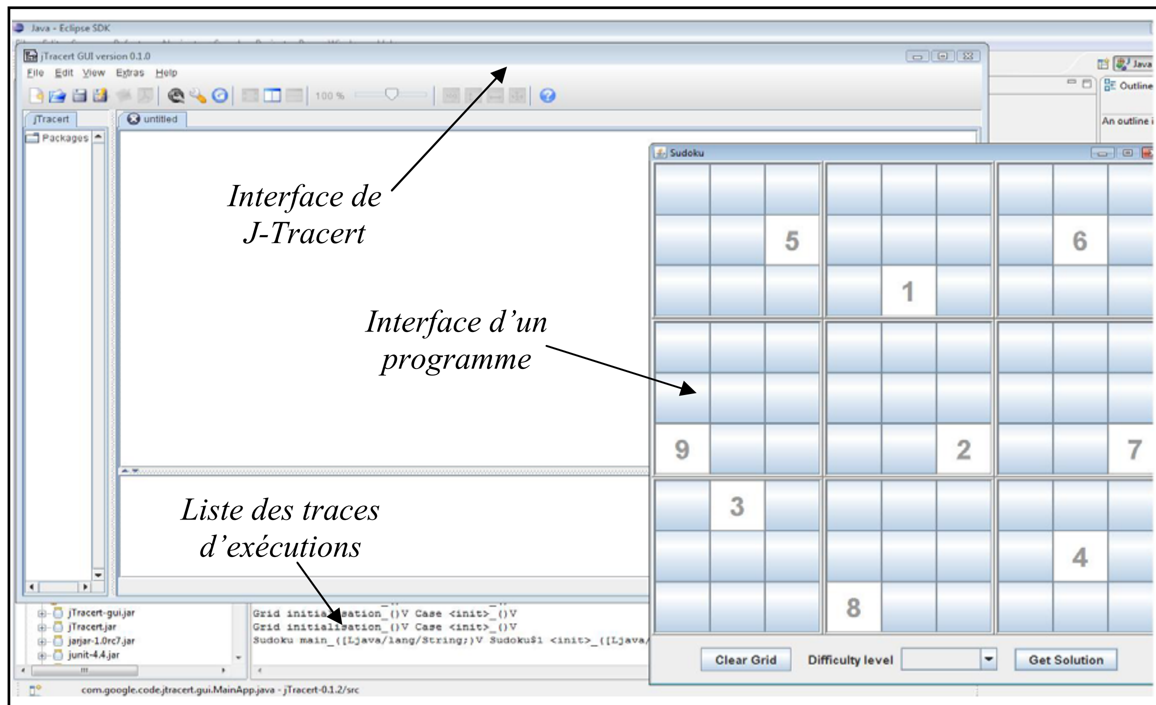


Figure 7. Exemple d'utilisation de *J-Tracert* sur un programme Java.

Une fois que nous avons collecté les traces d'exécutions, nous calculons les différentes valeurs de métriques de couplage sur ces traces. En utilisant la librairie *SSJ*, nous avons implémenté un autre simulateur qui, ayant les valeurs de métriques de couplage de chaque classe, génère automatiquement des histogrammes qui reflètent la distribution de probabilité de métriques de chacune des classes du programme. Notre simulateur simule les entrées du programme en se basant sur des lois de probabilité spécifiées à l'avance. Pour chacun des deux programmes que nous avons utilisés pour valider notre approche, nous avons généré un échantillon de n=1000 exécutions.

Dans les sections qui suivent, nous allons donner certaines observations que nous avons pu déduire à partir de nos résultats, et détailler ensuite deux études de cas en donnant chaque fois, une description du programme, le choix de la métrique de couplage dynamique, ainsi qu'une description du modèle probabiliste des entrées. Nous allons aussi présenter les résultats obtenus pour chacun des deux programmes étudiés et nous

expliquons comment ces résultats facilitent la compréhension des liens de dépendance et l'analyse du rôle des classes dans le programme.

5.3. Cadre expérimental

Dans cette section, nous allons donner le cadre expérimental de nos travaux. Dans ce contexte, nous allons commencer par décrire les deux programmes que nous avons considérés pour les deux études de cas. Ensuite, nous allons présenter les deux métriques de couplage dynamiques que nous avons utilisées pour les deux programmes (une métrique par programme). Nous avons considéré deux métriques de couplage différentes pour donner la possibilité de tester plusieurs métriques, ce qui permet entre autres de généraliser le couplage. Ces deux métriques ont été choisies à partir du papier d'Arisholm *et al.* [3]. Le choix de ces deux métriques est au hasard et n'obéit à aucun critère. Finalement, nous allons décrire pour chaque programme, le modèle probabiliste des entrées.

5.3.1. Description des programmes

5.3.1.1. Cas 1: Cas d'une chaîne de Markov (système d'ascenseurs)

Le premier programme que nous avons considéré est un système qui simule le fonctionnement des ascenseurs dans un édifice. Pour lancer le programme, l'utilisateur doit commencer tout d'abord par définir le système en spécifiant le nombre des ascenseurs et le nombre d'étages. Ensuite, il doit spécifier à chaque nouvelle itération le numéro de l'étage à visiter, ainsi que le numéro de l'ascenseur à utiliser pour ce faire. Une fois que l'utilisateur est devant le système des ascenseurs, il sélectionne une action (*montée* ou bien *descente*). Quand la direction de son mouvement est spécifiée, l'utilisateur choisit l'action *aller à* dès qu'il sera à l'intérieur de la cabine. La figure 8 suivante donne une idée sur l'interface de notre programme.

À chaque nouvelle itération, le programme vérifie que les entrées de l'utilisateur sont valides. En effet, si on note par a' le nombre des ascenseurs dans le système et par e le nombre d'étages à parcourir, alors, il faut vérifier que le numéro de l'étage à visiter ne dépasse pas e et que l'ascenseur demandé existe dans le système (c'est-à-dire, que le

numéro de l'ascenseur à appeler ne dépasse pas *a'*). Bien sûr, il faut que $a', e \in IN^*$ et que $a' \leq e$ (car on suppose que le nombre des ascenseurs ne peut excéder le nombre des étages dans le système). Si ces deux conditions sont satisfaites, le programme passe à vérifier les autres champs saisis. En particulier, si un ascenseur se trouve à un étage i à l'instant (t-1), alors, il n'est pas possible de le choisir de *monter* vers un étage j à l'instant t tel que $i > j$, et vice-versa. Si l'une ou l'autre des conditions précédentes a été violée, le programme arrête l'exécution courante, sinon il continue son exécution pour les prochaines itérations.

Figure 8. Interface du système des ascenseurs.

5.3.1.2. Cas 2: Cas d'un vecteur aléatoire (générateur de grilles de Sudoku)

Le second programme que nous avons considéré génère des grilles de Sudoku [25]. Ce programme prend 10 valeurs en entrée. Ce nombre de valeurs est fixé et est toujours le même quelque soit l'exécution. Pour cela, nous l'avons considéré pour le cas du vecteur aléatoire. Les premières 9 valeurs sont des entiers (de 1 à 9) qui doivent être placés sur une

grille de taille 9x9. Nous avons supposé que chaque cellule possède une position dans la grille qui est numérotée de 1 (en haut à gauche) jusqu'à 81 (en bas à droite). L'utilisateur commence tout d'abord par entrer les valeurs de 1 à 9 successivement sur la grille; le premier clic correspond à la valeur 1, le second clic pour la valeur 2, etc. La 10-ème entrée au programme correspond au niveau de difficulté de la grille de Sudoku, sur une échelle de 1 à 5 où 1 correspond au niveau de difficulté *VERY EASY*, et 5 au niveau de difficulté *VERY HARD*, comme le montre la figure 9 ci-dessous.

En commençant par les 9 valeurs entières entrées par l'utilisateur, le programme se charge de trouver une solution pour la grille en remplissant les 72 cases restantes. Au cas où le programme ne trouve pas de solution, un message d'erreur sera affiché pour indiquer que la grille ne possède pas de solution valide. Dans le cas où une solution est trouvée, le niveau de difficulté sera utilisé pour déterminer le nombre de cases à cacher sur la grille.

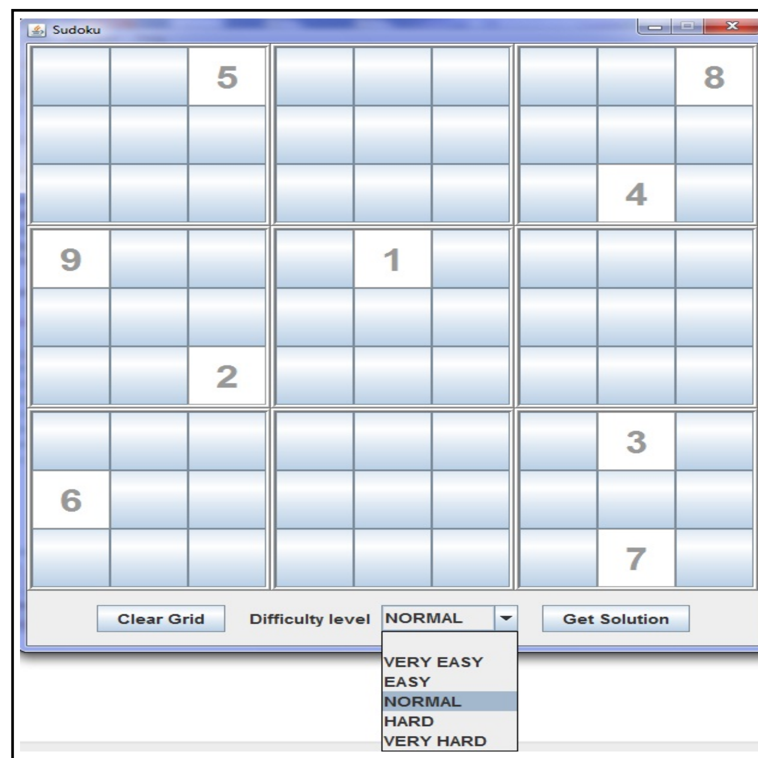


Figure 9. Exemple d'une grille de Sudoku initialisée.

5.3.2. Les métriques de couplage dynamiques

5.3.2.1. Cas 1: Cas d'une chaîne de Markov (système d'ascenseurs)

Pour notre programme d'ascenseurs, nous avons considéré la métrique *IC_CM* (*Import Coupling between Classes and Methods*) extraite des travaux d'Arisholm *et al.* [3]. Cette métrique calcule le nombre d'invocations de méthodes entre deux classes différentes. En effet, supposons que $m_1()$ est une méthode de la classe cl_1, et $m_2()$ est celle de cl_2 ($cl_1 \neq cl_2$). Si $m_1()$ appelle ou envoie un message à $m_2()$, ou bien $m_1()$ utilise un résultat de la méthode $m_2()$, alors, on augmente le couplage des deux classes correspondantes, à savoir cl_1 et cl_2. Ces invocations de méthodes sont capturées au moment de l'exécution du programme, d'où l'aspect dynamique. La figure suivante donne une idée sur le calcul de cette métrique de couplage.

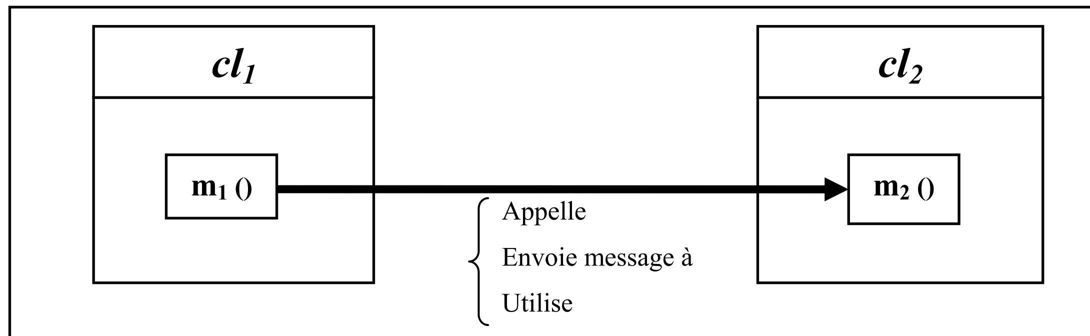


Figure 10. La métrique de couplage dynamique *IC_CM* (cl_1).

5.3.2.2. Cas 2: Cas d'un vecteur aléatoire (générateur de grilles de Sudoku)

Pour notre programme de Sudoku, nous avons considéré la métrique *IC_OM* (*Import Coupling between Objects and Methods*) extraite des travaux d'Arisholm *et al.* [3]. Cette métrique collecte des quadruplets de la forme *(méthode source, classe source, méthode cible, classe cible)* dans un ensemble *M*. La valeur de la métrique pour une certaine classe cl_1 est *IC_OM* (cl_1) = $|M|$.

En effet, $IC_OM(cl_1)$ (comme elle est définie dans [3]) s'intéresse au couplage entre les objets et les méthodes : un objet o_1 (instancié d'une classe cl_1) peut envoyer un message à un autre objet o_2 (d'une autre classe cl_2). Pour effectuer de telle communication, l'objet o_1 utilise une méthode $m_1()$ de sa classe qui communique avec une autre méthode m_2 ($m_2 \neq m_1$ car les auteurs ont exclu le cas des méthodes réflexives) de la classe cl_2 de telle sorte que l'objet o_2 peut récupérer à partir de $m_2()$ le message reçu. Il est à noter que cette métrique ne considère que les directions de couplage sortantes, c'est-à-dire, elle s'intéresse uniquement aux objets actifs qui envoient les messages (et non pas ceux qui reçoivent des messages d'autres objets). Le calcul de cette métrique ressemble à celui de la métrique *IC_CM* et est effectué au moment de l'exécution du programme, sauf que l'entité de mesure ici est l'objet (contrairement à la métrique *IC_CM* dont l'entité de mesure est la classe). La figure suivante permet de résumer cette métrique.

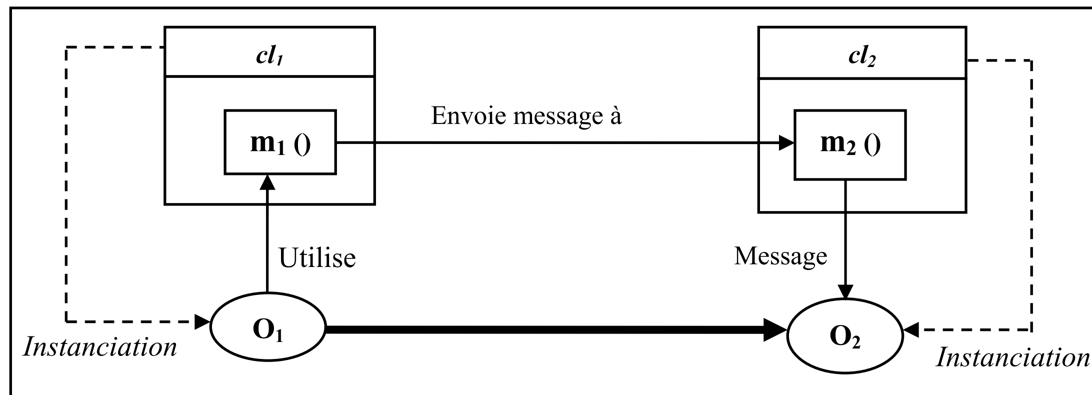


Figure 11. La métrique de couplage dynamique $IC_OM\ (cl_1)$.

5.3.3. Les modèles probabilistes des entrées

5.3.3.1. Cas 1: Cas d'une chaîne de Markov (système d'ascenseurs)

Il n'est pas possible de spécifier d'avance le nombre des entrées pour le cas du programme d'ascenseurs. Nous avons décidé de le présenter par un modèle de chaîne de Markov. On suppose qu'il y a un nombre arbitraire de personnes qui utilisent le système

des ascenseurs chaque jour et que les arrivées sont des variables aléatoires et indépendantes les unes des autres. Les temps entre deux arrivées successives sont i.i.d. (indépendants et identiquement distribués). La loi des inter-arrivées est exponentielle de moyenne $1/\lambda$. Dans notre programme de simulation, nous avons choisi $\lambda=0.5$. Chaque personne est modélisée par une chaîne de Markov qui sera déclenchée au moment où elle entre dans le système et s'arrête quand elle quitte le système ou bien l'une des conditions signalées dans la sous-section 5.3.1.1 précédente sera violée. Ces chaînes de Markov sont supposées être indépendantes les unes des autres. Pour chaque sujet, nous avons affecté une matrice de transition qui définit la probabilité de passer d'un étage à un autre ou bien rester dans l'étage courant. Nous avons également considéré que ces chaînes de Markov sont *homogènes*, c'est-à-dire, la matrice de transition est statique et ne change pas au cours du temps.

Notons par $\{S_j, j \geq 0\}$ la chaîne de Markov d'un sujet donné. Dans chaque nouvel état de la chaîne, il s'agit de spécifier le type de mouvement (*montée/descente*), le numéro de l'ascenseur à utiliser, ainsi que le numéro de l'étage à visiter. En se référant à la matrice de transition, on en déduit que chaque état de la chaîne dépend des états qui le précèdent, car le choix d'un mouvement doit dépendre généralement des types de mouvements précédents. Le temps d'arrêt τ est défini par la violation d'une des conditions précédemment mentionnées, ou bien que le sujet quitte le système après un certain temps τ. Ce dernier est généré aléatoirement par notre simulateur.

5.3.3.2. Cas 2: Cas d'un vecteur aléatoire (générateur de grilles de Sudoku)

En respectant les notations de notre cadre de travail, on peut modéliser le vecteur des entrées de notre programme de Sudoku par $\mathbf{X} = (X_1, \ldots, X_{10})$ où $X_1, \ldots, X_9 \in \{1, 2, \ldots, 81\}$ telle que $X_i \neq X_j$ chaque fois que $i \neq j$, et $X_{10} \in \{1, 2, \ldots, 5\}$ désigne le niveau de difficulté. Notons par N toutes les possibilités d'exécutions du programme. Alors, $N = 81 \times 80 \times \ldots \times 73 \times 5 \approx 2^{59}$ possibilités.

Pour pallier à cette explosion combinatoire, nous avons décidé d'utiliser un échantillon de 1000 exécutions représentatives pour simuler les entrées du programme (vu que le

nombre de toutes les possibilités d'exécutions est énorme). Dans notre modèle, nous avons supposé que toutes les valeurs d'entrée du vecteur **X** possèdent la même probabilité, c'est-à-dire, $1/N$. Ceci signifie que les positions des 9 premières entrées du programme seront choisies *aléatoirement*, *uniformément* et de façon *indépendante* les unes des autres. De ce fait, $X_i \sim$ Uniforme (1, 81), $\forall\ i \in \{1, \ldots, 9\}$. Le niveau de difficulté (10-ème entrée au programme) est également choisi de façon *aléatoire*, *uniforme* et *indépendamment* des autres entrées du programme. La distribution de probabilité du vecteur d'entrées **X** sera donc $F(\mathbf{X}) = F(X_1, \ldots, X_{10}) = F_1(x_1) \ldots F_{10}(x_{10})$ où $F_j(x_j) = \mathbf{P}(X_j \leq x_j)$ est la j-ème fonction de distribution marginale ($1 \leq j \leq 10$). On peut raisonner de la même manière en considérant que les 5 niveaux de difficulté ont la même chance d'être choisis, et par conséquent, X_{10} aura une distribution uniforme discrète sur $\{1, \ldots, 5\}$.

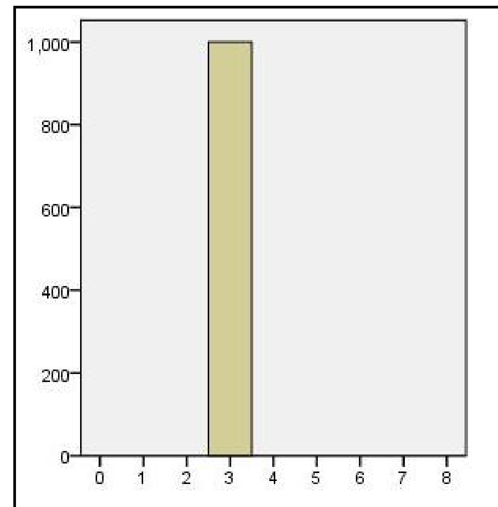
5.4. Observations

Pour chacun des deux programmes que nous avons considérés, et pour chaque classe, nous avons généré un histogramme qui décrit la distribution de probabilité des métriques de couplage dynamiques obtenues par l'échantillon d'exécutions. Dans la partie Annexe II et Annexe III, nous donnons la liste de tous ces histogrammes.

Nous avons constaté que certains types de distributions ont apparu plusieurs fois dans chacun des deux programmes. Ces observations nous ont menées à détailler ces histogrammes afin de comprendre pourquoi ces allures ont apparu souvent et de façon fréquente. Nous avons pu distinguer en total quatre *patrons de dépendance*. Dans les sous-sections qui suivent, nous allons présenter chaque fois un patron de dépendance que nous avons observé, et nous essayons de lui donner une interprétation en fonction des observations que nous avons pu déduire à partir des traces d'exécutions de nos deux programmes.

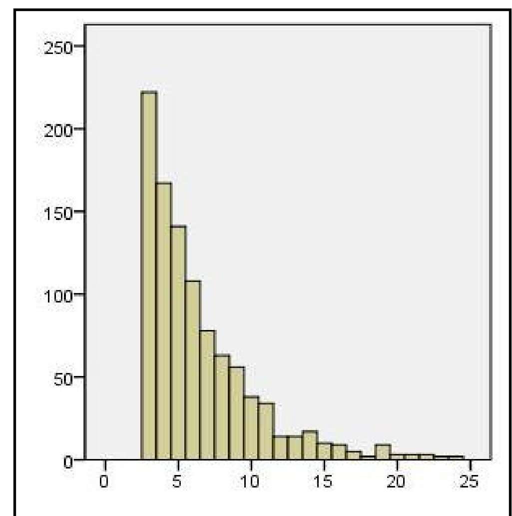
5.4.1. Patron de dépendance 1 : Chaîne de montage *(Assembly-Chain Worker)*

Nous avons observé que cette situation se produit quand une classe d'un programme effectue presque toujours la même variante (un scénario unique), c'est-à-dire la même fonctionnalité sans presque aucun traitement exceptionnel dans le programme. La distribution de probabilité correspondante prend la forme d'une barre verticale (avec possiblement une ou deux petites barres adjacentes). La valeur de couplage la plus fréquente pour cette classe donne une idée sur le nombre des interactions dont on a besoin pour réaliser le service demandé par cette classe.



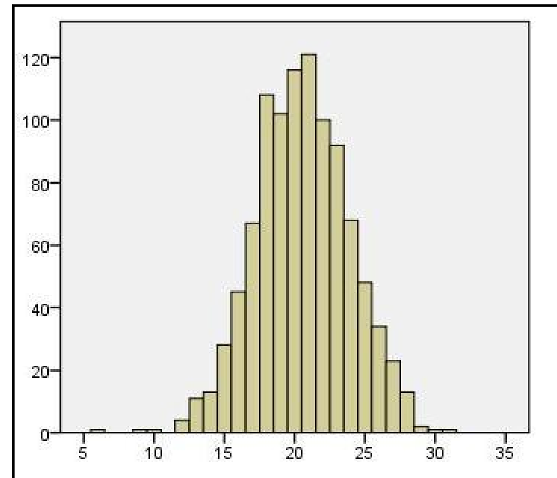
5.4.2. Patron de dépendance 2 : Employé *(Clerk)*

Ceci est une variante du premier patron de dépendance observé où le comportement unique est étendu de façon itérative pour la prise en compte de traitements de plus en plus exceptionnels. Nous avons constaté que le plus souvent, ce patron correspond à une distribution ayant une allure exponentielle et peut être représenté comme le montre la figure ci-contre. Dans ce cas, le traitement normal de la classe est le plus fréquent (c'est-à-dire, les scénarios standards que cette classe faisait la plupart du temps) et qui correspond généralement aux petites valeurs de couplage. Mais on trouve également d'autres traitements de plus en plus exceptionnels chaque fois que les valeurs de couplage de la classe augmentent. La fréquence des observations de scénarios d'exécutions diminue progressivement chaque fois que les



couplage augmentent, ce qui permet d'accumuler des comportements de plus en plus rares de la classe correspondante.

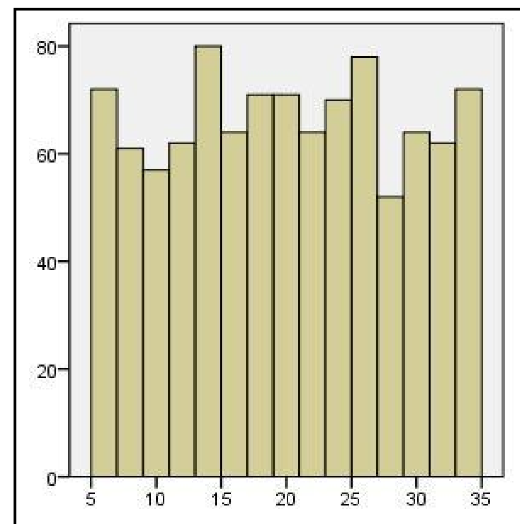
5.4.3. Patron de dépendance 3 : Soldat *(Soldier)*



Ceci est une variante des deux patrons de dépendance précédemment décrits. Nous avons observé que la classe correspondante effectue un traitement spécifique avec des scénarios plus ou moins rares. En effet, cette classe est censée effectuer un traitement minimal dans le cas où une pré-condition n'a pas été satisfaite, et des traitements standards qui correspondent généralement aux services dont la classe est supposée satisfaire, ainsi que des traitements de plus en plus rares qui décrivent des scénarios particuliers du programme. Nous avons observé que la distribution de probabilité de métriques de ce genre de classes correspond le plus souvent à une allure normale.

5.4.4. Patron de dépendance 4 : Secrétaire *(Secretary)*

Nous avons observé que les classes dont la distribution de probabilité de métriques présente ce genre de patrons sont généralement impliquées dans plusieurs fonctionnalités dans le système, avec des équiprobabilités. C'est



le cas des classes utilitaires dans le programme. La distribution de ces classes possède une allure uniforme, c'est-à-dire, les valeurs de couplages correspondantes aux différentes fonctionnalités du programme ont presque toujours la même fréquence dans la plupart des possibilités d'exécutions du programme. Ces types de classes sont souvent utilisés pour servir les autres classes du programme et/ou les utiliser pour

exécuter ses différentes fonctionnalités.

D'après les histogrammes que nous avons pu obtenir dans le cas des deux programmes, nous avons observé que si une classe a une distribution à allure exponentielle, alors, elle accumule le plus souvent les mêmes valeurs de couplage. Chaque fois que ces valeurs de couplage augmentent, le nombre des observations diminue progressivement. Dans ce cas, nous avons trouvé qu'une telle classe intervient le plus souvent dans des scénarios standards, mais elle est également impliquée dans des scénarios rares.

D'autre part, il est bien connu en statistique que pour une distribution normale, environ 60% des valeurs observées sont situées dans l'intervalle $[\mu - \sigma, \mu + \sigma]$ où μ et σ sont respectivement la moyenne et l'écart-type de cette distribution de probabilité, et environ 95% de ces valeurs sont situées dans l'intervalle $[\mu - 2\sigma, \mu + 2\sigma]$ [42]. Ainsi, si une classe possède une distribution à allure normale, alors, nous avons constaté (d'après nos observations) qu'elle présente dans la plupart des cas le même comportement. De telle classe peut également intervenir dans les scénarios rares, mais avec de faibles probabilités (par comparaison aux autres classes dont la distribution a une allure exponentielle). On peut même penser à examiner *le coefficient d'aplatissement* de cette distribution (« peakedness factor » en anglais, ou encore « kurtosis »). Si la valeur de ce coefficient d'aplatissement est faible, on peut conclure que les valeurs de couplage sont proches les unes des autres. Dans ce cas, nous avons observé que les scénarios d'exécutions de cette classe engendrent des comportements similaires du programme, et vice versa.

La distribution uniforme, quant à elle, est différente des deux autres distributions (exponentielle et normale). En effet, une classe dont la distribution de probabilité de métriques de couplage a une allure uniforme possède la propriété que la plupart des valeurs de couplage apparaissent de façon équiprobable. Dans ce cas, nous avons constaté que de telle classe interagit généralement avec le reste du système avec presque la même probabilité. En examinant ces genres de classes, nous avons trouvé qu'elles contiennent le plus souvent des méthodes utiles afin de servir les autres classes du programme.

Dans ce qui suit, nous allons donner plus de détails sur les résultats que nous avons observés, et nous allons expliquer comment ces patrons de dépendance peuvent nous aider à comprendre les liens de dépendance d'une classe avec les autres classes du programme, ainsi que son rôle dans le système.

5.5. Résultats et interprétations

5.5.1. Cas 1: Cas d'une chaîne de Markov (système d'ascenseurs)

5.5.1.1. Analyse de dépendance

Nous avons lancé notre système d'ascenseurs sur un échantillon de 1000 exécutions et nous avons examiné les histogrammes obtenus qui correspondent aux distributions de probabilité de métriques de chaque classe du programme. Nous avons pu identifier quatre *employés* (histogrammes à allure exponentielle), deux *soldats* (histogrammes à allure normale) et un *secrétaire* (histogramme à allure uniforme). La 8 ème classe ne correspond pas à un patron de dépendance particulier, mais elle peut être vue comme une extension du patron *chaîne de montage* (histogramme avec une barre). Dans ce qui suit, nous allons détailler chaque fois un patron de dépendance pour une classe particulière du programme.

Le patron le plus fréquent que nous avons identifié ici est l'*employé* (*Clerk*). Ceci n'est pas surprenant, car la plupart des classes interviennent à la fois dans des scénarios standards (pour faire une ou plusieurs fonctionnalités particulières), ainsi que dans des scénarios de plus en plus rares qui prennent en compte des situations exceptionnelles du programme.
La classe *Elevator* par exemple illustre ce type de patrons. En effet, les valeurs de couplage les plus fréquentes correspondent à un comportement régulier de la classe par des interactions avec les autres classes du programme comme *Floor* et *ArrivalSensor*. Il s'agit d'appeler les méthodes standards de la classe *Elevator* décrivant l'utilisation régulière des systèmes d'ascenseurs : un utilisateur se trouvant devant les portes d'un ascenseur, sélectionne son choix soit pour une *montée* (*Floor requestUp() Elevator getBestElevator()*) ou bien une *descente* (*Floor requestDown() Elevator getBestElevator()*). Une fois que les portes de l'ascenseur s'ouvrent (*Elevator openDoor() ElevatorGroup elevatorDisplay()*), il

entre à l'intérieur de la cabine et choisit l'étage qu'il veut visiter (*Elevator getNextDestination() Floor selectFloor()*). Les portes se ferment (*Elevator closeDoor() ElevatorGroup elevatorDisplay()*) et l'ascenseur bouge vers l'étage destinataire (*Elevator moveElevator() ElevatorGroup elevatorDisplay() – Elevator motorMoveUp() ElevatorControl <init> – Elevator motorMoveDown() ElevatorControl <init> – Elevator motorStop() ElevatorControl <init>*). Ces fonctionnalités décrivent les actions de base d'utilisation d'un système d'ascenseurs.

Chaque fois que les valeurs de couplage augmentent, la classe *Elevator* a tendance à effectuer des traitements de plus en plus exceptionnels. Par exemple, il arrive (plus ou moins souvent) lors du mouvement d'un ascenseur entre deux étages qu'on demande d'ajouter un arrêt lorsqu'un appel en cours de chemin survient. Dans ce cas, la classe *Elevator* doit gérer ces appels intermédiaires, par exemple :

Elevator addStop() Floor getFloorID() - ElevatorControl requestStop() Elevator addStop() Floor requestUp() Elevator addStop() - Floor requestDown() Elevator addStop()

Dans le cas où tous les ascenseurs sont occupés et qu'il y a plusieurs requêtes en attente de service, le programme doit pouvoir gérer cette situation en créant des files d'attente contenant les requêtes des clients en attente de service, ainsi que l'utilisation des objets de type *Thread* pour gérer la concurrence d'accès aux objets *Elevator* (lancer des threads, les mettre en veille, les arrêter, …). Ces cas surviennent rarement avec une faible probabilité. Des exemples d'appels observés pour ce cas sont les suivants :

ElevatorInterface getFromList() Elevator getFloorID()

Elevator run() ElevatorGroup elevatorDisplay()

ElevatorGroup startGroup() Elevator turnOn()

ElevatorGroup stopGroup() Elevator turnOff()

ElevatorGroup stopGroup() Elevator removeAllElevators()

Comme nous venons de le mentionner ci-haut, le patron *Soldat* a été trouvé deux fois dans notre programme. Un cas intéressant de ce patron est celui de la classe *ElevatorGroup*. Cette classe est responsable de la création des objets de type *Elevator*, ainsi que d'assurer la communication entre ces différents objets. Du point de vue comportement, cette classe se

comporte de façon similaire que la classe *Elevator*, c'est-à-dire, elle présente des comportements réguliers fréquents et des comportements spécifiques (exceptionnels) qui correspondent aux grandes valeurs de couplage observées. En effet, lors de son comportement standard, la classe *ElevatorGroup* utilise pratiquement les mêmes appels que ceux de la classe *Elevator* pour modéliser un schéma d'utilisation ordinaire des systèmes d'ascenseurs (c'est-à-dire, sélectionner un choix de montée ou de descente, ouvrir les portes de l'ascenseur, choisir un étage à visiter, fermer les portes de l'ascenseur,…). Parmi les appels observés dans ces scénarios, on en cite :

ElevatorGroup <init>   ArrivalSensor <init>

ElevatorInterface motorMoveUp()   ElevatorGroup getGroup()

ElevatorInterface motorMoveDown()   ElevatorGroup getGroup()

Elevator openDoor()   ElevatorGroup elevatorDisplay ()

ElevatorGroup motorMoving()   FloorInterface stopAtThisFloor()

Elevator closeDoor()   ElevatorGroup elevatorDisplay ()

ElevatorInterface requestStop()   ElevatorGroup elevatorDisplay()

Le comportement exceptionnel, quant à lui, correspond presque à celui de la classe *Elevator*, c'est-à-dire, tous les ascenseurs sont occupés et qu'il y a des requêtes à servir; dans ce cas, la classe *ElevatorGroup* se charge de gérer une file d'attente des requêtes en attente de service, ainsi que la définition des objets de type *Thread* pour gérer la concurrence d'accès aux objets ascenseurs. En effet, cette classe utilise la méthode *getFromList()* pour créer une liste des requêtes en attente de service. Dans ces scénarios, la classe *ElevatorGroup* communique avec différentes classes telles que *Elevator*, *ElevatorControl*, *ElevatorInterface*. Éventuellement, pour des scénarios de plus en plus exceptionnels, la classe *ElevatorGroup* commence à intervenir dans les traitements encore plus rares du programme et plus précisément, l'utilisation des objets *Threads* pour empêcher la concurrence d'accès aux ascenseurs en mouvement. Dans ce cas-là, la classe *ElevatorGroup* utilise sa méthode *startGroup()* pour communiquer surtout avec la classe *ElevatorControl*, *ElevatorInterface* et *Floor*.

Parmi les appels qu'on trouve dans ces cas, on peut en citer :

ElevatorInterface getFromList() *ElevatorGroup elevatorDisplay()*
ElevatorGroup startGroup() *Floor getSensor()*
ElevatorGroup startGroup() *ElevatorControl openDoor()*
ElevatorGroup startGroup() *ElevatorControl closeDoor()*
ElevatorGroup stopGroup() *ElevatorControl requestStop()*

Cependant, et contrairement à la classe *Elevator*, la classe *ElevatorGroup* présente parfois un traitement minimal dans lequel elle interagit peu fréquemment avec les autres classes du programme. Ceci se produit quand le nombre des ascenseurs et/ou le nombre des étages donnés en entrée viole certaines conditions, ce qui implique l'arrêt de l'exécution du programme. En effet, lors de chaque nouvelle exécution du programme, on doit définir le système en précisant le nombre d'ascenseurs et le nombre d'étages. Sauf que le programme exige certaines conditions sur ces entrées. Par exemple, il faut que le nombre d'ascenseurs et le nombre d'étages soient deux entiers strictement positifs et que le nombre d'ascenseurs n'excédent pas le nombre d'étages. Donc, si ces deux entrées saisies satisfont ces conditions alors, le programme termine son exécution de façon ordinaire, sinon, il affiche un message d'erreur et l'exécution est terminée. Dans ce cas, le constructeur de la classe *ElevatorGroup* utilise une instance de la classe *ElevatorInterface* pour vérifier ces tests. Une autre exception qui pourra arriver au début d'une exécution : initialement, tous les ascenseurs sont situés au rez-de-chaussée (étage 0), donc, pour chacun des ascenseurs du système, il faut que la première requête soit de type *requestUp()*, sinon, il y'aura une exception où le système affiche le message d'erreur « Wrong request !! » lors de l'appel de la méthode *motorMoveDown()* de la classe *ElevatorGroup*, et l'exécution du programme se termine.

Le troisième patron que nous avons identifié est de type *secrétaire*. Nous avons juste trouvé un seul exemple de ce type de patron dans notre programme. Ce patron correspond à la classe *FloorContol*. En effet, cette dernière interagit avec la majorité des autres classes du programme en différentes situations. Elle interagit avec la classe *FloorInterface* pour savoir si un ascenseur doit s'arrêter dans un étage donné ou non (*FloorInterface stopAthThisFloor()* *FloorControl <init>*), avec la classe *Floor* pour savoir si les appels

ont été correctement émis et qu'ils ont été servis (*FloorControl stopAtThisFloor() Floor requestUpMade() - FloorControl stopAtThisFloor() Floor requestDownMade() – FloorControl stopAtThisFloor() Floor requestUpServiced() - FloorControl stopAtThisFloor() Floor requestDownServiced()*). Elle interagit avec la classe *ArrivalSensor* pour choisir l'ascenseur le plus proche (*FloorControl <init> ArrivalSensor closestElevator()*), elle interagit également avec la classe *ElevatorGroup* pour demander à un ascenseur de bouger suite à la réception d'une requête de montée ou de descente (*ElevatorGroup motorMoving() FloorControl requestUp() - ElevatorGroup motorMoving() FloorControl requestDoxn()*). Finalement, la classe *FloorControl* communique avec la classe *Elevator* pour s'assurer que le mouvement demandé peut être réalisé; il est interdit de choisir un mouvement de type *descente* quand l'utilisateur est au rez-de-chaussée, ni un mouvement de type *montée* quand l'utilisateur est au dernier étage (*FloorControl stopAtThisFloor() Elevator getFloorID() – FloorControl requestUp() Elevator getDirection() - FloorControl requestDown() Elevator getDirection()*).

Le quatrième et dernier patron que nous avons pu identifier est de type *chaîne de montage* (une simple barre). Aucune classe dans notre programme n'avait exactement cette allure, sauf que l'histogramme de la classe *ArrivalSensor* peut être vu comme une extension de ce patron. En effet, cette classe faisait le plus souvent le même traitement dans le programme. Elle est responsable de choisir l'ascenseur le plus proche, l'appeler chaque fois qu'une nouvelle requête survient et l'arrêter dans l'étage destinataire (*FloorControl <init> ArrivalSensor closestElevator() – ArrivalSensor stopAtThisFloor() Elevator selectElevator() – FloorInterface getFloor() ArrivalSensor getTheFloor() – ArrivalSensor stopAtThisFloor() Elevator notifyNewFloor()*). Ces appels se font régulièrement presque dans chaque nouvelle exécution. C'est pour cette raison que cette classe se comporte de la même façon dans presque toutes les exécutions du programme.

5.5.1.2. Compréhension de dépendance

Dans la sous-section qui précède, nous avons analysé nos histogrammes en termes de dépendance et de comportement. Ces histogrammes reflètent la distribution de probabilité

de métriques de chaque classe du programme. Il reste alors à expliquer comment ces histogrammes peuvent nous aider à comprendre les liens entre les dépendances et le rôle d'une classe dans un programme.

Pour y parvenir, nous allons commencer par présenter les résultats statistiques que nous avons obtenus, à savoir les coefficients de similarité internes (normalisés) et externes, ainsi que l'ensemble des régions d'exécution obtenues pour chaque classe. Ces résultats sont présentés par le tableau 4 ci-dessous. Pour chaque classe du programme, nous avons donné les valeurs minimale, moyenne et maximale observées pour le coefficient de similarité interne, la liste des blocs qui constituent chaque région d'exécution, ainsi que la moyenne de similarité externe entre les blocs d'une région d'exécution donnée.

Nom de la classe	***NIS***	***Liste des régions d'exécutions***	***ES***
ArrivalSensor	Minimum : 80 % Maximum : 86.95 % Moyenne : 83.72 %	$R_1 = \{B_3, B_4, B_5\}$	67.5 %
Elevator	Minimum : 79.12 % Maximum : 100 % Moyenne : 95.24 %	$R_1 = \{B_9, B_{10}, B_{11}, B_{12}\}$ $R_2 = \{B_{13}, B_{14}, B_{15}, B_{16}, B_{17}\}$ $R_3 = \{B_{18}, B_{19}, B_{21}, B_{23}, B_{24}\}$	82.27 % 92.31% 88.07 %
ElevatorGroup	Minimum : 74.58 % Maximum : 100 % Moyenne : 95.67 %	$R_1 = \{B_9, B_{10}, B_{11}, B_{12}\}$ $R_2 = \{B_{13}, B_{14}, B_{15}, B_{16}, B_{17}, B_{18}, B_{20}, B_{21}\}$ $R_3 = \{B_{23}, B_{24}, B_{25}, B_{26}, B_{29}\}$	75.24 % 79.44 % 87.94 %
ElevatorControl	Minimum : 84.21 % Maximum : 100 % Moyenne : 93.90 %	$R_1 = \{B_4, B_5, B_6, B_7\}$ $R_2 = \{B_8, B_9, B_{10}, B_{11}, B_{12}, B_{13}, B_{14}\}$ $R_3 = \{B_{15}, B_{16}\}$	67.94 % 82.98% 100 %
ElevatorInterface	Minimum : 89.55 % Maximum : 100 % Moyenne : 96.57 %	$R_1 = \{B_{10}, B_{12}, B_{13}, B_{14}\}$ $R_2 = \{B_{15}, B_{16}, B_{18}, B_{19}, B_{20}, B_{22}, B_{23}, B_{24}\}$ $R_3 = \{B_{25}, B_{26}, B_{27}, B_{28}\}$	86.75 % 81.38 % 92.67 %
Floor	Minimum : 94.12 % Maximum : 100 % Moyenne : 98.11 %	$R_1 = \{B_{12}, B_{13}, B_{14}\}$ $R_2 = \{B_{15}, B_{16}, B_{17}, B_{19}, B_{20}\}$ $R_3 = \{B_{21}, B_{23}, B_{25}, B_{27}\}$	89.01 % 91.69 % 100 %
FloorControl	Minimum : 88.89 % Maximum : 100 % Moyenne : 96.24 %	$R_1 = \{B_6, B_8, B_{10}\}$ $R_2 = \{B_{12}, B_{14}, B_{16}\}$ $R_3 = \{B_{18}, B_{20}, B_{22}\}$ $R_4 = \{B_{24}, B_{26}, B_{28}\}$	67.5 % 90 % 92.31 % 100 %
FloorInterface	Minimum : 72.73 % Maximum : 100 % Moyenne : 92.99 %	$R_1 = \{B_2, B_3, B_4\}$ $R_2 = \{B_5, B_6, B_7, B_8\}$ $R_3 = \{B_9, B_{11}\}$	58.34 % 82.22 % 100 %

Tableau 4. Quelques statistiques sur les classes du système des ascenseurs.

En regardant ces résultats, nous constatons que les coefficients de similarité internes (normalisés) des blocs d'exécutions varient de 72.73% à 100% pour toutes les classes du programme. Rappelons d'après notre définition de ce coefficient que *IS* mesure le pourcentage des exécutions typiques à l'intérieur d'un bloc d'exécution, ou encore la diversité des appels entre les exécutions. La moyenne totale de ces coefficients pour tous les blocs d'exécution du programme est égale à 94.05%, ce qui signifie que dans chaque bloc, les classes du programme se comportent presque de la même façon puisque les exécutions sont typiquement similaires (du point de vue des appels de classes et de méthodes). Ceci implique entre autres que le rôle de la classe et ses les liens de dépendance ne changent presque pas d'une exécution à une autre à l'intérieur d'un même bloc. Ceci est important, car en se basant sur ces observations, il est possible de *généraliser* le comportement de notre programme ainsi que les liens de dépendance, non seulement entre les classes, mais aussi entre les éléments d'une même classe d'un programme. Nous avons également détaillé ces résultats pour chaque classe du programme. Dans le tableau suivant, nous donnons un aperçu de ces statistiques pour le cas de la classe *Elevator*.

Liste des blocs d'exécutions	***NIS***	***ES***	***Liste des régions d'exécutions***
B_9 B_{10} B_{11} B_{12}	92.31 % 93.02 % 93.62 % 94.12 %	$ES\ (B_9, B_{10}) = 90$ % $ES\ (B_{10}, B_{11}) = 81.82$ % $ES\ (B_{11}, B_{12}) = 75$ % $ES\ (B_{12}, B_{13}) = 15.38$ %	$R_1 = \{B_9, B_{10}, B_{11}, B_{12}\}$
B_{13} B_{14} B_{15} B_{16} B_{17}	94.55 % 100 % 100 % 100 % 100 %	$ES\ (B_{13}, B_{14}) = 100$ % $ES\ (B_{14}, B_{15}) = 100$ % $ES\ (B_{15}, B_{16}) = 84.62$ % $ES\ (B_{16}, B_{17}) = 84.62$ % $ES\ (B_{17}, B_{18}) = 50$ %	$R_2 = \{B_{13}, B_{14}, B_{15}, B_{16}, B_{17}\}$
B_{18} B_{19} B_{21} B_{23} B_{24}	79.12 % 96.20 % 96.55 % 96.84 % 96.97 %	$ES\ (B_{18}, B_{19}) = 100$ % $ES\ (B_{19}, B_{21}) = 90.48$ % $ES\ (B_{21}, B_{23}) = 82.61$ % $ES\ (B_{23}, B_{24}) = 79.17$ %	$R_3 = \{B_{18}, B_{19}, B_{21}, B_{23}, B_{24}\}$

Tableau 5. Quelques statistiques sur la classe *Elevator*.

Dans ces résultats, nous avons choisi nos valeurs seuils comme suit :
Internal_Threshold =0.6, *Relative_Frequency_Threshold* =0.5 et *External_Threshold* =0.6.
On constate une *chute* des valeurs de similarité externe (*ES*) entre les blocs (B_{12}, B_{13}) et entre les blocs (B_{17}, B_{18}) ce qui nous a permis de faire la distinction entre les régions d'exécutions. En effet, cette chute implique que les blocs B_{12} et B_{13} ne partagent pas le même comportement, donc, ils ne peuvent être regroupés dans une même région d'exécution. De même pour les deux blocs B_{17} et B_{18}.

La figure 12 montre la liste des régions d'exécutions correspondantes à la classe *Elevator* de notre programme. Le comportement général de cette classe peut être résumé en trois régions d'exécutions différentes. Nous avons analysé les appels générés dans chacune de ces trois régions et nous avons trouvé une grande analogie entre ces résultats avec notre définition du patron de dépendance *Employé*, car nous avons pu distinguer facilement les trois régions dans la classe *Elevator*, conformément avec notre patron. Les appels observés dans ces trois régions sont conformes avec notre interprétation des patrons de dépendance. En effet, la première région (R_1) définit le comportement général de la classe (à environ 66% des exécutions totales), c'est-à-dire, son traitement régulier. La seconde région (R_2) décrit un comportement plus spécifique de la classe relatif à des traitements conditionnels et des scénarios particuliers qui ne se produisent pas tout le temps. Quant à la troisième région d'exécution (R_3), elle correspond bien à des traitements exceptionnels et des scénarios rares de la classe. Nous avons déjà fourni les détails de ces traces d'exécutions dans la sous- section 5.5.1.1 précédente du présent chapitre.

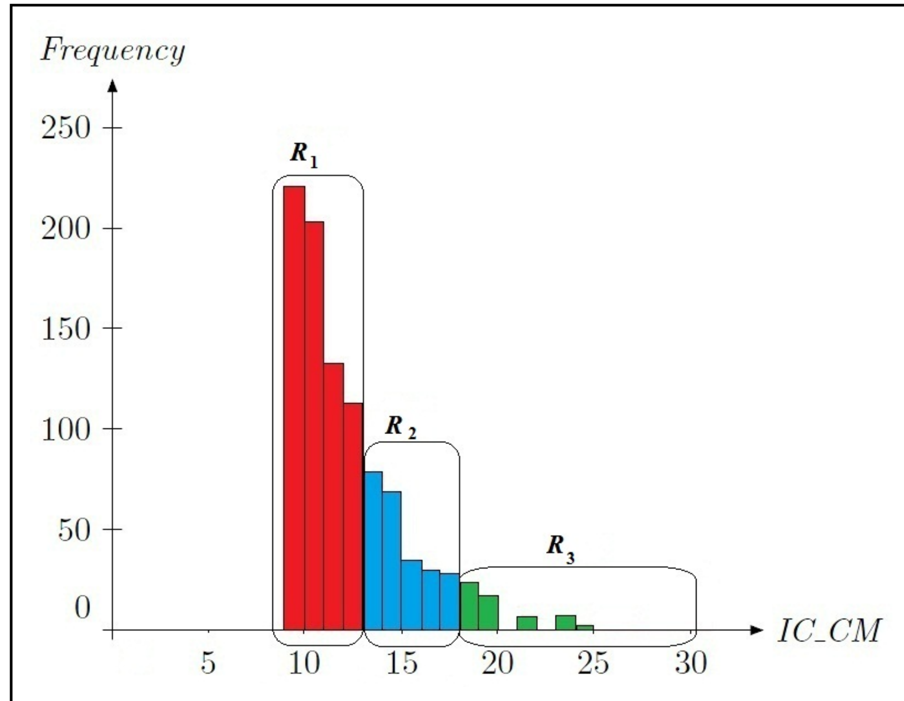


Figure 12. Les régions d'exécutions pour la classe *Elevator*.

Dans la figure 12 ci-dessus, on observe que certains blocs ont été omis de R_3, par exemple, les blocs B_{20}, B_{22}, B_{25}, B_{26} et B_{29}. En effet, lors du calcul du coefficient de la similarité interne de ces blocs, nous avons trouvé que ce coefficient est en dessous de notre valeur seuil *InternalThreshold* que nous avons fixée. De ce fait, de tels blocs ne définissent pas un comportement spécifique, c'est pour cette raison que nous ne les avons pas pris en compte lors du calcul des coefficients de similarité externe, ainsi que lors de l'identification des régions d'exécutions.

Une autre observation intéressante que nous avons trouvée est liée au patron de dépendance de type *Soldat*. En effet, comme nous l'avons mentionné avant, pour une distribution normale, environ 68% des valeurs sont localisées entre (μ - σ) et (μ + σ) où μ et σ désignent respectivement la moyenne et l'écart-type de la distribution [42], comme le montre la figure suivante extraite de [26].

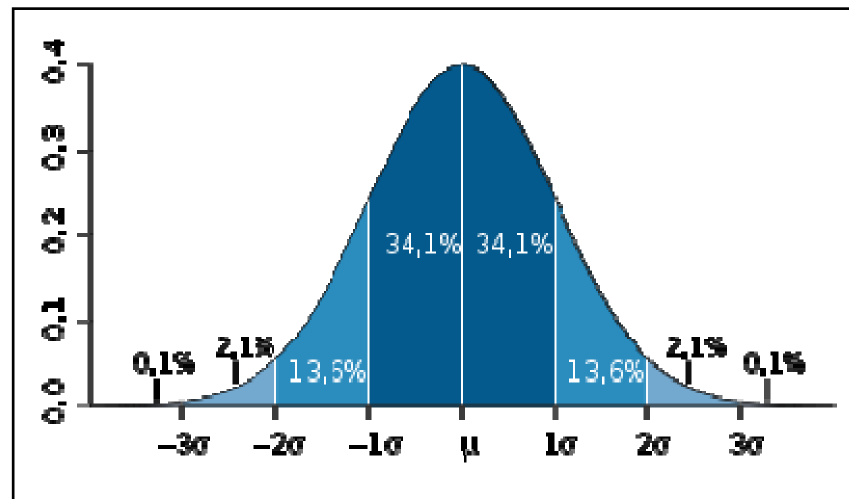


Figure 13. Proportions et intervalles des valeurs pour une distribution normale.

Nous avons constaté que pour les deux patrons de dépendance *Soldat* que nous avons identifiés pour notre programme, la seconde région d'exécutions correspond presque exactement à la plage de valeurs [μ' – σ', μ' + σ'] où μ' et σ' désignent respectivement la moyenne et l'écart-type de la distribution en question. A titre d'exemple, si on considère le cas de la classe *ElevatorGroup* (conférer partie Annexe II), on trouve que la moyenne de ses observations est μ' = 16.68 et son écart-type est σ' = 4.00. Par ailleurs, on s'attend à ce que la plage de valeurs [μ' – σ', μ' + σ'] = [12.67, 20.68] représente environ 68% des valeurs observées. Bien évidemment, d'après les résultats du tableau 4, on trouve que la région d'exécutions R_2 contient toutes les valeurs de couplage qui varient entre 13 et 21 et que les fréquences d'apparition de ces valeurs constituent environ 71% de toutes les fréquences d'exécutions du programme. Ces observations coïncident avec l'interprétation de nos patrons de dépendance. La figure suivante donne une idée sur la situation de la région d'exécutions R_2 par rapport à la plage de valeurs [μ' – σ', μ' + σ'].

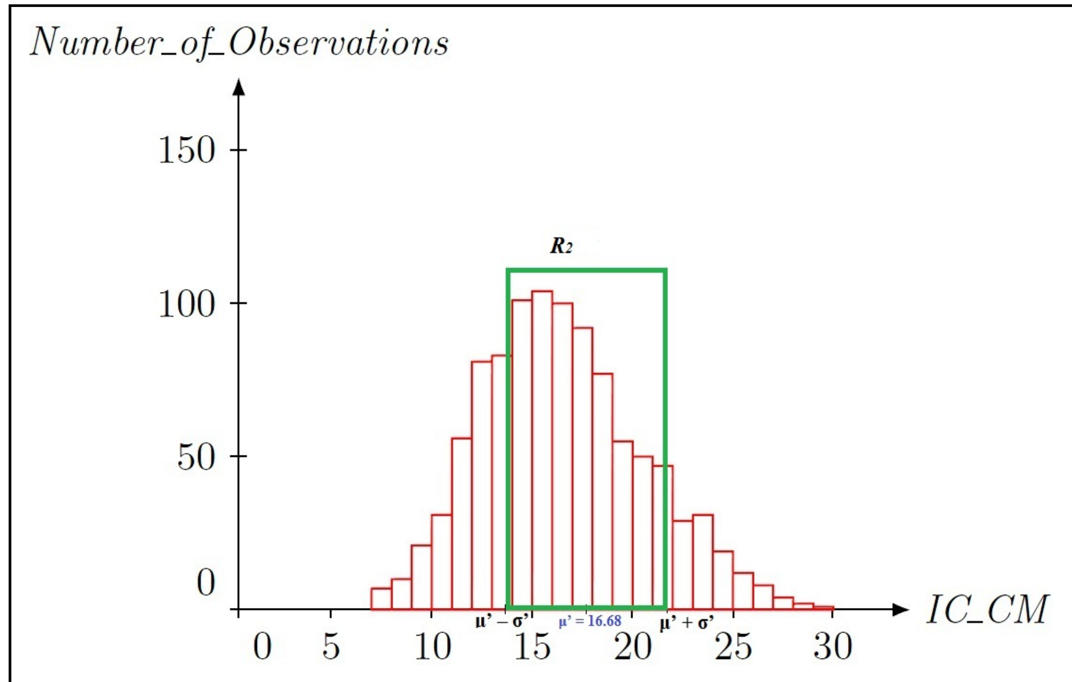


Figure 14. Situation de la région R_2 par rapport à la plage de valeurs [μ' – σ', μ' + σ'] pour la classe *ElevatorGroup*.

5.5.2. Cas 2: Cas d'un vecteur aléatoire (générateur de grilles de Sudoku)

5.5.2.1. Analyse de dépendance

Comme pour le cas du premier programme (le système des ascenseurs), nous avons exécuté le programme du générateur de grilles de Sudoku sur un échantillon de 1000 exécutions. Nous avons pu identifier quatre *chaînes de montage*, trois *employés*, deux *soldats* et deux *secrétaires*. Les deux autres classes restantes n'ont pas de forme qui correspond à un patron particulier. Dans la partie Annexe III, nous fournissons la liste complète de tous les histogrammes que nous avons obtenus et qui correspondent aux 13 classes de notre programme. Comme pour le cas de la chaîne de Markov, nous allons donner chaque fois un exemple de classe qui correspond à un patron de dépendance particulier et nous allons essayer d'examiner les appels observés dans ces classes.

Un exemple du patron *chaîne de montage* est celui de la classe *GridFrame*. En effet, cette classe est responsable de lancer l'interface graphique de la grille de Sudoku à chaque nouvelle exécution. Elle intervient juste au début de l'exécution via les 3 appels suivants : *Sudoku main() GridFrame <init>* - *GridGenerator createGrid() GridFrame <init> GridFrame <init> Grid initialisation()*. Ainsi, cette classe faisait le même traitement tout le temps, quelle que soit l'exécution.

Un exemple intéressant du patron *employé* est celui de la classe *Solver*. En effet, lorsqu'un utilisateur initialise la grille et qu'il demande une solution, cette classe se charge de chercher une solution valide pour la grille en fouillant dans les possibilités existantes, et finit par afficher cette solution (si elle existe). Pour cela, plusieurs appels seront générés, citons à titre d'exemple :

Solver searchSolution() Sudoku <init>

Solver searchSolution() Valid <init>

Solver getPossibilities() Case attrib_valeur()

Lorsqu'une solution valide existe, le programme l'affiche directement sur la grille de Sudoku (*Solver showSolution() Sudoku <init>*), sinon il affiche un message d'erreur pour dire qu'il n'existe pas de solution valide pour cette grille (*Solver showSolution() Sudoku no_Solution()*). Ceci correspond au traitement régulier de la classe *Solver*. En plus de cette fonctionnalité, cette classe utilise de temps en temps d'autres méthodes, par exemple pour faire des retours en arrière (« back-tracking » en anglais) en cas de besoin pour modifier la solution courante (au cas où elle n'est pas valide). Ce comportement est déduit à partir de certains appels comme :

Solver backtrack() Sudoku <init> - *Solver getPossibilities() Valid test_Valide()*

Dans le cas où l'utilisateur choisit de cacher certaines cases de la grille pour les deviner, la classe *Solver* utilise sa méthode *hide()*.

Dans des cas rares, le programme produit des scénarios exceptionnels et qui génèrent des erreurs. Par exemple, lorsqu'un utilisateur demande la solution d'une grille vide non initialisée ou bien qu'il cherche à cacher des cases de la grille quand elle est vide, alors, cet appel peut être généré : *Solver showSolution() Sudoku error_Solution()*.

Un exemple de classe dont le patron de dépendance est de type *soldat* est celui de la classe *Valid*. En effet, le rôle principal de cette classe est de vérifier qu'une solution proposée sur la grille est valide ou non. En d'autres termes, il s'agit de vérifier que les règles du jeu de Sudoku sont bien respectées (c'est-à-dire, on ne peut pas avoir la même valeur deux fois au niveau de la ligne, la colonne et le carré). Cette classe effectue un traitement minimal avec de faibles fréquences. Par exemple, si l'utilisateur n'a pas terminé la saisie des 9 entrées sur la grille et qu'il cherche rapidement une solution, alors, la classe *Valid* arrête l'exécution courante du programme et affiche un message d'erreur (*Sudoku userInputGeneration() Valid test_Valide()*). Le comportement habituel de cette classe est surtout de vérifier que les solutions générées sont correctes, et si la solution obtenue (quand elle existe) est unique ou pas. Exemples d'appels :

Solver getPossibilities() Valid test_Valide() - Valid <init> Case attrib_Valeur()

Dans certains cas rares, *une bonne grille de Sudoku* ne doit présenter qu'une et une seule solution [25]. Alors, il arrive (par hasard) que la grille en question soit unique. Dans ce cas, la classe *Valid* effectue certains appels comme :

Valid test_Unique() Grid <init>

Sudoku actionPerformed() Valid test_Valide()

Solver getPossibilities Valid test_Unique()

Le quatrième patron de dépendance que nous n'avons pas encore analysé est celui du type *secrétaire*. Un exemple de ce patron est celui de la classe *Util*. Comme son nom l'indique, cette classe interagit avec la plupart des autres classes du programme pour assurer ses différentes fonctionnalités. Elle interagit avec la classe *GridGenerator* pour créer une nouvelle grille et attribuer des valeurs (*GridGenerator create() Util newGrid()*), avec la classe *Case* pour attribuer des valeurs sur les cases de la grille en fonction des entrées de l'utilisateur (*Case attrib_Valeur() Util intToStr()*), avec la classe *Solver* pour anticiper lors de la recherche d'une solution pour la grille (*Solver getPossibilities() Util strToInt() - Solver showSolution() Util intToStr()*), avec la classe *GridPanel* lorsque le programme capte les actions de l'utilisateur sur l'interface graphique (*GridPanel*

actionPerformed Util strToInt()), etc. Ainsi, il est clair que cette classe est *utilitaire* dans le programme, vu qu'elle communique avec presque la totalité du système.

5.5.2.2. Compréhension de dépendance

Maintenant qu'on a expliqué les types de patrons de dépendance pour les différentes classes de notre programme, nous allons donner quelques résultats statistiques que nous avons obtenus. Dans le tableau 6, et pour chaque classe du programme, nous avons donné la valeur minimale, moyenne et maximale observées pour le coefficient de similarité interne, la liste des blocs qui constituent chaque région d'exécution, ainsi que la moyenne de similarité externe entre les blocs d'exécutions d'une région.

En effet, on constate que les coefficients de similarité internes (*NIS*) varient de 68.55% à 100%, en moyenne. Ce coefficient vaut 92.56% pour toutes les classes du programme. Rappelons que le coefficient de similarité interne permet de nous renseigner sur le pourcentage des exécutions typiques à l'intérieur d'un bloc d'exécution. À partir de là, on constate que notre programme produit presque souvent le même comportement ce qui implique que les liens de dépendance ne changent presque pas d'une exécution à une autre pour le même bloc d'exécution. En observant les régions qu'on a obtenues pour chaque classe, on remarque qu'elles coïncident généralement avec nos interprétations des patrons de dépendance observés. Prenons par exemple le cas de la classe *Sudoku*. Cette dernière présente un patron de type *Employé*. En effet, il est possible de distinguer trois parties dans la classe. La première correspond au traitement régulier (qui coïncide avec la région R_1), la seconde région correspond à un traitement plus spécifique (R_2) et la troisième à un traitement rare ou exceptionnel (R_3).

Nom de la classe	*NIS*	*Liste des régions d'exécutions*	*ES*
Solver	Minimum : 79.68% Maximum : 94.26% Moyenne : 88.71 %	$R_1 = \{B_5, B_6, B_7\}$ $R_2 = \{B_8, B_9, B_{11}\}$ $R_3 = \{B_{12}, B_{13}, B_{14}\}$	88.65 % 87.37 % 89.39 %
Sudoku	Minimum : 81.49% Maximum : 100 % Moyenne : 96.38 %	$R_1 = \{B_4, B_5\}$ $R_2 = \{B_6, B_7, B_8\}$ $R_3 = \{B_9, B_{10}\}$	91.42 % 79.28 % 92.23 %
Util	Minimum : 100 % Maximum : 100 % Moyenne : 100 %	$R_1 = \{B_1\}$ $R_2 = \{B_2\}$ $R_3 = \{B_3\}$ $R_4 = \{B_4\}$	100 % 100 % 100 % 100 %
Valid	Minimum : 80.12 % Maximum : 95.40 % Moyenne : 90.66 %	$R_1 = \{B_1, B_2\}$ $R_2 = \{B_3, B_4, B_5, B_6\}$ $R_3 = \{B_7, B_8\}$	94.55 % 64.91 % 89.74 %
GridGenerator	Minimum : 81.77 % Maximum : 94.35 % Moyenne : 90.67 %	$R_1 = \{B_4\}$ $R_2 = \{B_5, B_6, B_7\}$ $R_3 = \{B_8, B_9, B_{11}\}$	100 % 83.22 % 91.16 %
GridFrame	Minimum : 100 % Maximum : 100 % Moyenne : 100 %	$R_1 = \{B_3\}$	100 %
GridConfiguration	Minimum : 74.39 % Maximum : 94.15 % Moyenne : 85.20 %	$R_1 = \{B_4\}$ $R_2 = \{B_5, B_6\}$	100 % 76.20 %
Grid	Minimum : 69.43 % Maximum : 98.57 % Moyenne : 93.42 %	$R_1 = \{B_1\}$ $R_2 = \{B_2, B_3, B_4\}$ $R_3 = \{B_5\}$	100 % 89.07 % 100 %
Case	Minimum : 73.40 % Maximum : 100 % Moyenne : 87.25 %	$R_1 = \{B_1, B_2, B_3\}$	87.52 %
BoxPanel	Minimum : 100 % Maximum : 100 % Moyenne : 100 %	$R_1 = \{B_1\}$	100 %
ButtonPanel	Minimum : 68.55 % Maximum : 97.30 % Moyenne : 82.66 %	$R_1 = \{B_2\}$ $R_2 = \{B_3, B_4, B_5\}$	100 % 69.50 %
InitSquare	Minimum : 100 % Maximum : 100 % Moyenne : 100 %	$R_1 = \{B_3\}$	100 %
GridPanel	Minimum : 78.97 % Maximum : 97.56 % Moyenne : 88.27 %	$R_1 = \{B_3, B_4\}$	85 %

Tableau 6. Quelques statistiques sur les classes du générateur de grilles de Sudoku.

Nous avons également détaillé ces résultats pour chaque classe du programme. Dans le tableau suivant, nous donnons un aperçu sur ces statistiques pour le cas de la classe *Solver*.

Liste des blocs d'exécutions	***NIS***	***ES***	***Liste des régions d'exécutions***
B_5 B_6 B_7	79.68 % 94.26 % 88.31 %	ES (B_5, B_6) = 90.35 % ES (B_6, B_7) = 86.94 % ES (B_7, B_8) = 30.68 %	R_1 = {B_5, B_6, B_7}
B_8 B_9 B_{11}	83.14 % 90.07 % 89.16 %	ES (B_8, B_9) = 91.66 % ES (B_9, B_{11}) = 83.07 % ES (B_{11}, B_{12}) = 21.67 %	R_2 = {B_8, B_9, B_{11}}
B_{12} B_{13} B_{14}	91.76 % 88.39 % 93.62 %	ES (B_{12}, B_{13}) = 92.05 % ES (B_{13}, B_{14}) = 86.73 %	R_3 = {B_{12}, B_{13}, B_{14}}

Tableau 7. Quelques statistiques sur la classe *Solver*.

Remarquons *une chute* des coefficients de similarité externe (*ES*) entre les blocs (B_7, B_8) et entre les deux blocs (B_{11}, B_{12}). C'est grâce à cette chute que nous avons pu faire la distinction des différentes régions. En effet, il est facile de voir que les deux blocs B_7 et B_8 ne peuvent pas appartenir à une même région puisqu'ils ne partagent pas assez de traitements. À ce niveau-là, on trouve que le rôle de la classe *Solver* a changé. De même pour les deux blocs d'exécutions B_{11} et B_{12}. Il est aussi à noter que nous avons gardé les mêmes valeurs seuils pour les deux programmes que nous avons étudiés, c'est-à-dire, *Internal_Threshold* =0.6, *Relative_Frequency_Threshold* =0.5, et *External_Threshold* =0.6.

5.6. Discussion

Le travail que nous venons de présenter dans la cadre de ce mémoire est une première initiative pour comprendre les liens entre le rôle d'une classe et ses dépendances avec le reste du système, en utilisant les modèles probabilistes des entrées. Lorsque nous avons étudié ce problème, nous avons pu résoudre quelques points, mais d'autres restent encore à résoudre. Nous allons discuter ces points dans cette section.

En pratique, il est généralement difficile de définir des distributions de probabilité pour les entrées d'un programme. Lorsque le programme est utilisé, il est possible d'enregistrer les valeurs des entrées données par un utilisateur à travers plusieurs utilisations du programme et d'estimer plus tard la distribution de ces entrées à partir des données collectées. Cependant, lorsque le programme est encore en cours de développement (c'est-à-dire, il n'est pas encore prêt pour l'utiliser), ces données ne sont pas disponibles. Bien sûr, on peut estimer théoriquement la distribution de probabilité d'une certaine entrée du programme (par exemple normale). Néanmoins, ceci n'est pas encore suffisant, car il faudrait en plus de ça, spécifier les valeurs des paramètres de la loi (moyenne et variance pour le cas d'une distribution normale).

Un autre problème qu'on peut citer concerne les types des entrées d'un programme. Dans notre étude, nous avons considéré deux programmes de taille petite dont les entrées sont des entiers ou des chaînes de caractères. Sauf que dans la plupart des programmes, les entrées peuvent également être des chaînes de caractères plus compliquées (telles que les noms de personnes), des bases de données, des fichiers, etc. La génération aléatoire de chaînes de caractères basée sur des lois de probabilité peut être facile à réaliser, mais la génération aléatoire de fichiers tels que des codes sources pour des compilateurs n'est pas une tâche évidente.

Le passage à l'échelle (ou « scalability » en anglais) est aussi un autre problème à considérer. En plus, la précision de nos résultats est directement liée au nombre d'exécutions (c'est-à-dire, la taille de l'échantillon) comme il a été illustré dans le chapitre 3. Durant la phase de validation, environ 10 heures ont été nécessaires pour pouvoir calculer la distribution de métriques des 21 classes des deux programmes, avec un échantillon de 1000 exécutions réalisé sur un ordinateur portable de RAM 4GO et un processeur Core 2 Duo CPU – 32 bits. Donc, il s'avère indispensable d'utiliser des machines plus robustes en termes de calcul et de rapidité pour parvenir à extraire des distributions de métriques pour de grands programmes avec un nombre acceptable de simulations.

Éventuellement, nous avons mentionné souvent dans ce mémoire, que l'utilisation d'une simple moyenne des valeurs de métriques n'est pas toujours appropriée, et qu'il est plus intéressant de considérer la totalité de la distribution. Le problème est que les modèles prédictifs de qualité actuels demandent juste une simple valeur (unique) en guise d'entrée pour chaque métrique. Ceci est le cas des modèles prédictifs de qualité implémentés sous la forme de fonctions où les métriques sont traitées en tant que variables. Cependant, il existe d'autres types de modèles prédictifs de qualité où une distribution peut être utilisée comme entrée. Dans [20] et [39] par exemples, les modèles probabilistes ont été implantés sous la forme de réseaux bayésiens. Lorsqu'on exécute ces modèles pour des éléments particuliers d'un logiciel donné, la distribution de probabilité est donnée pour chaque métrique qui sera passée en entrée pour le modèle prédictif de qualité.

5.7. Conclusion

Dans ce chapitre, nous avons présenté les détails d'implémentation de notre approche et nous avons fourni deux études de cas, une pour le cas du vecteur aléatoire qui correspond à un programme dont le nombre des entrées est fixe, et une pour le cas d'une chaîne de Markov qui correspond à un programme dont le nombre des entrées n'est pas connu d'avance. Pour chaque cas, nous avons donné un aperçu sur le programme, la métrique choisie, ainsi que la description du modèle probabiliste des entrées. Nous avons ensuite fourni une liste d'observations basées sur les distributions de probabilité de métriques de couplage. Ces observations nous ont permis d'introduire quatre patrons de dépendance. Dans la section résultats, nous avons donné les distributions de probabilité de métrique obtenues, ainsi qu'une interprétation de ces histogrammes en termes de patrons de dépendance. Nous avons également expliqué comment ces patrons peuvent être utiles pour faciliter la compréhension de ces liens de dépendance et du rôle d'une classe dans un programme. Les résultats statistiques que nous avons obtenus sont conformes à l'interprétation de nos patrons de dépendance et aident à donner une idée générale sur le comportement global de notre programme.

Chapitre 6

CONCLUSION

6.1. Rétrospective

Vu la complexité ascendante des produits logiciels de nos jours, leur maintenance devient plus complexe et donc, plus coûteuse en termes de temps et de ressources. En particulier, la compréhension des programmes est en grande partie responsable de ce coût élevé. Pour agir sur la compréhension, on peut soit rendre le code compréhensible en utilisant les bonnes pratiques et le « refactoring », soit utiliser les outils existants qui permettent d'explorer en détail le code du programme telle que la visualisation à plusieurs niveaux.

En effet, comprendre un programme revient à comprendre les liens de dépendance entre ses différents éléments, ainsi que le rôle que peut jouer chaque classe dans le programme. D'ailleurs, le paradigme objet a été défini de telle sorte que tous les objets interagissent entre eux pour garantir le fonctionnement général du programme. Certes, spécifier le comportement de chaque objet est une tâche facile, car il suffit de regarder la liste des méthodes qu'il invoque, mais gérer les liens d'interaction ou de dépendance entre ces objets n'est pas évident. Pour cela, il s'avère que l'analyse des liens de dépendance est une tâche à la fois difficile et importante.

Au début, l'analyse de dépendance a été définie de façon à ce qu'elle se déroule de manière statique, c'est-à-dire, en extrayant les liens de dépendance à partir du code source du programme. Les liens de dépendance statiques sont extraits sur plusieurs possibilités d'exécutions du programme, sauf que cette dépendance statique ne reflète pas vraiment tout ce qui se passe au moment de l'exécution, et donne une vue globale sur le comportement du programme. En particulier, avec les dépendances statiques, il est difficile de prendre en

compte les mécanismes dynamiques des langages modernes tels que le polymorphisme et le chargement dynamique. D'autre part, les liens de dépendance dynamiques ont été proposés pour pallier à ce problème en analysant les interactions entre les éléments d'un programme et en observant son exécution, c'est-à-dire, par la prise en compte des différentes variations du comportement du programme. Cependant, cette dépendance dynamique n'est calculée que sur une seule exécution, donc, il n'est pas possible de généraliser son résultat sur tout le programme. Pour cela, nous avons pensé qu'il serait intéressant de faire une analyse dynamique des liens de dépendance, mais calculées sur plusieurs exécutions.

Dans ce mémoire, nous avons proposé une approche probabiliste semi-automatique pour analyser et comprendre les liens entre les dépendances et le rôle d'une classe dans un programme OO. Notre approche se déroule en plusieurs étapes. Il s'agit d'abord de définir un modèle probabiliste pour spécifier et caractériser les entrées d'un programme. Sur ce modèle se base un simulateur qui simule les entrées de ce programme. Pour y parvenir, nous avons appliqué des techniques de simulation stochastiques, plus précisément les techniques de Monte-Carlo. Entre temps que le programme s'exécute, les traces d'exécution seront enregistrées. Ces dernières vont nous servir pour calculer les valeurs de métriques de couplage dynamique de chaque classe, et sur chaque nouvelle exécution du programme. Les valeurs de métriques collectées sur un échantillon d'exécutions nous ont permis de générer automatiquement des histogrammes qui reflètent la distribution de probabilité de métriques de chaque classe du programme. Dans notre approche, nous avons fait la distinction entre deux types de programmes : ceux dont le nombre des entrées est fixé au préalable que nous avons choisi de modéliser leurs entrées par un vecteur aléatoire, et ceux dont le nombre des entrées n'est pas connu d'avance et dont on a décidé de modéliser leurs entrées par une chaîne de Markov (homogène).

Basés sur les distributions de probabilité de métriques, nous avons pu observer quatre patrons de dépendance, qui, en fonction du type de la distribution, permettent de comprendre le rôle de la classe et ses liens de dépendance avec le reste du système. Pour vérifier ces observations, nous avons introduit de nouvelles mesures qui calculent des coefficients de similarité interne et externe entre les possibilités d'exécutions du

programme. Ces mesures nous ont permis de distinguer les exécutions dans lesquelles une classe joue un rôle typique, ainsi que de déterminer des régions d'exécutions qui correspondent avec un comportement spécifique du système.

Éventuellement, nous avons évalué notre approche sur deux études de cas, l'un pour le cas du vecteur aléatoire (générateur de grilles de Sudoku) et l'autre pour le cas d'une chaîne de Markov (un système d'ascenseurs). Les résultats obtenus sont prometteurs et montrent que notre approche peut être un bon point d'entrée pour analyser et comprendre les liens de dépendance, ainsi que le comportement général des différentes classes d'un programme.

6.2. Contributions

Les recherches effectuées dans le cadre de ce mémoire ont amené plusieurs contributions significatives. De plus, plusieurs pistes observées peuvent encore être approfondies pour mener à des progrès concrets dans le domaine de l'analyse de dépendance.

6.2.1. Cadre global pour lier les dépendances au rôle d'une classe

Nous avons défini un processus complet semi-automatique qui, à partir des distributions de probabilité de métriques de couplage de chaque classe, permet de caractériser le rôle de cette classe, et de quantifier ses liens de dépendance avec le reste du système.

6.2.2. Définition des modèles probabilistes des entrées

Définir un modèle probabiliste pour les entrées d'un programme est indispensable, car il permet la génération d'un échantillon représentatif de toutes les possibilités des exécutions du programme par ce que le choix des entrées est guidé par un modèle mathématique bien défini. Dans nos modèles probabilistes, nous avons pris en compte tout type de programme, ayant un nombre d'entrées fixe ou non.

6.2.3. Introduction des patrons de dépendance

Basés sur les résultats de distributions de métriques obtenues, nous avons introduit quatre nouveaux patrons de dépendance, à savoir, *chaîne de montage*, *employé*, *soldat* et

secrétaire. En fonction du type du patron (pour une classe donnée), nous avons proposé une interprétation du comportement de la classe ainsi que ses liens de dépendance avec le reste du système.

6.2.4. Analyse et compréhension de dépendance

En plus des patrons de dépendance, nous avons également proposé de nouvelles mesures pour calculer des degrés de similarité internes (au sein d'un bloc d'exécutions) et externes (entre les blocs d'exécutions d'une même classe). Ces mesures sont conformes avec nos patrons de dépendance et permettent de définir des régions d'exécutions dans chaque classe telle que chaque région permet de caractériser un rôle particulier de la classe.

6.3. Perspectives futures

Bien que les premiers résultats de notre approche soient prometteurs, il reste encore du travail à faire. En effet, l'approche que nous venons de présenter dans le cadre de ce mémoire de maitrise n'est qu'une première initiative pour examiner les liens de dépendance des programmes OO en utilisant des modèles probabilistes des entrées. Nous avons déjà présenté dans la section « Discussion » du chapitre 5 quelques problèmes présentés par notre approche que l'on vise à résoudre dans des futurs travaux. En plus de ces problèmes, nous allons citer dans cette section quelques perspectives d'amélioration de nos travaux.

En effet, la prochaine étape de nos travaux consiste à valider notre approche sur des programmes de grandes tailles pour prendre en compte le facteur de mise en échelle. D'ailleurs, nous avons commencé à travailler sur un programme Java qui s'appelle « Pooka » [27] formé d'environ 440 classes. Il s'agit d'un client-courriel qui présente des options et des facilités pour lire et envoyer des courriels. Comme le nombre des entrées de ce programme n'est pas connu d'avance, alors, nous pensons qu'il est approprié de modéliser ses entrées par une chaîne de Markov.

Jusqu'à présent, nous avons défini juste des types de dépendance *intra-classe*, c'est-à dire, à l'intérieur d'une classe. Il serait peut être intéressant de regarder aussi les liens de dépendance *inter-classes*, c'est-à-dire, entre les différentes classes du programme. De cette

façon, on aura une idée plus précise et générale sur les liens de dépendance. Pour cela, nous pensons introduire une nouvelle mesure qui calcule le degré de similarité entre les régions d'exécutions de deux classes différentes d'un même programme.

En outre, il serait peut être intéressant de prendre en compte plus qu'une métrique de couplage dynamique, c'est-à-dire, calculer plusieurs métriques de couplage dynamiques pour une classe donnée et *combiner* les histogrammes générés.

Éventuellement, nous pensons à utiliser des modèles probabilistes d'entrées pour caractériser les fonctionnalités d'un programme et pour dériver automatiquement à partir de ceux-ci des modèles pour composer ces différentes fonctionnalités. Pour cela, nous allons tenter d'utiliser les *lignes de produits logiciels* (ou *SPL – Software Product Lines* en anglais). En effet, *SPL* consiste à définir les fonctionnalités et les propriétés communes à une famille de logiciels et à proposer des méthodes, techniques et outils pour les composer afin de créer des logiciels spécifiques [47][64][65]. Le développement initial et les changements subséquents sont réalisés dans un cadre bien défini et à un niveau d'abstraction élevé, ce qui réduit la complexité et les coûts de maintenance. L'objectif est de *tester* ces lignes de produits en se basant sur la définition de modèles probabilistes d'entrées et d'une analyse dynamique des liens de dépendance des programmes qui constituent ces lignes de produits.

BIBLIOGRAPHIE

[1] M. K. Abdi, H. Lounis, and H. A. Sahraoui. Predicting change impact in object-oriented applications with bayesian networks. In *COMPSAC (1)*, pages 234-239, 2009.

[2] G. Arévalo. Understanding behavioral dependencies in class hierarchies using concept analysis. In *Proceedings of LMO 2003 (Langages et Modèles à Object)*, Paris (France), 9(1-2):47-59, February 2003.

[3] E. Arisholm, L. C. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Trans. Softw. Eng.*, 30(8):491–506, 2004.

[4] S. Asmussen and P. W. Glynn. *Stochastic Simulation*. New York: Springer-Verlag, 2007.

[5] G. K. Baah, A. Podgurski, and M. J. Harrold. The probabilistic program dependence graph and its application to fault diagnosis. In *ISSTA '08*, pages 189-200, 2008.

[6] A. B. Binkley and S. R. Schach. Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. In *Proceedings of International Conference on Software Engineering*, pp. 452 - 455, 1998.

[7] M. Boukadoum, H. A. Sahraoui, and H. M.Chawiche. Refactoring Object-Oriented Software Using Fuzzy Rule-Based Prediction. In *the 8th Maghrebian Conference on Software Engineering and Artificial Intelligence (MCSEAI04)*, Sousse (Tunisia), pp. 411-424, May 2004.

[8] L. C. Briand, J. W. Daly, and J. Wust. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Software Eng.*, 25(1):91-121, 1999.

[9] L. C. Briand, P. T. Devanbu, and W. L. Melo. An investigation into coupling measures for C++. In *ACM/IEEE International Conference on Software Engineering*, pages 412–421, 1997.

[10] L. C. Briand, J. Wurst, and H. Lounis. Using coupling measurement for impact analysis in Object-Oriented Systems. In *IEEE International Conference on Software Maintenance*, pages 475-482, 1999.

[11] M. A. Caro, C. Calero, H. Sahraoui, and M. Piattini, A Bayesian Network to Represent a Data Quality Model, In *International Journal of Information Quality*, 1(3), 2007.

[12] S. R. Chidamber and C. F. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):293-318, June 1994.

[13] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object-oriented design. In *ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 197-211, 1991.

[14] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software engineering metrics and models*. Benjamin-Cummings Publishing Co., Inc., Redwood. City, CA, USA, 1986.

[15] T. A. Corbi. *Program understanding: Challenge for the 1990's*. IBM Syst. J., 28(2): 294–306, 1989. ISSN 0018-8670.

[16] M. D'Ambros and M. Lanza. Reverse engineering with logical coupling. *WCRE : Proceedings of the 13th Working Conference on Reverse Engineering*, pages 189–198, Washington, DC, USA. IEEE Computer Society. ISBN 0-7695-2719-1, 2006.

[17] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini. Defining and continuous checking of structural program dependencies. In *ICSE '08*, pages 391-400, 2008.

[18] N. E. Fenton. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, Boston, MA, USA, 2nd edition, 1996.

[19] N. E. Fenton and A. Melton. Deriving Structurally Based Software Measures. *Journal of Systems and Software*, vol. 12, pages 177-187, 1990.

[20] N. E. Fenton and M. Neil. Making decisions: Using bayesian nets and MCDA. *Knowledge-Based Systems*,14:307-325, 2000.

[21] R. Harrison, S. Counsell, and R. Nithi. Coupling metrics for object-oriented design. In *Proceedings of the 5th International Symposium on Software Metrics*, METRICS '98, page 150, Washington, DC, USA,1998. IEEE Computer Society.

[22] S. Hassaine, K. Dhambri, H. A. Sahraoui, and P. Poulin. Generating Visualization-based Analysis Scenarios from Maintenance Task Descriptions. In *5th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 41-44, 2009.

[23] M. Hitz and B. Montazeri. Measuring Coupling and Cohesion in Object-Oriented Systems. In *Proc. International Symp. Applied Corporate Computing*, pages 25-27, Monterrey, Mexico, October 1995.

[24] http://code.google.com/p/jtracert/

[25] http://fr.wikipedia.org/wiki/Sudoku

[26] http://upload.wikimedia.org/wikipedia/commons/thumb/3/37/Standard_deviation_diagram_%28decimal_comma%29.svg/400px-Standard_deviation_diagram_%28decimal_comma%29.svg.png

[27] http://www.suberic.net/pooka/

[28] M. Kaur, P. Batra, and A. Khare. Static Analysis and Run-Time Coupling Metrics. *International Journal of Information Technology and Knowledga Management*. Volume 3, No.2, pp. 707-710, July-December 2010.

[29] M. I. Kellner, R. J. Madachy, and D. M. Raffo. Software Process Simulation Modeling: Why? What? How? *Journal of Systems and Software*, 46(2-3):91-105, 1999.

[30] F. Khomh, S. Vaucher, Y-G. Guéhéneuc, and H. Sahraoui, A Bayesian Approach for the Detection of Code and Design Smells, In *Proceedings of the 9th International Conference on Quality Software (QSIC)*, pp 305-314, 2009.

[31] F. Khomh, S. Vaucher, Y.-G. Gueheneuc, and H. Sahraoui, BDTEX : A GQM-based Bayesian approach for the detection of antipatterns, In *Journal of Systems and Software (Elseveir)*, Volume 84, Issue 4, pp 559-572, 2011

[32] R. Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: A research survey. *Journal of Software Maintenance and Evolution: Research and Practice*, Volume 15 Issue 2, pages 87-109 , 2003.

[33] P. L'Ecuyer. *SSJ: A Java Library for Stochastic Simulation*, 2008. Software user's guide, available at http://www.iro.umontreal.ca/~lecuyer.

[34] G. Langelier, H.A. Sahraoui, and P. Poulin. Visualization–based Analysis of Quality for Large–Scale Software. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 214-223, Nov. 2005.

[35] P. K. Linos and V. Courtois. A tool for understanding object-oriented program dependencies. *Proc. Workshop on Program Comprehension*, pp. 20-27, 1994.

[36] J. S. Liu. *Monte-Carlo Strategies in Scientic Computing*. New York, NY: Springer-Verlag, 2001.

[37] Y. Liu and A. Milanova. Static analysis for dynamic coupling measures. In *CASCON*, pages 119-130, 2006.

[38] S. Lock and G. Kotonya. An integrated framework for requirement change impact analysis. In *Joural of Inf. Systems*, 6(2):38-63, 1999.

[39] G. Malak, H. A. Sahraoui, L. Badri, and M. Badri. Modeling web quality using a probabilistic approach: An empirical validation. *ACM Trans. Web*, 4:9:1-9:31, July 2010.

[40] S. Mirarab, A. Hassouna, and L. Tahvildari. Using bayesian belief networks to predict change propagation in software systems. *International Conference on Program Comprehension ICPC*, 0:177-188, 2007.

[41] A. Mitchell and J.F. Power. Using object-level run-time metrics to study coupling between objects. In *ACM Symposium on Applied Computing, Santa Fe, NM*, pp. 1456–1462, 2005.

[42] D. S. Moore and G. P. McCabe. *Introduction to the practice of statistics*. W. H. Freeman and Company - Cinquième édition, NewYork, 2006.

[43] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it? In *Proceedings of the 31st international Conference on Software Engineering*, pp. 287-297, 2009.

[44] R. B. Nelsen. *An Introduction to Copulas*, volume 139 of Lecture Notes in Statistics. Springer-Verlag, New York, NY, 1999.

[45] S. L. Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2001.

[46] G. Poels. An Analytical Evaluation of Static Coupling Measures for Domain Object Classes, *ECOOP Workshops,* pp. 260-263, 1998.

[47] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[48] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Softw. Engg.*, pp. 14:5-32, February 2009.

[49] D. M. Raffo and M. I. Kellner. Empirical Analysis in Software Process Simulation Modeling. *Journal of System Software*, pp. 53:31-41, July 2000.

[50] D. M. Raffo and S. Setamanit. A Simulation Model for Global Software Development Project. *The International Workshop on Software Process Simulation and Modeling*, St. Louis, MO, 2005.

[51] J. F. Ramil and N. Smith. Qualitative simulation of models of software evolution. *Journal of Software Process: Improvement and Practice*, 7(3-4), pp. 95- 112, 2002.

[52] C. P. Robert and G. Casella. *Monte-Carlo Statistical Methods*. Second ed. New York, NY: Springer-Verlag, 2004.

[53] M. P. Robillard. Topology Analysis of Software Dependencies. In *ACM Transactions on Software Engineering Methodology*, 17(4):18:1-18:36, August 2008.

[54] R. Santelices and M. J. Harrold. Exploiting program dependencies for scalable multiple-path symbolic execution. In *Proceedings of International Symposium on Software Testing and Analysis*, pp. 195–206, July 2010.

[55] S. Setamanit, W. Wakeland, and D. Raffo. Planning and improving global software development process using simulation. In *GSD: Proceedings of the International workshop on Global software development for the practitioner*, pages 8–14, New York, NY, USA, 2006.

[56] J. Shao. *Mathematical Statistics*. Springer-Verlag, New York, second edition, 2003.

[57] N. Smith, A. Capiluppi, and J. F. Ramil. A study of open source software evolution data using qualitative simulation. In *Software Process Improvement and Practice*, 10: pp. 287–300, 2005.

[58] I. Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 2000.

[59] P. Stevens and Rob Pooley. *Using UML: Software engineering with objects and components*. Object Technology Series. Addison-Wesley Longman, 1999. Updated edition for UML1.3: first published 1998 (as Pooley and Stevens).

[60] B. Stopford and S. Counsell. A framework for the simulation of structural software evolution. In *ACM Trans. Model. Comput. Simul.*, 18(4): pp. 1–36, 2008.

[61] K. Stroggylos and D. Spinellis. Refactoring–does it improve software quality? In *WoSQ: Proceedings of the 5th International Workshop on Software Quality*, pages 10–16,Washington, DC, USA, IEEE Computer Society, 2007.

[62] A. Tang, A. Nicholson, Y. Jin, and J. Han. Using bayesian belief networks for change impact analysis in architecture design. In *Journal of Syst. Softw.*, 80:127-148, January 2007.

[63] P. P. Texel and C. B. Williams. *Use cases combined with BOOCH/OMT/UML: process and products*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA,1997.

[64] E. Uzuncaova, D. Garcia, S. Khurshid, and D. Batory. A Specification-based Approach to Testing Software Product Lines, In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, SEC-FSE '07, pages 525-528, New York, NY, USA, 2007.

[65] E. Uzuncaova, D. Garcia, S. Khurshid, and D. Batory. Testing Software Product Lines Using Incremental Test Generation, In *Proceedings of the 19th International Symposium on Software Reliability Engineering*, pages 249-258, Washington, DC, USA, 2008.

[66] I. Vanderfeesten, J. Cardoso, and H. A. Reijers. A Weighted Coupling Metric for Business Process Models. In *Proceedings of the CAiSE 2007 Forum, CEUR Workshop Proceedings*, vol. 247, pp. 41–44, 2007.

[67] Z. Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported? An Eclipse case study. In *ICSM: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 458–468, Washington, DC, USA. IEEE Computer Society, 2006.

[68] S. M. Yacoub, H. H. Ammar, and T. Robinson. Dynamic metrics for object oriented designs. In *METRICS'99*, pages 50–6199, 1999.

[69] A. Zaidman and S. Demeyer. Automatic identification of key classes in a software system using webmining techniques. *Journal of Softw. Maint. Evol.*, 20(6): pp. 387–417, 2008.

[70] Y. Zhou, H. Leung, and P. Winoto. MNav: A markov model-based web site navigability measure. *IEEE Trans. Softw. Eng.*, 33(12):869-890, 2007.

[71] Y. Zhou, M. Würsch, E. Giger, H. C. Gall, and J. Lü. A bayesian network based approach for change coupling prediction. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 27-36, Washington, DC, USA, 2008. IEEE Computer Society.

Annexe I : Contexte et notions de base en simulation

Un *modèle* constitue une représentation abstraite et conceptuelle du monde réel. Il offre une vision schématique des éléments du système que l'on veut décrire ce qui permet d'aboutir à une compréhension commune et précise d'un problème donné. Comme le modèle est théorique et qu'on a besoin d'observer son comportement, alors, il est nécessaire de le *simuler* au sein d'un programme qui décrit ce que fait ce modèle de façon concrète et réaliste. De ce fait, *la modélisation (mathématique)* consiste à définir des équations, utiliser les lois de probabilité, des théorèmes, afin de décrire de façon précise et compréhensible le système à étudier.

Généralement, on a recours à *simuler* le fonctionnement d'un système (existant ou non) car réaliser l'expérience sur le système lui-même est coûteux en temps et en ressources. En plus, il sera préférable de faire des erreurs sur le programme de simulation, mieux que d'assumer les dégâts assez coûteux sur le système réel. *La simulation stochastique*, en particulier, met l'accent sur des aspects purement aléatoires.

La méthode Monte-Carlo constitue l'ensemble des techniques servant à résoudre des problèmes plus ou moins complexes en utilisant de l'*échantillonnage aléatoire*. Cette méthode se base essentiellement sur la génération d'une séquence de nombres aléatoires en fonction d'une certaine distribution de probabilité (uniforme/exponentielle/normale/…) connue d'avance. En outre, cette méthode englobe des techniques stochastiques pour fournir des solutions servant à résoudre des problèmes variées en mathématiques au cas où les techniques standards demeurent inefficaces pour donner une solution valide [33].

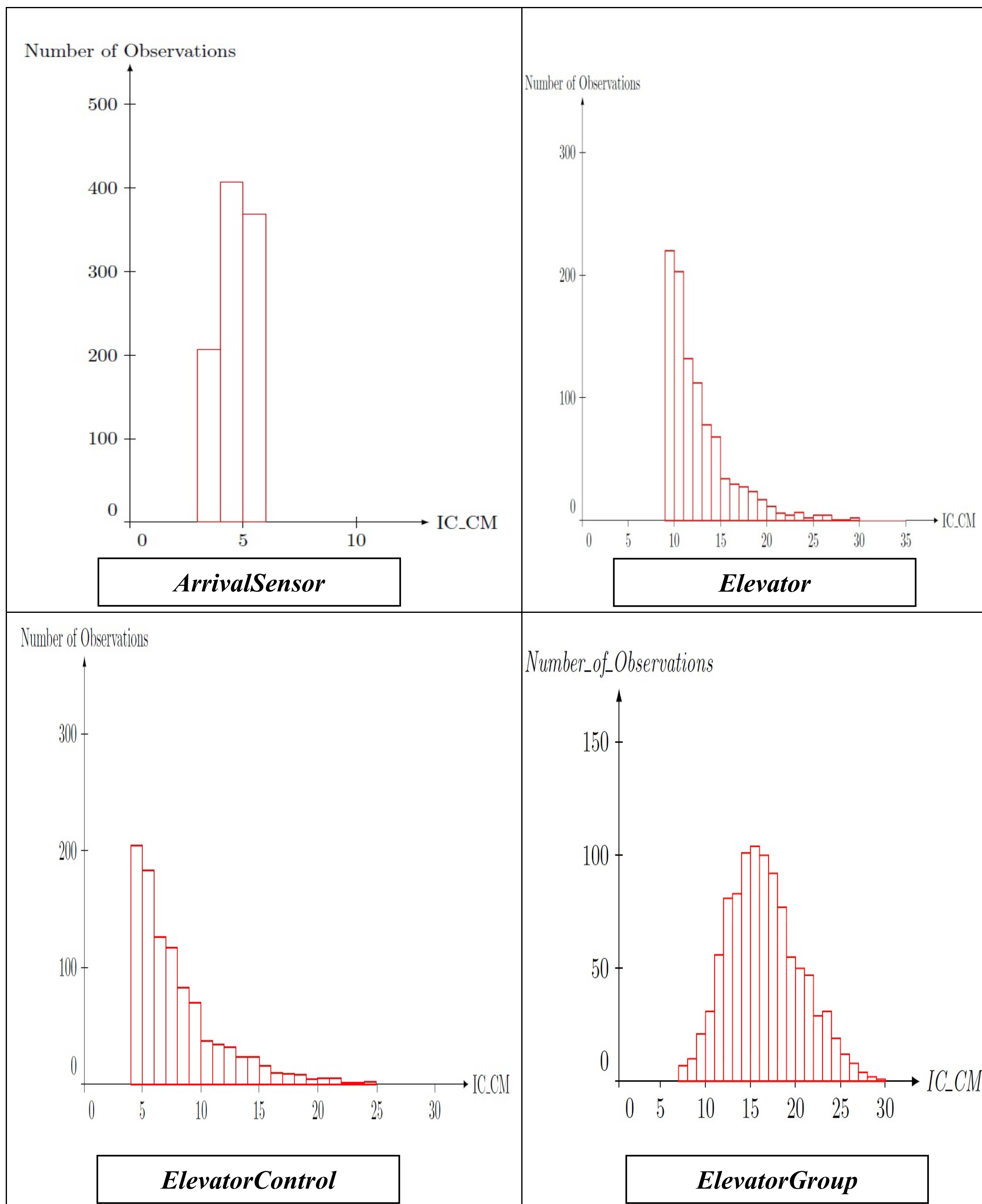
Dans certains cas, on cherche à estimer *la moyenne* théorique (moyenne exacte) μ de plusieurs observations de valeurs d'une variable aléatoire **X** donnée. Comme le nombre d'observations de **X** est généralement très grand, il est difficile, voire même impossible de prendre en compte toutes les observations. Grâce à la méthode Monte-Carlo, il est possible d'*estimer* cette moyenne en générant aléatoirement *n* copies (ou réalisations) de **X** ($\mathbf{X_1}, \mathbf{X_2}, \ldots, \mathbf{X_n}$) et de considérer ensuite la moyenne : $s = (1/n) \sum_{i=1, \ldots, n} \mathbf{X_i} = \overline{\mathbf{X}}_\mathbf{n}$. De ce fait, s (ou $\overline{\mathbf{X}}_n$) constitue un estimateur (biaisé ou non) de la moyenne exacte $\mu = \mathbf{E}(\mathbf{X})$ où **E**

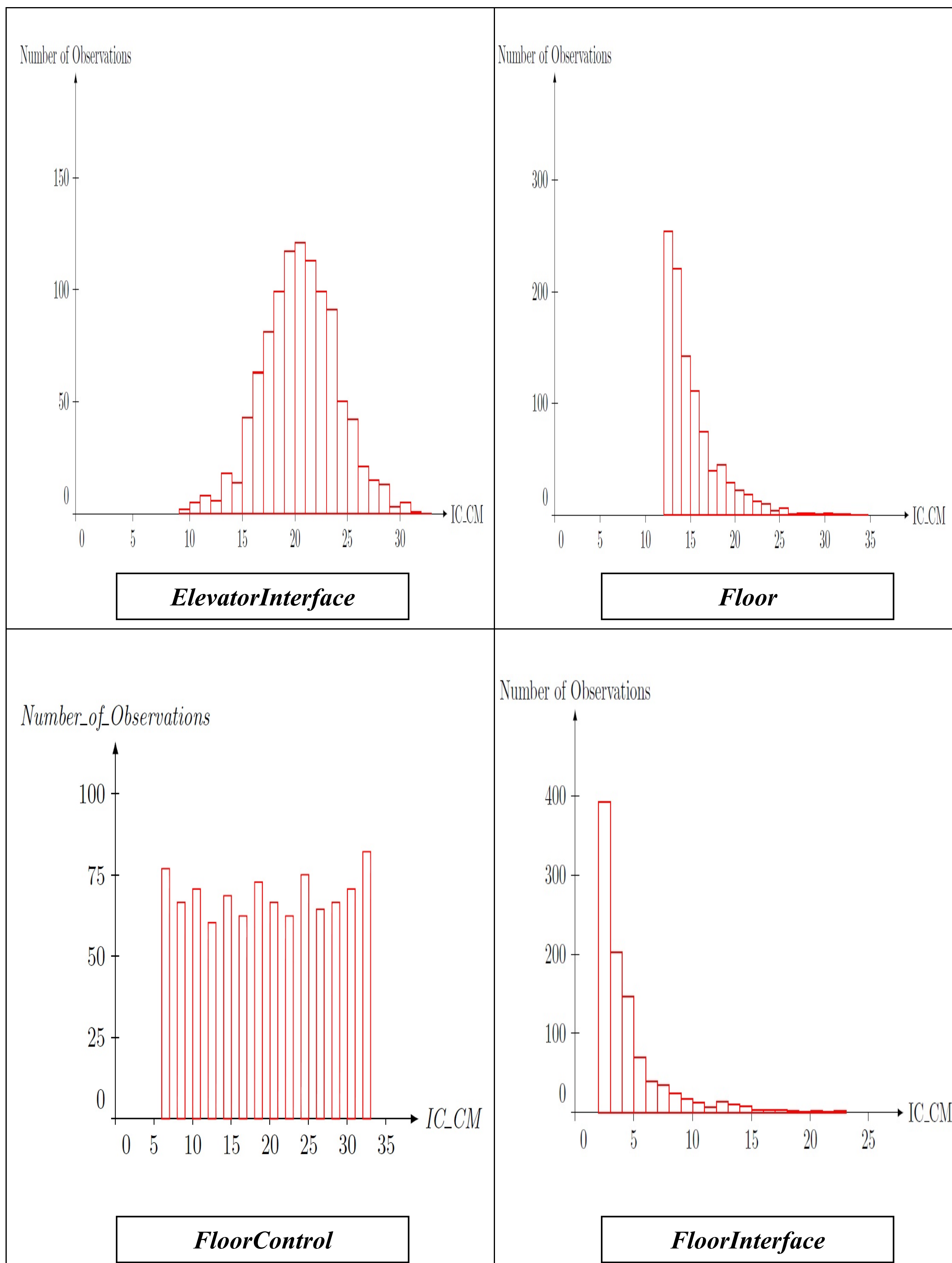
désigne l'espérance mathématique de la variable aléatoire **X**. D'autre part, il sera intéressant d'examiner les n réalisations, vis-à-vis de l'échantillon, afin de décider si ces réalisations sont groupées ou bien dispersées ce qui donnera une idée sur le degré de l'uniformité de notre échantillon d'observations. Pour y parvenir, on utilise habituellement *la variance* de **X** notée par Var(**X**) et donnée par : $\mathrm{Var}(\mathbf{X}) = \mathbf{E}(\mathbf{X} - \mathbf{E}(\mathbf{X}))^2$.

Finalement, et afin d'étudier le degré de précision de notre estimateur, il est possible de calculer *un intervalle de confiance*, ce qui permet de savoir (avec un certain degré d'exactitude) si la moyenne exacte μ se trouve entre les bornes aléatoires de cet intervalle $I = [I_1, I_2]$. De cette façon, on a $\mathbf{P}[I_1 \leq \mu \leq I_2] = 1-\alpha$ où $(1-\alpha)$ est le degré de précision souhaité où **P** désigne la *probabilité*. Par exemple, si on suppose que **X** a une distribution normale (ce qui n'est pas toujours vrai théoriquement, mais peut être une bonne approximation lorsque n devient très grand grâce au *théorème de la limite centrale*), alors, l'intervalle de confiance prendra la forme :

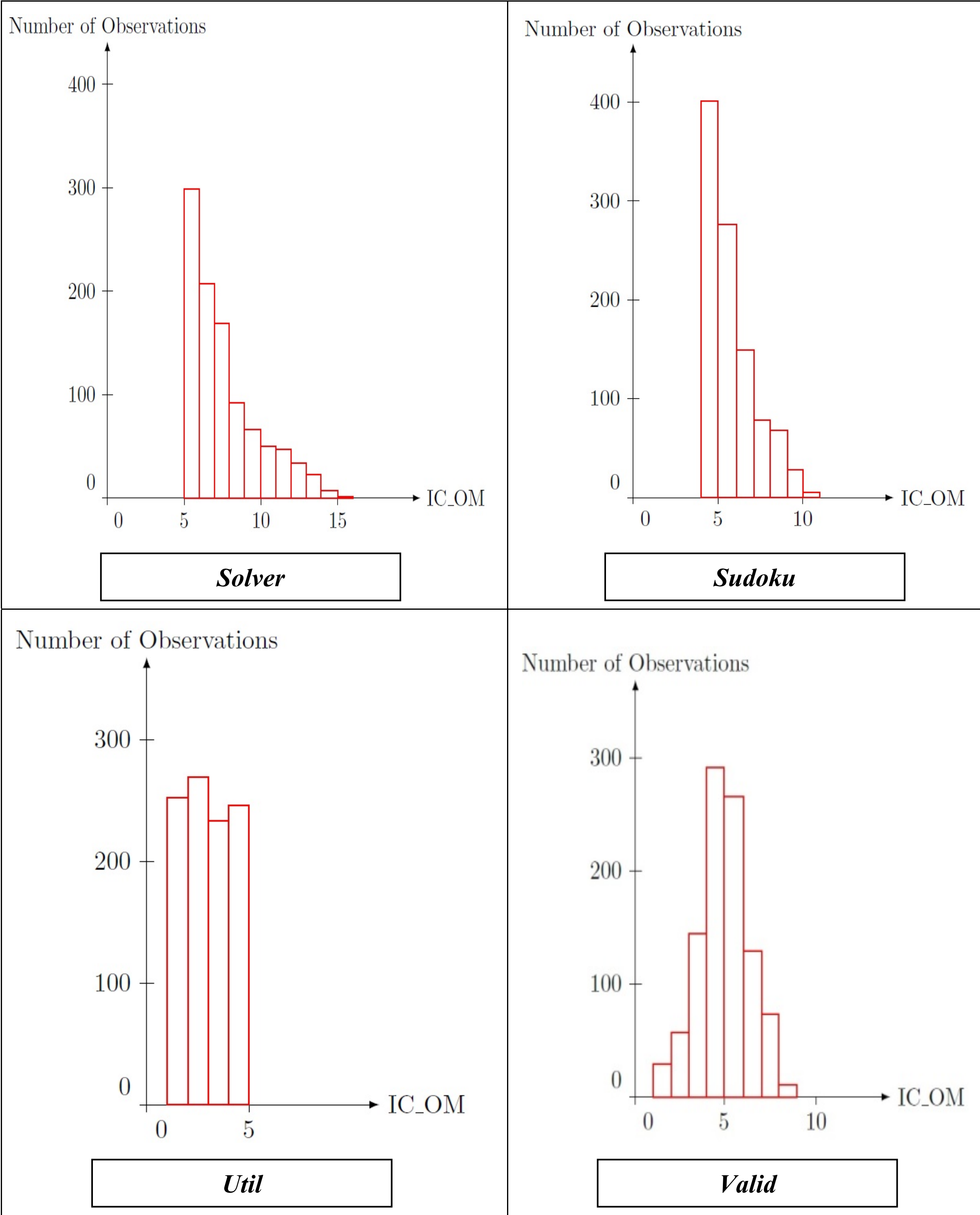
$[\overline{\mathbf{X}}_n - z_{1-\alpha/2} S_n / \sqrt{n}, \overline{\mathbf{X}}_n + z_{1-\alpha/2} S_n / \sqrt{n}]$ où S_n désigne l'*écart-type* de $X_1, X_2, \ldots, X_n$, c'est-à-dire, $S_n = \sqrt{\mathrm{Var}(\mathbf{X})}$, et le paramètre $z_{1-\alpha/2}$ satisfait la condition $\mathbf{P}[Z \leq z_{1-\alpha/2}] = 1-\alpha/2$, telle que Z est une variable aléatoire suivant la loi normale standard $\mathcal{N}(0, 1)$ [33].

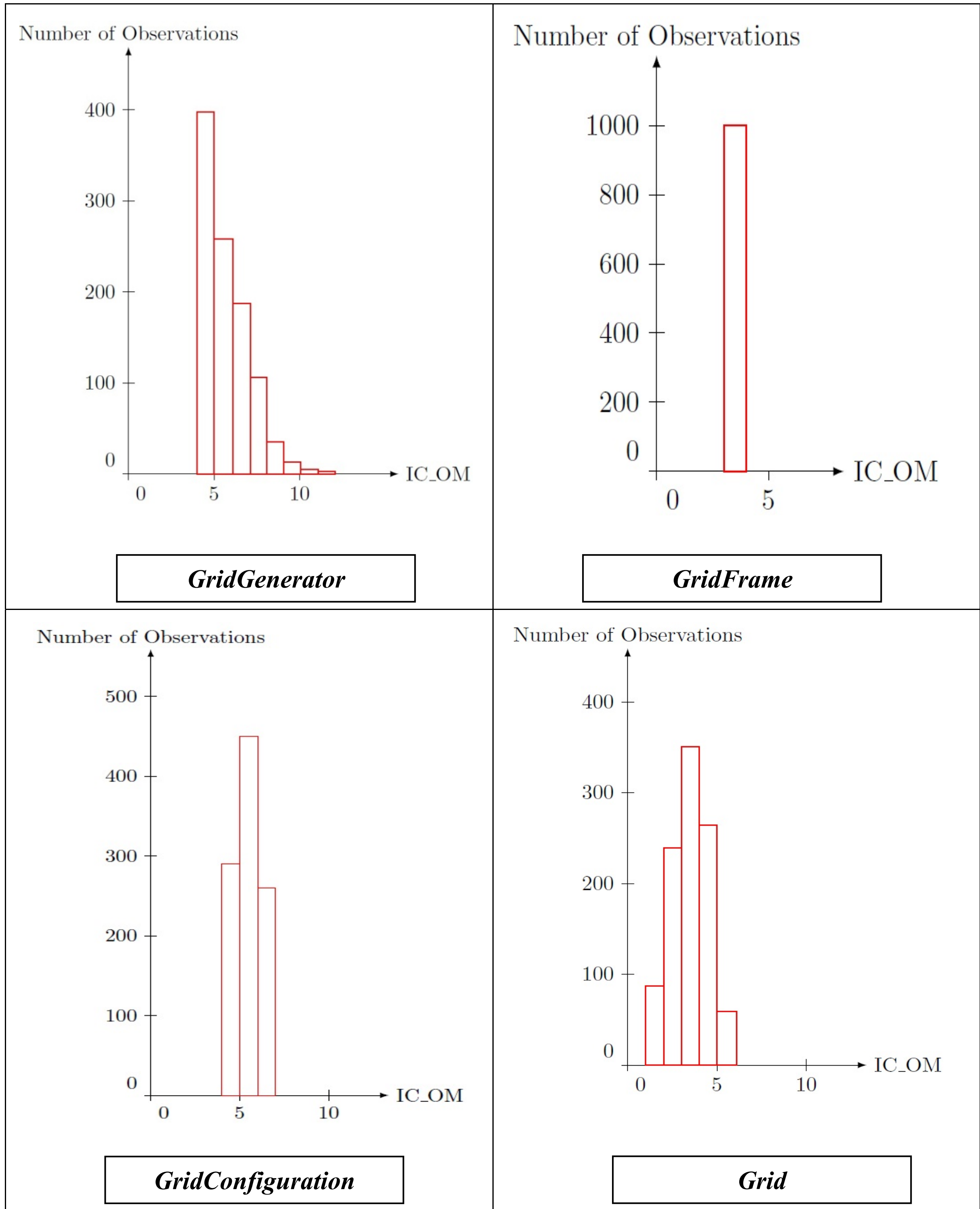
Annexe II : Distributions du système d'ascenseurs

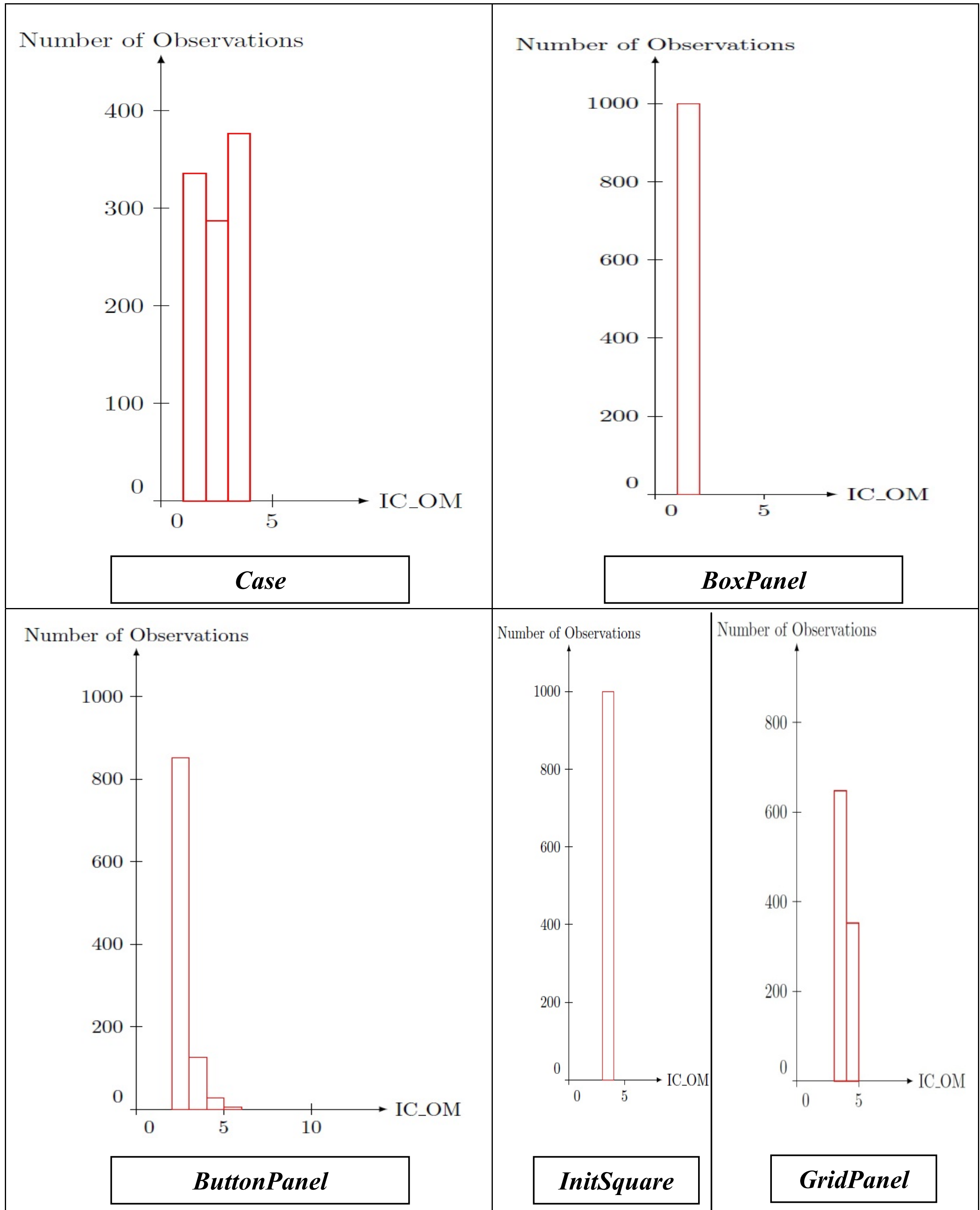



Number of Observations
150
100
50
0
0
5
10
15
20
25
30
IC_CM
ElevatorInterface
Number of Observations
300
200
100
0
0
5
10
15
20
25
30
35
IC_CM
Floor
Number_of_Observations
100
75
50
25
0
0
5
10
15
20
25
30
35
IC_CM
FloorControl
Number of Observations
400
300
200
100
0
0
5
10
15
20
25
IC_CM
FloorInterface

Annexe III : Distributions du générateur de Sudoku




Number of Observations
400
300
200
100
0
0
5
10
IC_OM
GridGenerator
Number of Observations
1000
800
600
400
200
0
0
5
IC_OM
GridFrame
Number of Observations
500
400
300
200
100
0
0
5
10
IC_OM
GridConfiguration
Number of Observations
400
300
200
100
0
0
5
10
IC_OM
Grid


Number of Observations
400
300
200
100
0
0
5
IC_OM
Case
Number of Observations
1000
800
600
400
200
0
0
5
IC_OM
BoxPanel
Number of Observations
1000
800
600
400
200
0
0
5
10
IC_OM
ButtonPanel
Number of Observations
1000
800
600
400
200
0
0
5
IC_OM
InitSquare
Number of Observations
800
600
400
200
0
0
5
IC_OM
GridPanel