UNIVERSITÉ FRANÇOIS RABELAIS DE TOURS

École Doctorale MIPTIS

Laboratoire d'Informatique (EA 6300)

Équipe Ordonnancement et Conduite (ERL CNRS 6305)

THÈSE présentée par :

Alexandre LISSY

soutenue le : 26 mars 2014

pour obtenir le grade de : Docteur de l'université François Rabelais de Tours

Discipline/ Spécialité : Informatique

Utilisation de méthodes formelles pour garantir des propriétés de logiciels au sein d'une distribution : exemple du noyau Linux

THÈSE DIRIGÉE PAR :

MARTINEAU Patrick	Professeur, Université François Rabelais de Tours

RAPPORTEURS :

LAWALL Julia	Directrice de Recherche, INRIA Paris-Rocquencourt
ROUX Olivier	Professeur, École Centrale de Nantes

JURY :

LAWALL Julia	Directrice de Recherche, INRIA Paris-Rocquencourt
ROUX Olivier	Professeur, École Centrale de Nantes
DI COSMO Roberto	Professeur, Université de Paris VII
LÈBRE Adrien	Chargé de Recherche, École des Mines de Nantes
STARYNKEVITCH Basile	Ingénieur de recherche, CEA
MARTINEAU Patrick	Professeur, Université François Rabelais de Tours

Résumé

Dans cette thèse nous nous intéressons à intégrer dans la distribution Linux produite par Mandriva une assurance qualité permettant de proposer des garanties de propriétés sur le code exécuté. Le processus de création d'une distribution implique l'utilisation de logiciels de provenances diverses pour proposer un assemblage cohérent et présentant une valeur ajoutée pour l'utilisateur. Ceci engendre une moindre maîtrise potentielle sur le code. Un audit manuel permet de s'assurer que celui-ci présente de bonnes propriétés, par exemple, en matière de sécurité. Le nombre croissant de composants à intégrer, et la croissance de la quantité de code de chacun amènent à avoir besoin d'outils pour permettre une assurance qualité. Après une étude de la distribution nous choisissons de nous concentrer sur un paquet critique, le noyau Linux : nous proposons un état de l'art des méthodes de vérifications appliquées à ce contexte particulier, et identifions le besoin d'améliorer la compréhension de la structure du code source, la question de l'explosion combinatoire et le manque d'intégration des outils d'analyse de l'état de l'art. Pour répondre à ces besoins nous proposons une représentation du code source sous la forme d'un graphe, et l'utilisons pour aider à la documentation et à la compréhension de l'architecture du code. Des méthodes de détection de communautés sont évaluées sur ce cas pour répondre au besoin de l'explosion combinatoire. Enfin nous proposons une architecture intégrée dans le système de construction de la distribution permettant d'intégrer des outils d'analyse et de vérification de code.

Mots clés : Linux, distribution, vérification, qualité logicielle, communautés

Abstract

In this thesis we are interested in integrating to the Linux distribution produced by Mandriva quality assurance level that allows ensuring user-defined properties on the source code used. The core work of a distribution and its producer is to create a meaningful aggregate from software available. Those softwares are free and open source, hence it is possible to adapt it to improve end user's experience. Hence, there is less control over the source code. Manual audit can of course be used to make sure it has good properties. Examples of such properties are often referring to security, but one could think of others. However, more and more software are getting integrated into distributions and each is showing an increase in source code volume: tools are needed to make quality assurance achievable. We start by providing a study of the distribution itself to document the current status. We use it to select some packages that we consider critical, and for which we can improve things with the condition that packages which are similar enough to the rest of the distribution will be considered first. This leads us to concentrating on the Linux kernel: we provide a state of the art overview of code verification applied to this piece of the distribution. We identify a need for a better understanding of the structure of the source code. To address those needs we propose to use a graph as a representation of the source code and use it to help document and understand its structure. Specifically we study applying some state of the art community detection algorithm to help handle the combinatory explosion. We also propose a distribution's build system-integrated architecture for executing, collecting and handling the analysis of data produced by verifications tools.

Keywords : Linux, distribution, verification, software quality, communities

Table des matières

1.1 Contexte 16
1.2 Objectifs 19
1.3 Distribution Mandriva/Mageia 20
1.4 Conclusion et axes d'étude 22

2 État de l'art **27**
2.1 Introduction 27
2.2 Choix de l'étude 27
2.3 Techniques générales 28
2.4 Axes de présentation 31
2.5 Coverity 31
2.6 Projet Coccinelle 36
2.7 Projet SLAM 48
2.8 Projet {CP,PR}-Miner 53
2.9 Projet Nooks 56
2.10 Projet Undertaker 57
2.11 Projet GCC MELT 59
2.12 Analyse statique (P.T. Breuer) 60
2.13 Autres approches 62
2.14 Conclusion 77

3 Caractérisation du noyau Linux **81**
3.1 Introduction 81
3.2 Construction du graphe 81
3.3 Étude du graphe 85
3.4 Analyses des noyaux 87
3.5 Conclusion 93

4 Application de la détection de communautés au noyau Linux **97**
4.1 Introduction 97

4.2 État de l'art de la détection de communautés 98
4.3 Méthodes applicables 132
4.4 Méthodologie d'étude 137
4.5 Résultats et analyse 142
4.6 Conclusion 144

5 Application et intégration **147**
5.1 Introduction 147
5.2 Analyse des besoins 148
5.3 Propositions et implémentation 171
5.4 Améliorations et extensions 182

6 Conclusion **191**
6.1 Vérification de code noyau 191
6.2 Cohérence du code source 192
6.3 Contributions pour Mandriva/Mageia 193
6.4 Perspectives d'évolution 194

Annexes **199**

A Analyse des graphes des noyaux **199**
A.1 Taille de la base de code 199
A.2 Résultat : occurrence des symboles 203
A.3 Résultat : densité du graphe 213
A.4 Résultat : taille moyenne du chemin 219
A.5 Résultat : degrés entrant, sortant et total 221
A.6 Résultat : carte de chaleur 228

B Analyse des résultats de la détection de communautés **239**
B.1 Résultats pour Blondel et al. (2008) 239
B.2 Résultats pour Clauset, Newman et Moore (2004) 240
B.3 Résultats pour Hofman et Wiggins (2008) 250
B.4 Résultats pour Pons et Latapy (2005) 255
B.5 Résultats pour Radicchi et al. (2004) 261
B.6 Résultats pour Zhang, Wang et Zhang (2007) 265

Liste des tableaux

1.1 Liste des paquets de Mageia 1 ayant plus de 100 fichiers modifiés 21

2.1 Classification et description des erreurs 45
2.2 Classification des types de variable et sûreté associée 66

3.1 Grandes familles de symboles, avec quelques exemples 89

4.1 Synthèse de la relation entre complexité et taille du graphe. Le nombre d'arcs est donné par m, le nombre de sommets est n. 133
4.2 Complexité des algorithmes proposés dans FORTUNATO. Le nombre d'arcs est donné par m, le nombre de sommets est n. 134
4.3 Liste des algorithmes proposant une implémentation 140

5.1 Liste des paquets de Mageia 1 ayant plus de 100 fichiers modifiés 157
5.2 Liste des paquets de Mageia 3 ayant plus de 100 fichiers modifiés 159
5.3 Liste des paquets de Mageia 4 ayant plus de 100 fichiers modifiés 160
5.4 Liste des 10 paquets de Mageia 1 ayant le plus gros effort de maintenance . 162
5.5 Liste des 10 paquets de Mageia 3 et 4 ayant le plus gros effort de maintenance 163

A.1 Grandes familles de symboles, avec quelques exemples 206
A.2 Utilisation des symboles – Noyau 3.0 – Top 20 207
A.3 Utilisation des symboles – Noyau 3.9 – Top 20 208
A.4 Utilisation des symboles par sous-répertoire (`kernel/`, `fs/`, `drivers/`, `net/`, `mm/`, `lib/`) – Noyau 3.0, instance **defconfig** 211
A.5 Utilisation des symboles par sous-répertoire (`kernel/`, `fs/`, `drivers/`, `net/`, `mm/`, `lib/`) – Noyau 3.0, instance **allyesconfig** 212
A.6 Utilisation des symboles par sous-répertoire (`kernel/`, `fs/`, `drivers/`, `net/`, `mm/`, `lib/`) – Noyau 3.9, instance **defconfig** 214
A.7 Utilisation des symboles par sous-répertoire (`kernel/`, `fs/`, `drivers/`, `net/`, `mm/`, `lib/`) – Noyau 3.9, instance **allyesconfig** 215
A.8 Comparaison des rapports du degré total par rapport au nombre de nœuds 223
A.9 Comparaison des rapports du degré entrant par rapport au nombre de nœuds 226

A.10 Comparaison des rapports du degré sortant par rapport au nombre de nœuds226
A.11 Rapport du degré entrant sur le degré sortant, pour les noyaux 3.0, 3.5 et 3.9, dans les instances **defconfig** et **allyesconfig**, pour les sous-répertoires. 229

Table des figures

1.1 Cycle de vie du logiciel : interactions entre les développeurs, la distribution et l'utilisateur 17

1.2 La boucle de la qualité : la distribution comme facteur stabilisant 18

3.1 Construction du graphe correspondant au noyau : `fonctionA()` est définie dans `fichier2.c`, `fonctionB()` et `variable1` sont définies dans `fichier3.c`, et `fonctionC()` est définie dans `fichier1.c`. Ce dernier utilise `fonctionA()`, `fonctionB()` et `variable1` ; `fichier2.c` utilise `fonctionC()`. 82

3.2 Processus pour analyser le graphe du noyau : de la construction aux résultats 85

3.3 Évolution de la quantité de code 88

3.4 Évolution de la taille moyenne du chemin dans le graphe, par sous-répertoire, entre les versions 3.0 et 3.9, pour l'instance **allyesconfig**. 91

3.5 Évolution du degré total dans le graphe, instance **defconfig** pour le noyau entre les versions 3.0 à 3.9 92

3.6 Cartes de chaleur du noyau 3.0, instance **allyesconfig**, pour le sous-répertoire `drivers/`. 94

3.7 Cartes de chaleur du noyau 3.9, instance **allyesconfig**, pour le sous-répertoire `drivers/`. 95

4.1 Processus de comparaison des différentes méthodes de détection de communautés 141

5.1 Volume de code suivant les langages de programmation utilisés dans les paquets de Mageia, version 1 152

5.2 Langages de programmation utilisés dans les paquets de Mageia, version 1 . 153

5.3 Volume de code suivant les langages de programmation utilisés dans les paquets de Mageia, versions 3 et 4 154

5.4 Langages de programmation utilisés dans les paquets de Mageia, versions 3 et 4 155

5.5 Volume de correctifs dans les paquets de Mageia, version 1 155

5.6 Quantité de fichiers modifiés par les correctifs dans les paquets de Mageia, version 1 156

5.7 Volume de modifications sur les paquets de Mageia, version 1 157
5.8 Volume de correctifs dans les paquets de Mageia, versions 3 et 4 158
5.9 Quantité de fichiers modifiés par les correctifs dans les paquets de Mageia, versions 3 et 4 159
5.10 Volume de modifications sur les paquets de Mageia, versions 3 et 4 160
5.11 Estimation de l'effort de maintenance sur les paquets de Mageia, version 1 . 161
5.12 Estimation de l'effort de maintenance sur les paquets de Mageia, versions 3 et 4 163
5.13 Taux de faute sur le noyau, versions 2.6.32 à 2.6.38 166
5.14 Taux de faute sur le noyau Mandriva, versions 2.6.33.6 à 2.6.33.7 167
5.15 Comparaison du taux de faute sur le noyau 2.6.33.7 entre la version *vanilla* et Mandriva 168
5.16 Taux de faute sur le noyau OpenSUSE, versions 2.6.30 à 2.6.37.1 169
5.17 Comparaison du taux de fautes entre le noyau *vanilla* 2.6.37 et le noyau Debian 2.6.37-2 170
5.18 Comparaison du taux de fautes entre les noyaux Debian 2.6.37-1 et 2.6.37-2 170
5.19 Architecture générale pour la plateforme de qualité proposée 175
5.20 Illustration du suivi temporel de la composition d'un paquet, avec l'exemple de `drakxtools` 180
5.21 Illustration de l'indication de l'état général de la distribution avec la répartition des langages utilisés 181
5.22 Illustration d'un correctif sémantique Coccinelle tel que visualisé avec les occurrences 182

A.1 Évolution de la quantité de code 200
A.2 Évolution de la quantité de code. (échelle log.) 201
A.3 Évolution de la taille du graphe. (échelle log.) 202
A.4 Distribution des symboles sur l'instance **defconfig** 203
A.5 Distribution des symboles sur l'instance **allyesconfig** 204
A.6 Histogramme des symboles sur l'instance **defconfig**, échelle logarithmique . 205
A.7 Histogramme des symboles sur l'instance **allyesconfig**, échelle logarithmique 205
A.8 Évolution de la densité générale du graphe. 216
A.9 Évolution de la densité du graphe, par sous-répertoire. 218
A.10 Évolution de la taille moyenne du chemin dans le graphe. 219
A.11 Évolution du degré total dans le graphe. 224
A.12 Évolution du degré entrant dans le graphe. 225
A.13 Évolution du degré sortant dans le graphe. 227
A.14 Cartes de chaleur des noyaux 3.0 et 3.9, instances **defconfig** et **allyesconfig**, pour le premier niveau de répertoires. 231

A.15 Cartes de chaleur des noyaux 3.0 et 3.9, instances **defconfig** et **allyesconfig**, pour le sous-répertoire `drivers/`. 233
A.16 Cartes de chaleur des noyaux 3.0 et 3.9, instances **defconfig** et **allyesconfig**, pour le sous-répertoire `fs/`. 234
A.17 Cartes de chaleur des noyaux 3.0 et 3.9, instances **defconfig** et **allyesconfig**, pour le sous-répertoire `kernel/`. 236
A.18 Cartes de chaleur des noyaux 3.0 et 3.9, instances **defconfig** et **allyesconfig**, pour le sous-répertoire `net/`. 238

B.1 Variations des différentes instances de Blondel et al. (2008) sur la taille et la concentration des communautés 241
B.2 Variations des différentes instances de Blondel et al. (2008) sur le nombre de communautés 242
B.3 Mesure de la taille des interconnexions des communautés avec Blondel et al. (2008) 243
B.4 Consommation mémoire et processeur pour l'exécution de Blondel et al. (2008) 244
B.5 Variations des différentes instances de Clauset, Newman et Moore (2004) sur la taille et la concentration des communautés 246
B.6 Variations des différentes instances de Clauset, Newman et Moore (2004) sur le nombre de communautés 247
B.7 Mesure de la taille des interconnexions des communautés avec Clauset, Newman et Moore (2004) 248
B.8 Consommation mémoire et processeur pour l'exécution de Clauset, Newman et Moore (2004) 249
B.9 Variations des différentes instances de Hofman et Wiggins (2008) sur la taille et la concentration des communautés 251
B.10 Variations des différentes instances de Hofman et Wiggins (2008) sur le nombre de communautés 252
B.11 Mesure de la taille des interconnexions des communautés avec Hofman et Wiggins (2008) 253
B.12 Consommation mémoire et processeur pour l'exécution de Hofman et Wiggins (2008) 254
B.13 Influence de la taille du chemin aléatoire dans Pons et Latapy (2005) sur la détection de communautés 257
B.14 Quantité de communautés détectées suivant la taille du chemin aléatoire avec Pons et Latapy (2005) 258
B.15 Mesure de la taille des interconnexions des communautés avec Pons et Latapy (2005) 259
B.16 Influence de la taille du chemin aléatoire dans Pons et Latapy (2005) sur l'exécution de l'algorithme 260

B.17 Variations des différentes instances de Radicchi et al. (2004) sur la taille et la concentration des communautés 262
B.18 Variation des différentes instances de Radicchi et al. (2004) sur le nombre de communautés 263
B.19 Mesure de la taille des interconnexions des communautés avec Radicchi et al. (2004) 264
B.20 Consommation mémoire et processeur pour l'exécution de Radicchi et al. (2004) 266
B.21 Variations des différentes instances de Zhang, Wang et Zhang (2007) sur la taille et la concentration des communautés 268
B.22 Variation des différentes instances de Zhang, Wang et Zhang (2007) sur le nombre de communautés 269
B.23 Mesure de la taille des interconnexions des communautés avec Zhang, Wang et Zhang (2007) 270
B.24 Consommation mémoire et processeur pour l'exécution de Zhang, Wang et Zhang (2007) 272

Introduction

Ce travail de thèse s'est déroulé en partenariat entre la société Mandriva (basée à Paris) et le Laboratoire d'Informatique de l'Université de Tours, dans le cadre d'une convention CIFRE (Convention Industrielle de Formation par la Recherche).

La société Mandriva a été fondée en 1998, sous le nom MandrakeSoft, afin de développer une distribution logicielle basée sur Linux. Dans l'écosystème du logiciel libre, les distributions travaillent à réaliser un assemblage intégré de logiciels pour répondre aux besoins des utilisateurs. Une version « poste de travail », à destination des ordinateurs de bureau a été maintenue pendant plusieurs années, sans jamais réellement trouver son marché. Une déclinaison serveur existe également. Malgré les différentes évolutions et péripéties de la structure, la mise à disposition et la maintenance d'une distribution sont toujours le cœur de métier, à côté du développement d'une activité de service et de gestion de parc. Le besoin de qualité et d'assurance est une clef importante pour reconquérir des clients et même pour accéder à certains marchés. Il est donc nécessaire pour Mandriva, d'acquérir la connaissance de ce domaine et de l'intégrer dans ses produits et processus. Par ailleurs, cette distribution est pilotée en « aveugle », par l'expérience (reconnue) de ses acteurs majeurs ; ceci pose problème en cas de départ ou d'indisponibilité, car il est plus difficile d'assurer l'intérim. Il est alors nécessaire d'avoir une vision globale et claire de l'état de la distribution : une sorte de tableau de bord.

Dans cette thèse, nous allons donc chercher et mettre en œuvre le nécessaire pour faciliter le travail de développement et de maintenance de la distribution, améliorer la qualité pour l'utilisateur final. Cela impliquera dans un premier temps de documenter la situation existante de la distribution et de retenir un composant « exemplaire » qui nous servira de base. Celui-ci devra avoir de bonnes propriétés pour faciliter par la suite l'extension du travail d'amélioration à tous les autres. Les résultats de cette première étude préliminaire nous amèneront à étudier le code source du noyau Linux, et notamment à nous intéresser à l'état de l'art en matière de vérification, d'analyse statique et de model-checking sur celui-ci : un objectif serait de pouvoir fournir aux utilisateurs des garanties de manière similaire à ce qui est proposé par une compagnie d'assurance.

À la suite de cet état de l'art, nous présentons un portrait de la situation actuelle sur les techniques et les outils disponibles pour effectuer de la vérification. Il ressort de cette situation l'existence d'outils de qualité, permettant de faire un premier bon niveau de vérification, et qui ne sont pas du tout ou très peu utilisés. Par ailleurs, la complexité liée à l'explosion combinatoire existe toujours. Nous proposons d'aborder cet aspect via l'utilisation d'outils de détection de communauté existant dans la littérature dans le but

de construire des sous-ensembles analysables du noyau. Cette approche nous permet également d'analyser et d'étudier un graphe du code source de Linux du point de vue de la maintenabilité, et proposer des bases pour aider et documenter la prise en main de sa complexité.

Enfin, nous présenterons notre analyse et une proposition de réponse sous la forme d'un prototype fonctionnel pour une architecture intégrant les différents outils d'analyse et d'aide à la compréhension du code à l'échelle de la distribution. Cette approche vise à permettre la constitution d'un tableau de bord accessible à la fois aux développeurs et aux responsables, permettant à chacun de piloter ce qui est sous sa responsabilité.

1.1 Contexte

1.1.1 Interactions logiciel-distribution

Une distribution Linux propose un système d'exploitation pour ordinateurs, à destination d'utilisateurs. Ces derniers peuvent être des experts ou des novices. La mise à disposition d'un système complet nécessite des outils qui se retrouvent dans tous les systèmes :

- Bibliothèques de base communes : `libc`, noyau, etc. ;
- Gestion des logiciels utilisateurs ;

Le premier point est crucial pour les développeurs d'applications et d'applicatifs : il va contraindre les choix technologiques et les possibilités offertes par le système. La surface définissant l'ensemble de ces bibliothèques peut varier, et des approches différentes se justifient : pour de nombreuses distributions Linux ces composants sont partie intégrante du cycle de vie de la distribution et évoluent avec elle ; à l'inverse, les systèmes BSD proposent une séparation nette entre un sous-ensemble critique et nécessaire formant une base précise et fournissant un jeu de services de base, le *basesystem*, au-dessus duquel les utilisateurs peuvent installer des logiciels depuis des `ports` qui vivent à leur rythme, déconnectés de la base. Il est ainsi possible de faire évoluer les deux séparément.

Le second point est également important : il permet aux utilisateurs (administrateurs en général) de choisir, installer, et suivre les mises à jour des différents logiciels nécessaires sur le système. Ce système est composé de « paquets » mettant à disposition les différents composants avec une granularité plus ou moins fine : par exemple, un moteur d'exécution et les données peuvent être séparés, afin de simplifier la maintenance. Il est toujours possible, bien sûr, d'effectuer l'installation des logiciels « à la main » ; mais dans ce cas, le suivi des mises à jour est laissé à l'utilisateur, et c'est rarement la solution que ce dernier souhaite, quel que soit son niveau. Par exemple, un utilisateur avancé sera capable techniquement de construire et faire le suivi des mises à jour à la main, mais sauf cas particulier, ce travail est chronophage et ce temps serait mieux utilisé à se concentrer sur son cœur de métier plutôt qu'à installer et mettre à jour ses logiciels. Un utilisateur novice, quant à lui, peut avoir besoin du logiciel sans avoir la moindre idée voire compétence pour le construire. Un bon exemple, pour illustrer, concernerait un navigateur Web : pour qui sait manier des outils de développement, sa construction et sa mise à jour régulière est tout à fait possible mais consomme du temps (humain et machine), mais un utilisateur novice aura besoin du

FIGURE 1.1 – Cycle de vie du logiciel : interactions entre les développeurs, la distribution et l'utilisateur

navigateur sans avoir la moindre idée des étapes nécessaires à sa construction.

Les développeurs de logiciels libres vont donc mettre à disposition leur code, qui sera pris en charge par les éditeurs de distribution, qui effectuent la mise en paquet et éventuellement ajustent l'intégration des différents composants, avant de livrer un tout cohérent et utilisable à l'utilisateur final. La distribution est donc un point de passage central et souvent obligé dans la relation développeur-utilisateur, même si en théorie rien n'empêche les développeurs de mettre à disposition leurs paquets, en pratique cela arrive peu. D'une part, la multitude des distributions rend ce travail impossible pour couvrir la totalité des utilisateurs, et même en visant seulement les plus importantes, il reste beaucoup de différences. Ensuite, la vision du développeur n'est pas forcément la même que celle de la distribution, tant d'un point de vue philosophique – par exemple, DEBIAN vise à expurger tous les composants non libres – que pratique, l'intégration effectuée par la distribution peut être beaucoup plus poussée et/ou différente de celle à laquelle le développeur d'origine pense. Ceci nous permet de proposer un schéma qui résume ces interactions, visible en figure 1.1.

Dans la section suivante, nous proposons de mettre en lumière le rôle précis et central de la distribution. Celle-ci est volontairement placée au centre de la relation : elle compose un tout cohérent. La relation entre les développeurs et la distribution pourrait être inversée, indiquant la fourniture de logiciels par celui-ci afin d'en permettre la mise à disposition aux utilisateurs. C'est un schéma qui correspond assez bien aux magasins d'applications en général (VALVE/STEAM, APPLE/APPSTORE, GOOGLE/PLAY STORE, ou encore le FIREFOX MARKETPLACE) qui se limitent à être un portail ouvert sur lequel les développeurs mettent à disposition leurs créations. Dans le cas de la distribution LINUX, celle-ci effectue un travail de recherche, de sélection, d'intégration (amélioration, correction de problème, rétro-portages de corrections, suivi de sécurité) sur les logiciels mis à disposition qui justifie de ne pas la considérer seulement comme consommatrice de logiciels proposés par les développeurs, mais comme actrice majeure et donc au centre.

1.1.2 La boucle de la qualité

La section précédente nous a permis d'établir la position de la distribution dans le cycle de vie du logiciel. La figure 1.2 propose de préciser son importance en détaillant le travail précis de l'éditeur de la distribution : nous appelons ce travail « la boucle de la qualité ». Le rôle le plus important, dans une distribution, est de proposer des logiciels utilisables : fonctionnels, intégrés. Le premier critère impose donc une série de tests pour s'assurer que les paquets qui seront mis à disposition des utilisateurs permettent de répondre au besoin. Le second impose de pouvoir effectuer des modifications pour adapter et améliorer

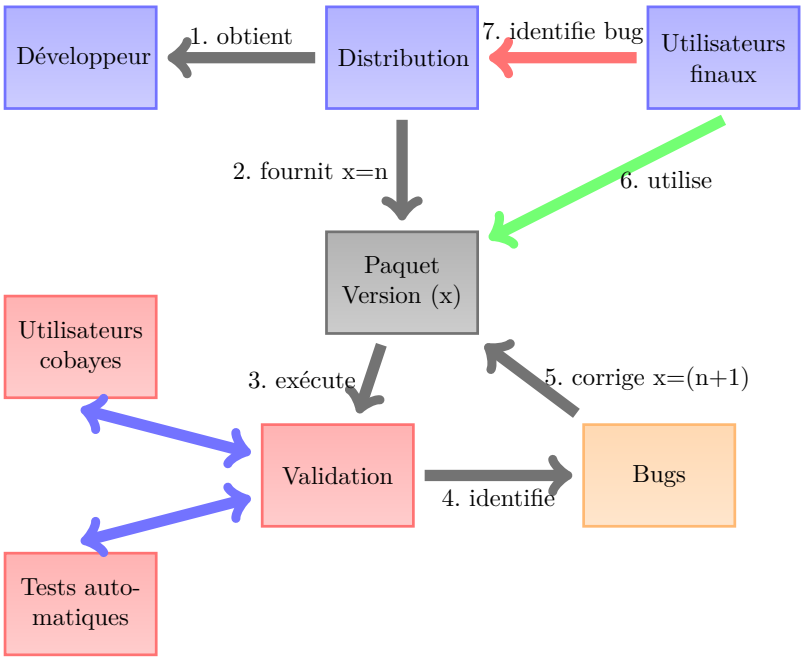


FIGURE 1.2 – La boucle de la qualité : la distribution comme facteur stabilisant

l'interaction entre les différents composants de la distribution. Ces changements peuvent avoir un impact sur l'aspect fonctionnel.

Pour chaque paquet, le processus va donc être répété un certain nombre de fois, le faisant évoluer d'une version n à une version $n+1$, jusqu'à ce que les critères permettent de valider son intégration et sa mise à disposition. Suivant la complexité du logiciel le nombre d'itérations peut varier en fonction de la quantité de changements et les interactions avec les autres composants de la distribution. La qualité de ce cycle, et donc des logiciels à leur sortie, constitue un facteur différenciateur fort pour les distributions : c'est un poste complexe, long, fastidieux, et qui impacte fortement les utilisateurs finaux. La systématisation et l'automatisation permettent de limiter la quantité de main d'œuvre nécessaire tout en augmentant la couverture possible.

De plus, comme indiqué dans le schéma, le cycle peut être déclenché à l'initiative de l'utilisateur final : lorsque celui-ci constate un problème, son réflexe est de se tourner vers celui qui met à disposition le logiciel. Ici, il s'agit donc de la distribution. Ce comportement est normal et voulu : les développeurs originels ne souhaitent pas, forcément, être submergés de problèmes rapportés qui sont liés à des modifications ou des interactions locales à une distribution. Celle-ci effectue donc un tri des problèmes rencontrés par les utilisateurs, et est chargée de les faire remonter si nécessaire : la maintenance des modifications locales n'est ni gratuite ni simple, et il est toujours préférable que celles-ci soient intégrées le plus en amont possible du cycle de développement. Quitte à ce que cette intégration soit effectuée et maintenue par des personnes de la distribution elle-même ; l'important est que ces changements soient réalisés en amont.

1.1.3 Détection de problèmes

Le coût des problèmes dépend du moment où ils sont identifiés : plus un souci est détecté tôt dans le cycle de développement, moins sa correction sera coûteuse. Dans le schéma 1.2, il est possible de lire deux sources de détection de problèmes : pendant le cycle de développement de la distribution, et pendant l'utilisation du logiciel. Le premier cas est donc le plus favorable puisqu'il arrive à un moment où nous sommes en mesure de *plus facilement* apporter des corrections et que l'impact sur les utilisateurs finaux est censé être très limité ; mais n'est pas forcément parfait car il dépend à l'heure actuelle de la capacité à utiliser le plus possible de fonctionnalités des logiciels. Or, cette utilisation reste manuelle, et cela implique d'avoir une population de test suffisamment importante et variée pour couvrir un maximum de cas. La capacité à automatiser la détection de problèmes est critique pour être en mesure de proposer une offre de logiciels assez importante avec un coût raisonnable. La société Mandriva souhaite donc automatiser la détection de bogues dans les paquets de sa distribution.

1.2 Objectifs

Nous avons présenté l'importance de la distribution dans la relation développeur-utilisateur, et notamment le rôle du gestionnaire de distribtion à la fois de testeur et de stabilisateur, mais également de filtre des remontées des utilisateurs. Ceci nous permet de définir les différents objectifs de ces travaux :

1. Documenter la situation existante ;
2. Identifier les axes d'amélioration ;
3. Intégrer les propositions dans la distribution.

Chacun de ces objectifs va être détaillé ci-après.

1.2.1 Documentation de la situation

Le premier objectif est de documenter la situation actuelle de la distribution : quels sont les langages de programmation majoritairement utilisés, quelle est leur importance d'utilisation (en volume et en nombre de paquets), quelle est la quantité de modifications locales à la distribution, c'est-à-dire des changements qui ne sont pas répercutés aux développeurs originaux.

Cette première étude servira de base pour atteindre l'objectif décrit dans la sous-section 1.2.2 et sera effectuée dans la section 1.3.

1.2.2 Identification des axes d'améliorations

Après avoir documenté la situation, nous allons présenter dans la section 1.4 les axes principaux sur lesquels nous pouvons travailler pour améliorer la qualité de la distribution. Ce sont ces axes qui seront développés dans les chapitres à venir, et qui serviront de base au travail final d'intégration.

1.2.3 Intégration dans le cycle de développement

La dernière étape importante est d'intégrer dans la distribution et dans son cycle de développement les propositions que nous ferons. Cela implique non seulement de participer à l'ajout des différents outils de vérification que nous pouvons trouver, mais également de proposer et développer leur utilisation pour la systématiser et l'automatiser le plus possible.

1.3 Distribution Mandriva/Mageia

Dans cette section nous allons présenter une analyse préliminaire de la distribution Mandriva. Cela permettra à la fois d'avoir un état des lieux rapide et partageable sur la distribution, afin de mieux comprendre comment elle est constituée, et également cela nous servira de base pour l'étude des axes d'améliorations qui interviendra par la suite. Nous nous concentrons sur les paquets logiciels qui sont les constituants de base du système. Les éléments principaux de cette étude sont proposés dans la sous-section 1.3.1, puis nous présentons certains des résultats obtenus en les commentant dans les sous-sections. Les détails sont disponibles dans la section 5.2.1.

1.3.1 Étude préliminaire des paquets Mandriva

Les paquets, comme cela a déjà été partiellement évoqué plus tôt, permettent de regrouper des logiciels dans la distribution. Ils contiennent non seulement le code source et les données, nécessaires pour les faire fonctionner, mais aussi les méta-données qui elles sont propres au système de gestion de paquets. Celles-ci indiquent les contraintes qui s'appliquent : dépendances (à des bibliothèques par exemple), incompatibilités, etc. Étudier ces paquets, c'est donc comprendre comment se constitue la base de code de la distribution.

Cette étude présentée en détails dans la section 5.2.1 nous permet de documenter l'état initial de la distribution mise à disposition par la société, et ainsi de pouvoir comprendre, définir et justifier nos futurs axes d'analyse et de travail. Différentes mesures calculées sur les paquets de la distribution ont été définies et sont présentées en détail dans la partie 5.2.1.1.

Code et langages Nous voulons comprendre et documenter la taille du code des différents paquets, ainsi que leur composition. Différents langages présentes des propriétés spécifiques et l'écosystème n'est pas identique pour tous. Documenter de quoi et dans quel volume est constitué notre base de code nous aidera pour nos choix.

Correctifs et modifications Les distributions ont besoin d'effectuer des changements à certains logiciels. Il s'agit de quantifier le volume et la taille de ces changements ainsi que leur distribution sur la distribution.

Maintenance Nous proposons de définir un indicateur pour chaque paquet qui nous renseigne sur la quantité de travail effectuée sur celui-ci. Pour des raisons de lisibilité et de simplicité, cet indicateur est pour le moment composé du nombre de personnes travaillant sur un paquet et de la quantité de modifications.

Paquet	Quantité de fichiers	Paquet	Quantité de fichiers
bash	100	grub	136
mailman	103	bacula	141
bsd-games	104	pulseaudio	178
apt	108	autofs	199
php	109	perl-Gtk2	205
mplayerplugin	114	vdr	212
qt3	115	initscripts	260
libreoffice	117	kernel-tmb	497
net-tools	119	kernel	621
mutt	122	kernel-linus	621
gcc	125	ncurses	676
netpbm	126	gdb	722
dietlibc	127	kernel-vserver	1040
john	133	kernel-xen	2420
hplip	134		

TABLE 1.1 – Liste des paquets de Mageia 1 ayant plus de 100 fichiers modifiés

1.3.2 Quantité de code et langages

En matière de volume de code nous observons dans le graphique 5.2 que les langages `C` et `C++` sont les plus importants dans la distribution. Ils sont de plus les seuls à être à une volumétrie de l'ordre de 10^8 lignes de code. Avec entre 7 et 9 millions de lignes de code, le noyau LINUX est de loin l'un des plus consommateurs. Un second graphique proposé en figure 5.1 nous renseigne sur l'utilisation faite par les paquets de chaque langage. Bien que classé cette fois troisième, le langage `C` reste l'un des plus utilisé ($\approx$ 2500 paquets) ; `C++` est un peu moins présent ($\approx$ 1000 paquets). Le `shell` reste le plus utilisé du point de vue du nombre de paquets.

Ces deux graphiques partagent une caractéristique sur la pente de la courbe, qui est décroissante : une petite part des langages vont concentrer à la fois le plus de lignes de code, mais également le plus de paquets utilisateurs. `C` et `C++` en sont deux bons exemples.

1.3.3 Quantité de correctifs et modifications

Les graphiques documentant ces résultats sont proposés en figure 5.5 et 5.6 et nous permettent de tirer deux conclusions rapides : une part importante des logiciels mis à disposition ne sont pas modifiés ; lorsqu'ils le sont, ces modifications peuvent être assez invasives. Le tableau ci-après 1.1 documente les paquets qui contiennent plus de 100 fichiers modifiés par des correctifs. Nous retrouvons ainsi la croissance très rapide observée et surtout, celle-ci est liée à une famille de paquets particulière : le noyau LINUX, avec cinq paquets ayant chacun des modifications différentes. Nous constatons aussi un écart notable entre les valeurs moyennes et le 95^{e} percentile ; c'est une indication que certains des paquets concentrent beaucoup de changements.

1.3.4 Effort de maintenance

Nous proposons enfin dans les graphiques présentés en figure 5.11 une mesure estimant l'effort de maintenance qu'a nécessité la production d'un paquet pour cette version. Comme documenté dans la définition de la mesure, il s'agit de constater le nombre de personnes différentes qui ont apporté une contribution sur la réalisation du paquet pour la version ciblée. Cette première approche de l'estimation de l'effort nous permet d'avoir une base

de comparaison assez simple à obtenir et à interpréter. Nous avons exclu les modifications apportées par les robots. L'effort moyen est très faible, et remonte un peu si nous nous limitons au 95^{e} percentile mais reste assez bas. Une minorité de paquets concentre une grosse partie du travail de maintenance. Cela peut être dû à plusieurs facteurs : taille de la base de code, complexité pour l'appréhender, difficile à tester, etc. Le tableau 5.4 résume les paquets qui sont concernés : `kernel-xen` est le pire en matière de maintenance.

1.4 Conclusion et axes d'étude

Grâce au résumé dans la sous-section précédente de l'étude préliminaire proposée en 5.2.1, nous avons des éléments tangibles pour avancer dans nos travaux. L'objectif de la société est d'améliorer la « qualité » de la distribution, ce qui nous permet de dégager trois axes de travail. Ces axes s'intègrent entre-eux et ils nous permettront de pouvoir proposer et implémenter un tableau de bord de la qualité logicielle de la distribution.

1.4.1 Résultats

Nous avons caractérisé la constitution de la distribution, qui peut se résumer comme un ensemble de logiciels ; une large majorité de ceux-ci présente à la fois un volume de code source faible et des adaptations locales limitées. Une minorité concentre à la fois une grosse quantité de code et des changements d'intégrations ou d'ajouts de fonctionnalités importants. Une étude rapide de ces paquets montrent que ce sont souvent des constituants de la base du système, et qu'ils sont donc importants voire critiques.

De plus, l'état de la qualité logicielle dans la distribution est embryonnaire : le processus pour publier des modifications sur les paquets est très, peut-être trop, libre ; à part sur quelques paquets spécifiques – le noyau Linux, la libc, ... –, les développeurs ayant accès au dépôt Subversion peuvent travailler sur n'importe quel paquet, et surtout, sans jamais avoir de processus de revue minimale. L'objectif derrière ce fonctionnement est en priorité de fluidifier le travail sur la distribution, et d'éviter de se retrouver bloquer pour des raisons de disponibilités d'autres utilisateurs. Dans le cadre d'un projet libre, pour lequel la majorité des contributeurs travaille sur son temps libre, cette contrainte est tout à fait compréhensible. La distribution Mageia est la base du produit Mandriva Business Server. Les ressources disponibles pour la maintenance étant limitées, il est bienvenue de pouvoir suivre au plus près les évolutions de la distribution.

Du point de vue du produit Mandriva Business Server, il est nécessaire d'avoir des outils pour connaître plus précisément la base de code, et aider à la maitriser tout en limitant les coûts. La mise en place d'un tableau de bord de la qualité logicielle sur la totalité du système serait une première composante de ce type d'outils. Il regrouperait :

- Navigation dans les sources des paquets, avec gestion de l'historique ;
- Intégration avec des outils de qualité sur les paquets déjà existants, tels que ceux développés dans le projet Mancoosi : vérification des dépendances, de l'installabilité, ... ;
- Base de connaissance sur les langages utilisés, leur répartition par paquet ;
- Gestion, suivi et visualisation de différentes métriques, telles que celles définies pré-

cédemment ;
– Intégration d'outils de validation de code plus poussés, applicables sur un maximum de paquets ;

En plus de faciliter la maintenance de la distribution, ces différents éléments, et notamment le dernier, constitueraient des éléments différenciateurs forts et permettant de proposer des services avancés et personnalisés. Les sous-sections à venir vont détailler les résultats dans l'optique de trois axes : vérification de code, cohérence du code source, intégration. Avant d'effectuer ces présentations, nous devons cependant justifier le choix de nous concentrer sur un paquet plus précis.

1.4.2 Le choix du noyau

Nous voulons être capables d'assurer une qualité minimale aux utilisateurs de la distribution, éventuellement proposer des garanties sur des propriétés du code. Cela signifie que nous devons faire de l'analyse de code, quels que soient les objectifs pratiques visés.

Pour réaliser ce type d'étude, nous avons besoin d'un paquet de la distribution qui présente de bonnes caractéristiques. Celles-ci peuvent se résumer :
– Composant important, critique, permettant de mettre en avant sa validation ;
– Composant présentant un code de « bonne qualité » en général ;
– Composant proposant une très large base de code ;
– Composant correctement maintenu ;

Les résultats présentés dans la section 1.3 permettent de justifier que le noyau Linux présente ces propriétés :
– L'aspect critique du noyau ne fait pas débat, même s'il est possible de trouver d'autres composants critiques dans une distribution ;
– Le code du noyau suit des règles et des standards très forts, ce qui est nécessaire vu le nombre de contributeurs. Cependant, certaines parties peuvent avoir un niveau différent, notamment dans le code des pilotes ;
– Les chiffres présentés montrent que la base de code du noyau est l'une des plus importantes dans la distribution ;
– La maintenance du noyau est très bonne, avec des nouvelles versions très régulièrement, certaines branches maintenues assez longtemps pour faciliter la vie des éditeurs de distribution serveur, et des développeurs qui sont très ouverts pour améliorer leur code ;

Ces résultats ont également permis de mettre en lumière l'importance que ce soit en volume ou en nombre d'utilisateurs du langage `C`. Celui-ci est utilisé par le noyau, nous pouvons donc espérer être en mesure de plus facilement pouvoir réutiliser des outils de vérification sur le code pour d'autres paquets. Des versions originales et alternatives, c'est-à-dire modifiées dans un but précis, du noyau ont également la propriété d'être parmi les paquets présentant le plus de modifications dans toute la distribution, comme présenté dans le tableau 5.1. Nous avons donc de nombreuses modifications locales, qui n'ont peut-être pas eu le même niveau de standards que le reste du code, et qui pourraient introduire des problèmes ; la plus-value de la vérification de ce code serait double. Si nous maintenons des modifications locales, c'est bien qu'elles rendent service à l'utilisateur final, la valeur ajoutée est donc là. Le fait que nous maintenions ces améliorations implique que

nous devons en assumer potentiellement les problèmes, effectuer un maximum d'analyses approfondies pour vérifier en amont limitera donc les risques. Enfin, le dernier résultat que nous avons proposé, le calcul de l'effort de maintenance, fait ressortir une des alternatives du noyau comme ayant l'effort le plus important.

Nous proposons de commencer en nous concentrant sur le cas du code source du noyau Linux.

1.4.3 Vérification de code noyau

Nous souhaitons valider au maximum, et automatiquement, que le code source du noyau Linux livré aux utilisateurs de la distribution respecte certaines garanties. La définition précise de ces garanties n'est pas de notre ressort, c'est le travail de la mise en place d'une ligne de produits. Cependant, nous pouvons donner des exemples : absence de déréférencement de pointeur invalide, utilisation correcte de la mémoire, etc. Il peut être souhaitable d'avoir la possibilité de proposer des garanties plus fortes encore, par exemple sur les capacités du système, son temps de réponse, etc.

Pour être en mesure de proposer ce type d'assurance, nous allons commencer par étudier l'état de l'art dans le chapitre 2 en matière de vérification de code, en nous focalisant sur le code noyau – qu'il soit Linux ou autres – ; la détection de problèmes et la validation de propriétés.

Nous en tirerons une vue non seulement des outils et des techniques existantes, des améliorations possibles, mais aussi leur pénétration auprès des développeurs – de logiciels ou de distributions –, ainsi qu'une vue générale sur ces thématiques, participant à la veille technologique nécessaire.

1.4.4 Cohérence du code source

Nous voulons être en mesure de *comprendre* le code source de la distribution. Une manière d'y parvenir, c'est savoir naviguer facilement dedans, être capable d'en dégager les axes et composants majeurs. Effectuer ce travail permet à la fois de faciliter le travail des développeurs qui effectuent la maintenance des paquets, mais aussi de partager plus facilement entre ces développeurs les connaissances sur les composants de la distribution. Enfin, savoir naviguer et tirer les fils qui constituent les logiciels permet de plus facilement suivre les impacts des modifications qui sont appliquées localement.

Pour atteindre cet objectif, le chapitre 4 propose de construire une représentation assez fidèle des interactions au sein du code source du noyau, et d'y appliquer des outils de détection de communauté. La structure de graphe proposée permet de répondre au besoin de navigation et de compréhension du code source, la détection de communauté nous permet d'aller plus loin dans la maintenance et d'aider à vérifier la cohérence du code source : l'organisation classique d'un code source, et *a fortiori* dans le cas du noyau, repose fortement sur l'arborescence du système de fichier. Les modules et les composants proches sont placés dans des répertoires proches. La détection de communauté nous permettra de vérifier que l'organisation physique est toujours en phase avec l'organisation logique, et également de les documenter et de les exposer simplement.

1.4.5 Intégration dans la distribution

Nous voulons naturellement que l'application de ces méthodes soit intégrée à la distribution. Cette intégration peut être vue sous deux angles : outils, processus. Les outils sont ceux qui ont été produits par d'autres, et qui font partie de l'état de l'art présenté dans le chapitre 2. Plusieurs des projets de recherche et des publications associées ont permis d'en obtenir de tout à fait fonctionnels, utiles, et permettant de trouver des problèmes intéressants. Nous pouvons citer, au moins, Coccinelle et Undertaker. Cependant, et malgré les efforts de leurs développeurs respectifs, les utilisateurs restent peu nombreux, même au sein des communautés concernées ; *a fortiori* dans le milieu des distributions Linux. L'intégration de ces outils est donc un préalable à l'intégration du point de vue processus : il s'agit, déjà, de les faire fonctionner dans l'écosystème visé. L'angle du processus correspond à l'utilisation des outils par la distribution : réfléchir, et proposer une implémentation de l'exploitation de ces outils, qui s'accordent avec le mode de développement de la distribution, tout en permettant de bénéficier du résultat de ces outils.

Nous évaluerons donc différents outils en vue de comprendre leur fonctionnement, leur bénéfice, et de pouvoir au moins dans un premier temps les intégrer dans la distribution. Une fois qu'ils seront fonctionnels, nous proposerons donc un processus s'intégrant dans la construction de la distribution et implémentant l'utilisation de ces outils : le résultat de cette combinaison serait un tableau de bord de la qualité ou des problèmes identifiés dans la distribution. Nous ferons ce travail dans le chapitre 5.

Chapitre 2

État de l'art

2.1 Introduction

Dans ce chapitre nous proposons une revue de la littérature sur le sujet de la vérification, et notamment du model-checking, appliquée au code source d'un noyau de système d'exploitation. En premier lieu, nous proposerons d'expliquer pourquoi nous nous sommes intéressés aux publications concernant la vérification du noyau Linux mais également de Microsoft Windows et de quelques autres systèmes d'exploitation. Ensuite, un résumé des principales techniques mises en œuvre sera proposé, avant de présenter en détail les articles s'y rapportant. Le but est de proposer une vue d'ensemble des techniques existantes, ainsi que l'état d'avancement des différents travaux et quelles seraient les voies intéressantes à explorer pour être en mesure de parvenir à notre objectif, i.e., intégrer la vérification de code dans une distribution, en commençant par le noyau Linux.

2.2 Choix de l'étude

La mise au point de l'état de l'art s'est effectuée dans le but d'appliquer une ou plusieurs techniques recensées, voire une amélioration de celles-ci, ou encore des techniques nouvelles à une vérification de code sur le noyau Linux dans le contexte d'une distribution libre. Une partie de cette recherche vise à documenter les théories existantes pour effectuer une telle vérification, ainsi que leur mise en œuvre technique dans des outils utilisables. Le code composant un noyau consiste principalement, du point de vue de la quantité de lignes de code, en des pilotes de périphériques, et c'est ce code qui est le plus souvent sujet à des problèmes, car il n'est pas forcément écrit par des experts du noyau. Ce premier point justifie de focaliser sur les thématiques liées aux pilotes.

Du point de vue des systèmes libres, la quantité de littérature disponible au sujet de leur qualité et de leur vérification est directement corrélée avec leur montée en puissance dans l'industrie : les premières publications datent de 2000, par l'équipe fondatrice de Coverity. Ces résultats feront date puisqu'ils sont cités par toutes les publications ayant suivi. Cette étude empirique ne s'est pas limitée à la vérification du code du noyau Linux. De plus, concomitamment, Microsoft commençait à travailler sur le sujet et à publier.

Enfin, malgré des différences d'implémentation ou même de design entre les différents types de systèmes d'exploitation, on peut supposer que les techniques de vérification de code sont agnostiques et peuvent s'appliquer partout. Ce second point justifie que l'on ne s'intéresse pas uniquement à Linux, mais à tous les systèmes « généralistes », et donc à la famille BSD (représentée par OpenBSD, notamment dans les travaux de [46]), ou bien Windows (qu'il aurait été difficile d'écarter vu l'importance des travaux de Microsoft sur ce sujet).

Enfin, les publications relatives à ces travaux suivent un modèle incrémental, sous la forme d'une série d'articles qui s'enchaînent en améliorant toujours la même idée de base. Cette structuration s'est retrouvée dans les travaux des différentes équipes et a été conservée pour la présentation de tous ces travaux, permettant ainsi de mieux voir la progression de chacune dans ses recherches, les incréments et les améliorations au cours du temps.

Avant de proposer cette présentation des travaux des différentes équipes dans la section 2.4, les différentes grandes idées principales contenues dans ces approches seront exposées dans la section 2.3 et les références vers les articles correspondants seront indiquées. Il est important de noter que les articles se référant à du code non libre sont moins intéressants pour nous. Cependant, le logiciel libre a horreur du vide, il est courant qu'une technique décrite dans un article soit ré-implémentée sous forme de code source libre.

2.3 Techniques générales

Pour réaliser une vérification de code sur le composant que représente le noyau Linux, et d'autres noyaux, les équipes de recherche publiant leurs résultats s'appuient sur un petit ensemble de techniques de base. Le but de cette section est de présenter rapidement ces méthodes, de présenter leur mode de fonctionnement global, et d'indiquer dans quels articles ou publications elles sont utilisées et comment elles le sont. Trois grandes familles se dégagent, accompagnées d'une quatrième plus restreinte : l'analyse statique et le model-checking (qui est une technique d'analyse statique), la méta-compilation, le confinement et enfin dans une moindre mesure la fouille de données.

2.3.1 Analyse statique

L'analyse statique de programmes permet, contrairement à l'analyse dynamique, de détecter et d'identifier des comportements non spécifiés en dehors de l'exécution réelle du code. Cela permet donc de détecter plus rapidement les problèmes potentiels et de limiter leur impact. Dans cette famille d'analyse, on distingue généralement deux sous-ensembles de méthodes :

- Le model-checking, qui permet de vérifier qu'un modèle respecte les règles décrites par une spécification.
- L'interprétation abstraite, qui permet d'extraire des informations sur le code sans dérouler son exécution.

Ces approches ont bien évidemment leurs partisans et leurs détracteurs, comme en atteste la publication [60] des auteurs de `Coverity`, dans laquelle ceux-ci proposent une

analyse comparative basée sur leur expérience, et qui est présentée en section 2.5.3. En quelques mots, chacune a ses avantages et ses inconvénients, aucune ne domine.

Parmi les publications, celles appartenant à la famille de l'analyse statique sont de loin les plus nombreuses. Outre les travaux relatifs à `Coverity` détaillé en section 2.5, ceux liés aux projets `SLAM` visibles en section 2.7 et par conséquent ceux de Post et Küchlin en section 2.13.2.2, qui visent à appliquer le même processus pour le noyau Linux, les travaux de Breuer et Valls présentés en section 2.12.1 en font partie. Les travaux de Witkowski et al. avec [188] sont partie prenante de cette famille. Il convient de ne pas oublier le projet `Undertaker` décrit en section 2.10 par Tartler et al. qui utilise aussi de l'analyse statique pour vérifier la configurabilité du logiciel. La boite à outils (qui est présentée dans [190]) propose de faire de l'analyse statique en utilisant les outils proposés par les solveurs SAT. Par ailleurs, dans [30] et [75] les auteurs décrivent de nouvelles méthodes d'analyse statique centrée sur un problème particulier et permettant de vérifier des parties préalablement délaissées : les opérations manipulant le tas (*heap*) d'un programme.

Les travaux menés par l'équipe à l'origine de `Coccinelle`, et qui sont présentés dans la section 2.6 entrent également dans la sphère de l'analyse statique.

2.3.2 Méta-Compilation

L'utilisation de méta-compilation pour effectuer des vérifications de code source a été popularisée par les travaux de Chou et al. lors de leur première publication proposant une analyse empirique de différents systèmes dans [46] : Linux, OpenBSD ainsi qu'un système embarqué FLASH. L'idée de base de la méta-compilation est d'avoir la possibilité d'augmenter les règles lors de la construction du système pendant la phase de compilation ; ainsi, le développeur qui a la connaissance, par exemple, des règles d'utilisation d'une bibliothèque ou d'une interface de programmation peut spécifier celles-ci et s'assurer qu'elles sont respectées. Les auteurs classifient indirectement leur approche comme une approche d'analyse statique dans [60].

Cette méthode est exploitée dans l'étude réalisée par Chou et al. dans [46] pour laquelle ils ont modifié le compilateur libre `GCC` en un `xGCC` et l'ont instrumenté avec un langage adapté, METAL. Une analyse plus détaillée est proposée dans la section 2.5.

Une autre exploitation importante de cette technique est proposée au travers du langage dédié (*Domain Specific Language*) `MELT` qui est présenté dans la section 2.11. Techniquement il s'agit d'un greffon au compilateur libre `GCC` et qui permet, via son langage dédié, de manipuler plus facilement les structures internes durant les différentes *phases* de compilation.

Du point de vue de la finalité, ces deux projets sont très proches, mais les briques mises en place pour y parvenir divergent : là où le code de `xGCC` et `METAL` n'est pas libre et où le langage se limite à exprimer les règles systèmes sous la forme d'une machine à état, `MELT` propose un langage beaucoup plus générique (qui sera traduit en C puis est compilé comme greffon `GCC`) et dont la totalité du code est libre, et maintenu. Pour `MELT`, les versions 4.6 à 4.8 de `GCC` sont supportées.

2.3.3 Confinement

Une autre catégorie de travaux peut se distinguer au regard des articles présentés, elle a été baptisée « confinement ». Elle consiste en un autre point de vue sur le problème des fautes dans le code : au lieu de chercher à détecter les erreurs dans le code, l'idée est de considérer que malgré tous les efforts possibles, il en restera toujours, et autant s'assurer de « limiter la casse » lorsqu'elles surviennent. C'est aussi un des moteurs justifiant le design des micro-noyaux, mais le but est ici, plutôt, d'adapter ce mécanisme de fonctionnement à l'existant.

C'est l'objectif principalement poursuivi par les auteurs du projet `Nooks` qui est présenté en section 2.9, et qui vise à isoler les pilotes du noyau Linux de sorte à ne pas les empêcher de générer des erreurs, mais détecter cet état, et procéder à leur arrêt et redémarrage. Bien qu'intéressante, cette approche n'a visiblement pas pris puisque les travaux n'ont pas vu d'autre publication par la suite.

Par ailleurs, au vu de leurs propriétés, de leur mode d'action et de leur expressivité, à la fois `MELT` – présenté en section 2.11 – et `Coccinelle` – présenté lui en section 2.6 – peuvent être considérés comme permettant de faire du confinement : `MELT`, en manipulant les structures internes lors de la compilation avec `GCC` permet non seulement de détecter des cas pathologiques, mais d'augmenter le code généré pour pouvoir les prendre en charge, ou rajouter des vérifications qui limitent les effets de bords ; `Coccinelle` procèdera de la même manière, bien que ne manipulant pas les structures internes de `GCC`, ses capacités à identifier du code et à y appliquer des transformations permettent également d'envisager de rajouter du code pour limiter, identifier voire contrer des effets de bords néfastes.

2.3.4 Fouille de données

Une dernière approche sur ces problèmes de vérification de code, qui sort notablement de l'ordinaire, est détaillée dans plusieurs publications, [104, 105, 106] : il s'agit de faire de la fouille de données. L'objectif est de détecter plusieurs types de problèmes :

- Ceux introduits à la suite de « copier/coller », comme présenté dans la section 2.8.2.
- Ceux introduits par le non-respect du protocole d'utilisation implicite d'une bibliothèque, comme présenté dans la section 2.8.1.

Il est intéressant de noter que la seconde catégorie recoupe ce qui est proposé par les projets `SLAM` et `Coccinelle`, qui proposent de faire la vérification de l'utilisation d'interfaces de programmation. La différence est ici que l'on s'intéresse à des règles qui sont implicites.

Dans le cas de la détection de copier/coller, le problème se résume à retrouver les plus longues séquences communes dans l'ensemble de données constitué par le code source du noyau Linux. L'hypothèse, qui se vérifie plutôt assez bien d'après différents auteurs, est qu'il arrive souvent que les développeurs fassent des opérations de copier/coller pour certaines parties du code, ou pour écrire un nouveau pilote, mais oublient d'en adapter correctement certaines parties, ce qui introduit des problèmes potentiellement sournois. La détection des parties de code qui ont été copier/coller permet ensuite de repérer les sections non changées et qui sont potentiellement à risque.

Pour le cas de la vérification de l'utilisation d'une interface de programmation, les auteurs proposent d'exploiter la fouille de données pour détecter les règles implicites d'utilisation d'une telle interface, pour ensuite vérifier si elles sont bien appliquées, en exploitant une approche probabiliste.

2.4 Axes de présentation

Comme indiqué dans la section 2.2, un regroupement par « projets » est proposé : souvent, les auteurs commencent par décrire une méthode ou une idée puis la développent, plus ou moins de manière incrémentale, avec des points de divergence potentiels. Malgré tout, une constance se retrouve dans ces articles, que ce soit les auteurs (voire équipes complètes) qui restent les mêmes, ou le sujet qui s'étoffe mais dont l'objectif ne diverge pas. Au sein de ces projets, les articles sont présentés dans l'ordre des contributions qu'ils apportent, ce qui correspond globalement à un ordre chronologique : ainsi, il est plus aisé de comprendre le fil des travaux et les améliorations successives. Une catégorie « autres » regroupe les articles connexes, notamment ceux apportant des bases théoriques.

Trois projets importants se dégagent réellement de l'environnement, particulièrement par leur longévité et leurs avancées :

- Coverity, issu de travaux de recherche à l'Université de Stanford, et qui a marqué le départ des vérifications avancées sur le noyau Linux, présenté en section 2.5.
- SLAM, issu de travaux de recherche au sein de Microsoft Research, et qui est contemporain des débuts de Coverity, qui propose une autre démarche, présenté en section 2.7.
- Coccinelle, issu de travaux de recherche entre l'`INRIA`, le `Lip6` et `DiKu`[1] commencé un peu après SLAM et les travaux précurseurs de Coverity, présenté en section 2.6.

2.5 Coverity

La société Coverity[2] a été créée suite à des travaux de recherche menés à l'Université de Stanford, et l'outil créé à l'occasion baptisé « Stanford Checker ». Il s'agit simplement de la version modifiée de GCC, `xgcc`, permettant de faire des analyses, de la méta-compilation, grâce au langage `METAL`. Lors de la commercialisation de l'outil et de la création de la société, `GCC` a été abandonné au profit d'`EDG`, comme indiqué par les auteurs dans [47]. Cette technique, la méta-compilation, est présentée en section 2.5.2. Par la suite, les mêmes auteurs proposent une première *étude empirique* sur les fautes dans un système d'exploitation en prenant l'exemple de Linux et OpenBSD principalement. Cette étude est présentée en section 2.5.1, et elle sera « mise à jour » une dizaine d'années plus tard, dans [132]. Quelques années et publications plus tard, Engler et Musuvathi proposent une étude [60] comparée entre les techniques d'analyse statique et le model-checking, basée sur leur expérience.

1. Université de Copenhague
2. http://www.coverity.com/

2.5.1 « An Empirical Study of Operating System Errors » [46]

Dans cet article les auteurs utilisent `xgcc`, présenté en 2.5.2, et l'appliquent à plusieurs systèmes dont Linux et OpenBSD, afin de faire une « étude empirique des erreurs dans les systèmes d'exploitation ». Leur approche est différente de la littérature classique en ce qu'elle propose une détection automatique des erreurs grâce à de l'analyse statique sur la totalité du noyau. L'étude vise à répondre à plusieurs questions, dont : « Les pilotes sont-ils la source d'erreur principale ? » ; « Quelle est la distribution des erreurs dans le code source ? » ou encore, « Quelle est la durée de vie des erreurs dans le noyau ? ».

L'utilisation du compilateur `xgcc` permet d'automatiser uniformément sur tout le noyau l'application des vérifications, permettant d'effectuer des comparaisons. L'audit des journaux et l'écriture des vérifications sont effectués manuellement. Quelques statistiques à partir des 21 versions analysées sont obtenues : entre les versions 1.0 et 2.4.1, le nombre de lignes de code a été multiplié par 16 ; puis par 2 entre les versions 2.3.0 (mai 1999) et 2.4.1 (janvier 2001). Les pilotes en sont la principale cause, et représentent entre 50% et 70% de la totalité du code source. Les auteurs définissent deux mesures : erreurs inspectées (vérifiées manuellement), et erreurs projetées (découvertes par des outils dont le taux de faux positifs est assuré comme « faible », par exemple *Block*, *Null*, *Float*, *Real*).

Différentes analyses sont exploitées, par exemple (non exhaustif) : *Block* (vérification de l'absence de deadlock) ; *Null*, *INull* (vérification de l'utilisation de pointeur) `null` ; *Range* (vérification des indices de tableaux et de boucles) ; *Lock* (pas de double acquisition, libération des verrous) ; *Free* (non ré-utilisation de mémoire déjà libérée) ; *Param* (non-déréférencement de pointeurs utilisateurs) ; *Size* (vérification de cohérence entre le type et la quantité de mémoire allouée).

Au total, 1025 erreurs sont ainsi relevées sur l'ensemble des vérificateurs utilisés. Au niveau du taux de faux positifs, les auteurs annoncent n'en avoir aucun pour *Var*, de l'ordre de 3% pour *Block* et 10% pour *Null*. Au-delà de la représentativité des erreurs trouvées, les auteurs émettent des critiques quant au traitement équitable de toutes les erreurs et arguent qu'il serait intéressant d'avoir une classification de gravité. Par ailleurs, du mauvais code qui ne contient pas les erreurs cherchées peut se faire passer pour du bon code ; c'est une des raisons qui motive à analyser plusieurs versions : du mauvais code devrait logiquement avoir des erreurs à un moment ou un autre de son cycle de vie.

La répartition des erreurs fait remonter quatre répertoires comme étant les plus importants : `drivers/`, `fs/`, `net/` et enfin `arch/i386/`. Le premier cas est notamment particulièrement repris dans le reste de la littérature, puisque les auteurs notent que les pilotes présentent jusqu'à sept fois plus d'erreurs. Plusieurs facteurs favorisant les erreurs sont également mis en lumière : taille des fonctions, complexité du code.

Un autre objectif de l'étude était de connaître et qualifier la durée de vie des erreurs. Cette durée de vie est un indicateur que l'efficacité du processus qualité du noyau : chaque nouvelle version du noyau vient avec son lot de correction d'erreurs et de nouvelles erreurs. Quelques exemples sont proposés pour certains types d'erreurs (en années) : *Block* $\approx$ 2.52 ($\pm$0.15), *Null* $\approx$ 1.27 ($\pm$0.10), *Var* $\approx$ 1.43 ($\pm$0.23), *Tous* $\approx$ 1.85 ($\pm$0.13). Les auteurs présentent les biais potentiels dans leur méthodologie : la quantification pose problème, la plupart des versions sont séparées de quelques mois, mais cette durée n'est pas constante

(d'un mois à un an). Ceci augmente artificiellement la durée de vie des erreurs. De plus, pour beaucoup, elles n'étaient pas corrigées dans la dernière version testée et la date de correction est déduite à partir des données précédentes par un estimateur Kaplan-Meier. Un autre biais vient des auteurs eux-mêmes qui, en découvrant et en rapportant des erreurs, réduisent potentiellement leur durée de vie.

2.5.2 « Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions » [61]

Dans cet article, les auteurs proposent de vérifier des règles sur l'utilisation des interfaces dans le code source, que ce soit au niveau des noyaux, dans les systèmes embarqués ou des bibliothèques. La méthode proposée par les auteurs est appelée *« Meta-Level Compilation » (MC)*. Les auteurs de ce papier justifient leur approche par rapport aux techniques de model-checking à cause des spécifications : coût et complexité de leur écriture, niveau d'abstraction trop élevé, manques et sur-simplifications liés. Cette contribution est contemporaine des travaux sur SLAM[12, 13] menés par Microsoft, visant à faire du model-checking. La technique de *Meta-Level Compilation* permet à l'auteur du logiciel de décrire la sémantique que son code doit suivre, pour pouvoir la vérifier. Ces travaux ont été « fondateurs », puisque quasiment la totalité des références trouvées sur ces thématiques y font référence. L'implémentation proposée pour les expérimentations de l'article se fait grâce à un langage spécial, `METAL`, implémenté dans le compilateur *xg++* (basée sur le compilateur *GNU g++*). Les règles de vérifications sont des machines à états, contenant des motifs que le compilateur doit trouver ainsi que des états qui sont activés lorsque les motifs sont détectés. Les auteurs rappellent les limitations de leur outil : ce n'est pas de la *vérification*, mais bien de la découverte de bogues ; on ne garantit pas l'absence de bogues.

Cette approche permet non seulement la recherche de bogues que l'on est en mesure de caractériser, mais aussi d'identifier des cas d'optimisation du code. Les résultats sur la vérification qui ont été menés sur le code source des noyaux Linux et OpenBSD sont plutôt encourageants puisque plus de 500 erreurs ont été trouvées (utilisation de `assert()`, effets de bord liés à son utilisation, gestion des pointeurs, allocation mémoire, primitive de synchronisation, etc.). Un peu moins de 300 faux positifs sur le total des vérifications, et des problèmes débusqués qu'il aurait été difficile de trouver à la main ou par des tests classiques.

2.5.3 « Static Analysis versus Software Model Checking for Bug Finding » [60]

Dans cet article, les auteurs proposent leur retour d'expérience après plusieurs travaux de vérification de code, avec des techniques d'analyse statique et du model-checking. Les travaux sur la cohérence des caches dans le système FLASH ainsi que l'implémentation du protocole AODV servent pour le retour d'expérience en analyse statique et model-checking ; l'implémentation de la pile TCP dans Linux illustre du model-checking seul.

Pour la vérification sur FLASH, ce sont des règles propres à ce système qui sont étudiées ; il a par ailleurs déjà servi pour appliquer des techniques de vérification dans le cadre de [61]. L'objectif de la comparaison est de montrer les avantages et les inconvé-

nients de chaque technique. Un premier résumé des résultats obtenus est l'efficacité de l'analyse statique : il faut moins de temps pour écrire et réaliser ce genre d'analyses et elles trouvent plus d'erreurs ; le code doit simplement pouvoir être *compilé*, et *tous* les chemins d'exécution peuvent être couverts. La difficulté principale réside dans la mise en place de l'environnement pour le model-checking, celui-ci permet de vérifier des propriétés plus riches (par exemple, qui nécessitent un raisonnement poussé). L'environnement doit être de bonne qualité sans quoi les erreurs n'ont pas beaucoup de sens. L'outil de model-checking, `CMC (Custom Model Checker)` a été écrit par les auteurs : de leur point de vue, la plupart des langages existants pour faire du model-checking ne permettent pas de couvrir les fonctionnalités du C (les pointeurs étant cités plus spécifiquement).

La méthodologie est présentée. Pour le model-checking, le choix se porte sur l'utilisation d'états explicites. Une des limitations reste l'écriture des spécifications. Les auteurs proposent deux approches réduisant cette difficulté. Pour l'analyse statique, ils utilisent `xgcc`[61] présenté en 2.5.2. Si cette approche peut retourner des faux négatifs, l'objectif reste de trouver un maximum d'erreurs en limitant les faux positifs.

La charge de travail pour l'écriture des spécifications pour la vérification du protocole `AODV` est évaluée à deux semaines pour le premier modèle, sans tenir compte du temps de développement de l'outil de model-checking. Le modèle est constitué d'un cœur de fichiers sources non modifiés, et d'un environnement modélisant le réseau avec des implémentations simplificatrices. Le model-checking fait ressortir 42 erreurs dont 35 uniques, et l'analyse statique permet d'en identifier 34. La comparaison des résultats montre que certains types d'erreurs sont identifiés par les deux techniques (mauvaise gestion des erreurs de `malloc()`, fuites mémoires, utilisation de mémoire libérée), d'autres sont identifiés par le model-checking seul, et enfin certains uniquement par l'analyse statique. Deux catégories sont proposées : les propriétés génériques (reprise sur erreur de `malloc()`, fuites mémoire, utilisation de mémoire déjà libérée) et celles spécifiques au protocole, détectées seulement par model-checking (entrée invalide dans la table de routage, message inattendu, génération de paquets invalides, erreurs dans les hypothèses du programme et boucles de routage).

Pour l'analyse de l'implémentation TCP dans le noyau Linux 2.4.19, les auteurs rappellent la difficulté principale du model-checking : création de l'environnement et l'extraction du code à vérifier, et préfèrent tester l'intégralité du noyau, la génération de fausses fonctions pour les besoins du test risquant d'induire des faux positifs qu'il serait difficile de débusquer. En faisant une couverture de 55% en matière de nombre de lignes de code, les auteurs montrent qu'ils couvrent 92% du protocole, avec trois modèles (variant le comportement des clients, avec des connexions simultanées). Pour des questions de simplicité, les auteurs notent qu'ils n'ont pas stimulé le code prenant en charge les paquets mal formés.

Plusieurs enseignements d'ordre général sont donnés : si l'on cherche des erreurs, on finit forcément par en trouver ; la description du respect d'une propriété est plus simple que son non-respect ; vérifier des propriétés dans du code réel est plus simple qu'un cas trop général ; il ne faut remonter que les erreurs importantes.

Pour l'analyse statique il s'agit principalement de la possibilité de parcourir tous les chemins de code, la compréhension partielle du code est suffisante, l'analyse est assez rapide (en heures quand le model-checking est en semaines), elle permet le traitement de plusieurs millions de lignes de code et surtout est capable de découvrir des milliers

d'erreurs. Pour model-checking, le meilleur modèle du code reste l'implémentation, mais il permet de viser la couverture la plus large possible, il ne faut pas négliger la nécessité de limiter tout travail manuel au maximum et surtout se limiter aux interfaces (API) publiques.

2.5.4 « A few billion lines of code later : using static analysis to find bugs in the real world » [26]

Dans cet article, les auteurs proposent un retour sur leur expérience suite à la fondation de la société Coverity, qui commercialise (depuis 2002) un logiciel exploitant les résultats obtenus grâce à `xgcc`. L'article ne propose pas de résultats scientifiques, par contre les informations fournies sur la méthodologie de déploiement auprès d'utilisateurs « finaux » est intéressante. Un premier point mentionné est l'évolution du point de vue de la communauté par rapport aux années 2000 : l'approche présentée dans [61] n'est pas parfaite (*unsound*) et cela faisait débat à l'époque ; c'est devenu un comportement normal pour les projets qu'ils soient de recherches ou commerciaux.

Le profil des utilisateurs apporte des contraintes très fortes. D'abord sur les retours de l'outil : ceux-ci doivent être le plus clair possible, sinon, selon les auteurs, les utilisateurs vont remettre en cause l'outil lorsqu'il s'avère que le retour était correct mais mal compréhensible. Les exemples donnés concernent notamment du code C invalide, et donc qui ne peut pas être pris en charge par Coverity, mais qui compile avec les outils de l'utilisateur ; ce qui oblige à mettre en place, pour être en mesure de signer les contrats, des contre-mesures.

Une autre contrainte, liée au profil, concerne la vitesse de traitement de l'outil. Pour être en mesure de s'intégrer dans les systèmes de compilation, notamment les compilations nocturnes quotidiennes, et détecter au plus tôt les régressions et les erreurs, il est nécessaire de pouvoir traiter la totalité du code source pendant une nuit. Les auteurs avancent, sur la base de leur expérience, une vitesse à atteindre de l'ordre de 1400 lignes de code par minute.

Les erreurs remontées par l'outil doivent être facilement compréhensibles par les développeurs. Sans quoi ceux-ci vont mal utiliser Coverity, et rapporter des faux positifs sur de vraies erreurs, et ainsi détériorer la qualité de l'analyse et de la détection de ces faux positifs, engendrant un cercle vicieux. Les auteurs justifient ainsi l'élimination des erreurs trop complexes à comprendre, pour limiter l'impact. Le taux de faux positif a également une influence, puisqu'au-delà de 30% l'utilisateur considère que l'outil fonctionne mal : un objectif d'un maximum de 20% est donc considéré lorsque des compromis sont à faire (analyse de meilleure qualité contre faux positifs).

2.5.5 Conclusion

Les travaux initiés par Engler et al. dans [61] ont été particulièrement féconds : ils ont non seulement permis la fondation de la société Coverity qui propose « gratuitement » ses services sur certains projets libres afin de démontrer la qualité du système. La technique de Méta-Compilation, décrite dans [61] et appliquée dans [46], permet une intégration dans

la chaîne de compilation bienvenue comme rappelé dans [26] ; on notera que `MELT`[42, 158] reprend le même principe en exploitant la nouvelle infrastructure de greffons de `GCC`, ce qui permettrait de réaliser un équivalent iso-fonctionnel aux travaux de `xgcc`. Par ailleurs, le langage `METAL` utilisé pour spécifier les règles peut être vu comme un précurseur des correctifs sémantiques `SmPL` utilisés par Coccinelle. Les bases théoriques sur lesquelles ces travaux reposent restent malgré tout simples, et les analyses effectuées sont efficaces en partie grâce à l'avance prise par les auteurs : travaux commencés de longue date, de nombreux projets déjà analysés, stratégie limitant les remontées de faux positifs.

2.6 Projet Coccinelle

Le « projet Coccinelle » regroupe un ensemble de travaux menés afin d'améliorer la qualité du code source du noyau Linux. Le début du projet correspond au besoin de simplifier la vie des développeurs noyaux tout en fiabilisant le processus sur un cas pratique : comment réaliser efficacement une modification sur une API du noyau. Ce point de départ [90] permet de justifier facilement l'approche, en prenant exemple de plusieurs erreurs qui ont été introduites par ce type d'évolutions, et qui sont restées plus ou moins longtemps ; les auteurs définissent ainsi le « problème des évolutions collatérales ». Au fil des années, ils bâtissent ainsi tout une pile d'outils dérivant de ce premier constat, et forment le corpus documentaire qui nous intéresse. Un découpage en trois sous-parties majeures est proposé :

1. Gestion des évolutions collatérales (en section 2.6.1)
2. Détection d'erreurs dans le code source (en section 2.6.2)
3. Suivi du cycle de vie des erreurs (Herodotos) (en section 2.6.3)

2.6.1 Gestion des évolutions collatérales

Dans cette première sous section sont regroupées les publications concernant les travaux initiaux sur l'outil Coccinelle, qui se bornent à traiter le problème de la mise en œuvre des évolutions collatérales sur le code source.

2.6.1.1 « Tarantula : Killing Driver Bugs Before They Hatch » [90]

Dans [90] les auteurs du futur projet Coccinelle présentent leurs premiers résultats sur l'aide à l'application des évolutions collatérales. Il ne s'agit pas là de « vérifier » les interfaces à proprement parler, mais leur objectif est d'aider à ce que les modifications importantes d'APIs puissent se propager plus simplement : plusieurs exemples sont donnés de modifications qui ont été longues à finir totalement. Notamment, la fonction `check_region()` qui peut présenter des problèmes suite à des changements dans la manière d'initialiser les pilotes est à remplacer par `request_region()` accompagnée de quelques lignes pour gérer le cas où l'initialisation échoue. Ce changement a débuté dès la version **2.4.2** (février 2001) et jusqu'à la version **2.6.10** (décembre 2004) il n'était toujours pas fini (bien que la fonction était considérée dépréciée depuis la version **2.5.54**, janvier 2003). Bien sûr, des erreurs ont été introduites avec ces changements. L'explication donnée pour le temps nécessaire à faire ces modifications provient du fait que les changements sont

complexes à appliquer. Ils justifient ainsi les correctifs sémantiques sur leur forme et sur leur objectif : documenter les évolutions des interfaces, pouvoir les appliquer plus facilement sur une grosse base de code, et décrire celles-ci sous une forme compréhensible par les développeurs.

2.6.1.2 « Semantic Patches for Documenting and Automating Collateral Evolutions in Linux Device Drivers » [128]

Cet article traite d'un sujet un peu plus éloigné du model-checking que les précédents : l'objectif des auteurs est de proposer des améliorations à la gestion des correctifs pour le noyau Linux, de sorte à rendre plus solide les processus évolutifs sur les APIs du noyau. Ils commencent donc par définir le problème de l'évolution collatérale puis décrivent la solution qu'ils proposent : les correctifs sémantiques. L'aspect model-checking apparait lorsque le mécanisme qui permet d'exploiter ces correctifs sémantiques est décrit. Cela repose sur le moteur de transformation Coccinelle : le code source C est traduit en un graphe de contrôle de flux qui sert de modèle ; le correctif sémantique est traduit en une formule de logique temporelle CTL augmentée de fonctionnalités supplémentaires. Il est intéressant de trouver un morceau de code réel correspondant à une réelle évolution d'API dans le noyau sous la forme de correctif sémantique, et de trouver la traduction en logique CTL adaptée par les auteurs de ce même correctif. Cette manière de faire est donc une application typique du model-checking, mais non dans le but de vérifier des propriétés sur le code : en réutilisant le formalisme classique, le code devient modèle, et les propriétés à vérifier sont celles décrites par le correctif sémantique.

2.6.1.3 *Towards Documenting and Automating Collateral Evolutions in Linux Device Drivers* [129]

Dans [129] les auteurs du projet Coccinelle présentent l'utilisation des correctifs sémantiques à des fins de documentation (entre autres). Une première contribution relative à la classification des erreurs dans ce rapport de recherche donne les erreurs classiques liées à l'application manuelle des évolutions collatérales dans le noyau Linux :

- Erreurs, suite à une coquille typographique, une erreur d'inattention, etc. Les auteurs rapportent notamment le cas d'erreurs qui se glissent et ne sont activées que lors de l'utilisation d'options de compilation spécifiques, montrant aussi l'intérêt d'Undertaker (section 2.10).
- Mauvaise compréhension, très souvent lorsque la personne qui modifie l'interface de la bibliothèque n'est pas la même que celle qui met à jour les pilotes qui l'utilisent : par exemple l'ajout d'une couche de compatibilité dans la bibliothèque (`check_region()` remplacée par `request_region()`), ou lorsque le code ne fait pas partie de la branche principale du noyau.
- Conflits, lorsque l'évolution collatérale est faite sur une version non totalement à jour du noyau Linux.

2.6.1.4 « SmPL : A Domain-Specific Language for Specifying Collateral Evolutions in Linux Device Drivers » [125]

Dans [125] les auteurs présentent en détail le langage de description des correctifs sémantiques utilisés dans Coccinelle : `SmPL`. Ils ont été dans un premier temps créés pour décrire les évolutions des interfaces utilisées au sein du noyau Linux et ensuite les appliquer de manière automatique aux pilotes. Ces correctifs décrivent les transformations à faire au niveau du graphe de contrôle de flux ce qui selon les auteurs permet de cibler la sémantique du code C et non simplement sa syntaxe. La justification pour utiliser des correctifs est simple : les développeurs du noyau Linux ont déjà l'habitude de manipuler ceux-ci, de les relire pour correction avant inclusion du code. Il est donc important de conserver un format « connu » pour faciliter l'adoption de l'outil. Cette contribution repose sur les notions suivantes, illustrée sur un cas concret, la modification du prototype d'une fonction du sous-système SCSI :

- Remplacements, utilisé pour changer le prototype d'une fonction par exemple. L'expression est identique à celle utilisée dans les correctifs classiques.
- Métavariables, qui permet de s'abstraire des noms des identifiants (par exemple les paramètres dans la fonction dont on change le prototype). Ceux qui permettent de définir la fonction sont déclarés comme `identifier`, et afin d'indiquer au programme qu'il convient de trouver un identifiant qui n'interfère pas avec ceux en place, les auteurs proposent de le repérer avec `fresh identifier`. De manière similaire, un mot clef `struct X y` permet de déclarer une métavariable `y` de type `struct X`.
- Séquences, pour identifier le code correspondant aux changements, i.e., les utilisations du paramètre qui est supprimé (`hostno` dans l'exemple). Ces séquences sont symbolisées par `....` Les auteurs indiquent que les opérations sont effectuées sur le graphe de contrôle de flux, ce qui permet de s'abstraire du code pratique et de gagner (encore une fois) en généricité. Dans le cas où il faut appliquer un changement partout dans une région, l'opérateur `<...`changements`...>` permet de le faire.
- Isomorphismes. Ce sont ce que les auteurs appellent des « équivalences sémantiques ». Un exemple simple concerne la manière de vérifier un pointeur `NULL` : `!pointeur`, `pointeur == NULL`.

Par ailleurs, les auteurs fournissent quelques informations quantitatives sur les correctifs sémantiques en reprenant l'exemple utilisé précédemment (nommé `proc_info`) : le correctif « normal » fait 614 lignes soit une moyenne de 32.3 lignes par fichier, alors que le correctif sémantique (qui s'applique sur tous les pilotes, mêmes ceux non inclus officiellement) se limite à 33 lignes de code. Les auteurs proposent une version plus détaillée et à jour dans [127] qui est par ailleurs présentée en section 2.6.2.2.

2.6.1.5 « Semantic Patches, Documenting and Automating Collateral Evolutions in Linux Device Drivers » [124]

Dans cet article, les auteurs présentent leur outillage réalisé dans le cadre du projet Coccinelle, suite à l'introduction de la notion d'évolution collatérale dans [126] : les correctifs sémantiques. Cet ajout permet de répondre au besoin de gérer de manière plus efficace les évolutions collatérales : jusqu'à présent, elles sont effectuées à la main, par les développeurs. Ce dernier point est problématique, puisqu'il a été montré précédemment

que l'application manuelle de ces évolutions n'était pas exempte de problèmes : c'est une tâche rébarbative, d'envergure potentiellement large ; des erreurs sont introduites et ne seront corrigées que plus tard. Les auteurs proposent donc un langage de description des évolutions collatérales, chargé à la fois de documenter l'évolution, i.e., faire en sorte qu'elle soit plus facilement lisible ; mais également de l'automatiser, l'appliquer sur tout le code source du noyau. C'est le double rôle des correctifs sémantiques proposés. L'utilisation du format de `correctif` comme base est justifiée pour l'intégration et l'acceptation de la part des développeurs noyaux déjà habitués à travailler avec ces fichiers qui décrivent des changements dans le code, ligne par ligne ; l'idée est de passer à des correctifs ayant du *sens*, des correctifs *sémantiques*, qui décrivent plutôt *ce qu'il faut changer*, laissant à un autre outil la tâche de trouver *où il faut changer*. La manipulation décrite reste cependant axée autour des opérations d'ajout ou de suppression de ligne.

Afin de présenter leur proposition aux développeurs, les auteurs prennent comme exemple une évolution collatérale qui a déjà été effectuée, nommée « `video_usercopy` », correspondant à un changement (simplification de l'utilisation des `IOCTLs` par la factorisation dans une fonction générique de l'API) dans l'interface du sous-système multimédia du noyau :

- Ajout de nouveaux paramètres à une fonction
- Suppression du code correspondant à l'ancien comportement
- Utiliser les structures mises en place par la nouvelle fonction

Ils proposent donc un correctif sémantique qui effectue ces changements, en 50 lignes. Sur les 49 que les auteurs ont écrites et testées, 92% des changements ont été appliqués avec succès, les cas restants étant liés à des problèmes de lecture de fichiers.

2.6.1.6 « Understanding Collateral Evolution in Linux Device Drivers » [126]

Dans cet article, les auteurs présentent la notion d'« évolution collatérale », qui sert de base aux travaux pour construire le projet Coccinelle. Une définition est d'abord donnée : les évolutions collatérales sont les changements induits suite à des modifications dans les interfaces de programmation. Ces évolutions sont à distinguer de la refactorisation de code : il s'agit de maintenance de pilotes, souvent, pour adapter ces derniers aux évolutions des APIs. Plusieurs contributions sont présentées :

- Formalisation du problème des évolutions collatérales
- Clarification de la structure du noyau Linux vis-à-vis de celles-ci
- Présentation de trois exemples d'évolutions (Linux 2.5)
- Étude de la cinétique des évolutions collatérales
- Estimation de la charge de travail liée aux évolutions collatérales

Les auteurs définissent également la notion de protocole d'une bibliothèque noyau : il s'agit de l'enchaînement des appels de fonctions, la gestion des erreurs, etc. Une taxonomie des changements possibles est proposée :

- Fonctions exportées
  - Ajout/suppression d'arguments
  - Changement de nom
  - Changement de type de retour
- Callbacks

  - Ajout/suppression d'arguments
  - Changement de type de retour
- Structures
  - Découpage/recollage des structures
  - Ajout d'un niveau d'indirection
  - Conversion d'un membre en getter/setter
- Protocoles
  - Ajout/suppression d'appels nécessaires
  - Changement dans le séquencement
  - Changement des besoins de verrouillage
  - Ajout de vérification d'erreur

Les auteurs proposent d'étudier 72 évolutions collatérales découvertes dans le noyau Linux 2.5, qualifiées de représentatives ; dans un premier cas, il s'agit du changement du prototype d'une fonction dans la bibliothèque (suppression d'un argument). L'application de la suppression elle-même est assez simple, mais il faut également retirer le code lié à cet argument. Les autres cas sont également passés en revue.

2.6.2 Détection d'erreurs dans le code source

Suite au succès des travaux permettant l'application des évolutions collatérales de manière plus mécanique et sûre qu'en opérant les changements à la main, les auteurs font évoluer l'utilisation de leur outil vers la détection d'erreurs.

2.6.2.1 « Towards Easing the Diagnosis of Bugs in OS Code » [170]

Dans cet article, les auteurs étudient l'utilisation de Coccinelle dans l'objectif d'aider au diagnostic d'erreurs dans le code source d'un système d'exploitation. Il s'agit dans un premier temps d'exploiter le moteur pour trouver des erreurs ; puis de se servir des capacités de transformation pour insérer ou changer du code et avoir une prise en charge pro-active des erreurs potentielles. Les exemples donnés sont la mise en place de contournements, l'ajout de messages de debogue et de la journalisation, etc.

Pour découvrir des erreurs, les auteurs proposent *simplement* d'écrire des correctifs sémantiques. À titre d'illustration des exemples déjà publiés en `METAL` (dans le contexte du projet Coverity, confere section 2.5) sont donnés pour montrer la différence d'expressivité entre les deux langages. Les résultats en terme quantitatif (nombre d'erreurs trouvées) sont également proposés entre les deux approches. Ainsi, suivant les cas étudiés (gestion des interruptions, réutilisation de mémoire libérée, déréférencement de pointeurs `NULL`), Coccinelle fait soit aussi bien que `METAL` soit un peu moins bien, principalement pour des erreurs de lecture du code source. De plus, les auteurs notent que certaines erreurs non identifiées par `METAL` sont détectées par Coccinelle, grâce à son architecture. Un autre point de comparaison concerne le nombre de faux positifs : les deux outils présentent des taux à peu près similaires, avec cependant un petit désavantage pour Coccinelle. Pour améliorer ce point, des propositions d'amélioration sont formulées : rajouter une maitrise sur le flux (data flow) de l'analyse avec l'introduction d'un nouveau mot clef dans les correctifs sémantiques, **where**. Un exemple de faux positif que permet d'éviter cette modification

est donné avec la libération de mémoire d'un tableau, qui était considérée comme une utilisation de mémoire déjà libérée.

2.6.2.2 « Documenting and Automating Collateral Evolutions in Linux Device Drivers » [127]

Dans cet article, les auteurs prennent la suite du papier précédent, [124] mais présentent leurs résultats sous un angle plus formel. Ils présentent d'abord les quatre challenges inhérents à l'objectif du projet Coccinelle – documenter et automatiser les évolutions collatérales dans les pilotes de périphériques du noyau Linux – qui imposent des choix sur la réalisation :

- Facilité d'utilisation, pour maximiser les chances de son acceptation et utilisation par les développeurs
- Conservation des règles de codage, pour faciliter la maintenance du code source
- Généricité, parce que tout le code à mettre à jour n'est pas forcément limité au dépôt du noyau officiel
- Efficacité, pour permettre une utilisation interactive

Notamment, les auteurs présentent ici le langage SmPL qui n'est qu'évoqué dans [124] et le fonctionnement moteur de transformation correspondant. En résumé, Coccinelle va d'abord lire le code source, avec son propre analyseur (notamment pour être capable de traiter les directives du préprocesseur sans les interpréter, dans le souci du respect du code source), puis en extraire le graphe de contrôle de flux (*Control Flow Graph*). Le correctif sémantique est lu, et traduit en une formule de logique temporelle `CTL`. Ensuite, cette formule et le graphe du code source sont utilisés au sein d'un algorithme de model-checking, qui donne ainsi les nœuds du graphe qu'il convient de modifier, la partie du correctif sémantique correspondant, et les métavariables associées.

Les auteurs proposent des relevés de temps d'exécution sur l'instance `proc_info` (une évolution collatérale identifiée précédemment, concernant le sous-système SCSI) : 5838 fichiers à analyser pour appliquer cette évolution sur tout le noyau. Au total la modification est réalisée en 2 minutes, et 19 fichiers sont concernés.

Par la suite, les auteurs présentent deux nouvelles analyses : les méga-évolutions collatérales, ainsi que les évolutions collatérales qui génèrent beaucoup d'erreurs. Dans le premier cas, ce sont des évolutions collatérales sur des bibliothèques du noyau qui sont très utilisées, et qui vont avoir un gros impact. Une comparaison entre l'application manuelle de ces méga-évolutions et leur application avec Coccinelle est proposée : le taux de succès de Coccinelle est rapporté entre 63% et 100%, avec beaucoup de cas entre 90% et 100%. Dans 94% des fichiers, l'évolution est correctement appliquée. À noter que la notion de « bien appliqué » n'est pas explicitement donnée. Le temps d'exécution pour les différentes évolutions collatérales présentées est de l'ordre de quelques secondes, avec certains cas de l'ordre de la dizaine de secondes. Trois évolutions ont dépassé la limite maximale de 90 secondes. La complexité du code a un impact, de même que la taille du correctif sémantique.

2.6.2.3 « Hunting bugs with Coccinelle » [169]

Au travers de son mémoire de thèse, l'auteur nous narre sa traque passionnée des bogues avec Coccinelle. On remarquera là la malice de ce nom, puisque les coccinelles sont des insectes qui se nourrissent d'insectes que l'homme qualifie de « nuisibles ». La contribution majeure de l'auteur dans le domaine consiste en une analyse statique plus poussée au sein de Coccinelle, *data flow analysis*, i.e., une analyse de la propagation des valeurs au sein du graphe de contrôle du programme. Une justification de cette analyse est donnée en exemple : si l'on souhaite détecter des dépassements de tampons (*buffer overflows*), il est nécessaire de pouvoir traquer les tailles possibles d'un tableau ainsi que les valeurs potentielles qui sont utilisées pour accéder à celui-ci. L'auteur a au préalable cherché à se baser sur l'existant (GCC, CIL et CLang), mais aucun ne permettait simplement de faire ce qui était voulu : une seule analyse de ce type sera implémentée dans Coccinelle dans le cadre de son travail, permettant de raisonner sur les valeurs des variables ; il fournit en référence plusieurs autres analyses qui pourraient être implémentées. La technique implémentée est connue dans la littérature comme *Generalised Constant Propagation*, et son utilisation principale originale est pour optimiser l'utilisation des registres par un programme. Avec les besoins de parallélisation, la technique a été adaptée pour déterminer les plages de valeurs de variables et plus seulement de constantes. Et des références vers l'utilisation de cette technique pour trouver des bogues sont données. L'implémentation dans Coccinelle se base sur les travaux de [179], et un exemple sous forme de script Python qui se charge de détecter des dépassements de tampons est donné. Une variante est proposée pour détecter des dépassements de pile, de tas. L'analyse des résultats montre que les règles proposées fonctionnent, et ont été en mesure de trouver de *vrais* bogues présents dans le noyau Linux 2.6, l'expérimentation s'étant attachée à essayer de retrouver des bogues connus. Cependant, en conclusion, l'auteur souligne le fort taux de faux positifs qui vient détériorer la qualité de la détection, puisque dans un cas, on tombe à 0.2% de taux de succès (dans le cas de la détection des dépassements de tampons) ; sur une autre base de code (*tbaMUD*), le taux est de 67%. La détection de l'utilisation après libération est meilleure dans le noyau Linux, puisque le taux de succès est à 40%. La principale explication donnée pour les mauvais résultats sur la détection des dépassements dans le noyau est l'importance des simplifications effectuées sur l'algorithme dans l'implémentation réalisée ; cette dernière peut cependant être améliorée pour permettre des résultats plus fidèles.

2.6.2.4 « Generic Patch Inference » [4]

Dans [4], les auteurs présentent comment à partir d'une interface de programmation existante, que l'on fait évoluer, il est possible de générer automatiquement un correctif sémantique utilisable avec Coccinelle. Un langage plus simple que le langage C (cible de l'analyse) est également utilisé pour réaliser cet objectif. Les auteurs introduisent plusieurs aspects plus théoriques sur les correctifs tels :

- Les termes du langage, qui peuvent être accompagnés de méta-variables et former ainsi des motifs.
- Un correctif de remplacement de termes est défini comme une combinaison de motifs et donne la mécanique du remplacement de tout terme ou sous-terme qui correspond.
- Un correctif générique, qui est soit un correctif de remplacement de termes soit une

séquence de correctifs génériques

Par ailleurs, plusieurs propriétés doivent être respectées par les correctifs et posent également des contraintes :

- Sûreté, pour s'assurer que les correctifs n'effectuent que des modifications désirées.
- Compacité, pour avoir un correctif générique décrivant les modifications, et le plus petit possible.

Des travaux sur la génération des correctifs sémantiques ont été menés, dans [4], en suivant une idée simple : laisser le développeur corriger le problème dans un pilote, puis en déduire un correctif sémantique qui permettra de propager la correction. La principale contribution de ce travail est l'algorithme *SPFIND*, qui déduit le « plus grand sous-correctif commun » par rapport au langage de transformation d'un correctif générique. Par ailleurs, la notion de « plus grand sous-correctif commun » est introduite et définie formellement, et ce indépendamment du langage. Enfin, des exemples reprenant des problèmes déjà identifiés au cours de la vie du noyau Linux ont été utilisés pour valider la démarche. À l'heure de cet article, toutes les évolutions ne peuvent être prises en charge, parce que le langage d'expression des correctifs sémantiques est trop riche pour la version actuelle de l'algorithme *SPFIND*.

2.6.2.5 « A Foundation for Flow-Based Program Matching Using Temporal Logic and Model Checking » [38]

Dans l'article précédent [128] les auteurs faisaient référence à une variante de la logique temporelle CTL. Cette variante est présentée au sein de cet article, sous le nom de CTL-VW (*CTL with Variables and Witnesses*) et les contributions autour de cette dernière sont :

- Sémantique de la logique ;
- Algorithme de Model-Checking ;
- Correction de l'algorithme par rapport à la sémantique ;
- Traduction SmPL vers CTL-VW.

L'introduction de CTL-VW est faite après justification de la non-adaptation des logiques CTL-FV (*CTL with Free Variables*) et CTL-V (*CTL with Variables*). Bien que ces deux logiques ne soient pas bien adaptées pour les besoins de Coccinelle, les auteurs notent qu'elles apportent des éléments de base intéressants. Par ailleurs, lors de l'étape de transformation du correctif sémantique en logique CTL-VW, ce n'est pas tant la complexité (taille) de cette dernière qui va influer sur les performances que la complexité du code source analysé qui va réellement avoir un impact. D'ailleurs, c'est la lecture du code source qui prend le plus de temps par rapport à la totalité de l'exécution du processus.

Par ailleurs, deux versions de l'algorithme de model-checking existent : une version « basique », permettant de comprendre le fonctionnement, et une version « optimisée » (à la fois la traduction depuis SmPL vers CTL-VW et l'algorithme en lui-même) qui est celle implémentée dans Coccinelle, et dont le gain va jusqu'à 150 (l'analyse passant ainsi de 7 secondes à 0.9 secondes dans le pire cas) notamment grâce à la réduction des besoins mémoire.

2.6.2.6 « Enforcing the Use of API Functions in Linux Code » [93]

Dans cet article [93] les auteurs proposent d'étudier les « erreurs faites par les experts » sur les protocoles d'utilisation des APIs (gestion de la mémoire) et le respect des conventions de codage. Ces interfaces de gestion de la mémoire sont multiples (génériques ou spécialisées) et donc propices à confusion. Les auteurs étudient cinq hypothèses quant à ces APIs, via le taux d'erreur :

1. Les erreurs sont introduites par les développeurs les moins expérimentés ;
2. Les APIs courantes sont plus sujettes à des bogues mais rapidement corrigées ;
3. Les APIs rares sont peu sujettes à des bogues, mais plus difficilement corrigées ;
4. Les conventions de codage sont bien connues des développeurs expérimentés ;
5. La fréquence d'un problème varie inversement à son impact.

La modélisation est le correctif sémantique de l'outil utilisé, Coccinelle. Un schéma généraliste (indépendant de l'API) résume le protocole d'utilisation : choix de la taille et des drapeaux (`GFP_KERNEL`, `GFP_ATOMIC`, ...), construction de l'appel, vérification de la valeur de retour, libération du résultat lorsque nécessaire. Il est évalué pour les APIs : *Standard*, *Node* (architectures NUMA), *Cache*, *Boot* (gestion mémoire pendant le démarrage). Herodotos est utilisé pour suivre les erreurs sur les différentes versions du noyau, en exploitant les informations Git pour attribuer aux différents auteurs les erreurs. Pour chacune des hypothèses, les résultats obtenus sont :

1. Les erreurs sont plutôt introduites par des développeurs débutants. Cependant, pour les APIs *Node* et *Boot* le niveau moyen des développeurs est plutôt haut. Le faible échantillon et l'objectif de ces APIs explique ces résultats ;
2. L'API *Standard* est beaucoup plus utilisée que les autres, et on constate une fréquence d'erreurs plus élevée dans certains cas et plus faible dans d'autres. La durée de vie est généralement d'un an, alors que pour l'API *Boot* elle est à deux ans ;
3. Les APIs « rares » (*Node*, *Cache*, *Boot*) présentent le plus haut taux d'erreur ;
4. Les conventions de nommage sont mal connues des développeurs expérimentés ;
5. Plusieurs classes de problèmes présentés comme fréquents ont un impact important (plantage, gel, etc.).

Les auteurs proposent une classification des erreurs présentée dans le tableau 2.1 et les classent selon leur impact (nomenclature anglophone) : **buffer overflow** (*No deref.*, *Array alloc*), **coding style** (*Sizeof*), **plantage** (*NULL test*), **hang** (*Flag*), **memory leak** (*Free*) et **inefficient** (*Memset*).

2.6.2.7 *WYSIWIB : A Declarative Approach to Finding Protocols and Bugs in Linux Code* [91] et « WYSIWIB : A Declarative Approach to Finding Protocols and Bugs in Linux Code » [92]

Dans ces articles, les auteurs étudient l'identification et la vérification des protocoles, de manière la plus automatique possible. L'approche *WYSIWIB*, *What You See Is Where It Bugs* repose sur trois étapes :

1. Description générique d'une classe de protocole et collecte sur tout le noyau ;
2. Construction d'une collection d'erreurs potentielles pour ces protocoles ;
3. Application sur tout le noyau pour trouver d'éventuelles violations.

Nom	Description
Sizeof	Argument de taille passé comme taille d'un type
No deref.	Argument de taille ne correspondant pas au type pointé
Flag	Drapeau autorisant le verrouillage dans un contexte d'interdiction
Cast	Modification du type de retour d'une fonction d'allocation
NULL test	Absence de test sur la valeur de retour d'un pointeur
Free	Non libération d'un pointeur accessible seulement localement
Memset	Mise à zéro explicite et inutile
Array	Allocation d'un tableau sans utiliser une fonction appropriée

TABLE 2.1 – Classification et description des erreurs

L'idée est que celui qui connait les bonnes pratiques décrive les modèles d'erreurs pour pouvoir en automatiser l'application sur tout le noyau et ainsi détecter ces violations d'utilisations d'APIs. Par rapport aux travaux précédents, le langage SmPL évolue pour faire de la recherche de code. Un premier exemple est donné : le couple `nlmsg_new(), nlmsg_free()` s'utilise conjointement, et `kfree_skb()` ne peut être appliqué. Le correctif sémantique recherche les appels incohérents sur une même variable et indique l'emplacement de l'erreur. Pour cette étude, les auteurs proposent le cadre logiciel suivant :

Search Recherche de protocole propice aux erreurs ;

Instantiate Génération d'une collection de correspondances sémantiques ;

MultiSearch et **MakeBugReport** Recherche sur tout le noyau, préparation de résultats au format, respectivement, `Instantiate` ou rapport d'erreur pour `emacs`.

Les auteurs présentent ensuite trois études de cas en utilisant ce protocole :

– Détection des vérifications d'erreurs incohérentes
– Détection des fonctions d'allocation et de désallocation
– Détection d'erreurs, en se basant sur l'exemple `netif`

Dans les deux premiers cas, le processus est similaire : d'abord, une détection des protocoles, puis la mise en application pour trouver des erreurs. Le dernier cas repose sur le principe que plusieurs manières de retourner une erreur existent : un pointeur `NULL`, un booléen ou une valeur construite avec `ERR_PTR()`. La vérification doit être en accord avec la manière de retourner l'erreur et quatre catégories de fonctions existent, toutes avec très peu de faux positifs : `NULL` (2394 fonctions), `ERR_PTR()` (480 fonctions), `NULL` ou `ERR_PTR()` (100 fonctions) et celles pour lesquelles on ne peut pas décider (6940 fonctions) Par manque de temps, la validation des résultats a été faite en tirant 50 fonctions au hasard dans les trois premières catégories puis en vérifiant à la main les résultats. Au total, 535 erreurs sont identifiées comme n'effectuant pas de test avant déréférencement. L'étude sur les paires (allocation, désallocation) permet d'en identifier 602. La même vérification au hasard a été faite, montrant 37 protocoles corrects et 13 faux positifs, soit 26%. Cependant tous les protocoles ne peuvent être détectés.

En guise de conclusion, les auteurs notent que Coccinelle peut être utilisé pour ces travaux mais que les correspondances sémantiques peuvent être complexes. La seconde publication, [92] apporte une nouvelle information : une analyse du flux de données commence à être développée, pouvant notamment servir à faire de la vérification sur les indices de bornes de tableaux.

2.6.2.8 *Herodotos : A Tool to Expose Bugs' Lives* [130]

Dans [130], les auteurs du projet Coccinelle présentent leur outil Herodotos. Celui-ci permet :

- d'identifier les bogues dans le code source d'un projet, en utilisant les correctifs sémantiques et l'outil Coccinelle
- de faire un suivi de la vie des bogues identifiés au cours des versions

Étant donné l'utilisation de Coccinelle, on peut considérer que les erreurs sont modélisées par les correctifs sémantiques. Les auteurs font également mention de la classification des défauts dans plusieurs études antérieures qui citent toutes une référence commune, *Common Weakness Enumeration (CWE)*[3]. Cette classification est utilisée dans la section de leur rapport évaluant les résultats.

2.6.2.9 *Finding Bugs in Open Source Software using Coccinelle* [149]

Ce rapport de projet montre l'utilisation d'un outil connu pour l'analyse statique de pilotes dans un autre contexte, certes similaire : la découverte de bogues dans des logiciels n'étant pas des éléments du noyau. Il est rappelé que Coccinelle procède en une analyse statique du code, i.e., sans l'exécuter, à la différence d'une procédure de test « classique ». Bien que le logiciel analysé soit différent, puisqu'il s'agit d'OpenSSL, il est possible d'y appliquer *certaines* des règles proposées par Coccinelle : celles qui ne sont pas spécifiques au code noyau. Un exemple donné est le correctif sémantique qui vérifie que les pointeurs sont bien comparés à la valeur `NULL` et non `0`. Il est intéressant de constater que non seulement ces correctifs sémantiques ont du sens en dehors du code du noyau Linux, mais également que leur application au cas OpenSSL (sur une version stable) a permis de détecter des bogues réels, puisque déjà corrigés dans la version de développement. Plusieurs nouveaux correctifs sémantiques sont proposés : `malloc_style`, `use_after_free`, `openssl_malloc_free`, `openssl_malloc`. Il s'agit principalement de s'adapter aux règles de codage utilisées par le projet OpenSSL.

2.6.3 Suivi du cycle de vie des erreurs (Herodotos)

Une nouvelle utilisation de Coccinelle apparait suite à l'évolution du point de vue des auteurs, et ils proposent de greffer autour une suite d'outils chargée de permettre de suivre le cycle de vie des erreurs dans le code source : c'est le projet Herodotos.

2.6.3.1 « Tracking code patterns over multiple software versions with Herodotos » [131]

Dans cet article, les auteurs du projet Coccinelle présentent un outil, Herodotos, chargé de suivre les évolutions du code source et notamment les erreurs. Le lien avec le projet Coccinelle est trivial : afin d'être en mesure de suivre les évolutions dans le code source, Herodotos utilise un outil de mise en correspondance du code, Coccinelle dans le cas présent. L'étude présentée ne se limite pas au noyau Linux, mais considère également le

3. http://cwe.mitre.org

code des projets : API Wine, bibliothèque OpenSSL et le lecteur multimédia VLC. Les contributions dans l'article sont donc non seulement de proposer un outil et un processus (Herodotos) pour effectuer le suivi, mais également un langage de configuration pour qualifier l'environnement du test – chercher et visualiser les occurrences – ainsi qu'une stratégie pour corréler le code au travers des versions. Enfin, une étude « qualitative » est aussi proposée sur les types d'erreurs rencontrées.

Après avoir défini un environnement de suivi, le processus consiste à trouver des occurrences grâce à l'outil de mise en correspondance (Coccinelle dans le cas présent) ; cette collecte est effectuée sur toutes les versions du logiciel qui est étudié. Ensuite, il convient de faire la corrélation entre ces différentes versions : si le code ne bouge pas, elle est triviale ; malheureusement c'est rarement le cas et dès lors Herodotos repose sur `diff` pour être capable de suivre ces changements. Une fois les occurrences du motif et les différences entre chaque version identifiées, la corrélation entre motifs et versions est effectuée. Les auteurs introduisent ensuite le *Herodotos Configuration Language (HCL)* pour décrire un projet.

L'étude des différents logiciels libres que sont Linux, VLC et OpenSSL permet de trouver un total de 22 000 défauts répartis sur trois ans. 99% de ceux-ci sont inférés automatiquement par Herodotos : 19371 corrélations, et seules 282 sont proposées à l'utilisateur pour revue manuelle ; après annotation manuelle, 163 sont considérées équivalentes et 25 sans rapport. Il en ressort également que la typologie des erreurs est propre à chaque projet.

2.6.3.2 *Faults in Linux : Ten Years Later* [132]

Dans cet article les auteurs reprennent les travaux initiés par Chou et al. dans [46] et présentés en section 2.5.1, présentant ainsi une mise à jour de l'analyse avec leurs outils (Coccinelle, Herodotos), en reprenant les mêmes critères que l'article original. Celle-ci est nécessaire, une partie importante du code actuel n'existait pas lors de cette première étude. Les résultats proposés sont :

- Vérification de la pertinence du type d'erreurs analysées ;
- Taux de correction supérieur au taux d'introduction des erreurs ;
- Améliorations visibles dans `drivers/`, les besoins sont sur `arch/` et `fs/` ;
- Durée de vie des erreurs toujours proche et dépendante de l'impact ;
- Beaucoup de travail côté recherche à effectuer.

Une première partie de la publication concerne le protocole expérimental : les auteurs présentent les difficultés liées à l'absence de code disponible. Les vérificateurs retenus sont : **Block** (absence de deadlocks), **Null** et **INull** (pointeurs retournés `NULL`), **Var** (allocation statique $< 1k$), **Range** (validité des bornes et indices), **Lock** et **Intr** (libération des verrous, pas de double acquisition, restauration des interruptions), **Free** (réutilisation de mémoire libérée), **Float** (utilisations de nombres décimaux) et **Size** (allocation mémoire correspondant au type).

Par manque d'information `Real` et `Param` ne sont pas réimplémentées. La suite de l'étude s'assure que les mesures effectuées par Chou et al. sont comparables à celles obtenues ici. Certaines vérifications rapportent moins d'erreurs (`IsNull`), d'autres plus (`NullRef`) et finalement le résultat reste comparable. La plus grosse concentration d'er-

reurs est également détectée dans le répertoire `drivers/` (8 fois plus d'erreurs quand Chou et al. en ont 7 fois plus). Contrairement à Chou et al., beaucoup d'erreurs sont détectées pour `Var` dans les répertoires `arch/` et « autres ». Par ailleurs l'analyse de la distribution des fautes proposée par Chou et al. montrait une distribution logarithmique de $\Theta = 0.567$ ($\chi^2 = 79\%$). La même analyse donne ici $\Theta = 0.581$ et $\chi^2 = 81\%$, ce qui leur permet de conclure que les résultats sont comparables.

La suite de l'étude porte sur les noyaux 2.6.0 à 2.6.33 (6 ans) : 3860 erreurs, 2322 valides. Le nombre absolu de fautes détectées est à peu près constant dans chaque version : la densité de fautes chute de 51% entre 2004 et 2010 ; la raison est que chaque nouvelle version apporte de nouvelles erreurs, mais en corrige plus encore. Une erreur a une durée de vie moyenne de l'ordre de deux ans, l'importance de son impact limite celle-ci. Plus du tiers des fautes est introduit avec le fichier qui les contient, et 12% disparaissent en même temps que ce dernier. Les auteurs proposent d'essayer de prévoir la qualité du code source. Ils se basent sur trois métriques : code *churn* (nombre de modifications d'un fichier), âge des fichiers et taille des fonctions. Pour le noyau 2.6.33, la moitié des fichiers les plus récents présente un taux de fautes deux fois plus important que le prochain quart de fichiers récents ; pour les fichiers plus âgés, la relation est moins évidente : le taux de fautes atteint un minimum sur les fichiers de 5 ans, puis un maximum vers 9 ans, et ensuite diminue sous le minimum précédent au-delà de 15 ans. La taille de la fonction se corrèle au taux de fautes : plus elle est grande et plus elle présente d'erreurs.

2.7 Projet SLAM

2.7.1 *Boolean Programs : A Model and Process for Software Analysis* [12]

Ce document initie le travail qui donnera naissance au projet SLAM et SDV [16]. Contrairement à [13], il ne s'agit pas d'un article de recherche ou d'une présentation en conférence, mais d'un rapport technique sur l'utilisation des programmes booléens. Il commence par rappeler que le principal problème pour le model-checking reste de trouver un modèle pour le logiciel, ce pour quoi ils proposent l'introduction des programmes booléens. Un algorithme pour réaliser son model-checking est également proposé. Une première analyse fait le parallèle avec le model-checking de circuits tel qu'il est réalisé aujourd'hui : il existerait une représentation R pour modéliser le logiciel, à la manière des machines à état finis pour le matériel. Les programmes booléens permettent de faire ce R, avec une petite différence de taille par rapport à ces dernières : la prise en charge de la récursivité. L'algorithme de model-checking proposé se base sur le problème *Context-Free-Language Reachability*. En fait, le problème *Boolean Program Reachability* est posé : Soit une instance $< B, l >$ avec B un programme booléen et l un label tel que $l \in L(B)$. La réponse au problème est « oui » si l est atteignable dans B et « non » si l n'est pas atteignable dans B. L'auteur montre que le problème d'atteignabilité dans les programmes booléens se réduit au problème $CFL - reachability$, et justifie ainsi l'utilisation d'un algorithme proposé pour ce problème pour effectuer le model-checking. Les détails sur la création des programmes booléens à partir du programme source sont donnés, ainsi que la manière d'éliminer les chemins infaisables dans le programme booléen. En conclusion, il est indiqué

les évolutions envisagées au moment de l'écriture de ce rapport, notamment :
- Prise en compte des pointeurs, pointeurs de fonction, allocation dynamique et concurrence ;
- Efficacité des algorithmes, notamment en utilisant des techniques de model-checking symbolique ;

2.7.2 « Checking Temporal Properties of Software with Boolean Programs » [13]

Un des problèmes principaux pour effectuer du model-checking consiste, comme cela a déjà été précisé, à définir le modèle. Cet article présente une manière de modéliser, les programmes booléens, en vue de réaliser du model-checking. Un algorithme pour effectuer le model-checking est également proposé. Cette modélisation est la base des futurs travaux de Microsoft. L'idée est de transposer ce qui se fait au niveau matériel (machines à état finis) pour le logiciel : une représentation cible que n'importe quel langage pourrait fournir. Un programme C, C++ ou Java pourrait être compilé en cette représentation. Contrairement aux machines à états finis, les programmes booléens sont capables de contenir de la récursivité. La suite consiste donc, à partir de ce modèle, à vérifier qu'un programme séquentiel P est en mesure de respecter une propriété ϕ. Un programme booléen B est construit à partir de P par raffinages successifs en conservant le graphe de contrôle du programme P, supprimant chaque déclaration de variable, et en remplaçant chaque assignation par une opération vide (« skip »). D'autres modifications sont également réalisées. À chaque étape du processus de raffinage, les chemins infaisables dans B sont supprimés de P tout en vérifiant que les chemins réalisables sont conservés.

2.7.3 « Automatically validating temporal safety properties of interfaces » [11]

Le sujet étudié consiste à vérifier la *bonne utilisation* des interfaces dans le noyau de Windows. Cette vérification est effectuée à l'aide de SLAM[14]. Une contribution notable est de montrer qu'il est possible de vérifier des propriétés de sûreté grâce au Model-Checking, sans que l'utilisateur n'ait besoin de fournir des annotations ou des abstractions. C'est une étape importante, puisque cela indique que l'on peut amener la vérification sur une base de code existante et dans laquelle il n'est pas envisageable (que ce soit pour des raisons techniques, politiques ou pratiques) d'imposer aux développeurs l'ajout d'annotations, ou l'écriture d'abstractions quelconques comme cela peut être le cas pour le noyau Linux. Cela ne veut pas dire que la vérification se fait ex-nihilo : il convient malgré tout de décrire les propriétés à vérifier. Cette étape se fait avec le langage SLIC, un langage de description d'interfaces qui est décrit dans le papier. La méthode permet notamment d'obtenir un contre-exemple en cas de propriété réputée violée. Tout s'articule autour de trois outils développés :
- C2BP, un outil transformant un programme C en programme booléen [13] ;
- Bebop, un outil permettant le Model-Checking de programmes booléens ;
- Newton, un outil pour affiner le programme booléen par rapport à la faisabilité des chemins dans le programme C.

La cible privilégiée d'application dans ce cas précis est les pilotes exploitant l'interface Windows NT I/O. Un exemple réalisé avec un pilote PCI est proposé, montrant les différentes étapes (instrumentation, bogue, contre-exemple). Des détails sur l'implémentation efficace de Bebop sont donnés. Une étude plus réelle est proposée, pour vérifier deux types de propriétés :

- Séquences d'utilisation du verrouillage ;
- Fonctions `discorrectif` devant soit compléter une requête, soit la mettre en attente pour un traitement futur.

La conclusion rappelle l'inspiration originale, à savoir les travaux [61]. Mais l'introduction des programmes booléens avec Model-Checking permet d'aller beaucoup plus loin, grâce à *Bebop* et *Newton*.

2.7.4 « The SLAM project : debugging system software via static analysis » [14]

Dans cet article les auteurs présentent l'objectif du projet SLAM comme permettant de trouver les erreurs dans un logiciel via de l'analyse statique : il s'agit de vérifier la bonne utilisation des bibliothèques, notamment dans les pilotes du noyau Windows. Un élément clef dans la capacité à pouvoir traiter de plus gros volumes de code est la complexité de l'abstraction en programme booléen.

Par ailleurs, trois challenges importants pour la suite du projet sont énoncés :

- Le fardeau des spécifications : la création de spécifications correctes prend beaucoup de temps. Il faut donc capitaliser dessus.
- Le fardeau des annotations : annoter consiste pour le programmeur à rajouter des informations de vérification. Dans SLAM les invariants sont déduits automatiquement.
- Gestion de la sortie : il est important de donner de bonnes informations sur les erreurs. Il faut qu'elles soient à la fois courtes et détaillées.
- Robustesse, complétude et utilisabilité : une analyse est robuste si toute erreur réelle est remontée, elle est complète si toute erreur rapportée est une vraie erreur et utilisable si elle trouve des erreurs qui intéressent les développeurs.

2.7.5 *The Fugue protocol checker : Is your software Baroque ?* [55]

Dans cet article, les auteurs présentent un autre outil chargé de faire de la vérification de protocole. Ce sont des travaux parallèles à SLAM commencés simultanément après une réunion du groupe *Software Productivity Tools* en 1999, posant les bases de la vérification des pilotes chez Microsoft : trois projets avaient été démarrés, dont SLAM :

- Vault est un langage de programmation où des pré- et post- conditions sont définies sur les types de données. Ces travaux ont été réutilisés dans le cadre du *byte-code MSIL* pour la machine virtuelle *CLR*. Un greffon Visual Studio, *Fugue*, a été créé.
- ESP visait le même objectif que SLAM, mais avec une implémentation et des choix différents pour l'analyse statique mise en œuvre. Il est difficile d'obtenir plus d'informations sur ce projet.

C'est donc ce projet Fugue qui est présenté ici : un outil de vérification statique pour les langages compilés sur la *CLR*, et qui utilise des spécifications et annotations données par le développeur. Les exemples de vérifications effectuées sont : non-libération d'une ressource, utilisation après libération, mauvaise utilisation d'une bibliothèque, etc. Sur les objectifs, on peut constater que SLAM vise les mêmes, la différence se fait donc sur les moyens : le développeur doit ici écrire des spécifications dans son code. À hypothèses différentes, résultats différents, Fugue est capable d'analyser 13 000 méthodes en moins d'une minute (bibliothèque principale de la *CLR*, `mscorlib.dll`). Les spécifications sont de deux types :

- Protocole de ressources : indiquer quelles fonctions permettent d'acquérir une ressource, et lesquelles permettent de la libérer ; ceci permet à Fugue de vérifier qu'aucune ressource n'est utilisée après sa libération et que toutes sont libérées
- Protocole de machine à états : spécifier l'ordre d'appel des fonctions d'une bibliothèque. Un exemple est donné avec la gestion des *Sockets* : on ne peut pas appeler la méthode `down` avant d'avoir eu une connexion établie, on ne peut pas se connecter sans avoir défini d'hôte ni de port.

Les auteurs proposent un algorithme de vérification du modèle utilisé pour le tas, constitué d'un environnement de types et d'un ensemble de capacités. Plusieurs exemples de vérification de bibliothèques sont donnés, dont l'utilisation du composant `ADO.NET` qui gère l'accès aux bases de données.

2.7.6 « SLAM and Static Driver Verifier : Technology transfer of formal methods inside Microsoft » [17]

Dans ce rapport technique, les auteurs relatent la mise en place technique de leur outil de vérification de pilotes et documentent son contexte : échecs rencontrés, motivations, succès, et politique. Les besoins justifiant leur approche sont :

- Vérification de pilotes critique mais accessible et permet des gains ;
- S'appuyer sur des méthodes formelles connues et éprouvées ;
- L'environnement de la recherche au sein de leur société ;

Les pilotes sont une des clefs de la stabilité de la totalité du système : pour Windows XP ils sont responsables de 85% des plantages du système [15]. Une API spécifique existe pour les pilotes, Windows Device Model (WDM) et expose plus de 800 fonctions, proches de ce qui se trouve dans l'espace noyau Linux (allocation mémoire, entrées/sorties asynchrones, processus, évènements, verrouillages, files, etc.), une différence majeure étant leur stabilité Windows. C'est nécessaire avec le modèle de développement fermé adopté par Microsoft, où le travail d'écriture des pilotes est confié à des développeurs tiers qui n'ont pas accès au code source. L'origine de SLAM remonte à une session de réflexion au sein de la société : trois projets permettant de faire de la vérification d'utilisation d'interfaces ont été lancés, SLAM, Vault et ESP. Ils sont présentés dans la section 2.7.5. La différence réside dans la manière d'exprimer les règles. À la date du rapport, les deux autres projets étaient encore actifs. Vault est utilisé dans MSIL (*Microsoft's Intermediate Language*, une brique de la CLR, *Common Language Runtime*). Aujourd'hui seul le projet SLAM [4] semble avoir continué, les autres n'étant pas présents sur le site Microsoft Research. Ces projets annexes ont influencé SLAM.

4. http://research.microsoft.com/en-us/projects/slam/

2.7.7 « Thorough static analysis of device drivers » [15]

Dans cet article les auteurs présentent la plateforme de vérification statique de tous les modules noyaux pour Windows, *SDV, Static Driver Verifier*. Ils posent comme hypothèse que le nombre d'états à analyser est assez faible par rapport au nombre d'états au total du pilote, et que la difficulté réside dans la séparation de deux espaces d'états : « utiles » et « inutiles ». Pour un pilote de 100 000 lignes dont on veut vérifier la correcte utilisation d'une bibliothèque de verrouillage (*spinlock*), à chaque ligne on garde l'état du verrou (un booléen), soit 200 000 états à conserver, ce qui est admissible. Une première contribution est la présentation de l'outil d'analyse du code source. Un retour d'expérience pour définir un ensemble de règles correspondant à ce qu'est un *bon* pilote (*Windows Driver Model*). Une étude de l'exécution de l'outil d'analyse statique sur un ensemble de 126 pilotes, avec un jeu de 60 règles, et un ensemble de 40 pilotes *Kernel-Mode Driver Framework*, avec un jeu de 40 règles. Entre 75% et 80% des erreurs détectées sont confirmées. Les autres sont soit dues à des erreurs dans les règles ou dans l'environnement.

L'environnement est modélisé de manière « hostile », le code du pilote étant sollicité dans beaucoup de cas limites afin de pouvoir faire ressortir un maximum d'erreurs. Un élément important pour parvenir à ce point est l'introduction de non-déterminisme dans le modèle, pour simuler l'exécution de chemins inattendus. Selon les auteurs, la combinaison de ce non-déterminisme, d'analyse statique et d'exécution symbolique permet de couvrir tout le code exécutable du pilote. Quelques-unes des règles sont proposées comme exemple :

- Vérifier le comportement de la méthode `AddDevice` : elle doit appeler certaines APIs ;
- Vérifier l'absence de multiples requêtes d'entrées/sortie avec `IoCompleteRequest` ;
- Vérifier l'utilisation correcte des verrous tournants ;

Les pilotes qui ont été analysés pendant l'étude faisaient entre 48 et 130 000 lignes de code et 12 000 en moyenne : entrées/sorties, stockage, USB, firewire, clavier, souris, etc. Les exemples du Driver Development Kit ont également été analysés. Sur ces 126 pilotes, 206 erreurs ont été détectées. 65 ont été étudiées de près, dont 53 étaient pertinentes et 12 des faux positifs. Ces erreurs ont été générées par 40 des 60 règles. Pour l'analyse des pilotes KMDF, 20 sont utilisés et 18 erreurs détectées : 12 pertinentes et 6 faux positifs. Les auteurs proposent un petit aperçu des erreurs qui ont été trouvées, présentent et analysent les raisons des faux positifs :

- Modèle C du code noyau Windows très simplifié ;
- Nombreux cas limites dans les règles d'utilisation des APIs ;
- Comportement différent quant à l'ordre d'exécution de l'outil de vérification ;

Les auteurs concluent à un succès de 93% des tests de SDV. Sur les 118 restants, il y a 32 erreurs, échec dans 26 cas et dépassement du temps limite (2000 secondes) dans 64 cas. Sur les 717 tests où SDV s'est bien exécuté, les auteurs documentent le comportement de la boucle d'affinage : en moyenne, 5.63 itérations sont nécessaires – i.e. 5 chemins infaisables ont été trouvés et éliminés avant de pouvoir établir une preuve ou un chemin d'erreur – (écart-type 11.21, médiane 2). Le temps d'exécution moyen est de 101 secondes (écart-type 267, médiane 17). Environ 500Mo de mémoire sont consommés. Les auteurs placent quelques limites de leur outil : la mémoire n'est pas vérifiée ; il n'y a aucune prise en compte de la concurrence ; les entiers sont considérés non bornés, des erreurs de dépassement sont donc possibles ; il n'y a pas de validation de la vivacité du module noyau.

2.7.8 « The Static Driver Verifier Research Platform » [18] et « A Decade of Software Model Checking with SLAM » [16]

Cet article de synthèse reprends des éléments déjà évoqués autour du projet SLAM de Microsoft, et propose de prendre du recul sur la démarche qui a mené à la concrétisation de ce projet. La méthodologie d'abstraction depuis le programme C en un programme booléen est rappelée, ainsi qu'une explication un peu plus détaillée que ce qui était jusqu'à présent disponible sur *SLIC*, le langage de description utilisé pour décrire les règles à respecter. Par ailleurs, une représentation synthétique et claire du processus CEGAR (*Counter-Example Guided Abstraction Refinement*) est proposée, permettant de bien saisir le mécanisme de fonctionnement et qui explicite l'origine de certains soucis avec quelques pilotes : la boucle de vérification s'emballe, et il est impossible de mener à terme le processus. Heureusement, ce cas est très rare. Au-delà de l'aspect purement « retour sur l'utilisation des outils », se trouve une discussion sur le côté plus « social » au sein du processus de développement de la place du model-checking. Notamment, l'outil SDV (*Static Driver Verifier*) qui utilise SLAM et qui est intégré au kit de développement de pilotes depuis longtemps (distribué publiquement dès les versions beta du kit pour Vista). Sur les règles utilisées par SLAM, il est intéressant de noter que si les premières ont été écrites par l'équipe elle-même sur indication des experts des APIs de pilotes, rapidement le besoin est arrivé de faire en sorte que « n'importe qui » soit en mesure de les écrire. Sur les 470 règles maintenant présentes, seules 60 ont été écrites par les auteurs de SLAM. Plus intéressant encore, la complexité de l'écriture de certaines règles a influée sur les interfaces pour les pilotes pendant la réflexion sur Windows Drivers Foundation, amenant à la simplification de ces dernières.

Dans [18, 16], les auteurs du projet SLAM présentent la plateforme de recherche de vérification de pilotes *SDVRP, Static Driver Verifier Research Platform*, prenant la suite de *SDV, Static Driver Verifier* permettant de vérifier des propriétés de sûreté temporelle : il s'agit de s'assurer que les interfaces que le noyau met à disposition des pilotes sont bien utilisées. Le côté plateforme de recherche apporte un changement majeur et la destine aux chercheurs : le modèle de la plateforme et les règles des APIs sont modifiables dans ce cas. Selon les auteurs, cette modification permet d'appliquer la vérification à d'autres éléments que les seuls pilotes de périphériques. La plateforme est modélisée par deux composants :

- Le *platform manager model* qui va « stimuler » le module ;
- Le *platform API model* propose une implémentation (simplifiée) des APIs.

2.8 Projet {CP,PR}-Miner

2.8.1 « PR-Miner : Automatically extracting implicit programming rules and detecting violations in large software code » [104]

Dans [104] les auteurs utilisent de la fouille de données pour : identifier les règles de programmation implicites et vérifier qu'elles sont correctement appliquées. Ces objectifs sont proches de ceux Coccinelle mais l'originalité réside dans l'aspect découverte de règles implicites, qui intervient plus tard pour Coccinelle. Les auteurs donnent un exemple de la construction d'une règle avec un pilote SCSI. Les fonctions s'enchaînent avec certains types : `alloc`, `add`, `scan` et `Scsi_Host`. Dans cet exemple, 14 ($2^k - 2$, k le nombre

d'éléments) règles peuvent être construites à partir de ce ensemble en partitionnant les trois fonctions et le type en deux sous-ensembles. Un exemple est {`add`}⇒{`alloc`, `scan`, `Scsi_Host`}. Une optimisation est proposée reposant sur la notion de fermeture largement employée par les auteurs et présentée dans [70]. Un taux de confiance est calculé pour chaque règle. Cette approche est sujette à des faux positifs.

Pour avoir un système assez efficace, il n'est pas nécessaire de vérifier toutes les règles de programmation. Une limite basse est définie pour la confiance d'une règle en deçà duquel, elle est éliminée de la phase de génération. Si une règle est à un niveau de confiance de 100% alors il n'est pas nécessaire de la vérifier puisque cela indique l'absence de violation. Seul l'intervalle de confiance $[t; 100\%[$ nous intéresse. Un exemple de détection est proposé : soit la règle $\{a, b\} \Rightarrow \{d\}$ avec un support de 100 et $\{a, b\}$ a un support de 101. Alors on sait que seul un cas sur 101 présente $\{a, b\}$ mais pas $\{d\}$. Ce dernier cas viole donc la règle. Si par contre le support de $\{a, b\}$ est de 200, alors la confiance de la règle tombe à 50% et elle sera éliminée. Il n'y a donc pas de modélisation précise du concept d'erreur ou de bogue. Après élimination des faux-positifs les erreurs sont triées par ordre de confiance. Une étude du support des règles montre que le nombre de règles diminue lorsque le support augmente, suivant approximativement une loi de décroissance exponentielle. Une autre étude sur la taille des règles montre que 14% des règles ont seulement deux éléments, et 9% des règles en ont plus de 10 (un exemple est donné sur un périphérique PCMCIA). Les auteurs ont regardé de près les 60 premières violations, pour essayer de trier les vraies erreurs des faux positifs, et trouvent plusieurs bogues (16 validés dans le noyau Linux, 6 dans PostgreSQL et 1 dans Apache). Ils mettent également en avant que leur méthode détecte des bogues *sémantiques* qui sont plus difficile à repérer que les erreurs plus basiques, surtout quand les règles incluent plus de trois éléments. 20 erreurs potentielles dans le noyau Linux sont également rapportées, et 24 faux positifs identifiés. Les chiffres sont potentiellement pires encore pour le cas de PostgreSQL et Apache, avec respectivement 45 et 6 faux positifs. Plusieurs causes à ces problèmes d'identification sont suggérées : le copier-coller engendre des faux négatifs, les macros du langage C interfèrent en générant des faux positifs ou des faux négatifs, des collisions dans les noms de fonctions peuvent se produire pendant la génération des règles, les règles de programmation inter-procédurales ne peuvent pas être prises en compte et la granularité de l'analyse (au niveau fonction) fait que si une fonction a deux parties dont l'une qui viole la règle et l'autre non, alors aucune erreur ne sera remontée.

2.8.2 « CP-Miner : a tool for finding copy-paste and related bugs in operating system code » [105] et « CP-Miner : Finding Copy-Paste and Related Bugs in Large-Scale Software Code » [106]

Dans cet article les auteurs présente une technique de détection d'erreurs issues de copier-coller de code. Des études récentes, dont [46], pointent que cette pratique comme une part significative des problèmes rencontrés dans le code des systèmes d'exploitation dont les pilotes de périphériques. Chou et al. indiquent [46] par exemple pour le répertoire `drivers/i2o`, 34 erreurs sur 35 ont pour origine un copier coller : une erreur a été copiée à 10 endroits et une autre à 24. Distinguer l'original de la copie n'est pas l'objectif ici. L'approche proposée par auteurs est capable de traiter en moins de 20 minutes une base

de code de 3 millions de lignes (Linux, FreeBSD), et moins d'une minute pour analyser le code source d'Apache ou de PostgreSQL. Ils proposent une étude de l'utilisation de copier/coller dans différentes bases de code, systèmes d'exploitation compris. Trois techniques de détection de code ayant été copier/coller existent : chaînes, arbre d'analyse lexicale, jetons.

Ils retiennent une technique basée sur des jetons car elle permet d'identifier plus finement des changements, et elle n'est pas sensible aux faux positifs. Le reste de l'analyse repose sur le problème *frequent subsequence mining* et l'algorithme utilisé est CloSpan[192]. Le processus d'identification se décompose en :

- Construction d'une base de données de séquences à fouiller ;
- Recherche de segments copier-collés de base
- Suppression des faux positifs
- Composition de plus grands segments à partir de ceux de base ;

L'identification d'erreurs liées au copier coller repose sur la mise en correspondance des identifiants dans les sections de code identifiées pour vérifier la cohérence des changements d'identifiants. Cela peut indiquer que le développeur a oublié de changer le nom d'une variable après avoir fait une copie. Des bornes sont posées : trop d'incohérences peuvent indiquer, par exemple, que les deux morceaux de code ne sont pas liés. Les auteurs calculent donc un « ratio de non changés » en comptant le nombre d'identifiants inchangés rapporté sur le nombre total d'identifiants, et plus il est faible, plus il y a de chance qu'une erreur existe, sauf si le ratio est nul auquel cas les deux morceaux ne sont pas liés. Un seuil existe pour les valeurs de ratio qui trop importantes. Les auteurs auditent le code de Linux, FreeBSD, Apache et PostgreSQL. Ils y trouvent de nombreuses erreurs mais très peu sont confirmées. L'origine de ces faux positifs est analysée par les auteurs : mauvaise identification des segments copier-collés (beaucoup de `case` ou `if`), importance de l'ordre dans les expressions. Ils comparent CCFinder et CP-Miner : le premier identifie toujours plus (entre 17% et 52%) d'erreurs, quel que soit le corpus utilisé.

Vient enfin l'étude sur l'utilisation du copier-coller dans les différents systèmes. Les points étudiés sont : taille des segments et granularité, modifications dans les segments, copie de code entre modules et évolution dans le temps des pratiques. Les morceaux de code qui sont copier-collés sont de petite taille, la majorité ($60-64\%$) étant composée de 5 à 16 segments, et même 35% à 40% sont compris entre 5 et 8 segments. La couverture, mesurée par l'indice `CP_Coverage` (défini comme étant le rapport entre le nombre de lignes de code dans les segments identifiés et le nombre de lignes de code dans la totalité du code analysé), suit la même distribution dans le cas de Linux et de FreeBSD. Dans un peu plus de 6% des cas, plus de 8 copies du code ont été effectuées, même si dans 60% des cas seule une copie existe. La granularité est également similaire que ce soit dans le cas de Linux ou de FreeBSD : autour de 9% des segments identifiés sont de l'ordre du bloc de base, et entre 11% et 13% sont de l'ordre de la fonction.

Les auteurs s'intéressent aux modifications effectuées entre les segments : il en ressort qu'un tiers n'est pas modifié, dont 8% n'ont qu'une seule modification, et les deux tiers restant nécessitent des changements dans les identifiants. De plus, un tiers des segments présente une altération (ajout, suppression, modification). Pour l'étude des copies de code entre modules, les auteurs découpent Linux et FreeBSD en 9 catégories et pour Linux, la majorité du code fautif est situé dans `drivers` et `arch` (71% au total). Pour FreeBSD c'est

le module `sys` (60%). L'explication est la structure des pilotes, tous similaires, qui incite les développeurs à reprendre par copier-coller. Enfin l'étude de l'évolution des pratiques montre que la croissance du code source en matière de nombre de lignes de code est plus rapide que celles des copier-collers.

2.9 Projet Nooks

2.9.1 « Nooks : an architecture for reliable device drivers » [172] et « Improving the reliability of commodity operating systems » [171]

Dans [172, 171] les auteurs présentent un mécanisme modifiant les interfaces pour améliorer la fiabilité du système, en permettant de relancer un pilote en cas de problème. Cette approche rappelle une des propriétés des micro-noyaux. Le principe du *Nooks Isolation Manager* est détaillé dans l'article [171]. L'approche proposée se limite aux pilotes, à cause de la question de la sécurité : les auteurs considèrent que la plupart des pilotes sont dignes de confiance, et dès lors, l'exécution sûre d'extensions du noyau se limite à la sûreté de fonctionnement, et non à la sécurité. Différentes approches sont comparées : ajouts de vérifications autour des interfaces noyau (*kernel wrapping*), protection mémoire matérielle, limitation de privilèges, pour empêcher l'utilisation d'instructions privilégiées, protection logicielle des fautes, pour garantir la sûreté des adresses et des instructions, utilisation de langages plus sûrs. Chacune présente des avantages et des inconvénients : modification du code, récupération depuis les erreurs simples, performances sur des petits et des gros ensembles de données, protection contre la corruption mémoire et les deadlocks. Les auteurs introduisent *nook* : environnement protégé pour l'exécution de pilotes. Tous ne s'exécutent pas forcément dans un tel environnement (pilote SCSI donné en exemple), et certains peuvent, pour des raisons de performances s'exécuter dans un *nook* commun. Cette interface s'intercale en transférant les interruptions et en gérant l'accès aux registres ainsi que les appels noyau-pilote. L'architecture de *Nooks* répond à deux impératifs : être résistant aux fautes (i.e., ne pas forcément chercher à gérer **toutes** les fautes), ne pas protéger des abus. Les auteurs dérivent de ces principes trois buts : **isolation** (détecter les problèmes avant qu'ils n'infectent d'autres parties du noyau), **récupération** (permettre à une application de continuer à fonctionner malgré un module défaillant), **compatibilité ascendante** (modifications des pilotes réduites au strict minimum).

Seule la couche *Nooks Isolation Manager (NIM)* présentée dans [171] nous intéresse. Elle propose quatre services : protéger les espaces mémoires, protéger les appels de fonctions entre modules et noyau, suivi des ressources du noyau utilisées par les modules et processus de récupération après erreur. Afin d'évaluer leur implémentation, les auteurs injectent des fautes : modification du registre (source ou destination), modification d'un pointeur, réutilisation de la valeur actuelle d'un registre plutôt que le paramètre passé, suppression d'une branche d'instructions, inversion de la condition d'arrêt dans une boucle, modification d'un bit dans une instruction, et suppression d'une instruction. Sur 365 plantages détectés, l'outil a permis d'en éviter 360. L'impact sur les performances est étudié dans [172] (2002) : la bande passante mesurée TCP augmentent avec *Nooks* (de 611MiO/s à 630MiO/s), la latence augmente peu (de 236µs à 241µs). Le débit plus élevé s'explique par le nombre moyen de paquets reçus pour chaque interruption (de 7.1 à 8.6).

2.9.2 « Recovering Device Drivers » [115]

L'approche proposée dans [115] est similaire à celle de *Nooks* : réaliser une couche entre pilotes et applications, *shadow driver*. C'est un pilote fantôme qui vérifie le bon fonctionnement du pilote réel, le relançant au besoin, et assurant un service minimum ne perturbant pas le système pendant le temps nécessaire à la coupure. Quatre grands principes sont au cœur du système : séparation entre les erreurs et les utilisateurs des pilotes, processus de redémarrage centralisé et générique, pas de perturbation en utilisation nominale. Les auteurs ne traitent que des pilotes de périphériques permettant faciliter l'objectif de non-perturbation. La plupart des problèmes dans les pilotes sont liés à des entrées ou des évènements inattendus : déterministes (déclenchés par une séquence de requêtes ou d'entrées/sorties), transitoires (effets de bords d'autres éléments du noyau). Ce premier cas est complexe : rejouer la requête peut re-déclencher le problème.

D'autres erreurs, *fail-stop*, peuvent être détectées et arrêtées avant d'atteindre le pilote ou l'application. L'approche proposée ne fonctionne que pour des bogues transitoires et *fail-stop*. L'infrastructure *Nooks* est réutilisée pour la détection et l'isolation. La détection des erreurs se fait toujours sur accès à une référence mémoire invalide ou le passage d'un paramètre invalide. Le noyau est modifié : pour chaque pilote chargé, il existe un pilote fantôme. Tous les appels de configuration au pilote passent donc via le fantôme, qui enregistre les données. Pour certains périphériques (stockage), seul le dernier état est conservé. Les limitations de l'approche sont aussi présentées : le pilote doit pouvoir être retiré puis rechargé, si la communication entre noyau et pilote se passe hors des interfaces standard la méthode ne fonctionne pas, les erreurs ne doivent pas être irréversibles (corruption de données sur le disque), un état interne d'un pilote qui est déduit des entrées du périphérique ne pourra pas être restauré. La détection des erreurs a également ses limites : il est difficile voire impossible de détecter une valeur de retour valide mais incorrecte. Une solution proposée serait de faire une détection par **classe** de pilotes.

2.10 Projet Undertaker

Dans le cadre de la vérification de logiciels, le projet Undertaker apporte une autre approche, en s'interrogeant sur la configurabilité. L'exemple est pris du noyau Linux, qui est très modulaire et dont la modularité est croissante. Dans [165] (en 2010), les auteurs rapportent plus de 8000 « fonctionnalités », i.e., des possibilités de configuration. Dans un autre article [173] publié en 2011, les mêmes auteurs parlent maintenant de l'ordre de 10000 « fonctionnalités », sans rien changer autre que la version du noyau étudiée. Le problème de la configurabilité du noyau et des logiciels en général est introduit dans [165] : l'idée simple est de vérifier la cohérence entre les options de configuration activables (`Kconfig` au niveau du noyau) par l'utilisateur, et les options de configuration utilisées dans le code source du logiciel (dans le cas du noyau, ce sont des directives `#ifdef`). En effet, au cours de la vie du logiciel, des options peuvent disparaitre d'un côté ou de l'autre, entrainant des modifications dans le code compilé qui peuvent avoir des répercussions importantes, par exemple :

- Code mort (jamais activé)
- Code vivant (toujours activé)

Les auteurs rapportent un cas concret montrant l'intérêt : la fonctionnalité de branchement à chaud des processeurs est restée cassée pendant plus de six mois, faute à une directive de configuration mal orthographiée. Conséquence, un administrateur souhaitant bénéficier du changement de processeur à chaud sur une machine le supportant ne pouvait pas, ce qui peut être très problématique.

Les auteurs contribuent donc un système pour vérifier les incohérences. Trois niveaux sont identifiés : modèle, génération et implémentation. Le modèle est apporté par `Kconfig` et permet de décrire les options disponibles. Le niveau génération correspond à l'utilisation, par le processus de compilation du noyau (`Kbuild`), de ces directives pour savoir les objets (« unités de compilation ») à produire (sous forme de module ou bien directement dans le noyau). Le niveau implémentation correspond à l'activation conditionnelle dans le code source à l'aide de directives `#ifdef`. Quatre conditions sont proposées pour détecter les problèmes :

1. Chaque référence à une *option de configuration* doit être définie dans le *modèle* ;
2. Chaque *option de configuration* du *modèle* doit être utilisée dans le code source ;
3. Chaque *option de configuration* doit être sélectionnable par les outils manipulant le *modèle* ;
4. Chaque condition référençant une *option de configuration* doit être activée par au moins une confiuration valide obtenue à partir du *modèle* ;

L'analyse consiste ensuite à extraire du code source les informations liées à la compilation conditionnelle ainsi que les options disponibles dans le modèle. Une formule booléenne représentant le modèle est ainsi obtenue et permet de vérifier que l'implémentation (ce qui est extrait du code source) est correcte par rapport au modèle.

2.10.1 « Facing the Linux 8000 Feature Nightmare » [165]

Dans [165] les auteurs du projet Undertaker cherchent à identifier une autre catégorie d'erreurs présentes dans le code source du noyau Linux : celles liées à la configurabilité. Undertaker est présenté dans la section 2.10. Il s'agit de vérifier la cohérence entre les options de configuration pour le noyau qui sont à disposition de l'utilisateur via l'outil `Kconfig` et les directives de compilation conditionnelles présentent dans le code source. Quatre conditions de cohérence sont proposées en dehors desquels l'outil détectera une erreur :

1. Chaque référence à une *option de configuration* dans le code source doit être définie dans le modèle *Kconfig*
2. Chaque *option de configuration* définie dans le modèle *Kconfig* doit êre présente dans le code source
3. Chaque *option de configuration* doit être sélectionnable dans le processus de configuration proposé par les outils *Kconfig*
4. Chaque branche conditionnelle `#if` qui contient des *options de configuration* doit être activée par au moins une configuration valide produite par les outils *Kconfig*

Les deux premières conditions sont classées dans la catégorie référencement, les deux dernières sont des conditions sémantiques.

2.10.2 « Feature Consistency in Compile-Time Configurable System Software » [173]

Dans [173], les auteurs apportent plus de résultats pratiques et montrent l'évolution dans le noyau de ce type d'erreurs, ainsi que le problème de la remontée de ces erreurs : les auteurs ont proposé des correctifs pour corriger les défauts identifiés (123 correctifs). Le processus menant à la rédaction des correctifs est indiqué, et outre l'identification avec Undertaker, consiste à faire corriger les problèmes par deux étudiants. Ces derniers envoient ensuite leur contribution aux mainteneurs noyaux, qui peuvent accepter, refuser, ou demander des modifications. Cette manière de procéder permet d'évaluer la pertinence des défauts trouvés par Undertaker. Plus de 70% des correctifs ont amené une réponse : ils sont donc faciles à vérifier (30% ont eu une réponse en moins d'une heure), et pertinents (ils ont été acceptés). Dans quelques cas (15) les mainteneurs ont reconnu l'existence d'un problème, mais n'ont pas voulu le corriger : besoin de conserver des références obsolètes à des fins de documentation, et développement ayant lieu hors de l'arbre du noyau classique.

La raison de la présence de ce type de bogue n'est pas encore identifiée, mais une piste sérieuse concerne les copier/coller où l'auteur va prendre les directives de compilation conditionnelle en même temps.

2.11 Projet GCC MELT

2.11.1 « Les greffons du compilateur GCC : Votre compilateur GCC sur mesure ! » [42]

Cette communication présente les possibilités du compilateur GCC en matière d'extensibilité : deux types d'interfaces sont possibles, dont une qui nous met directement au cœur du processus de compilation. En fait, dans GCC, comme dans n'importe quel compilateur, ce processus comporte trois phases :

- Front-End, où les langages pris en charges sont lus et transformés en une première représentation : Generic dans GCC ;
- Middle-End, où tout un ensemble de passes est appliqué au code ; c'est là que se font les optimisations, vérifications, sur une seconde représentation : Gimple ;
- Back-End, où les cibles prises en charges sont traitées grâce à une représentation *Register Transfer Language*.

Les greffons GCC nous mettent au cœur de l'étape Middle-End, c'est-à-dire que l'on a accès à une représentation abstraite et indépendance du code, qui a tout de même des références vers ce qui est disponible dans le fichier source, et qui est classiquement utilisée par les passes d'optimisations. C'est l'endroit idéal pour faire de l'analyse statique avec ces greffons.

2.11.2 « Extending the GCC compiler with MELT to suit your needs » [158]

Dans cette communication, l'auteur présente le langage d'extension `MELT`, destiné à la réalisation de greffons pour le compilateur GCC, langage fonctionnel inspiré de Lisp permettant notamment de facilement faire de la mise en correspondance de motifs : ceci

permet de repérer des structures dans le code source. Le langage MELT travaille au niveau du code source après son traitement par le compilateur, et permet ainsi de réaliser des opérations similaires à ce que permet Coccinelle, présenté en section 2.6 ou ce que permet `METAL` dans le cadre de `xgcc` présenté en section 2.5.

2.12 Analyse statique (P.T. Breuer)

2.12.1 « Static deadlock detection in the Linux kernel » [37]

Le problème des interblocages est régulièrement étudié de part ses caractéristiques : une mauvaise utilisation des primitives de verrouillage peut entraîner ces interblocages, et ils sont souvent difficile à déceler par le test, courants et engendrent des problèmes gênants. Les auteurs de cet article s'intéressent au cas particulier des `spinlock`, une primitive de verrouillage du noyau Linux très utilisée et qui implémente un simple verrou tournant : ce mécanisme a un fonctionnement facile à comprendre, puisqu'il s'agit d'une attente active. Dans Linux, les verrous tournants ne sont pas récursifs, on ne peut donc pas les acquérir plusieurs fois, et si l'utilisateur essaye de le faire, une erreur sera déclenchée. Le noyau propose des outils pour faciliter la découverte de ce genre de cas, mais cela suppose d'avoir un cas de test reproductible. Une erreur très courante est le cas de s'endormir après avoir pris un verrou tournant : le processus qui l'a acquis n'est plus exécutable, et un autre peut tenter de prendre le même. Les auteurs définissent ensuite ce qu'est une portion de code « bien formée » : les appels aux primitives de verrouillage et de déverrouillage doivent être équilibrés, i.e., il doit y avoir autant d'appels pour verrouiller que pour déverrouiller. L'analyse se fait suivant les sorties empruntées par le code :

- *Normal* : le code arrive à la fin de la fonction et noté N ;
- *Return* : le code sort après un mot-clef `return` et noté R ;
- *Break* : le code sort après un mot-clef `break` et noté B.

À partir de ceci, les auteurs sont en mesure de calculer le déséquilibre d'appels. L'article ne propose pas d'autre résultat expérimental qu'un exemple sur le pilote `sbull`, qui est lui-même un exemple de pilote de disque, où trois fonctions du pilotes sont montrées comme fautives.

2.12.2 « One million (LOC) and counting : Static analysis for errors and vulnerabilities in the Linux kernel source code » [32]

Cet article prend la suite du précédent, et commence en rappelant bien que le sujet ne traite pas du model-checking, et propose une présentation plus complexe de la méthodologie : d'abord, on réalise une description de l'état initial (un exemple étant que « 0 verrou pris »). Ensuite, il faut construire une description de chacun des états atteignable. Ces descriptions sont des prédicats d'un ensemble très restreint, composé de conjonctions et de disjonctions de propositions simples (par exemple, de la forme $x <= a, x <= b$ avec a, b des constantes). Finalement, on obtient un cube situé dans un espace à n dimensions. L'intérêt de cette approche vient de la possibilité de vérifier l'inclusion d'un « cuboïde » p dans un autre q avec un algorithme de programmation linéaire, ce qui permet de valider ou non l'implication $p \leftarrow q$. Toute la logique introduite dans l'article précédent autour de

la notion de point de sortie (*Normal*, *Return*, *Break*) est décrite de manière plus théorique. Grâce à la logique introduite, une manière de détecter l'accès à de la mémoire déjà libérée est proposé :

- Sur un appel `kfree()`, on incrémente un compteur correspondant à la zone mémoire manipulée ;
- Lors de l'assignation d'une valeur, on vérifie que le compteur correspondant doit bien à 0, sinon il y a risque d'accès à de la mémoire déjà libérée.

L'application de cette règle sur 1151 fichiers source du noyau Linux 2.6.3 a permis de repérer des erreurs potentielles dans 30 d'entre-eux, dont trois contenaient de réelles erreurs. La répétition de cet exercice sur un noyau plus récent, 2.6.12.3 et sur un ensemble de fichiers plus large (1612) montre que toutes ces erreurs ont été corrigées : plus aucune n'est détectée. À noter que dans la première analyse, nombre de faux positifs sont remontés à cause d'un bogue. Les auteurs ont étendu la couverture de l'analyse de leur outil au cas de la détection de mémoire accédée après libération dans cet article. Ils annoncent avoir trouvé trois bogues de ce type par millier de fichiers dans le code source du noyau 2.6.

2.12.3 « Verification in the large via symbolic approximation » [35]

Suite aux travaux précédemment réalisés [37, 32], l'auteur présente la technique développée au cours de ceux-ci dénommée « approximation symbolique ». L'idée de base est de réimplémenter la sémantique du langage C utilisé pour construire le noyau Linux au sein d'un domaine symbolique avec une notion d'approximation de programme bien définie et en considérant :

1. qu'il ne fait pas sens de traiter en détails tous les états du programme pour des raisons d'explosion du nombre d'états
2. que l'approximation est contrôlée pour explorer plus de chemins et d'états que n'en existeront en réalité

Les approximations sont une forme d'interprétation abstraite, et la version abstraite ainsi obtenue d'un programme propose une correspondance entre un prédicat formel décrivant l'état courant et un prédicat formel décrivant l'état final. Un programme C ainsi abstrait peut être exécuté de plusieurs manières différentes résultant en diverses *perspectives* liées à des points précis du code source, formant des *jugements*. Chaque jugement dans une perspective précise et à un point donné fourni des informations sur la violation d'une assertion à cet endroit : par exemple, lire de la mémoire non initialisée. Globalement, les constructions de base du langage C sont traduites en des prédicats inspirés de la logique d'Hoare. Deux tableaux résument la construction du langage et de ses briques de base ainsi que les conditions de branches. La notion d'approximation est donnée formellement et sa définition est : *si* `a` *est plus spécifié que* `b` *alors on peut en dire plus à propos de ce que* `a` *fait qu'à propos de ce que* `b` *fait*. On retrouve également une définition plus formelle de la logique `NRBC` (*Normal*, *Return*, *Break*, *Goto*). L'article [36] intitulé « Detecting deadlock, double-free and other abuses in a million lines of linux kernel source » [36] des mêmes auteurs est une version plus orientée sur un cas pratique : *détection de deadlock, de double libération mémoire et quelques autres abus*. Une version plus étoffée et reprenant la totalité de l'historique du projet est par ailleurs proposée dans [33].

2.12.4 « Symbolic approximation : an approach to verification in the large » [34]

L'approche présentée dans ce papier n'est **pas** du model checking, mais une alternative permettant d'arriver à un résultat similaire, sans avoir l'épée de Damoclès que constitue le problème de l'explosion du nombre d'états. L'approximation implique nécessairement une perte de sémantique par rapport au programme. Par conséquent, il est nécessaire d'adapter l'approximation en ajustant la logique appliquée : pas de risque de faux négatifs, mais potentiellement des faux positifs. L'article fait évidemment référence au projet SLAM de Microsoft[17, 14, 16], indiquant par ailleurs que le sujet de cet article est dans la globalité similaire à SLAM, à l'exception du model-checking. Une description détaillée de la méthode d'abstraction est donnée, avec de solides fondements mathématiques, reposant notamment sur la logique d'Hoare. Il est intéressant de noter que, l'objectif étant de travailler sur du code du noyau Linux, les particularités liées au C GNU sont prises en compte. Par la suite, les auteurs s'intéressent aux programmes C sous la forme d'une boite grise : une composition entre une approche boite noire et boite blanche ; le contenu du programme n'est pas connu pendant l'exécution (boire noire), mais celle-ci peut-être arrêtée pour regarder les détails (boite blanche). En pratique, les points d'observations possibles sont les points de sorties exceptionnelles : *return*, *break* et *goto*. La logique précédemment introduite se voit donc étendue par la logique Normal, Return, Break, Goto (NRBG). Les instructions imposant des sauts en arrière ne sont pas prises en charge. Le tableau 4 est particulièrement intéressant, puisqu'un résume la logique augmentée de NRBG de manière claire et synthétique. Mais la logique en elle-même nécessite plus de temps pour être maitrisée réellement. En conclusion, il est indiqué que la méthode décrite fonctionne en temps raisonnable sur une base de code telle que le noyau Linux. Surtout, la logique d'approximation est configurée par un expert, mais ensuite il n'est pas nécessaire d'avoir des compétences précises pour appliquer les analyses. La notion d'expert reste vague, cependant on peut supposer qu'elle fait référence au code à tester, i.e. la personne doit bien connaître le code dont elle prépare l'approximation, puisqu'il y a un compromis à trouver entre approximation et perte d'information. Enfin, à noter que les analyses présentées par les auteurs ont permis de détecter deux cas de deadlock par milliers de fichiers analysés ainsi que trois fichiers pour mille qui effectuent des accès à de la mémoire déjà libérée.

2.13 Autres approches

2.13.1 Vérification de code et Model-Checking

Ces trois premiers articles sont regroupés car ils introduisent la notion de vérification de code par le model-checking. Le premier est à considérer comme fondateur de la technique de model-checking et les deux suivants permettent de comprendre les techniques.

2.13.1.1 « Automatic verification of finite-state concurrent systems using temporal logic specifications » [48]

Dans [48] les auteurs proposent de faire de la vérification sur un système concurrent à états finis. La spécification à vérifier est exprimée dans une logique propositionnelle temporelle à branches. L'algorithme proposé est de complexité linéaire suivant la taille du graphe à vérifier et la taille de la spécification, et les auteurs proposent de l'appeler un *model-checker* (la publication date d'avril 1986), et le modèle proposé est une structure de Kripke. La logique proposée pour décrire la spécification à vérifier est dans un premier temps CTL, mais par la suite, les auteurs proposent d'autres logiques qui sont plus expressives : CTL^*, CTL^+.

2.13.1.2 « Développement et validation de logiciels. Méthodes formelles » [24]

Cet article de la revue « Techniques de l'Ingénieur » propose un état de l'art des différentes méthodes formelles applicables au logiciel :

- Méthodes formelles ;
- Méthodes ensemblistes ;
- Logiques ;

Des éléments connexes sont par ailleurs donnés : bases mathématiques nécessaires pour appréhender ces techniques, notions de types abstraits algébriques, etc. Notamment on trouve une classification des différentes méthodes formelles, selon des niveaux :

- Niveau 0 : Pas de méthode formelle, spécification sous forme de documents en « langage naturel », tests manuels ;
- Niveau 1 : Notations et concepts sont formalisés, sans pour autant permettre de vraies preuves ;
- Niveau 2 : Spécifications rédigées dans un langage mathématique ;
- Niveau 3 : Spécifications mathématiques rigoureuses où l'on peut certifier la conformité du logiciel à ces dernières ;

La complexité et les coûts vont croissant avec le niveau.

2.13.1.3 « Program Analysis via Graph Reachability » [53]

Dans cet article les auteurs s'intéressent à l'analyse de programme et la résolution de ce problème en se rapportant au problème de l'atteignabilité dans un graphe, et plus précisément, les problèmes d'atteignabilité pour les langages non contextuels (*CFL-reachability problems*). Les problèmes d'analyse de programme abordés avec cette méthode sont :

- Découpage inter-procédural
- Problèmes d'analyse de flux de données en approche inter-procédurale
- Approximation des *formes* de structures de données allouées sur le tas

Selon les auteurs, la transformation en un problème d'atteignabilité dans un graphe permet notamment d'obtenir un algorithme efficace pour le problème d'analyse de programme correspondant. Ils citent également la possibilité d'obtenir une approximation rapide de la solution, ainsi que des indices sur la parallélisation du problème. Sur ce point plus précis, les auteurs rappellent qu'il a été montré que :

- Le découpage inter-procédural est log-space complet pour P
- L'analyse de flux interprocédurale est P-difficile
- Les problèmes d'analyse de flux interprocédurale avec des ensembles de faits sous formes d'ensembles finis sont également log-space complet pour P

Et qu'à moins de montrer $P = NC$ [5], il n'existe pas d'algorithme pour le découpage inter-prodécural et l'analyse de flux de données inter-procédurale dont :

1. Le nombre de processeurs est borné par un polynôme fonction de la taille d'entrée
2. Le temps d'exécution est borné par un polynôme en log de la taille d'entrée.

2.13.1.4 *Vérification de logiciels* [25]

Dans cet ouvrage, les auteurs proposent une introduction aux techniques de vérification de logiciel, en utilisant le model-checking. Un premier rappel concerne le modèle à définir : il n'existe pas de « bonne méthode de travail » qui soit générique, il est important de connaître le système que l'on étudie pour le modéliser fidèlement. Par système, l'auteur entends ici n'importe quel ensemble de matériel ou de logiciel, pas uniquement des programmes ; les exemples donnés sont :

- Un gestionnaire d'impression ;
- Un ascenseur.

Bien évidemment, les systèmes qui se prêtent le plus au jeu du model-checking sont ceux qui sont facilement modélisables sous la forme d'un automate à états finis. La notion de logique temporelle pour permettre de vérifier des propriétés est amenée grâce à des propriétés élémentaires liées aux états élémentaires. Le problème de l'explosion du nombre d'états est également introduit, par le besoin de modularisation du système à tester : les automates des différents modules doivent être synchronisés, ce qui engendre beaucoup d'états. L'utilisation de réseaux de Pétri pour effectuer la synchronisation au sein de l'automate est proposée. Ensuite, le rôle précis d'une logique est donné : exprimer des propriétés élémentaires pour chercher à les montrer par la suite. Une première explication avec la logique du premier ordre montre la lourdeur de celle-ci dans la vie réelle, et justifie l'utilisation de logique « temporelle » : langage formel, opérateurs « naturels ». Malgré tout, cela nécessite une certaine habitude qui limite la diffusion du model-checking. La logique temporelle *CTL* (Computation Tree Logic*)* est donnée, puis les versions restreintes utilisées par des model-checkers présentées : *CTL* (implémentée par *SMV*), *PLTL* (implémentée par *Spin*). Des pistes pour aider au choix entre les différentes logiques sont données :

- Description des comportements attendus : privilégier *PLTL* ;
- Efficacité du Model-Checking : privilégier *CTL*.

Une autre logique, *FCTL*, corrigeant des lacunes de *CTL* est évoquée, mais son usage est dit peu répandu. Les présentations étant faites, l'ouvrage passe à un élément fondamental : l'algorithme de model-checking de la logique temporelle *CTL*, articulé autour d'une « procédure de marquage » essentielle. Le même travail est proposé pour la logique *PLTL*. Dans le premier cas, la complexité est définie. La logique *PLTL* ne permet pas l'usage de la procédure de marquage. On peut faire le lien avec les expressions régulières : il suffit s'associer à toute formule f de cette logique une expression régulière E_0. Le problème

5. *NC*, Nick's Class, classe de problèmes inclue dans P résolus en temps poly-logarithmique sur une machine parallèle à nombre de processeurs polynomial

revient, après quelques transformations, à l'étude du langage reconnu par un automate, et s'il est vide, on sait que la propriété est vérifiée.

Une fois les bases posées, la suite de l'ouvrage propose d'aller plus loin en se penchant sur le model-checking symbolique, et l'utilisation des *BDD (Binary Decision Diagrams)* pour passer outre le problème de l'explosion du nombre d'états : ces diagrammes permettent la représentation symbolique d'états. Ils sont efficaces, simples, généraux et utilisables avec tous les types d'automates. Une manière d'encoder les états et les transitions, en utilisant un codage binaire, est proposée. Elle a l'avantage d'être efficace est facile à automatiser.

Enfin la prise en compte de contraintes temps-réel est introduite avec une logique temporelle temporisée (*TCTL*) et des automates temporisés.

2.13.1.5 « An Efficient Algorithm for Solving the Dyck-CFL Reachability Problem on Trees » [193]

Les auteurs de cet article présentent une méthode efficace qui permet de résoudre le problème *Context-Free-Language Reachability* dont il a été démonté par ailleurs l'importance dans le cadre de l'analyse statique, notamment dans [12, 13, 17, 11]. Une classe particulière de ces problèmes, identifiée sous le nom *Dyck CFL* peut voir sa complexité réduite considérablement rapport au cas général : passant de $O(k^3n^3)$ pour une grammaire de taille k à $O(kn^3)$. La solution proposée dans cet article effectue un pré-calcul, dont la complexité temporelle est annoncée à $O(n \log n \log k)$ et $O(n \log n)$ pour la complexité spatiale ; une fois ce traitement effectué, la complexité de n'importe quelle demande est en $O(1)$. Bien qu'étant éloigné, c'est un papier qui propose un résultat intéressant qu'il convient de garder en mémoire. Notons la conclusion qui résume les limites des travaux : il s'agit de se limiter au cas où le graphe est un arbre bi-orienté de taille n, et où la grammaire est un langage de Dyck de taille k.

2.13.2 Vérification noyau

Les articles qui suivent touchent à la vérification de code noyau, que ce soit uniquement des pilotes ou bien des noyaux complet.

2.13.2.1 « Categorization of common coupling and its application to the maintainability of the Linux kernel » [194]

Dans [194] les auteurs s'intéressent au problème de couplage au sein du noyau Linux, et à ses répercussions sur sa maintenance, en se focalisant sur la version 2.4.20. Un premier résultat sur l'analyse de plus de 400 versions successives montre que, si le nombre de lignes de code croit linéairement, les couplages entre les modules croissent exponentiellement. Les auteurs retiennent comme définition du couplage le degré d'interaction entre deux « unités » composant le logiciel. C'est donc une notion inhérente à la modularité du code source, et un excès de dépendance entre les composants va rendre l'ensemble complexe à maintenir notamment. La définition retenue par les auteurs pour leur mesure du couplage vient de Offut et al. [122] : deux modules P et Q sont couplés généralement si P et

	k->k safe	minimally k->k safe	k->k unsafe
nonk->k safe	Catégorie 1	Catégorie 2	Catégorie 3
nonk->k unsafe	Catégorie 4	-	Catégorie 5

TABLE 2.2 – Classification des types de variable et sûreté associée

Q partagent des références aux mêmes variables globales. Par ailleurs, trois raisons sont avancées pour se concentrer sur ce problème :

- Des études précédentes montrent que la part la plus importante de couplage introduite pendant la phase de maintenance est le couplage commun.
- Le couplage commun est à peu près le seul à faire l'unanimité sur son risque induit.
- La notion de couplage commun clandestin, plutôt à même de se retrouver dans des pilotes de périphériques.

Les auteurs référencent également trois cas d'utilisation qui présentent des risques différents. Soient M_1, M_2 deux modules couplés parce qu'ils référencent une variable globale gv :

- Seul M_1 peut changer la valeur de gv et seul M_2 l'utilise.
- Seul M_2 peut changer la valeur de gv et seul M_1 l'utilise.
- M_1 et M_2 peuvent tous les deux changer la valeur de gv.

Seul ce dernier cas est vraiment problématique. Grâce à l'analyse réalisée avec l'outil `LXR`, les auteurs proposent une classification des variables globales :

1. Variable globale définie dans des modules et utilisée dans aucun.
2. Variable globale définie dans un module noyau et utilisée dans un ou plusieurs.
3. Variable globale définie dans plus d'un module noyau et utilisée dans un ou plusieurs.
4. Variable globale définie dans un ou plusieurs modules non-noyau et utilisée dans un ou plusieurs modules noyaux.
5. Variable globale définie dans un ou plusieurs modules non-noyau et définie dans un ou plusieurs modules noyaux.

De plus, une classification des risques est proposée :

k->k safe Une variable globale est définie comme étant sûr de noyau à noyau si un changement depuis un module noyau n'affecte pas le noyau.

k->k unsafe Une variable globale qui n'est pas k->k safe.

minimally k->k unsafe Une variable globale qui n'est définie que dans un module noyau et utilisée dans un ou plusieurs modules noyaux.

nonk->k safe Une variable globale dont un changement dans un module non-noyau n'affecte pas le noyau.

nonk->k unsafe Dans le cas contraire.

Cette sûreté des variables est présentée pour chacune des catégories :

Les auteurs analysent ensuite les différents types de catégories dans le cas du noyau Linux. Un exemple est donné avec la variable `current`, qui permet de faire référence au processus en cours, classée en catégorie 5, avec un graphe en étoile qui représente les dépendances entre modules du noyau. Ce qui importe pour la maintenance est principalement les instances de couplage. Les auteurs montrent qu'il y a beaucoup d'instances sur des

définitions nonsafe et d'après eux, ceci présente un risque pour l'avenir à long terme du noyau.

2.13.2.2 « Integrated static analysis for Linux device driver verification » [138]

En se basant sur les travaux de Microsoft avec sa plateforme *SDV (Static Driver Verifier)*, les auteurs proposent de faire un travail semblable pour les pilotes du noyau Linux. Quelques adaptations sont cependant nécessaires :

- Intégration sous la forme d'un plugin Eclipse ;
- Utilisation du model-checker libre CBMC ;
- Langage de spécification des règles SLICx inspiré de SLIC ;
- Intégration des règles au code source ;
- Vérification à la compilation par CBMC.

En fait, les problèmes qui se sont posés étaient les suivants :

- Non-disponibilité de l'outil SLIC, et absence de prise en charge du C utilisé par le noyau Linux, d'où la réimplémentation et l'extension avec SLICx ;
- Pas de modèle de pilote Linux similaire au WDM sous Windows, d'où l'écriture de nouvelles règles à partir de la documentation des APIs du noyau.

Notamment, une grammaire de SLICx est donnée. Une différence majeure entre SLIC et SLICx est que dans le premier, les fonctions de transfert (celles manipulant l'automate de sûreté) se terminent nécessairement. Comme SLICx est utilisé en combinaison avec un model-checker borné, cette contrainte disparait. Un problème est abordé, mais non résolu, il s'agit de la création automatique du modèle de l'environnement. Une première modélisation du cycle de vie d'un module est proposé. Les auteurs mettent également en exergue la difficulté à passer sur un processus totalement automatisé dans le cas de pilotes Linux, leurs résultats sur l'application de la technique à de vrais pilote montrant de nombreuses étapes manuelles. De leur point de vue, une fois le problème du modèle du système d'exploitation résolu, il sera possible d'égaler voire de surpasser ce qui a été fait avec SDV.

2.13.2.3 « Model checking concurrent linux device drivers » [188]

Les auteurs de cet article ont un objectif intéressant en ce sens où ils veulent faire du Model Checking concurrent de pilotes dans le noyau Linux. Ils se placent, eux-mêmes, dans la lignée des travaux menés sur les pilotes Windows avec SLAM[17, 14, 16]. Leurs contributions, au-delà de la confirmation dans le monde Linux des résultats obtenus par SLAM sur les pilotes Windows, permettent la vérification automatisée des pilotes Linux avec un modèle fidèle des parties nécessaires du noyau. Surtout, ils apportent une technique de vérification pour les programmes concurrents avec mémoire partagée. Cet aspect est intéressant, car les problèmes engendrés sont souvent complexes à comprendre, et sont des cas limites très difficilement reproductibles par les tests. Une première de leurs contribution est donc l'intégration des model checkers `Cadence SMV` et `Boppo` avec l'outil de vérification basé sur l'abstraction de prédicat `SatAbs`. Un modèle non déterministe de l'environnement du pilote dans le noyau (i.e., d'APIs importantes) est proposé. Un exemple d'assertion permettant de trouver des bogues qui n'étaient pas déjà connus (mauvaise utilisation de

`request_region()`) est donné. Cette mise en exergue est d'autant plus intéressante que le code contenant ce bogue est exécuté à chaque chargement du module. Une limite principale à l'heure de cet article (2007) concerne les performances des outils de model-checking utilisés.

2.13.2.4 « seL4 : formal verification of an OS kernel » [83]

Un travail intéressant visant non seulement la vérification formelle mais également la preuve sur un noyau de système d'exploitation a été réalisé et présenté notamment dans [83]. La cible est un noyau précis, *seL4*, une version « sécurisée » du micro-noyau L4. Dans le cadre de ce qui nous intéresse, et en mettant de côté l'aspect micro-noyau, cet article est important puisqu'il s'attaque à vérifier formellement (et prouver) un noyau quasi-complètement : seul le code correspondant au démarrage n'est pas encore concerné ; ce qui laisse tout de même, selon les auteurs, 8700 lignes de `C` et 600 lignes d'assembleur. Le fait de se limiter à un micro-noyau rends la preuve possible bien que nécessitant l'assistance d'outils (Isabelle). Ces vérifications et preuves se font au travers de plusieurs contributions. La première concerne une méthode de prototypage rapide d'un micro-noyau en écrit C, grâce au langage Haskell, et permet d'aider à la réalisation de la preuve. Cette dernière est obtenue grâce à l'assistant Isabelle/HOL. Ceci conduit également à l'obtention de plusieurs niveaux de spécification :

- spécifications abstraites, décrivant les *fonctionnalités du système*
- spécifications exécutables, décrivant le *fonctionnement bas niveau du système*

Une autre contribution est une modélisation d'un sous-ensemble du langage C99. Celle-ci est nécessaire puisque le code C fait partie de la phase de vérification. Beaucoup de comportements de C99 ne sont cependant pas pris en comptes, parmi lesquels pas d'utilisation de l'opérateur &, éviter au maximum l'utilisation de références (à cause de l'absence de garantie sur l'ordre d'évaluation des expressions), les pointeurs de fonctions ne sont pas autorisés, de même que les instructions `goto` ou `switch`. Ceci s'explique bien dans leur contexte, mais reste une limite forte pour le nôtre. Quelques hypothèses fortes sont prises, notamment que l'on peut faire confiance au matériel et au compilateur. Le résultat est donc un micro-noyau réel et vérifié, `seL4`, et sans perte de performance contrairement aux préjugés liés à la preuve de code, ce dernier est performant. Ceci est en partie possible grâce à l'architecture de prototypage avec Haskell et Isabelle/HOL mise en place. Un autre point intéressant est l'analyse du coût d'une telle preuve : selon les auteurs, leur méthode est à la fois beaucoup moins cher et beaucoup plus efficace que la certification EAL6.

Dans [83], également présenté en détails en section 2.13.2.4, les auteurs présentent une vérification formelle d'un micro noyau, `seL4`, basé sur `L4`. Celle-ci vise la presque totalité du code, que ce soient les 8700 lignes de code C ou les 600 lignes de code assembleur. Cette version de `L4` vérifiée est d'autant plus intéressante que le préfixe `se` de son nom signifie *secure embedded*, i.e., ce micro noyau est pensé pour être sécurisé et vise les systèmes embarqués. La preuve est à la fois faite au niveau des spécifications et de l'implémentation. De plus, les auteurs proposent des outils pour aider à la création d'un noyau vérifié avec `Haskell` et `Isabelle/HOL`. Les résultats montrent non seulement qu'un micro noyau totalement prouvé (à l'exception du code de démarrage) est faisable, mais les auteurs arguent que cette sûreté n'a pas à être faite au détriment des performances, même s'il existe un

coût, il reste marginal. Par ailleurs, la quantité de travail nécessaire pour réaliser ce micro noyau a été comparée à celle d'autres, `Pistachio` en profil *embarqué*, et elle se révèle inférieure dans le cas de `seL4` : l'étape de prototypage avec `Haskell` a réduit le travail globalement nécessaire. La preuve en elle-même accapare une partie importante du temps, presque le double du total nécessaire à `Pistachio`. Cependant, les auteurs affirment que faire une vérification similaire sur un nouveau noyau en conservant la technologie permettrait d'arriver à un coût global légèrement supérieur au cas de `Pistachio`, pour une sûreté bien mieux garantie.

2.13.2.5 « Establishing Linux Driver Verification Process » [82]

Dans [82], les auteurs proposent un *processus de vérification des pilotes Linux*, constitué de deux éléments principaux : un dépôt de problèmes potentiels, et un ensemble d'outils de vérification. Les sous-processus qui s'en dégagent sont :

surveillance noyau – suivi des modifications sur le noyau Linux via la liste *Linux Kernel Mailing List*
- identification des messages pertinents au regard de l'objectif poursuivi
- extraction des nouvelles règles d'interaction avec les APIs
- enregistrement dans le dépôt

formalisation – pour chaque enregistrement présent dans le dépôt, un expert décide si un outil de vérification existe
- si cette vérification automatique est possible, une spécification de la règle est ajoutée dans le dépôt (dépendante de l'outil sélectionné)

identification de problèmes potentiels Il s'agit d'une vérification automatique en utilisant les données précédentes, qui permet de dégager :
- une liste de problèmes potentiels dans le code source
- une liste de recommandations pour améliorer le code source
- des informations de compatibilité du pilote avec plusieurs versions du noyau

analyse des résultats cette dernière étape est manuelle et effectuée par un expert, qui est chargé de déterminer si les problèmes identifiés sont des faux positifs ou non et prend les mesures appropriées (contact avec le mainteneur du pilote, retour pour améliorer les outils de vérifications).

Comme indiqué, l'alimentation des outils de vérification se fait depuis le dépôt. Une règle de vérification est représentée par :
- Un modèle d'échec
- Un modèle de noyau
- Un descripteur de traçabilité
- Un type de vérification et une liste d'outils de vérification

C'est le modèle d'échec qui représente ce que nous considérons comme la modélisation des erreurs. Ce modèle dépend de l'outil utilisé pour effectuer la vérification. Les auteurs donnent l'exemple de l'outil `BLAST` qui ne peut vérifier uniquement la non-violation d'instructions `assert`, et c'est pourquoi l'exemple de règle donné est adapté à cet outil, en utilisant des assertions pour *BLAST*. La fiche de la règle de vérification contient également une description humainement compréhensible. Pour réaliser le modèle du noyau,

les auteurs proposent de réutiliser les fichiers d'entêtes en utilisant des versions simplifiées de l'implémentation. Un banc de test modélisant les requêtes effectuées au pilote est également proposé. Les règles à vérifier sont classifiées selon trois classes :

syntaxique ces règles permettent de décrire des restrictions quant à l'utilisation de certaines constructions syntaxiques du langage (types, champs, fonctions)

sûreté ce sont des règles écrites pour vérifier l'atteignabilité de certaines portions du code source

vivacité ces règles s'assurent de la terminaison du programme

Il convient de noter que ces travaux ne font que présenter l'idée, mais qu'ils ne sont pour le moment pas suivi d'effet.

2.13.3 Model-Checking de structures dynamiques

Des travaux récents proposent de s'intéresser précisément au cas de la vérification de propriétés sur des structures dynamiques telles des listes simplement ou doublement chaînées. Deux articles plus précisément, le premier, [30] introduit la méthode *Abstract Regular Tree Model-Checking* alors que le second, [75] présente l'utilisation de cette méthode pour vérifier l'utilisation du tas dans un programme.

2.13.3.1 « Abstract regular (tree) model checking » [30]

Dans ce premier article, les auteurs proposent une amélioration sur le regular model-checking, technique classée dans le model-checking symbolique, qui permet de vérifier des systèmes à nombre infini d'états. Les configurations du système sont encodées sous la forme de mots ou d'arbres sur alphabet fini, et les transitions sont des tranducteurs à états finis. Des automates finis sont utilisés pour représenter et manipuler les ensembles de configuration (potentiellement infinis). L'atteignabilité des états se calcule en calculant les fermetures transitives des transducteurs (si l'on cherche des relations d'atteignabilité) ou en calculant des images des automates par itération des transducteurs (dans le cas où l'on recherche des ensembles d'atteignabilité). Ces calculs ne sont pas décidables dans le cas général, et les auteurs rappellent que pour faciliter la terminaison, différentes méthodes *d'accélération* existent. Le reste de l'article consiste à présenter une accélération par abstraction : une combinaison entre le Regular Tree Model-Checking et les boucles CEGAR, avec calcul de point fixe abstrait sur certains domaines finis d'automates. Ces calculs sont garantis pour la terminaison et fournissent une sur-approximation de l'ensemble d'atteignabilité.

Des retours sur une première expérimentation sont proposés grâce à une implémentation d'un prototype écrit en Prolog avec la bibliothèque FSA. L'objectif était de démontrer l'applicabilité à la vérification d'une vaste gamme de systèmes, parmi lesquels :

1. Réseaux paramétrisés de processus : une version idéalisée d'algorithmes d'exclusion mutuelle pour un nombre arbitraire de processus. Vérification de la propriété d'exclusion mutuelle de ces algorithmes ainsi que la vivacité (commune et individuelle) dans le cas de `Bakery`.

2. Systèmes push-down : système de fonctions récursives. Vérification d'une propriété de sûreté sur un algorithme de dessin : correction de l'ordre des instructions.
3. Systèmes avec files : vérification de l'ordre de remise des messages sur le système `Alternating Bit Protocol`.
4. Réseaux de Pétri et systèmes avec compteurs : vérification d'un réseau de Pétri (assimilable à un système avec compteurs infinis), notamment un modèle lecteur/écrivain étendu avec la gestion dynamique de processus. Vérification de l'exclusion mutuelle sur ces processus (entre les lecteurs et un écrivain, et entre les écrivains).
5. Structures de données liées dynamiquement : vérification d'une fonction pour inverser une liste simplement chaînée non vide.

Pour cette évaluation, plusieurs schémas d'équivalence d'automates ont été comparés (ils sont définis dans l'article), et un premier résultat concerne le choix de la valeur initiale et son incrémentation : cela influe énormément sur les performances de ces schémas. Il en est de même du choix du prédicat initial pour d'autres schémas. Le second résultat concerne les temps de calculs (évalués sur un Intel Pentium 4, 1.7GHz, avec le prototype présenté précédemment) : les deux approches (langage à prédicat ou langage bornés) se révèlent complémentaires, justifiant de s'intéresser aux deux.

Une dernière section s'intéresse à la vérification de programmes contenant des pointeurs. Les auteurs se limitent aux programmes C séquentiels non récursifs, manipulant des structures de données dynamiquement liées par des pointeurs et stockant des valeurs de domaines finis. Les propriétés que l'on veut tester sont classiques : non-déréférencement ni assignation de pointeurs null, ne pas faire référence à un élément supprimé. De plus, la définition de *testeurs* (portions de code C avec une légère extension du langage C) permet de vérifier certaines conditions (invariants, cycles, etc.). Ces testeurs deviennent également une partie du code qui est vérifié. Un prototype a été réalisé, utilisant les bibliothèques Mona, pour vérifier des fonctions manipulant des structures telles des listes simplement et doublement chaînées, arbres, listes de listes ainsi que des arbres avec des feuilles liées. L'arithmétique des pointeurs n'est pas prise en compte. Malgré la qualité de prototype, les auteurs sont plutôt enthousiastes des résultats expérimentaux obtenus (sur Opteron 2.8GHz en 64 bits) : les vérifications sur les listes chaînées se font en quelques secondes à dizaines de secondes (1 seconde pour la création d'une liste simplement chaînée, 5 secondes pour son inversion, 10 secondes pour l'insertion d'un élément dans une liste doublement chaînée). Seule le cas des `task-list` est moins encourageant, avec presque 2 minutes pour la suppression ; et plus de 11 minutes pour l'insertion. Cependant les auteurs indiquent que c'est le cas le plus complexe de l'étude, et qu'il s'inspire fortement de ce qui est fait dans les systèmes réels (comme `struct task_struct` dans le noyau Linux).

2.13.3.2 « Forest automata for verification of heap manipulation » [75]

Dans ce second article, les auteurs propose d'exploiter la technique présentée précédemment, *Abstract Regular Tree Model-Checking*, dont ils ont déjà montré l'intérêt pour vérifier des structures de données dynamiques complexes, pour vérifier les manipulations du tas dans un programme. Le tas est ainsi représenté par un ensemble d'arbres d'automates, pouvant se faire référence entre eux, et peuvent contenir d'autres arbres d'automates

imbriqués afin d'obtenir une représentation hiérarchique.

Les auteurs précisent s'intéresser en particulier à la manipulation de pointeurs en langage C, même si les résultats sont valable pour tout autre langage similaire, notamment pour vérifier des listes doublement chaînées complexes ainsi que des arbres sous formes diverses. Les propriétés de sûreté qui les intéressent sont classiques :

- absence de déréférencement de pointeur null
- absence de double libération
- gestion des pointeurs suspendus
- fuites mémoires

Ils proposent, de plus, et comme dans l'article précédent, l'utilisation de *testeurs* pour valider certaines propriétés de forme. Une première partie se propose de montrer le cheminement depuis un ensemble de tas à une forêt hiérarchique d'automates, de sorte à introduire la notion d'hypergraphe et de forêt d'automates qui est par la suite centrale.

Les tas peuvent être vus comme des graphes orientés dont les nœuds sont des zones mémoire allouées et les arcs sont les pointeurs qui font le lien entre ces cellules. Des points de coupures sont définis (des nœuds qui sont pointés par des variables du programme ou bien ayant plusieurs arcs entrants) et numérotés suivant un parcours en profondeur d'abord. Puis ils servent à découper le graphe du tas en arbres à des points particuliers. Ils introduisent ensuite *canonicity respecting forest automata (CFA)* qui permet d'encoder des ensembles de tas décomposés de manière canonique : dans le tuple choisi, il n'y a aucun arbre avec une racine qui ne corresponde pas à un point de coupure et que ces arbres sont bien triés en profondeur d'abord. La forme respectant la canonicité permet de tester l'inclusion sur les ensembles de tas représentée par CFA. De plus cet encodage permet de représenter des ensembles de tas avec un nombre borné de points de coupure. Cependant les auteurs notent que pour traiter nombre de cas de structures de données classiques, il est nécessaire de le faire avec un nombre non borné de points de coupure, et proposent de résoudre le problème avec une représentation hiérarchique, en créant des boites de sous-graphes (appelés composants), lesquels peuvent être remplacés par un simple hyperarc ayant pour label la boite appropriée. Ainsi l'ensemble de tas peut être transformé en un ensemble hiérarchique d'hypergraphes de tas. Finalement, ils proposent une décomposition canonique d'hypergraphe hiérarchique permettant de décider l'inclusion pour des ensembles d'hypergraphes de tas représentés par des forêts d'automates, permettant de vérifier une approximation de l'inclusion sur les tas de représentés hiérarchiquement, et indiquent qu'elle semble bien fonctionner en pratique.

Une seconde partie de l'article est consacrée à la formalisation de la notion d'hypergraphe et de forêt d'automate, et proposent des résultats : représentation d'hypergraphe en forêt, forêts minimales et canoniques, forêt d'automates. Une fois ces résultats introduits et démontrés, la notion d'hypergraphe hiérarchique est introduite, avec notamment des résultats sur l'inclusion et la bonne connexité d'ensemble hiérarchiques de forêts d'automates. Au sein de ces résultats, on trouve également des résultats intermédiaires tels que *proper boxes and well-formed hypergraphs*, *box-connectedness*, *canonicity respecting hierarchical forest automata*.

En se basant sur tous ces résultats, les auteurs peuvent maintenant expliciter leur procédure de vérification. Dans la suite, *Sel* correspond à l'ensemble de tous les sélecteurs

(chaque cellule de mémoire allouée peut avoir plusieurs pointeurs sélectionnant l'élément suivant) ; *Data* correspond au domaine des données et les variables sont *Var*. Ils rappellent d'abord la représentation choisie du tas comme un hypergraphe déterministe avec certains labels (*Sel*, *Data* et *Var*) ainsi qu'une fonction de classification telle que $\#(x) = 1 \approx x \in Sel$ et $\#(x) = 0 \approx x \in Data \cup Var$. Le calcul symbolique des configurations de tas atteignables est effectué sur un graphe de contrôle de flux obtenu à partir du programme source. Le calcul de point fixe se fait sur le même graphe de contrôle.

Finalement les auteurs proposent quelques résultats expérimentaux grâce à un premier prototype sous force de greffon au compilateur *GCC* : *Forester*. Ils considèrent ces résultats préliminaires comme encourageants quant à la généricité de l'outil, la précision des invariants ainsi que les temps de calculs (de l'ordre du dixième de seconde). Notamment, comparé à d'autres outils, leur proposition semble traiter plus de cas (*Space Invader* ayant plusieurs échecs d'exécution symbolique et *ARTMC* semblant très complexe à dompter et trop lourd). Les cas de tests tournent toujours autour des listes simplement et doublement chaînées (insertion, suppression, tris), ainsi que les arbres.

2.13.4 « Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking » [84]

L'objectif affiché dans ce papier est de vérifier la cohérence du comportement entre un programme écrit en C et un circuit décrit avec Verilog. Il existe un lien fort et trivial entre ces deux éléments : le programme C est un modèle qui est utilisé afin de valider un circuit avant de passer à une implémentation bas niveau Verilog. Vérifier que les deux soient cohérent est non seulement essentiel mais aussi naturel. La technique proposée est du Model Checking Borné (Bounded Model Checking) : il s'agit de dérouler l'exécution du système (représenté par un automate fini) pour obtenir une formule booléenne qui est vérifiée grâce à un solveur SAT. Ces travaux sont intéressants pour l'aspect de conversion ANSI-C vers formule booléenne qu'ils proposent et qui peuvent être une source d'inspiration. Notamment, une procédure qui effectue cette traduction est proposée dans l'article : réduction du problème Model Checking d'un programme C à la détermination de la validité d'une équation de vecteurs de bits. Une suite de modifications est appliquée au programme C d'origine :

- Suppression des boucles `while` ;
- Suppression des conditions `if` ;
- Suppression des sauts inconditionnels `goto` ;
- Suppression des assignations ;

Une version détaillée de l'algorithme proposé est donné dans [85]. La gestion des pointeurs est également abordée : les déréférencements (que ce soit avec l'opérateur `*` ou avec l'opérateur `[]`) sont transformés en une fonction ϕ qui est définie par cas dans l'article. Chacun de ces cas permet de prendre en compte l'un des cas sémantiques du pointeur en C. Les cas non définis par ANSI-C sont traités en utilisant une valeur erreur dont il sera vérifié que le cas n'est jamais atteint pendant l'exécution du programme. La prise en compte de l'allocation dynamique de la mémoire est rapidement évoquée, et tous les détails sont donnés dans [85]. Il n'est pas fait mention lors de la discussion sur les résultats expérimentaux de la taille du code analysé, mais on peut déduire des cas étudiés qu'elles

sont probablement assez faible :

- Unité de récupération d'instructions dans un processeur Torch ;
- Arbitreur entre 4 clients exploitant une unité de DRAM ;
- Interface PS/2 ;
- Microprocesseur DLX ;

Sur ces exemples, le temps de calcul est de l'ordre de quelques minutes (51 secondes à 380 secondes) et la consommation mémoire de l'analyse va jusqu'à 1.4GiO à 2GiO dans certains cas.

2.13.5 « MOPS : an infrastructure for examining security properties of software » [44] et *MOPS : an Infrastructure for Examining Security Properties of Software* [43]

L'utilisation du model-checking à des fins de sécurité est abordée dans cet article : une propriété de sûreté temporelle, qui donne l'ordre adapté pour une séquence d'opérations liées à la sécurité. L'utilisation de cette technique se fait après la modélisation du programme à analyser sous la forme d'un automate à pile, lorsque les propriétés de sécurité que l'on veut vérifier le sont sous la forme d'automates à état finis. Le model-checking intervient seulement à ce moment, pour vérifier sur un état qui viole les propriétés est atteignable. Rappels sur la sécurité : c'est le maillon le plus faible de la chaîne qui définit l'ensemble ; ce n'est pas seulement des erreurs de tampon en C, mais par exemple une erreur dans un programme telle que celui-ci viole les contraintes implicites d'un appel système qui peut introduire des vulnérabilités. Des exemples sont l'utilisation de `chroot()`/`chdir()`, ou `stat()`/`open()`. L'étude se propose d'aller jusqu'à trouver les cas où la séparation de privilèges n'est pas effectuée. Le framework proposé, *MOPS*[44] permet de réaliser une analyse inter-procédurale, et grâce à une nouvelle technique décrites dans [43], l'outil est en mesure de faire une analyse sur de gros codes. La vérification de propriétés temporelles de sécurité ressemble beaucoup aux travaux de SLAM[17, 14] chez Microsoft. L'approche retenue génèrera inévitablement des fausses alertes. Au-delà de la modélisation sous forme de machine à états finis, un point intéressant abordé concerne la relation avec le système d'exploitation : définir les modèles de sécurité. L'approche retenue consiste à explorer exhaustivement et automatiquement les opérations liées à la sécurité et à créer le nécessaire dans le modèle. Un exemple sur la gestion des privilèges dans Linux 2.4.17 est donné. La possibilité de réaliser des sauts long peut-être une source de vulnérabilité, et l'exécution de l'outil sur le code de *wu-ftpd 2.4beta11* a montré une faiblesse avec la gestion des privilèges et des signaux. Bien évidemment, les auteurs font références aux travaux sur SLAM[17, 14] en notant cependant que ce dernier n'est pas en mesure d'analyser aussi efficacement (en matière de temps) le code que ne le permet MOPS, celui-ci faisant un compromis orienté en faveur de la quantité de code analysable et de l'efficacité quand SLAM garde pour objectif la précision. De plus, SLAM étant itératif, il peut ne pas converger, chose qui ne peut arriver avec MOPS.

2.13.6 « Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level » [113]

Comme l'indique le titre, le sujet de cette thèse concerne la vérification de SoC. Ce besoin de vérification est devenu nécessaire à cause de ce que l'on appelle le *design gap*, lié au fait que les performances techniques (productivité du design, capacité des puces) augmente plus rapidement que la productivité des gens qui réalisent le design des puces, et au fait que le cycle de développement doit être le plus court possible. La puce est donc simulée pour permettre le développement du logiciel sans qu'elle ne soit disponible réellement grâce à un modèle transactionnel (uniquement ce qui est nécessaire au logiciel est modélisé) qui, développé au plus tôt dans le cycle de conception de la puce, permet de trouver un maximum de bogues avant qu'ils ne deviennent coûteux à corriger. Il s'agit donc, notamment, de faire du model-checking des modèles transactionnels. Ces systèmes sur puce sont développés avec SystemC, dont il convient d'extraire l'information. Ceci est réalisé grâce à deux outils développés dans le cadre de cette thèse :

- *Pinapa*, qui réalise une extraction « syntaxique » ;
- *Bise*, qui réalise une extraction « sémantique ».

La sortie de ces outils sera manipulée pour ensuite permettre l'utilisation de model-checkers classiques (*Lustre*, *SMV*). Dans les limites évoquées à propos de la transformation depuis SystemC, on retrouve la question du compromis lié à l'approximation qui est faite. De même, le passage à l'échelle est potentiellement un problème. Il faut noter notamment la mise en place d'une représentation intermédiaire, *HPIOM (Heterogeneous Parallel Input/Output Machines)* qui, si elle apporte ses limitations (structure de contrôle figée, pas de processus dynamiquement créables, etc.), permet aussi de faire des transformations et des optimisations. Il est intéressant de noter que le modèle d'exécution des transactions étudiées se rapproche des systèmes distribués : pas de variable partagée, pas de référence de temps commune. Une liste de méthodes de vérification formelle est donnée :

- Preuve de théorème : bien, mais pas utilisable dans le cas présent (interactivité, non-automatisation) ;
- Model Checking : bien, correspond au besoin, mais besoin de techniques d'abstraction ;
- Interprétation abstraite et SAT ne convenant pas au besoin ;

Parmi les contributions apportées avec *Pinapa*, une peut plus particulièrement nous intéresser : il s'agit de l'extraction depuis SystemC pour alimenter un outil d'analyse. La possibilité de scinder une preuve en plusieurs sous-preuve est évoquée, pour réduire sa complexité en taille : on peut prouver la séquence de boot, et séparément l'exécution du programme. C'est déjà une proposition qui est faite par seL4[83]. Les techniques utilisées dans le cadre de cette thèse sont donc le model checking symbolique, ainsi que les diagrammes de décision binaires, deux méthodes plus puissantes déjà évoquées dans [25]. Mais pour arriver à prouver de gros programme, il reste conseillé d'effectuer la preuve « par morceaux ». Une limite évoquée du model-checking concerne le cas des structures dynamiques, par exemple un simulateur de jeu d'instruction : c'est certes un cas problématique dans le cadre de System-on-Chip, mais cela confirme qu'il n'y a assez peu de limitations quant à ce qui est possible.

2.13.7 « Saturn : A scalable framework for error detection using Boolean satisfiability » [190]

Description d'une méthode « basique » de détection d'erreur exploitant les capacités liées aux solveurs SAT. Le programme à analyser est transformé en un ensemble de contraintes booléennes, sur lesquelles on pourra inférer et vérifier des propriétés. Cette méthode a plusieurs avantages :

- Précision, puisqu'aucune abstraction n'est réalisée ;
- Flexibilité, puisque l'expressivité naturelle des contraintes booléennes permet de modéliser plusieurs langages ;
- Compacité, qui permet de simplifier les formules booléennes dans le cas d'une analyse intra-procédurale ;

L'analyse inter-prodécurale est facilitée grâce au pré-calcul d'une sorte d'empreinte, de signature, pour chaque fonction analysée : ainsi, des programmes beaucoup plus importants en matière de quantité de code peuvent être analysés de manière plus efficace que les précédents systèmes basés sur SAT. De plus, cela autorise la parallélisation, ce qui est très intéressant sur de grandes quantités de code : l'analyse de fuites mémoire dans le noyau Linux (5MLOC) passe ainsi de 23 heures à 50 minutes. Les auteurs avertissent bien que SATURN n'est pas un outil de validation du code, mais uniquement un outil destiné à trouver des bogues, en minimisant le nombre de faux-positifs. La traduction d'un langage de programmation vers le framework de détection d'erreur est présentée à partir d'un exemple simple, un langage de programmation bas niveau.

Presque 200 erreurs identifiées grâce à cet outil, et une base de donnée de l'utilisation des mécanismes de verrouillage dans le noyau. Quelques faux-positifs, mais des pistes sont données pour améliorer ce point, en s'inspirant d'autres outils tels que SLAM[14] ou BLAST.

Cependant, la base de données n'est plus disponible, ce qui est assez dommageable puisque l'on perd une information potentiellement utile. De plus, plusieurs limitations de l'outil sont présentées :

- Gestion des boucles perfectible : il existe deux manières de faire, la première peut ne pas trouver des bogues qui n'apparaissent qu'après plusieurs itérations, et la seconde peut rencontrer des soucis sur les pointeurs qui sont modifiés pendant l'exécution de la boucle ;
- Aucune prise en charge de la récursivité, ce qui peut-être assez limitant ;
- Distinction des pointeurs : basée sur des heuristiques, il existe des risques de faux-négatifs ;
- Lors du calcul des signatures des fonctions, certains aspects du comportement ne sont pas pris en compte ;
- Certaines constructions du langage C ne sont pas gérées : unions, tableaux, arithmétique des pointeurs.

L'outil SATURN est toujours maintenu par l'Université de Stanford.

2.13.8 « A Framework for Verification of Software with Time and Probabilities » [87]

Les auteurs s'intéressent à des techniques dites de « vérification quantitative » dont un exemple donné est « la probabilité que l'airbag ne se déploie pas ». Bien qu'il ne parle pas directement de model-checking, l'article propose une approche d'abstraction quantitative par raffinage qui permet la construction et l'analyse automatique d'abstractions de programmes temps-réels probabilistes. Un exemple concret est donné avec la vérification de SystemC. L'aspect stochastique est présenté comme étant, par exemple, la perte de composants dans un système ou la perte de messages durant la communication de deux périphériques réseaux. On peut étendre ces cas à n'importe quelle perturbation qui va avoir un impact sur le code exécuté. Le principe de cette analyse est la construction d'un modèle mathématique pour permettre la vérification de propriétés quantifiées ; ce qui peut être très intéressant si l'on souhaite mener une étude sur les propriétés que l'on peut garantir dans un noyau Linux temps réel. Il existe des outils ayant été utilisés pour faire du model-checking probabiliste, des références vers `PRISM` et `MRMC` étant données. La limite étant la nécessité de l'écriture d'un modèle par l'utilisateur. Une contribution est la formalisation de la notion de *programme probabiliste temporisé*, définie grâce à des processus de décision markovien à états infinis (*infinite-state MDPs*) ; et qui sont globalement des automates probabilistes temporisé avec des variables à valeurs discrètes. Ces processus markoviens servent de base pour modéliser les systèmes à la fois non-déterministes et à comportement probabliste. Une extension dite des « jeux stochastiques à deux joueurs » permet de modéliser deux choix non-déterministes contrôlés par des joueurs différents. C'est cette notion qui est exploitée par le framework proposé. Le processus de raffinage présenté a déjà été appliqué avec succès pour la vérification :

- D'une extension d'`ANSI-C` probabiliste ;
- D'un automate probabiliste temporisé ;
- De systèmes concurrents probabilistes.

Les deux non-déterminismes s'expliquent par, d'une part, celui induit par le modèle original, et d'autre part, par celui introduit par l'abstraction des programmes probabilistes temporisés (PTPs). La vérification de SystemC se fait d'abord en effectuant une traduction vers des PTPs : basiquement, il s'agit de faire de chaque méthode C++ un automate. Un intérêt particulier de ces travaux réside dans l'ajout de la notion *quantitative* à la vérification, puisque des travaux sur la vérification de SystemC existent déjà. Ceci permet de voir plus finement les différences propres à l'aspect quantitatif. On notera qu'à l'accoutumée le principal problème réside dans le passage à l'échelle sur des gros systèmes, et que dans cet objectif, l'utilisation de model-checking est envisagée même s'il est complexe à adapter au cas probabiliste.

2.14 Conclusion

Après avoir conclu sur les différents articles présentés, nous dresserons un constat plus général sur les difficultés communes à la fois sur les plans théoriques et pratiques.

2.14.1 Travaux présentés

Depuis les travaux menés par Chou et al., on a pu voir plusieurs belles initiatives afin d'améliorer la qualité du code qui s'exécute en espace noyau, avec des influences croisées entre les systèmes propriétaires et les systèmes libres. L'étude empirique des erreurs des systèmes d'exploitation a permis de mettre en lumière l'aspect critique des bogues des pilotes de périphériques, même si cela peut paraitre comme une évidence, la démarche a permis de montrer et quantifier l'état. Concomitamment les équipes de Microsoft travaillaient à améliorer de leur côté également la qualité des pilotes, aboutissant à un outil de vérification du code des pilotes inclus dans le kit de développement, et qui ira jusqu'à influencer la conception des APIs de pilotes pour Windows 7.

Plusieurs projets parallèles ont été mis en œuvre pour améliorer également la qualité du noyau Linux, que ce soit en essayant d'isoler les pilotes pour limiter les conséquences d'un plantage, s'inspirant des micro-noyaux, au travers des initiatives *Nooks* ; ou bien pour automatiser les tâches rébarbatives et sources d'erreurs telles que les évolutions majeures d'interfaces de programmation. Sur ce dernier point, c'est le projet *Coccinelle* qui a permis cette avancée, simplifiant le travail de maintenance. De part sa conception, il a également pu être utilisé afin de détecter des motifs de code sujets à caution. Il a ensuite été secondé par un autre outil, *Herodotos*, utilisant ses capacités pour faire un suivi du cycle de vie des problèmes identifiés, et permettant 10 ans après de réactualiser l'étude originelle de Chou et al.

D'autres personnes se sont intéressées aux problèmes de configurabilité très vaste que propose ce même noyau, en apportant des outils de vérification de la cohérence entre les options existant dans le code et celles sélectionnables par l'utilisateur, grâce à *Undertaker*. Toujours dans l'optique de pouvoir plus facilement vérifier le code, le projet *GCC MELT* vise à proposer une implémentation d'un langage de *détection de motifs* sur le code source, pendant la phase de compilation. Cette approche est ainsi très proche de celle utilisée par Chou et al. dans leur outil de vérification *xgcc*, et permettrait d'implémenter et d'intégrer assez facilement des vérifications génériques, ou par projet. Un exemple est proposé avec le projet *Talpo*[6] proposant un petit ensemble de vérifications de base qui propose un début de couverture des fonctions de base de la `libc`.

Une dernière branche de vérification sur le noyau Linux a été effectuée dans un ensemble d'articles présentés par Breuer et Valls, et visait à trouver spécifiquement certains types de bogues.

Les efforts de toutes ces initiatives n'ont pas été vains et dans leur mise à jour de l'étude de Chou et al., les auteurs du projet *Coccinelle* notent la nette amélioration de la qualité du code des pilotes, sur lesquels le principal de l'effort a été mis.

Une dernière initiative récente, présentée en section 2.13.3, propose une méthode pour effectuer du model-checking symbolique explicitement adapté aux structures dynamiques, i.e., toutes celles construites à base de pointeurs. C'est un point qui a souvent été souligné dans les travaux précédents, la difficulté de gérer le cas des pointeurs, amenant souvent à des limitations. Ces travaux récents sont d'autant plus intéressant que l'approche proposée semble assez générique pour prendre en compte beaucoup de cas de structures de données,

6. http://pvittet.com/index.html#talpo,https://gitorious.org/talpo

les exemples proposés étant des listes chaînées (simplement et doublement), des arbres, ainsi qu'une structure inspirée d'un cas réel, `task-list`. De plus, malgré l'aspect prototype de l'outil, les auteurs sont plutôt enthousiastes sur ses performances. D'un point de vue intégration, il s'agit d'un greffon pour le compilateur `GCC`, ce qui tranche avec la quasi-totalité des précédents outils, qui sont à chaque fois autonome (avec la complexité liée à la lecture du code source, soulignée à maintes reprises).

2.14.2 Explosion combinatoire

L'explosion combinatoire n'est pas un phénomène nouveau, et touche les systèmes contenant un grand nombre d'états. C'est typiquement le cas pour la mise en place de model-checking mais cela s'étend à l'analyse de n'importe quel code source, et plusieurs articles [35, 34, 15, 25, 30] en font mention. Les stratégies usuellement utilisées pour contourner ce problème sont de l'ordre de l'évitement : décomposition en sous-problèmes que l'on sait plus facilement traiter, ce qui est par exemple le cas lorsque l'on se limite à une analyse intra-procédurale ou bien que l'on limite l'application de l'outil au code source correspondant à un pilote ; ou bien simplification du modèle pour limiter le nombre d'états à traiter.

La seconde technique est illustrée par la mise en place des *Binary Decision Diagrams* présentée dans [25, 88], qui permettent de représenter efficacement des fonctions booléennes.

La limitation du périmètre d'évaluation se retrouve par exemple dans `Coccinelle`, qui se limite à une analyse intra-procédurale. Souvent, bien que liées au problème de l'explosion combinatoire, ces limitations se justifient également d'un point de vue pratique : faire plus n'est pas nécessaire, pas utile, voire pas souhaitable.

Il ne semble pas cependant y avoir eu de percée fondamentale permettant de passer outre, et les deux stratégies présentées restent pertinentes. Dans cette logique, il parait intéressant de se demander si l'on pourrait opérer la découpe en sous-problème à partir d'une représentation du noyau Linux.

2.14.3 Quelle utilisation des outils ?

Plusieurs outils utilisables ont été produits à la suite des différents projets réalisés ces dernières années. Si l'on se limite à la vérification de code qui soit libre, citer `Coccinelle` ainsi qu'`Undertaker` parait un minimum. Cependant, l'existence des outils et leur utilisation ne va pas nécessairement de paire, c'est particulièrement visible dans le rapport de recherche [132] sur la figure 16 qui présente l'utilisation de différents outils de vérification dans le temps. Cette utilisation se base sur les notifications de problèmes rapportés sur la liste de diffusion du noyau, les auteurs de ce type de rapports faisant mention de l'outil détectant. L'étude porte sur les versions du noyau produites depuis le passage à l'outil de gestion de version `Git`, à partir du noyau 2.6.12 (mi-2005), jusqu'au 2.6.33 (début 2010), et les outils relevés sont : `Coccinelle`, `Coverity`, `Smatch` et enfin `Sparse`. Globalement, et comme détaillé en section 2.6.3.2, l'utilisation de `Coccinelle` semble largement majoritaire à présent, `Coverity` disparaissant presque. Les auteurs en concluent à une préférence des outils libres.

Cependant, avec le passage à la version 2.6 du noyau Linux, l'ancienne organisation en une branche dite « stable » et une branche de « développement » a été abandonnée. Dans un message [7] du 2 mars 2005, l'architecte du noyau, Linus Torvalds, justifie ce changement par la trop grande latence entre les différentes sorties, engendrant un gros travail de rétro-portage et de gros bouleversements dans l'arbre des sources. En adoptant un cycle de développement beaucoup plus court, le but est de permettre de rendre les changements moins brutaux ce qui a posé des problèmes dans la gestion de la version 2.4. Quelque temps plus tard, un autre message [8] expose une partie des conséquences en matière de stabilisation pour les distributions Linux : une partie du travail leur revient maintenant, et la logique voudrait que les développeurs du noyau choisissent une version à maintenir « un peu plus longtemps », ce qui arrivera finalement. La discussion avec certains des développeurs noyaux chargés de la maintenance de ces versions montre qu'ils connaissent ces outils de vérifications (ils en ont entendu parlé), qu'ils pensent que c'est utile (notamment lorsqu'ils doivent faire du rétro-portage de corrections), mais qu'ils n'ont pas forcément le temps d'apprendre à les utiliser, ni le réflexe de les exploiter.

De plus, le transfert de stabilisation à l'encontre des distributions marque un peu plus leur rôle central : sas de communication entre les développeurs et les utilisateurs, elles auront toujours à intégrer des modifications pour plusieurs raisons (sécurité, intégration, régressions, etc.). Elles se retrouvent aussi avec le besoin potentiel de vérifier leur base de code, puisqu'elle diverge de la base originale. À notre connaissance, ni Mandriva ni Mageia n'ont cette démarche. Les outils comme `Coccinelle` ne sont pas utilisés par les mainteneurs de ces distributions au moins. Ceci plaide pour un travail d'intégration de ces outils de vérification et plus généralement de l'automatisation des tâches permettant de s'assurer de la qualité du code lors de la construction de toute la distribution. Une version préliminaire de cet état de l'art a été communiquée [101] au cours de l'édition 2011 du FOSDEM (Free and Open source Software Developers' European Meeting) [9] afin de présenter les premiers résultats sur l'état de l'art de ces travaux, et débattre des perspectives pour les distributions. Ceci nous a permis de rencontrer non seulement les développeurs de différentes distributions, mais également les auteurs de certains des outils, notamment Undertaker.

7. https://lkml.org/lkml/2005/3/2/247
8. https://lkml.org/lkml/2005/12/3/94
9. Le FOSDEM est un évènement annuel international regroupant des milliers de contributeurs (bénévoles et employés) travaillant sur différents logiciels libres

Chapitre 3

Caractérisation du noyau Linux

3.1 Introduction

Le chapitre précédent a introduit l'état de l'art et montré le besoin de mieux comprendre la structure et l'organisation du noyau **Linux** : un système d'exploitation complet basé sur celui-ci est constitué par une *distribution*. Celle-ci propose un assemblage complexe de composants hétéroclites, qui peuvent nécessiter des adaptations. Son rôle est également de s'assurer que le résultat final est utilisable et répond aux besoins des utilisateurs. Cela implique de vérifier et de valider le fonctionnement des différents composants. Le noyau est l'un d'entre eux : il est formé d'un code source organisé sous la forme de « modules » correspondant à une arborescence sur le système de fichier.

Plus précisément, dans la section 2.14.2 nous avons évoqué l'idée de découper le problème de l'analyse du noyau en sous problèmes qui seraient bien caractérisés. Le but de ce chapitre est de présenter une première approche de cette démarche. Dans la section 3.2 nous présenterons la manière de construire un graphe du noyau et la justifierons, sans omettre de présenter les limites de cette première approche. Dans la section 3.3 nous nous intéresserons à l'analyse du graphe ainsi obtenu afin d'en ressortir un maximum d'informations permettant de mieux comprendre la structure du code source du noyau. En plus d'être en mesure d'exposer un découpage du noyau en sous-modules permettant d'effectuer une analyse complète de manière efficace, nous espérons pouvoir mettre en lumière que le découpage physique du code correspond à l'utilisation effective.

3.2 Construction du graphe

Dans cette section nous allons présenter la procédure de construction du graphe du noyau Linux. Nous commençons par exposer rapidement comment, physiquement, se structure le code, puis ce en quoi cette structuration justifie l'utilisation d'un graphe pour la modélisation, avant de montrer quels éléments sont utilisés pour sa construction.

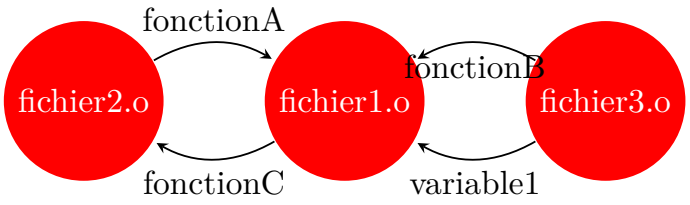


FIGURE 3.1 – Construction du graphe correspondant au noyau : `fonctionA()` est définie dans `fichier2.c`, `fonctionB()` et `variable1` sont définies dans `fichier3.c`, et `fonctionC()` est définie dans `fichier1.c`. Ce dernier utilise `fonctionA()`, `fonctionB()` et `variable1` ; `fichier2.c` utilise `fonctionC()`.

3.2.1 Noyau et code source

L'objectif de cette modélisation est d'être capable de découper le code source en des sous-ensembles analysables :

- dont la taille est maitrisable voire contrainte ;
- dont les relations sont caractérisées clairement.

3.2.1.1 Utilisation d'un graphe

Le noyau Linux est écrit en C, il est donc déjà plus ou moins découpé en modules cohérents composés par les fichiers sources `.c` qui sont compilés en `.o`. Il nous manque néanmoins une information cruciale : quelle est la relation entre ces fichiers sources (et objets, par transitivité) ? Par définition, on sait déjà que des modules peuvent exporter ou importer des éléments externes : variables, fonctions. Cette notion se retrouve dans les fichiers sources (bibliothèque) et donc dans le code compilé, sous la forme de « *symboles* ».

Intuitivement, entre l'arborescence utilisée sur le système de fichier et la notion de modules héritée du langage C, l'idée de modéliser ces relations sous la forme d'un graphe est assez naturelle. Le choix de cette modélisation classique permet de bénéficier des travaux et des outils existants. Nous proposons donc de construire un graphe qui va représenter les relations entre les modules du code source :

- Les sommets du graphe sont constitués par les modules objets, i.e., les fichiers `.o`, qui correspondent souvent directement aux fichiers sources `.c` ;
- Les arcs indiquent l'utilisation des symboles par les modules : soient A et B deux sommets dans le graphe et un arc S représentant un symbole. Celui-ci est défini dans le fichier correspondant au sommet B et est utilisé par le fichier correspondant au sommet A : alors, il y a une dépendance de A à B, i.e., $A \rightarrow B$;

La construction est résumée dans le schéma disponible en figure 3.1, et elle se décompose ainsi : le fichier `fichier1.c` utilise les fonctions `fonctionA()`, `fonctionB()` ainsi que `variable1` ; et `fichier2.c` utilise `fonctionC()`.

3.2.1.2 Collecte des données

Pour être en mesure de construire un tel graphe il est nécessaire de collecter les données correspondant aux symboles définis ainsi que leur utilisation. Ces données sont obtenues à partir du code source. Différents articles dans l'état de l'art font mention de la complexité et de la difficulté d'écrire son propre analyseur syntaxique pour le code source, et cela s'illustre bien par le recours au code de l'*Edison Design Group* par la société *Coverity*. De plus, il peut être intéressant de comparer différents cas du code source pour identifier la prévalence de certaines options de compilation : par exemple, comparer la topologie d'un noyau sans aucun module additionnel à celle d'un noyau avec tous les pilotes correspondant à une plateforme. Pour être en mesure d'obtenir cette finesse d'information, il convient d'interpréter les informations contenues dans `KConfig`, qui est l'outil permettant de gérer la configuration de la compilation.

Une solution alternative serait d'appuyer l'analyse sur le résultat de la compilation : l'idée devient d'analyser les fichiers objets pour en extraire les informations. La correspondance entre les sources et les fichiers objets peut être obtenue assez facilement : en général, un `.o` correspond à un (ou plusieurs) `.c`. Dans le cas où il y a correspondance 1 : 1, le nom est identique. Dans le second cas, une analyse du fichier `Makefile` permet rapidement d'identifier les correspondances, et son format reste assez simple ; de plus, `make` permet d'obtenir une version « calculée ». Un premier prototype a été réalisé en exploitant cette deuxième solution, qui s'est avérée à la fois simple et assez efficace.

L'analyse des fichiers objet est assez simple, la liste des symboles est facilement consultable par des outils classiques permettant de manipuler les binaires `ELF`, et nous indique quasi-directement si le symbole est **défini** dans le fichier ou bien s'il est **utilisé** (et donc, défini ailleurs).

Le processus de collecte des données implique donc plusieurs étapes :
- Identification des fichiers objet à analyser
- Éventuellement, mise en correspondance avec les fichiers sources
- Extraction de la liste des symboles de chaque fichier
- Stockage de la liste des symboles de chaque fichier, avec les attributs permettant de les qualifier par la suite (fonction, objet, variable, etc)

La méthodologie retenue d'analyser des fichiers objet permet de plus facilement transposer ce type d'analyse à d'autres bases de code source : la contrainte principale étant la production de ce type de fichiers, ce qui est le cas pour une large part de langages compilés. Être en mesure de généraliser cette approche à une large part des logiciels de la distribution, tout du moins ceux que nous jugeons importants, est un avantage certain.

3.2.1.3 Préparation des sources

Afin d'être en mesure d'obtenir une collection de fichiers objets, il est nécessaire de compiler les sources, i.e., construire un noyau. La sélection de ce qui soit être compilé doit être évoqué, bien que la compilation en soi ne soit pas un problème particulier. Le noyau est composé de nombreux **modules** qui peuvent être activés ou non pendant l'étape de configuration. Une large part de ces modules correspond à des fonctionnalités non nécessaires, des pilotes de périphériques, etc. Le nombre de fonctionnalités proposées par

le noyau Linux est constamment croissant, et assez important, [165] faisant référence en 2010 à plus de 8000 fonctionnalités, ce qui limite très fortement l'option d'une sélection manuelle.

De plus, il parait logique que les pilotes forment une partie « spécifique » du noyau : nous nous attendons à ce que ces éléments soient fortement liés à certains sous-ensembles du code (piles applicatives, APIs) et soient assez indépendants entre eux. Permettre deux analyses, l'une incluant le code des pilotes et l'autre l'excluant, permettrait d'avoir une mise en avant (le cas échéant) de ce comportement, et plus généralement de voir l'influence des pilotes sur la topologie du graphe.

Dans cette optique, nous proposons de préparer les sources de deux manières :
- une première en effectuant une configuration par défaut, qualifiée de **defconfig** : tous les choix par défaut sont retenus, les modules ne sont pas activés. Cette alternative doit représenter un sous-ensemble « de base » du noyau.
- une seconde en activant tous les modules possibles. Cette alternative est nommée **allyesconfig**, et doit représenter un noyau le plus complet possible.

Dans tous les cas, il est nécessaire de garder à l'esprit que ces configurations vont être dépendantes de l'architecture matérielle pour laquelle la compilation est effectuée. Cela implique que nous n'avons pas une couverture totale du code source du noyau. Pour les besoins de notre étude, nous nous sommes limités à l'architecture qui intéresse la société Mandriva, **amd64**. Les systèmes uniquement **x86** (32 bits) sont en voie de disparition en général, et en particulier depuis plusieurs années sur le marché des serveurs. En toute rigueur il est faisable d'évaluer l'approche sur d'autres architectures, mais cela implique de reconstruire le noyau avec un compilateur croisé.

Le temps nécessaire à la réalisation de l'analyse est assez important : il dépend du temps nécessaire à obtenir la construction du noyau, puis l'analyse des données obtenues. Pour réaliser l'étude ci-après, nous avons utilisé une machine virtuelle constituée de 22 processeurs et 48GiO de mémoire. L'hôte physique comportait 24 processeurs (Xeon E5-2630) et 96GiO de mémoire. La construction du noyau varie de l'ordre de 5-10 minutes pour le cas **defconfig** jusqu'à 30 minutes pour **allyesconfig**. Le reste de l'analyse prend plusieurs heures dans le premier cas, plusieurs jours dans le second, et le facteur limitant premier réside dans les entrées/sorties : les fichiers objets sont de taille importante (puisqu'ils n'ont pas été compressés ni débarrassés des symboles dont nous avons besoin), leur lecture prend donc du temps, et ils sont plusieurs milliers.

3.2.1.4 Processus complet

Le schéma proposé en figure 3.2 résume les différentes étapes entre la récupération des sources du noyau jusqu'à l'analyse du graphe obtenu. Le code couleur permet de regrouper les étapes sœurs :
- Le vert est utilisé pour les étapes de préparation des sources et des fichiers objets ;
- Le jaune est utilisé pour les étapes d'extraction des symboles ;
- L'orange sert à mettre en commun les étapes de création du graphe ;
- Les étapes d'analyse sont indiquées en rouge ;

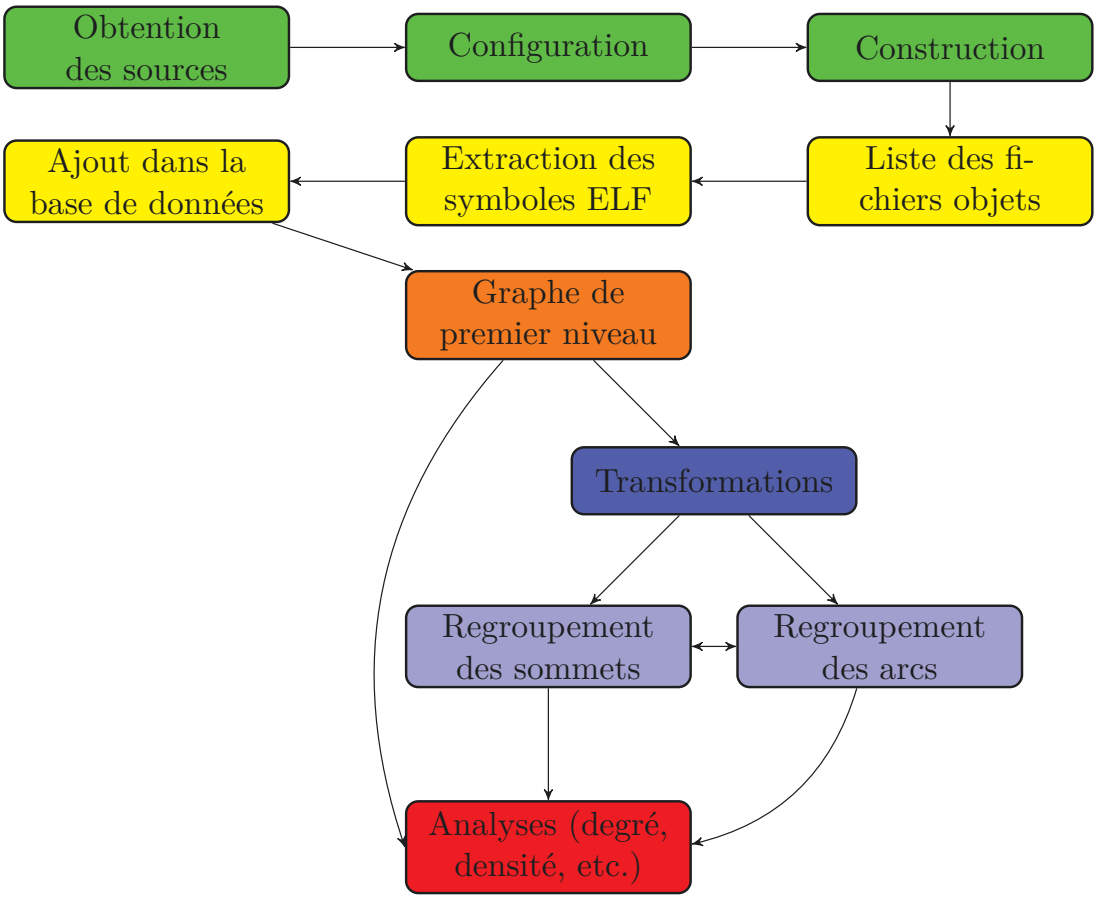


FIGURE 3.2 – Processus pour analyser le graphe du noyau : de la construction aux résultats

– Les transformations sont en bleu foncé ;
– Les regroupements de sommets et d'arcs sont en bleu clair ;

Le graphe « de premier niveau » mentionné dans la figure 3.2 indique que ce graphe sert de base à d'autres graphes, dont ceux obtenus par les différentes transformations proposées. Il peut être nécessaire de procéder à des regroupements sur le graphe pour simplifier certaines analyses. Par exemple, regrouper tous les sommets qui appartiennent à un même sous-répertoire. Deux variantes de ce graphe sont calculées, l'une contenant en plus l'arborescence du système de fichier sous la forme d'un graphe : ceci permettra de pouvoir corréler plus facilement certaines propriétés.

3.3 Étude du graphe

Après avoir présenté la manière dont nous construisons le graphe et justifié son application, le but de cette section est de présenter les différentes analyses que nous allons appliquer aux graphes que nous obtenons. Les résultats ainsi que leur explicitation seront

fournis dans une section suivante, 3.4. Dans la suite de cette section nous allons présenter pour chaque mesure effectuée : sa définition ainsi que ce qu'elle indique sur le code source. Pour chacune des mesures, il est à noter que le choix entre une vue « globale » et une vue « par composants » (pour le moment assimilés à des répertoires) peut avoir une influence.

3.3.1 Taille de la base de code

Les différentes versions du noyau Linux évoluent : les changements ajoutent et enlèvent du code, mais finalement, la quantité augmente toujours. Certaines analyses par la suite étant quantitatives, il est utile d'avoir comme référence la taille du code source qui a été analysé. Ce calcul est délégué à l'outil `SLOCCount`.

3.3.2 Mesure : occurrence des symboles

Comme présenté dans la section précédente 3.2, dans le graphe que nous construisons, les arcs correspondent aux symboles qui sont utilisés par les modules. Nous allons donc compter simplement le nombre de fois qu'un symbole est utilisé. Cela permet d'avoir une idée des sous parties les plus critiques du point de vue du nombre d'utilisateurs. Soit, pour un graphe $G = (E, V)$ et un symbole S :

$$occur_S = \sum_i |E_i|^S$$

Avec $|E|^S$ le nombre d'arcs portant le label correspondant au symbole S.

3.3.3 Mesure : densité du graphe

Pour avoir une idée du niveau de couplage au sein du noyau, il est intéressant de regarder la mesure de la densité d_G d'un graphe $G = (E, N)$, définie comme :

$$d_G = \frac{|E|}{|N| \times (|N| - 1)}$$

Notamment, entre les instances **defconfig** et **allyesconfig**, la densité devrait varier : elle devrait être plus importante dans le premier cas que dans le second cas, puisque les modules et pilotes doivent, logiquement, uniquement dépendre du cœur du noyau, et n'avoir que peu ou pas de dépendance entre eux. De plus, cette densité devrait potentiellement varier au sein des sous-répertoires.

3.3.4 Mesure : taille moyenne du chemin

La taille moyenne du chemin permet de mesurer la distance entre deux sommets, en moyenne. Par définition [1], c'est la somme des plus courtes distances de toutes les paires distinctes de sommets divisée par le nombre de ces paires. Pour une paire de nœuds non connectés, la distance la plus courte est fixée à 0. Soit, pour un graphe $G = (E, V)$:

1. provenant de la bibliothèque `Tulip`

$$avg_G = \frac{\sum_{i,j} d(v_i, v_j)}{|E| \times (|E| - 1)}$$

Avec, $d(v_1, v_2)$ la distance la plus courte entre les sommets v_1 et v_2, et $d(v_1, v_2) = 0$ si $v_1 = v_2$ ou s'il n'y a pas d'arc entre v_1 et v_2.

Entre les instances **defconfig** et **allyesconfig**, il devrait y avoir une évolution similaire de la densité et de la taille moyenne du chemin.

3.3.5 Mesure : degré entrant et sortant

Le degré d'un sommet nous permet d'avoir une vue sur la quantité de symboles qui sont utilisés :

– Le degré sortant indique le nombre de symboles qui sont exportés par le module ;
– Le degré entrant indique le nombre de symboles qui sont importés par le module.

Les piles applicatives au sein du noyau devraient donc avoir un degré sortant très élevé. Ainsi, dans le cas de l'instance **defconfig** il devrait y avoir beaucoup plus de sommets ayant un degré sortant élevé, et dans le cas de l'instance **allyesconfig**, le nombre de sommets avec un degré sortant faible devrait être beaucoup plus important.

3.3.6 Mesure : carte de chaleur

Pour avoir une meilleure vue des dépendances entre les sous-répertoires du noyau, nous proposons une visualisation sous la forme de « cartes de chaleur » : il s'agit d'un graphique ayant en ordonnée et en abscisse les sous-répertoires, et dont l'intersection correspond au nombre normalisé d'arcs entre ces deux modules. L'intérêt principal de cette représentation réside dans sa lisibilité. Le processus pour la construire se décompose en trois étapes :

1. D'abord, regrouper ensemble les sommets qui sont à une certaine profondeur par rapport à la racine du système de fichier, en conservant les arcs ;
2. Ensuite, compter le nombre d'arcs entre chaque paire de ces nouveaux sommets ;
3. Enfin, normaliser le nombre d'arcs.

Par construction, ces cartes présentent une matrice composée des répertoires étudiés et dont l'intersection contient la valeur décrite précédemment. L'abscisse correspond donc aux répertoires utilisateurs, et l'ordonnée aux répertoires utilisés.

3.4 Analyses des noyaux

La section 3.3 nous a permis de présenter les mesures prises, leur définition et leur intérêt, et nous allons maintenant exposer quels sont les résultats principaux, présentés par mesure, que nous pouvons retenir de cette étude. Les détails des analyses sont disponibles dans l'annexe A.

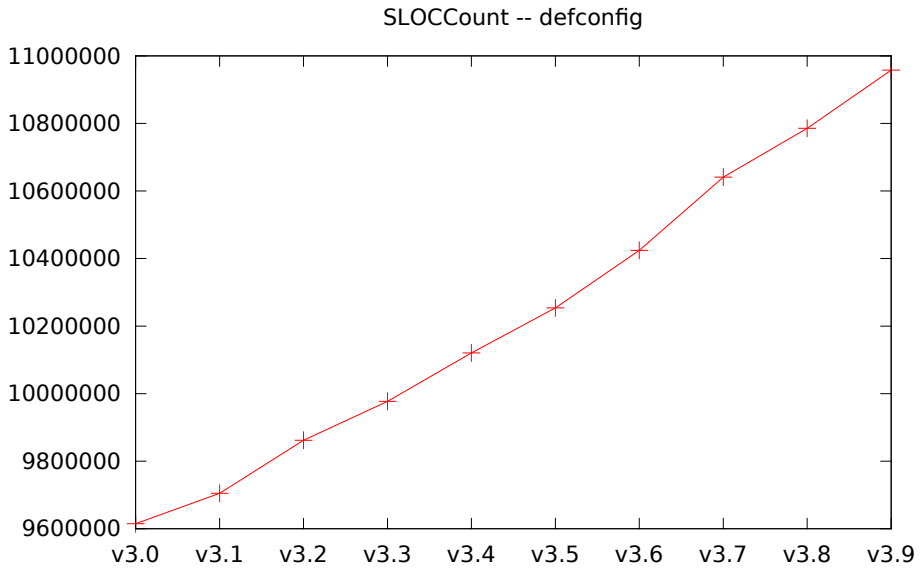


FIGURE 3.3 – Évolution de la quantité de code

3.4.1 Taille de la base de code

L'analyse de la taille du code du noyau est proposée dans l'annexe A.1 en exploitant à la fois la quantité de code et les dimensions du graphe, que ce soit sur la totalité du noyau mais également en s'intéressant de manière plus détaillée à chaque sous-répertoire. Nous reprenons dans la figure 3.3 le graphique documentant l'évolution de la quantité de code composant le noyau entre les versions 3.0 et 3.9 : près de 1.5 millions de lignes de code ajoutées. Ceci permet de montrer l'évolution constante, régulière et importante de la volumétrie du noyau. Dans le détail par sous-répertoire, proposé en annexe A.1.1.2, nous pouvons observer que seuls les répertoires `drivers/` et `kernel/` dépassent le million de lignes, la moitié des répertoires étant sous les 100 000.

L'analyse détaillée de la taille du graphe construit est proposée dans la section A.1.2, et un premier résultat est l'impact de la configuration choisie : passer de **defconfig** à **allyesconfig** augmente globalement le nombre de sommets et d'arcs, le facteur étant dépendant du sous-répertoire. L'évolution pour chacun n'est pas nécessairement équivalente. L'impact des différentes versions du noyau est également documenté : certains sous-répertoires voient leur quantité de code augmenter quand elle stagne pour d'autres. Le noyau est en expansion, de version en version, et cette expansion est d'autant plus marquée avec **allyesconfig**.

3.4.2 Utilisation des symboles

L'extraction des données depuis les fichiers objets et la construction du graphe nous donnent également l'occasion d'avoir une vue sur l'utilisation qui est faite des symboles (variables partagées, fonctions). Dans l'annexe A.2.1 nous détaillons la répartition de cette

Affichage	Mémoire	Verrouillage
`printk()`	`kfree()`	`mutex_unlock()`
`sprintf()`	`kmalloc_caches()`	`mutex_lock()`
	`kmem_cache_alloc_trace()`	`_raw_spin_lock()`
	`__kmalloc()`	`_raw_spin_unlock_irqrestore()`
	`memcpy()`	`_raw_spin_lock_irqsave()`
	`_copy_to_user()`	
	`_copy_from_user()`	

TABLE 3.1 – Grandes familles de symboles, avec quelques exemples

utilisation suivant leur fréquence d'apparition. Nous en déduisons d'abord que la distribution de l'utilisation des symboles présente des caractéristiques proches d'une distribution de PARETO, avec quelques symboles qui sont particulièrement utilisés, et ensuite une décroissance rapide. Nous constatons de plus que l'échelle des fréquences est faible. Dans le tableau 3.1 sont regroupés certains des symboles les plus utilisés, classés par famille.

Les symboles présents et les plus utilisés dépendent aussi de l'instance choisie, nous observons dans **allyesconfig** une augmentation importante de ceux liés à la gestion des pilotes (`dev_get_drvdata()` et `dev_set_drvdata()`). Cet impact est logique, étant donné la quantité de pilotes qui sont activés dans cette instance. Celle-ci active également certaines options de contrôle du code, qui font ressortir des symboles spécifiques liés à GCC (GCOV) et FTRACE (outil de traçage du code).

Nous étudions également quels sont les symboles les plus utilisés mais en considérant chaque sous-répertoire au lieu du noyau dans son ensemble. Les versions 3.0 et 3.9 dans les instances **defconfig** et **allyesconfig** sont proposés, dans les annexes A.2.3.2, A.2.3.3, A.2.3.4 et A.2.3.5. Il ressort ainsi que tous les sous-répertoires n'ont pas le même comportement, et certains ont des caractéristiques précises :

- Utilisation de familles de symboles générales (`kernel/`) ;
- Utilisation de familles de symboles spécialisées (`lib/`) ;
- Utilisation de familles de symboles générales et spécialisées correspondant à la pile concernée (`net/` ou `fs/`).

Au-delà des fréquences, il n'y a pas de différence majeure sur cette analyse entre les instances **defconfig** et **allyesconfig**. Les mêmes constats peuvent être formulés pour le noyau dans sa version 3.9 : le nombre de symboles augmente, mais la répartition et l'utilisation restent très proches.

3.4.3 Densité du graphe

La densité telle que définie dans la section 3.3.3 est documentée en détails dans l'annexe A.3. Une première vue de l'ensemble du noyau est proposée sur l'intervalle de versions 3.0 à 3.9. L'instance **defconfig** se montre chaotique mais le bilan est stable (moyenne à 0.01516) alors que **allyesconfig** expose une baisse faible mais régulière. La moyenne des valeurs dans ce cas est plus faible (0.0033), ce qui s'explique par le plus grand nombre de nœuds et l'augmentation relativement plus faible du nombre de liens. Cette répartition est également

plus dispersée, l'écart-type augmentant.

L'analyse par sous-répertoire permet de mieux comprendre comment chacun évolue, et la caractéristique de chacun se lit dans la densité : les répertoires contenants des pilotes (`drivers/`, `fs/`, `net/`, ...) ont une densité faible au contraire de ceux fournissant un service (`mm/`, ...). Ce constat se visualise en comparant les instances **defconfig** et **allyesconfig**, dans laquelle ce type d'élément plutôt indépendant à plus de chance d'exister. Le répertoire avec la densité la plus importante et constante est `ipc/`. La lecture de l'évolution temporelle par sous-répertoire met également en évidence les modifications de sous-systèmes : nous pouvons ainsi identifier des changements brutaux. La migration d'une partie du code depuis `fs/partitions/` vers `block/partitions/` entre les versions 3.2 et 3.3 : baisse importante dans le sous-répertoire `block/`, augmentation importante mais moindre sur `fs/`.

3.4.4 Taille moyenne du chemin

La taille moyenne du chemin, qui renseigne sur la largeur du graphe, est documentée en détails dans l'annexe A.4 : par sous-répertoires pour chaque instance (**defconfig**, **allyesconfig**). Cette valeur évolue entre 1 et 3 arcs à traverser pour parcourir le sous-ensemble du graphe dans la largeur, suivant les répertoires analysés. Nous en déduisons que certains répertoires ont un comportement instable dans le temps, qui correspondent à des évolutions importantes dans le code (`block/`, `init/` et `arch/`). Nous pouvons former des groupes de répertoires suivant leur valeur de taille moyenne de chemin, qui correspondent à des besoins du noyau similaires : `lib/` met à disposition des bibliothèques et présente une taille moyenne de chemin de 1.0 ; `kernel/` et `mm/` fournissent des sous-systèmes de base et sont à une valeur comprise entre 1.9 et 2.0 suivant l'instance considérée ; `drivers/`, `fs/` et `net/` qui contiennent des pilotes et dont la valeur de taille moyenne évolue entre l'instance **defconfig** et **allyesconfig**. Après une autre analyse sur la totalité des sous-répertoires, nous constatons que ceux contenant des bibliothèques avec peu de liens entre elles ont une valeur de taille moyenne < 1.5, les composants de base du noyau évoluent entre 1.9 et 2.3 alors que les pilotes sont au-delà de 2.6. La figure 3.4 montre l'évolution par sous-répertoire, entre les versions 3.0 et 3.9 du noyau.

3.4.5 Degrés des sommets

Dans l'annexe A.5 une étude sur les degrés des sommets est proposée (degré entrant, sortant, et total), et s'intéresse à caractériser l'évolution temporelle de ces valeurs. Une illustration est proposée en figure 3.5 avec l'instance **defconfig**. L'impact de l'instance se mesure facilement en lisant les graphiques, puisque l'échelle de ceux-ci est multipliée par 10 entre **defconfig** et **allyesconfig**. Les variations pour les différents sous-répertoires ne sont pas uniformes mais restent proches. Par exemple, après normalisation des données, nous pouvons estimer une progression d'un facteur 1.062 sur le sous-répertoire `lib/` pour l'instance **defconfig**, alors que le répertoire `drivers/` montre dans ce cas un facteur de 1.114. Ce facteur passe respectivement à 1.242 et 1.217 pour l'instance **allyesconfig**. Une comparaison du rapport entre le degré et le nombre de nœuds est documenté dans le tableau A.8. Elle met en lumière des chiffres stables dans le temps et dépendant du

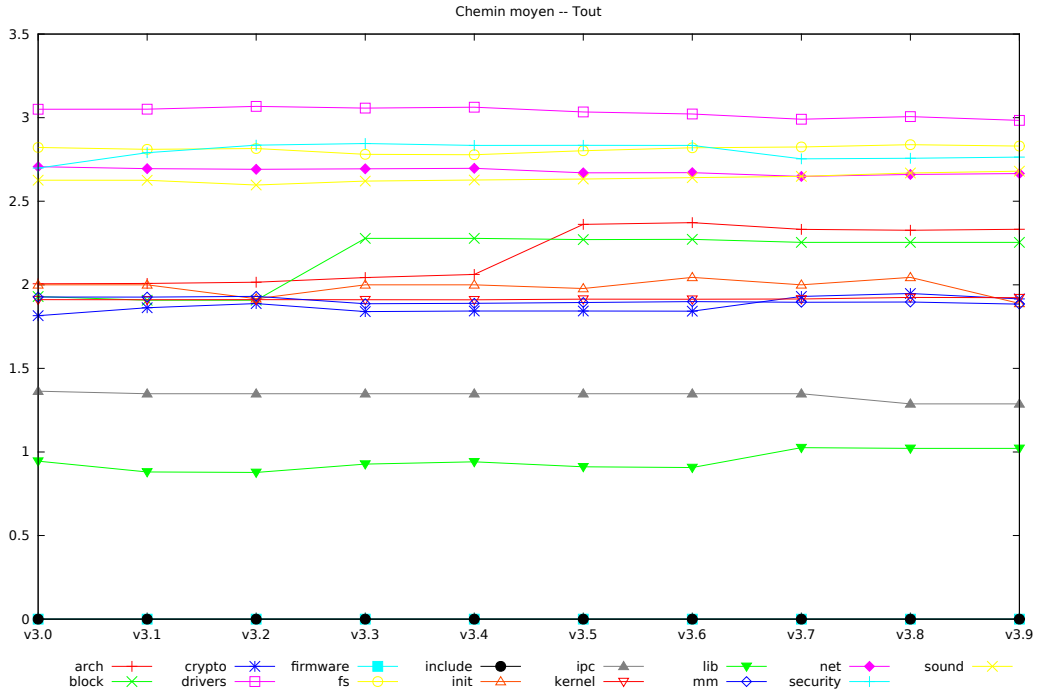


FIGURE 3.4 – Évolution de la taille moyenne du chemin dans le graphe, par sous-répertoire, entre les versions 3.0 et 3.9, pour l'instance **allyesconfig**.

répertoire considéré et de la configuration choisie.

Les degrés entrants (détaillés dans l'annexe A.5.2.1) et sortants (détaillés dans l'annexe A.5.2.2) n'évoluent pas de manière similaire. Nous effectuons la comparaison par rapport à la mesure de degré total. Par exemple, sur le sous-répertoire `lib/`, le passage de l'instance **defconfig** à **allyesconfig** se traduit par une augmentation du degré total d'un facteur de 6.5, le degré entrant est multiplié par un facteur 7.7. La répartition par sous-répertoires des valeurs de degrés entrant permet de mieux constituer des groupes : un premier groupe constitué des répertoires `kernel/` et `drivers/`, un second groupe formé par `mm/`, `lib/`, `fs/`, `arch/` et `net/` et un dernier groupe où l'on retrouve notamment `block/`, `crypto/`, `security/`. Le répertoire `sound/` se retrouve isolé. Le degré sortant évolue de manière différente et complémentaire.

Enfin le tableau A.11 proposé en annexe compare sur les versions 3.0, 3.5 et 3.9 le rapport calculé entre les degrés entrant et sortant. Pour l'instance **defconfig**, ce rapport est supérieur à 1 pour certains répertoires : `arch/`, `kernel/`, `lib/` et `mm/` ; et il évolue entre 0.5 et 0.75 pour `drivers/`, `fs/`, `net`, `security` et `sound`. Ceci est une bonne illustration du rôle de chacun, et la comparaison avec l'instance **allyesconfig** confirme ces résultats.

3.4.6 Carte de chaleur

Pour visualiser et documenter de manière plus lisibles la structure du noyau, nous proposons une vue sous la forme d'une carte de chaleur. Les détails sont donnés dans l'annexe

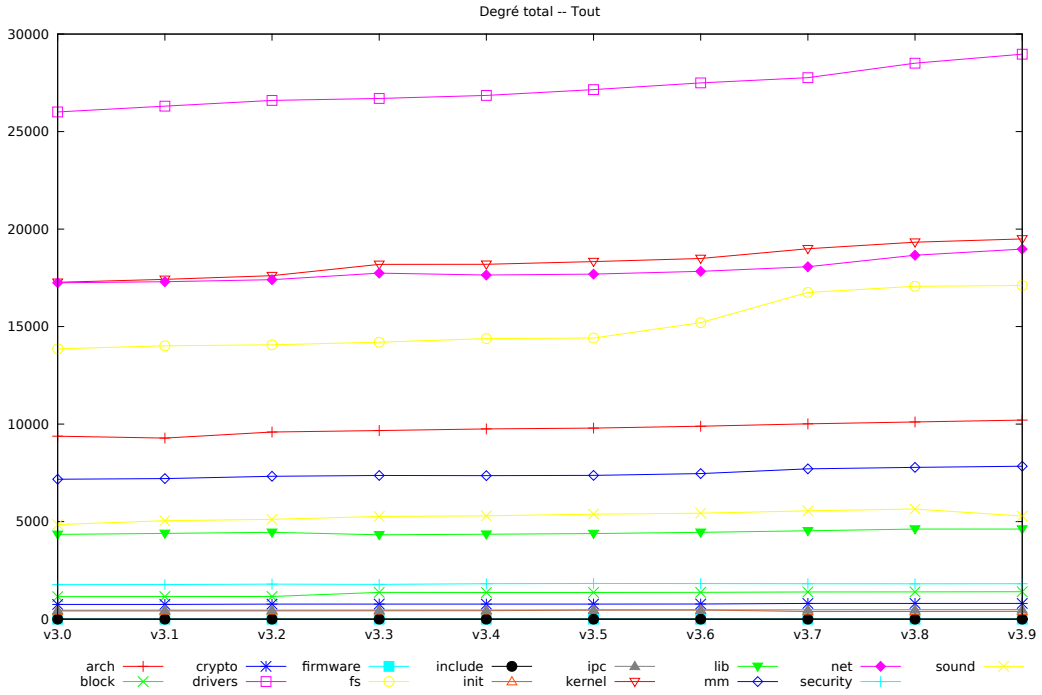


FIGURE 3.5 – Évolution du degré total dans le graphe, instance **defconfig** pour le noyau entre les versions 3.0 à 3.9

A.6, et l'analyse est proposée à la fois sur l'intégralité du noyau (étude des interactions entre le premier niveau de répertoires) et sur certains répertoires. L'intensité de chaleur est représentée par une couleur à l'intersection de deux répertoires et indique la quantité d'arcs orientés entre ces deux répertoires.

Deux résultats importants sont disponibles sur toutes ces cartes : la mise en lumière de répertoires transversaux, et la position de la première bissectrice. Dans les cartes, nous pouvons observer la présence de lignes (horizontales) quasi continue, d'intensité moyenne : celles-ci traduisent l'utilisation d'un répertoire par beaucoup d'autres, et correspondent à des bibliothèques importantes ou bien à des piles applicatives majeures (USB, PCI). Les sous-répertoires `mm/`, `lib/` et `kernel/` sont particulièrement utilisés par les répertoires `drivers/`, `fs/` ainsi que `net/`. La première bissectrice apparait également de manière très lisible et elle indique un fort taux de symboles définis et utilisés au sein du répertoire. C'est le cas des répertoires `drivers/` et `net/`. Ces résultats sont valables sur toutes les versions étudiées du noyau, et avec toutes les instances de configurations (**defconfig**, **allyesconfig**). Des variations d'intensité existent, mais la tendance générale est la même.

Les sous-répertoires `drivers/`, `fs/`, `kernel/` et `net/` sont étudiés en détails respectivement dans les annexes A.6.2, A.6.3, A.6.4 et A.6.5. En effectuant la comparaison entre les cartes des versions 3.0 et 3.9, nous pouvons constater les évolutions qui ont lieu dans le temps : apparition de nouveaux sous-systèmes, déplacements de code, développement de piles nouvelles. Nous proposons les cartes pour l'instance **allyesconfig** du noyau sur le sous-répertoire `drivers/` dans les figures 3.6 et 3.7. Les échelles de valeurs sont les

mêmes ; et nous visualisons bien à la fois la première bissectrice, et deux piles applicatives importantes (`base/` et `pci/`).

3.5 Conclusion

Dans ce chapitre, nous avons proposé une modélisation du noyau Linux en vue de pouvoir l'analyser : identifier des composants, caractériser les liens entre ces composants. L'objectif est à la fois de documenter la structure du code, expliquer comment interagissent les parties, détailler les liens qui unissent les composants ; également de pouvoir exploiter les informations ainsi obtenues pour détecter des « communautés » au sein du code, i.e., des sous-ensembles qui interagissent fortement entre eux. La mise en exergue de ces communautés permettrait de vérifier de potentiels soucis d'organisation au sein du code source (couplage trop fort entre certains composants), mais également, et surtout, d'être en mesure de procéder à un découpage en vue d'une analyse statique approfondie. En effet, les outils qui effectuent ces analyses restent limités par la quantité d'états à traiter. Savoir découper le code source en des sous-ensembles qui auraient une limite sur le nombre d'états à traiter autoriserait à appliquer ses outils ; connaître les relations entre ces sous-ensembles devient nécessaire afin de proposer des vérifications fortes et intéressantes.

Nous avons donc proposé une modélisation sous la forme d'un graphe orienté, composé des fichiers objets obtenus pendant la compilation – cela limite donc la portée de l'analyse, potentiellement à l'architecture pour laquelle cette compilation a été effectuée – et des symboles qui sont utilisés entre ces fichiers objets. Cette modélisation permet déjà de tirer quelques enseignements sur les caractéristiques du noyau :

- Au travers des différentes versions, le noyau voit sa quantité de code augmenter ;
- L'activation de tous les modules optionnels a un impact non négligeable, et change de manière assez importante certaines relations ;
- Des chiffres permettant de quantitifier non seulement le nombre de lignes de code (une information souvent fournie lors des annonces de nouvelles versions), mais surtout une quantification du nombre de nœuds et d'arcs, permettant de mieux saisir la complexité ;
- La présentation de la distribution des symboles ainsi qu'une mise en avant des symboles les plus utilisés, par une analyse de chacun des sous-répertoires ;
- Une documentation de la puissance des liens entre les différents sous-répertoires grâce à une analyse de la densité, de la taille du chemin moyen, ainsi que l'étude des degrés des différents sommets.

Nous avons également proposé une manière de mieux « visualiser » les liens entre les sous-répertoires et leur importance au travers de cartes de chaleur.

Toutes ces analyses ont éte effectuées sur une série de 10 versions du noyau, du 3.0 (22 juillet 2011) au 3.9 (29 avril 2013), couvrant ainsi deux ans d'évolution. Afin de rendre compte de l'impact des différents modules activables, les configurations **defconfig** et **allyesconfig** ont été comparées pour chaque version du noyau, même si pour l'illustration il arrive que seules les versions 3.0 et 3.9 soient présentées. Une première version a été présentée [103] lors de la conférence *Ottawa Linux Symposium 2012*, pour débattre de la pertinence de l'approche choisie et des premiers résultats obtenus. Une seconde communi-

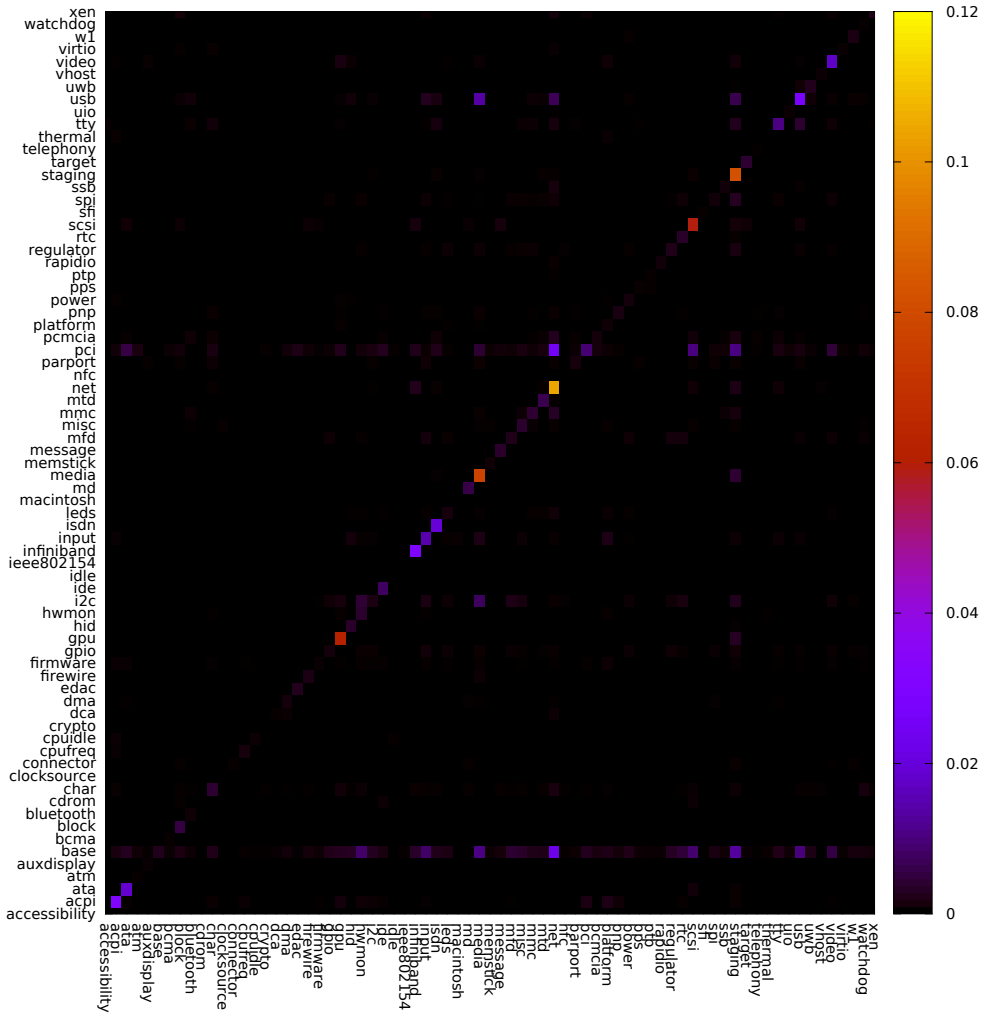


FIGURE 3.6 – Cartes de chaleur du noyau 3.0, instance **allyesconfig**, pour le sous-répertoire `drivers/`.

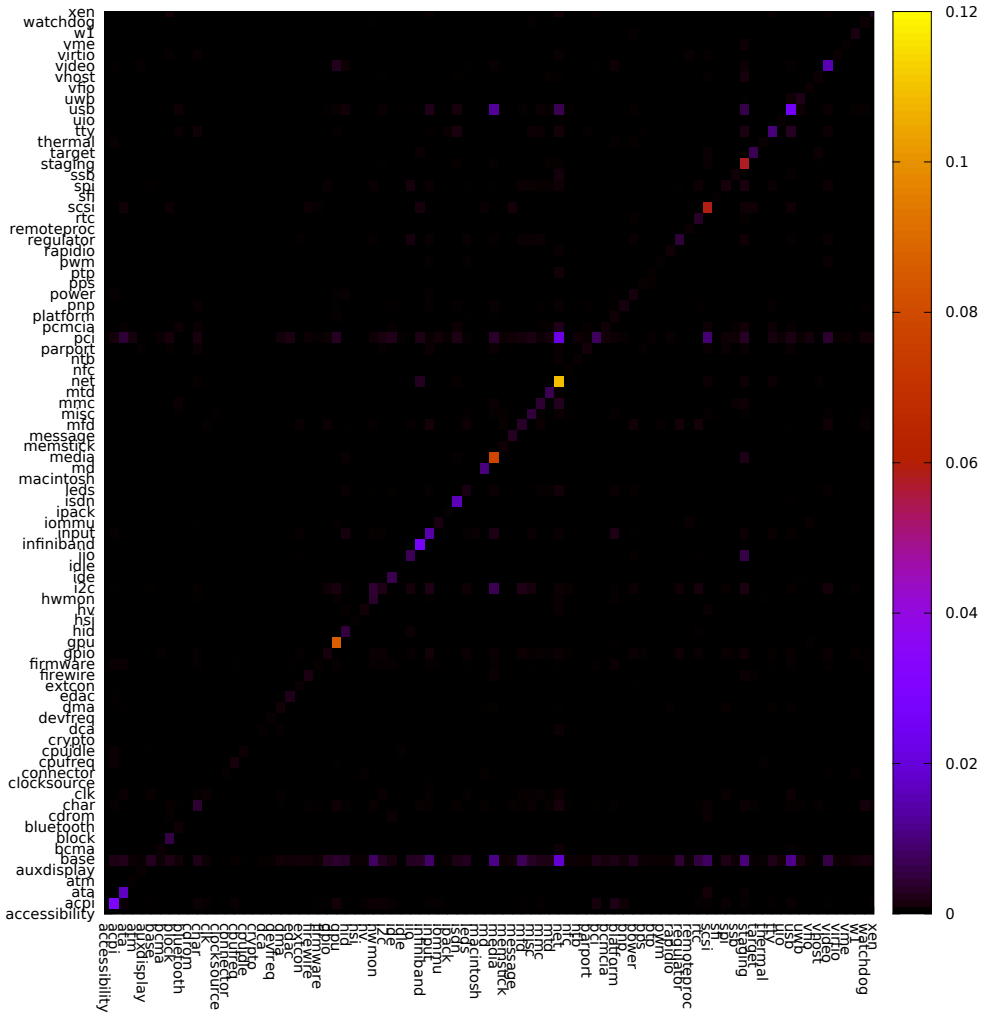


FIGURE 3.7 – Cartes de chaleur du noyau 3.9, instance **allyesconfig**, pour le sous-répertoire `drivers/`.

cation a eu lieu [99] pendant le *Linux Driver Verification Workshop (ISoLA 2012)*.

Une limite importante de cette analyse est qu'elle n'est valable que pour l'architecture ciblée, à savoir **amd64**. Les combinaisons d'options et le comportement du sous-répertoire `arch/` pourraient être différents sur d'autres architectures. Une autre limitation tient au choix de la construction du graphe : nous sommes partis de fichiers objets, i.e., après la compilation. Ce point a déjà été abordé dans la section 3.2.1.2. Il implique notamment :

- La perte d'un peu d'information par rapport à l'analyse des sources ;
- Un temps de construction et d'extraction loin d'être négligeable (plusieurs jours pour la série sélectionnée).

Ces résultats permettent de mieux comprendre la topologie du graphe que nous avons créé, et seront utiles pour sélectionner les différentes possibilités lors de la détection de communauté : en effet, suivant les caractéristiques recherchées et/ou le graphe étudié, les algorithmes de la communauté ne donnent pas les mêmes résultats.

Chapitre 4

Application de la détection de communautés au noyau Linux

4.1 Introduction

L'étude de l'état de l'art a mis en évidence que la mise en application d'outils de model-checking restait limitée par le nombre d'états à analyser. Nous avons proposé d'effectuer une modélisation du code composant le noyau sous la forme d'un graphe pour être en mesure de l'analyser et de le « découper » en sous parties analysables. L'étude des caractéristiques de base de ce graphe a montré que la modélisation était pertinente : nous pouvons mesurer et visualiser certaines caractéristiques propres à certains sous-systèmes qui le composent. Cette analyse repose sur les liens qui unissent les fichiers objets entre eux, en suivant la structure hiérarchique que compose l'arborescence des sources : c'est déjà une manière de constituer des groupes de code source, ceux qui ont, normalement, vocation à vivre ensemble. Cette structure imposée étant arborescente, elle limite de fait la vision communautaire du code source. Utiliser les outils existants de détection de communautés au sein d'un graphe nous permettra, notamment :

- De vérifier la pertinence de l'arborescence du code source
- De définir des communautés basées sur les liens dérivés de l'utilisation des symboles
- De calculer des communautés qui optimisent certains critères

L'étude de ces communautés permettra également de mieux comprendre et explorer le code source du noyau.

Dans une première section 4.2 nous présenterons l'état de l'art en nous appuyant sur une référence existante et reconnue du milieu de la détection de communautés, afin de présenter les grandes méthodes possibles dans la section 4.3 et faire le parallèle avec notre problème et ses caractéristiques. Par la suite, la section 4.4 nous permettra de présenter la méthodologie pour évaluer les différentes méthodes retenues et les comparer, puis les résultats obtenus seront présentés et analysés dans la section 4.5.

4.2 État de l'art de la détection de communautés

Sur la thématique de la détection de communautés dans les graphes, l'article [66] de Fortunato, fait figure de référence avec plus de 2000 citations dans la litérature selon Google Scholar. Nous allons nous baser sur celui-ci pour évaluer l'état de l'art. Le but dans ce cas n'est pas de présenter une sélection de méthodes intéressantes, mais de montrer leur étendue et classification ; la confrontation avec notre contexte sera proposée dans la section 4.3. Nous reprenons donc en partie le plan de l'auteur original et nous proposerons de résumer son analyse. Dans une première sous-section 4.2.1 nous introduisons les définitions existantes des notions de communauté, une mise en contexte avec les différents domaines d'applications possibles, et les objectifs de l'état de l'art. La sous-section 4.2.2 concentre le cœur de ce résumé, et propose de faire le tour des différentes approches existantes référencées dans cet état de l'art. La dernière sous-section 4.2.3 propose de faire le lien avec les cas réels, notamment en étudiant quelques propriétés de vraies communautés ainsi que l'application à des cas réels.

4.2.1 Communautés et graphes, premiers pas

Les graphes sont un outil mathématique assez ancien, et leur utilisation pour modéliser des systèmes n'a fait que croître au cours du 20^{e} siècle. Un exemple donné concerne l'étude des réseaux sociaux, c'est-à-dire l'étude d'un graphe qui représente les liens de relations entre des individus, qui a décollé dès les années 1930, devenant un sujet important en sociologie ; par ailleurs la taille des graphes qui peuvent être traités a considérablement augmenté.

Dans la sous-section 4.2.1.1 nous introduirons la notion de graphe aléatoire nécessaire pour définir ce qu'est une communauté du point de vue du graphe ; puis dans la sous-section 4.2.1.2 nous présenterons des domaines d'applications et les objectifs associés qui illustrent l'intérêt de la détection de communautés.

4.2.1.1 Qu'est-ce qu'une communauté ?

Dans un premier temps, l'auteur introduit la notion de graphe aléatoire, dont la caractéristique principale est l'homogénéité de la distribution des arcs : le degré suit une loi binomiale. L'absence de cette homogénéité dans les graphes construits à partir de la réalité est l'essence de la notion de communauté : des regroupements de nœuds avec un haut degré. Particulièrement, le graphe dans son ensemble va présenter une répartition des degrés qui sera fortement hétérogène. Cette notion ne se limite pas à l'organisation et les relations sociales, mais se retrouve également dans de nombreux systèmes : biologie, économie, politique, etc.

Quelques exemples de communautés sont proposés pour souligner son impact :
- Étude des réseaux d'interaction entre les protéines : les communautés du graphe représentent celles qui ont des fonctions similaires ;
- Dans un graphe correspondant à Internet, les communautés vont représenter les pages qui traitent d'un sujet similaire ou proche ;

Le but de la détection de communauté est donc d'identifier des caractéristiques du graphe (identifier des modules, leurs relations), seulement en exploitant la topologie de celui-ci.

Cette définition est assez large. En fait, l'auteur insiste sur l'absence d'une définition pleine et entière de la notion de communauté ou de partition ; c'est ainsi qu'il propose d'exposer les différentes définitions des communautés et des partitions. Dans le premier cas, après avoir introduit les bases, il présente les définitions locales puis les globales avant de terminer avec celles basées sur la similarité des nœuds. Pour la notion de partitions, après avoir présenté les bases, ce sont les fonctions de qualité liées à la modularité qui sont introduites. Nous allons résumer les éléments importants de ces définitions.

4.2.1.1.1 Communautés

4.2.1.1.1.1 Bases Dans un premier temps, l'auteur introduit certaines bases communes aux définitions : d'abord l'intuition (sic), basée sur les remarques précédentes, qu'une communauté dans un graphe correspond à une zone où il y a plus d'arcs entre les sommets par rapport au reste du graphe. Cependant, cette première approche n'est pas suffisante ni unique et l'auteur précise que la définition reste liée au système étudié.

L'auteur introduit ensuite deux mesures de densité. Soit un sous-graphe $\mathcal{C}$ d'un graphe $\mathcal{G}$, ayant respectivement n_c et n sommets. Et soient $E_{int}(\mathcal{C})$ le nombre d'arcs internes à $\mathcal{C}$, $E_{ext}(\mathcal{C})$ le nombre d'arcs sortants de $\mathcal{C}$.

Une première mesure proposée est la *densité intra-cluster* $\delta_{int}(\mathcal{C})$, définie par :

$$\delta_{int}(\mathcal{C}) = \frac{E_{int}(\mathcal{C})}{n_c \frac{(n_c - 1)}{2}}$$

Une seconde mesure proposée est la *densité inter-cluster* $\delta_{ext}(\mathcal{C})$, définie par :

$$\delta_{ext}(\mathcal{C}) = \frac{E_{ext}(\mathcal{C})}{n_c(n - n_c)}$$

Grâce à ces définitions, l'auteur indique que la recherche de communautés peut commencer : pour qu'un sous-graphe $\mathcal{C}$ puisse être considéré comme tel, alors $\delta_{int}(\mathcal{C})$ devrait être très largement supérieur à (sic) la densité moyenne du graphe dans sa totalité donnée par $\delta(\mathcal{G})$; et $\delta_{ext}(\mathcal{C})$ devrait être largement inférieur (sic). L'auteur ne fournit pas, du moins à ce stade, plus de précision sur l'écart attendu.

Par la suite, trois familles de définitions sont présentées : locales en paragraphe 4.2.1.1.1.2 (les plus nombreuses), globales en paragraphe 4.2.1.1.1.3 et enfin celles basées sur la mesure de similarité en paragraphe 4.2.1.1.1.4.

4.2.1.1.1.2 Définitions locales Dans l'approche locale, le focus est donné sur le sous-graphe que l'on étudie en négligeant le reste du système. L'auteur se base notamment sur les travaux de Wasserman et Faust [185] autour de l'analyse de réseaux sociaux pour retenir quatre critères :

- Mutualité complète
- Atteignabilité
- Degré du sommet
- Comparaison de la cohésion interne et externe

Les communautés qui répondent à ces critères sont des sous-graphes maximaux, i.e., auxquels on ne peut ajouter d'arc ni de sommet sans perdre leurs propriétés (maximalité). Ces communautés sociales peuvent ainsi être des cliques au sens des graphes : tous les sommets sont adjacents (mutualité complète) ; cependant l'auteur relève que cette approche est beaucoup trop stricte pour coller à la réalité.

Ce qui justifie une relaxation avec des sous groupes *proches* d'une clique : une *n-clique* est un sous-graphe maximal tel que la distance entre chaque paire de sommets ne dépasse pas n. Avec $n = 1$, on retrouve la définition d'une clique classique. Cependant, cette définition pose encore des soucis : si le diamètre du graphe excède n, et si le sous-graphe est déconnecté. L'auteur cite les travaux [112] de Mokken qui proposent deux alternatives : *n-clan*, une *n-clique* dont le diamètre ne dépasse pas n, et *n-club*, un sous-graphe maximal de diamètre n.

Toujours dans le domaine de l'étude des réseaux sociaux, une autre définition se base sur le fait qu'un sommet doit être adjacent à un minimum d'autres dans le sous-graphe : ainsi, un *k-plex*[161] est un sous-graphe maximal dans lequel chaque sommet est adjacent à tous les autres sauf au plus k sommet, ; dans le cas *k-core*[160], chaque sommet est adjacent à au moins k autres sommets.

L'auteur présente ensuite la définition de communauté basée sur la cohésion interne et externe : définie par les travaux [108] de Luccio et Sami comme *LS-set* puis *strong community* par Radicchi et al. dans [139], il s'agit d'un sous-graphe tel que le degré interne de chaque sommet est supérieur à son degré externe. Une définition relaxée est donnée dans [139].

La dernière définition locale proposée par l'auteur repose sur l'utilisation d'une mesure de fitness permettant de calculer la satisfaction d'un sous-graphe à une propriété liée à sa cohésion : plus cette mesure sera forte, plus la communauté sera « sûre ». L'auteur cite comme première approche la mesure de densité intra-cluster préalablement définie, $\delta_{int}(\mathcal{C})$; ainsi que la densité relative donnée par $\rho(\mathcal{C})$, dont l'auteur cite [163] prouvant que la recherche d'un sous-graphe d'une taille donnée avec la contrainte d'une limite sur $\rho(\mathcal{C})$ est NP-complet. Cette mesure est définie comme le rapport entre les degrés internes et total du sous-graphe $\mathcal{C}$.

4.2.1.1.1.3 Définitions globales Si les communautés sont des parties essentielles – i.e., leur retrait a un impact non négligeable – du graphe, alors l'auteur propose de s'intéresser à des définitions qui en tiennent compte. L'idée principale de la littérature est qu'un graphe qui a des communautés est différent d'un graphe aléatoire : ceci permet d'introduire la notion de *null model*, un graphe qui est aléatoire sauf pour certaines fonctionnalités où il correspond au graphe originel. Si plusieurs modèles existent, l'auteur en retient un particulier, par Newman et Girvan dans [119] : celui-ci est à la base de la mesure de *modularité* qui sera présenté plus tard.

4.2.1.1.1.4 Similarité des sommets L'étude de la similarité des sommets est utilisée dans de nombreuses techniques de détection de communautés, l'auteur propose de présenter certaines des mesures populaires dans la littérature. Une première série propose de placer le graphe dans un espace Euclidien à n dimensions, puis d'utiliser des mesures de distance pour calculer la similarité : distance Euclidienne, distance de Manhattan, distance infinie voire encore une mesure de similarité cosinus.

Dans le cas où le graphe ne peut être adapté à un espace Euclidien, l'auteur relève la proposition d'une autre distance exploitant les matrices d'adjacence, ainsi qu'une méthode mesurant le recouvrement entre les voisins de deux sommets. Une dernière technique basée également sur l'équivalence structurelle introduite par [107] est la corrélation de Pearson, sur les colonnes et les lignes de la matrice d'adjacence.

L'auteur présente ensuite une autre méthode qui est populaire dans la littérature : il s'agit de compter le nombre de chemins indépendants entre deux sommets, ce qui revient plus ou moins au problème de flux maximum.

Le dernier cas proposé exploite les propriétés des « marches aléatoires » : étudier le nombre moyen d'étapes à un marcheur aléatoire pour partir d'un sommet et atteindre un autre sommet.

4.2.1.1.2 Partitions

4.2.1.1.2.1 Bases Une partition est un sous-ensemble du graphe, tel que chaque sommet appartienne à un cluster. L'auteur ne précise pas à cet instant si l'appartenance est stricte ou non. Ces partitions peuvent être ordonnées hiérarchiquement : un exemple donné par l'auteur est celui d'un réseau social d'enfants dans une ville. Il est possible de les regrouper de différents manières : suivant les écoles fréquentées, mais aussi suivant les classes dans les écoles, etc. L'auteur insiste sur le fait que la structure hiérarchique est une caractéristique forte des réseaux observés dans la réalité, et rappelle l'utilisation des dendogrammes pour représenter cette structure.

4.2.1.1.2.2 Modularité Afin de distinguer une *bonne* partition d'une *mauvaise*, il est nécessaire d'avoir des critères : l'auteur introduit ainsi la notion de fonction de qualité permettant de calculer cette information. Étant donné que certains algorithmes vont proposer plusieurs partitions dans le graphe, ces fonctions permettent d'effectuer un tri qualitatif : plus le score sera élevé, plus la partition sera jugée de bonne qualité.

L'auteur commence par présenter une première mesure, dénommée *performance* (P) d'une partition $\mathcal{P}$, et définie par :

$$P(\mathcal{P}) = \frac{|i,j \in E, C_i = C_j| + ||i,j \notin E, C_i \neq C_j|}{n(n-1)/2}$$

Par définition, $0 \leq P(\mathcal{P}) \leq 1$. Le but de cette mesure est de compter le nombre de paires de sommets qui ont été correctement déduites.

La modularité Q définie par Newman et Girvan dans [119], reste la mesure la plus populaire pour l'évaluation d'un partitionnement de nœuds dans un graphe. Celle-ci est

obtenue par la différence moyenne entre les densités des classes données par la partition et celles obtenues si les arêtes étaient dispersées de manière aléatoire dans le graphe (*null model*). Ceci est traduit dans la formule suivante :

$$Q = \frac{1}{2m} \sum_{i,j} \Big(A_{i,j} - P_{i,j} \delta(C_i, C_j) \Big)$$

- m est le nombre total d'arcs dans le graphe
- A représente la matrice d'adjacence
- $P_{i,j}$ est le nombre d'arcs entre deux sommets i et j du modèle nul
- $\delta(C_i, C_j)$ vaut 1 si les sommets i et j appartiennent à la même communauté, 0 sinon

Il est à noter que la modularité dépend fortement du modèle null choisi. Ainsi, l'auteur insiste sur ce choix qui va avoir un impact important et propose plusieurs pistes pour l'effectuer. Ensuite, plusieurs résultats quant à la formule de modularité exposée précédemment sont formulés, sous les contraintes de *bons* modèles. De plus il est indiqué qu'il faut utiliser cette mesure avec parcimonie ; notamment, comparer des modularités de structure de communauté dans des graphes de taille très différente n'est pas conseillé.

4.2.1.2 Domaines d'applications et objectifs

Comme indiqué dans la sous-section précédente, tous les domaines où il est possible de modéliser sous la forme d'un graphe peuvent bénéficier de la détection de communautés : ce mécanisme est au cœur des systèmes de recommandation de produits pour les magasins en ligne, l'objectif ici étant de maximiser le panier moyen du client ; la sélection de serveurs à utiliser pour répondre à des requêtes de clients web qui partagent des caractéristiques communes (intérêts, position géographique) illustre l'utilisation dans le cadre d'un mécanisme d'optimisation. D'autres applications sont proposées avec les réseaux de télécommunication autonomes, où les communautés ont l'avantage de fournir des sous-groupes dont la table de routage est compacte.

Par la suite, l'auteur indique que l'identification des groupes et surtout de ses frontières dans le graphe permet une classification, particulièrement en rapport avec la position structurelle au sein des groupes : des nœuds partageant un grand nombre d'arcs avec d'autres groupes peuvent avoir une fonction de contrôle et de stabilité ; alors que ceux vivant à proximité des frontières du groupe vont avoir un rôle quant aux interactions et aux communications entre les groupes. Ces classifications sont particulièrement valables dans le cadre de réseaux sociaux et métaboliques.

Enfin la détection des communautés permet également d'opérer une classification d'un point de vue hiérarchique entre ces modules : des exemples sont cités, que ce soit l'organisation du corps humain (composé d'organes eux-mêmes composés de cellules), une entreprise (avec une organisation pyramidale). L'auteur cite notamment une étude [164] de 1962 sur ce sujet, montrant que l'organisation d'une société sous forme de petites unités structurées ensemble permet finalement un processus plus robuste.

Le domaine du calcul parallèle s'intéresse également à ce sujet : la communication entre des tâches qui s'exécutent sur des processeurs différents a un coût très important qui va avoir un impact notable en matière de performance. L'étude de problème correspond à

celui de partitionnement de graphe.

4.2.1.3 Exemples de communautés

L'auteur propose un inventaire de quelques communautés qui ont été étudiées dans la littérature voire qui servent à comparer les algorithmes.

D'abord, dans la catégorie des réseaux sociaux, une communauté test récurrente est le réseau des membres d'un club de Karaté introduit par Zachary dans [196]. Ce graphe est construit de l'observation des 34 membres d'un club de sport aux États-Unis : chaque sommet est un membre, et un arc représente des interactions sociales en dehors du club. Suite à une scission du groupe autour du président et du professeur respectivement, l'auteur cherchait à retrouver la composition de ces deux groupes à partir du graphe précédent la scission. Deux communautés doivent être détectées au sein de ce graphe, chacune centrée sur l'un des responsables (président, professeur). Des arcs existent entre ces deux groupes et sont souvent mal classifiés.

Un second exemple proposé concerne une étude des collaborations entre les chercheurs du *Santa Fé Institute* : les sommets correspondent aux auteurs des études, et un arc existe quand ils collaborent pour une publication. Les groupes observables correspondent essentiellement aux disciplines : il y a peu de travaux interdisciplinaires.

Lusseau et al. étudie un groupe de 62 dauphins : le graphe est construit en plaçant un arc entre deux animaux s'ils se croisent plus que ce que le hasard ne les aurait fait se croiser. Ce cas est souvent utilisé pour valider les algorithmes de détection de communautés.

4.2.2 Méthodes présentées

Dans la suite de son état de l'art, l'auteur expose en détail les différentes méthodes de la littérature pour effectuer de la détection de communauté. Notre objectif au sein de cette sous-section est d'en présenter un rapide résumé qui permettra d'avoir une vue d'ensemble de ces méthodes, puis dans la sous-section suivante, de retenir et présenter les caractéristiques discriminantes qui justifient la sélection des techniques que nous étudierons. Le plan des approches qui sont présentées suit celui de l'auteur original.

4.2.2.1 Partitionnement traditionnel

L'auteur commence par rappeler le problème de partitionnement de graphe : diviser les sommets en g groupes de taille prédéfinie, en minimisant la taille de la coupe. Il est important d'imposer le nombre de groupes, sinon une solution triviale existe : tous les sommets dans un seul cluster, ce qui donne une coupe minimale parfaite. L'auteur présente plusieurs variantes de ce problèmes qui sont NP-difficile.

4.2.2.1.1 Partitionnement de graphe L'algorithme de *Kernighan-Lin* [81] est souvent utilisé, rarement seul, et est l'un des premiers qui a été proposé pour ce problème : à l'origine il permettait de travailler sur le placement des composants pour les cartes électroniques, et le fonctionnement repose sur l'optimisation d'une valeur de qualité. De

manière assez similaire, l'auteur évoque l'optimisation de la conductance d'un graphe, une mesure liée à la taille des coupes, pour trouver des communautés. Trouver une coupe de conductance minimale est NP-difficile ; cependant, contrairement au partitionnement évoqué précédemment, la méthode utilisant la conductance ne présuppose rien quant à la taille des clusters.

Une autre technique qui rencontre du succès est la bissection spectrale [22], proposée par Barnes et qui exploite les propriétés de la matrice Laplacienne. L'auteur présente plus en détails l'utilisation de cette matrice et indique que cette approche propose de bonne performances, et peut être combinée avec la méthode précédemment présentée : appliquer *Kernighan-Lin* sur les partitions obtenues par cette analyse pour les améliorer.

L'application du théorème de flux maximum coupes minimales peut également servir à identifier des communautés dans un graphe : l'auteur mentionne les travaux de Flake, Lawrence et Giles, Flake et al. qui l'ont exploité pour identifier des communautés dans un graphe du Web [64, 65].

En conclusion, l'auteur présente les limites de cette approche pour la détection de communautés : nécessité de fournir le nombre voire la taille des groupes à former, méthode de découpe non fiable.

4.2.2.1.2 Regroupement hiérarchique Imposer des contraintes fortes en matière de nombre ou de taille des groupes à détecter est souvent peu pratique voire non envisageable. Par contre, détecter une hiérarchie de cluster dans un graphe fait sens, et l'auteur indique que c'est souvent le cas dans l'analyse de réseaux sociaux, en biologie, en ingénierie, ou en marketing. Cette méthode repose sur définition et le calcul d'une mesure de similarité des sommets, qu'ils soient connectés ensemble ou non : cela permet de construire une matrice de similarité.

Deux familles d'algorithmes peuvent être identifiées :

- Ceux fonctionnant par agglomération de sommets dont la similarité est suffisamment grande ;
- Ceux fonctionnant par division en déconnectant les sommets de similarité trop faible ;

Le principal avantage de la méthode de construction hiérarchique est son agnosticisme quant à des paramètres externes ; mais elle ne permet pas de classifier les partitions pour savoir laquelle représente le mieux les communautés du graphe. Surtout, la prise en compte de gros graphes est indiquée comme problématique par l'auteur, suivant les techniques de calcul de similarité, la complexité varie entre $O(n^2)$ et $O(n^2 log(n))$, et cela peut empirer si la mesure de distance est plus complexe.

4.2.2.1.3 Regroupement par partitionnement Pour cette approche, on cherche à former un nombre limité de groupes, k, et les sommets sont projetés dans un espace métrique. La distance devient ainsi une mesure de similarité entre ces sommets ; et l'objectif est de construire les k groupes en optimisant une fonction coût basée sur ces distances. L'auteur propose une série de ces fonctions qui sont les plus utilisées :

- Minimisation du diamètre des groupes (*minimum k-clustering*) ;
- Minimisation de la distance moyenne au sein des groupes (*k-clustering sum*) ;
- Minimisation de la distance maximum au centroïde (*k-center*) ;

– Minimisation de la distance moyenne au centroïde (*k-median*) ;

Mais la plus populaire des méthodes reste *k-means clustering* [110], où la fonction coût est le total de la distance intra-cluster. Plusieurs heuristiques de la littérature peuvent être utilisées pour choisir les centroïdes, notamment celle de Lloyd, qui a l'avantage de converger très rapidement. Une extension pour prendre en compte l'appartenance d'un sommet à potentiellement plusieurs groupes existe, *fuzzy k-means clustering*.

L'auteur relève deux limitations à cette méthode : d'abord, la nécessité de spécifier le nombre de groupes ; ensuite, le risque potentiel de la projection du graphe dans un espace métrique, qui peut être difficile dans certains cas.

4.2.2.1.4 Regroupement spectral L'analyse spectrale regroupe toutes les méthodes et techniques qui exploitent les vecteurs propres directement ou indirectement dérivés de la matrice S construite grâce à la fonction de similarité : il s'agit d'une transformation d'un ensemble d'objets (points dans un espace ou sommets d'un graphe) en un ensemble de points dans un espace dont les coordonnées sont des éléments des vecteurs propres. Ensuite, des techniques classiques comme *k-means* peuvent être appliquées. L'auteur indique que l'utilisation des vecteurs propres permet de mettre en évidences les propriétés intrinsèques des données d'origine. De plus, il relève que les principales contributions de la littérature utilisent la matrice Laplacienne. Trois méthodes sont particulièrement populaires dans la littérature : *unnormalized spectral clustering* et deux méthodes *normalized spectral clustering*. La normalisation ou non intervient sur la matrice Laplacienne.

Par ailleurs, étant donné la proximité entre l'analyse spectrale et le partitionnement des graphes, il est possible de transformer des problèmes de minimisation des coupes en des problèmes de regroupement spectral. L'auteur note également que le problème des marches aléatoires sur un graphe se rapproche de l'analyse spectrale : en minimisant le nombre d'arcs entre les groupes, on force les marcheurs à passer plus de temps au sein de ces groupes et à se déplacer plus rarement entre les groupes.

Quelques résultats de complexité sont aussi évoqués : le calcul des vecteurs propres est $O(n^3)$, ce qui peut être important sur des instances composées de grand graphes, mais des heuristiques sont proposées.

Enfin, le choix de la matrice Laplacienne est discuté : si les degrés des sommets sont proches, alors choisir une matrice normalisée ou non n'a pas grande importance ; dans le cas contraire, l'auteur précise que chacune a des propriétés particulières. Notamment, dans le cas général, la matrice Laplacienne normalisée est plus adaptée : les analyses vont effectuer une double optimisation où la densité intra-cluster est forte et où celle inter-cluster est faible ; alors que la matrice non normalisée est plus liée à la seule densité inter-cluster.

4.2.2.2 Approches par décomposition

Comme indiqué précédemment, les approches par décomposition partent d'un graphe puis en retirent des arcs pour identifier ceux qui relient les communautés. L'auteur emphase la différence principale avec la décomposition hiérarchique : ici, on retire les arcs entre les clusters plutôt que les arcs entre les paires de sommets de faible similarité, et il n'y a aucune garantie a priori que des arcs inter-clusters relient des sommets de faible similarité. Deux

cas sont étudiés, d'abord l'algorithme de Newman et Girvan, puis les autres méthodes.

4.2.2.2.1 Algorithme de Newman et Girvan Cette approche a déjà été rapidement présentée dans le paragraphe 4.2.1.1.1.3. Le but est de sélectionner les arcs suivant une mesure dite « centralité de l'arc » (*edge centrality*) qui donne une indication de l'importance de l'arc par rapport au graphe.

L'auteur donne les grandes étapes principales de l'algorithme :
- Calcul des centralités des arcs ;
- Retrait des arcs avec la plus forte centralité. Les cas ambigus sont résolus au hasard ;
- Recalcul des centralités sur le graphe en cours ;
- Retour à la première étape.

Pour identifier l'importance des arcs, Newman et Girvan introduisent le concept d'intermédiarité (*betweeness*). Trois définitions sont comparées : intermédiarité géodésique (*geodesic edge betweeness*), intermédiarité du marcheur aléatoire (*random-walk edge betweeness*) et intermédiarité de flux (*current-flow edge betweeness*), qui sont présentés par la suite. L'auteur indique que dans la pratique, la meilleure mesure semble être l'intermédiarité géodésique, et cite des études numériques qui montrent l'importance de l'étape 3 de l'algorithme, justifiant une évolution de la complexité : $O(m^2n)$ dans le cas général, et $O(n^3)$ pour des graphes creux. Il indique également que la structure des communautés va impacter le temps de calcul, mais estime que l'algorithme est plutôt lent, et n'est applicable qu'à des graphes creux de l'ordre de 10^3 sommets, à la vue des puissances de calcul actuellement disponibles (l'auteur ne donne pas de référence cependant).

Des références à plusieurs versions « accélérées » de l'algorithme sont proposée : une première a été adaptée pour traiter des gènes et calculer leur co-occurrences, en calculant l'intermédiarité par une méthode inspirée de Monte-Carlo sur un nombre limité de centres. Les résultats montrent à la fois un gain de temps de calcul et de bons résultats pour la co-occurrence des gènes, sans citer de valeurs. Une seconde a été appliquée sur des correspondances par courriel. Une troisième méthode est évoquée [142], qui ramène la complexité à $O(m)$, et a été évaluée avec de bons résultats sur le graphe test des auteurs d'articles scientifiques.

4.2.2.2.1.1 Intermédiarité géodésique Cette mesure correspond au nombre de plus courts chemins entre toutes les paires de sommets qui passent par l'arc, et permet d'exprimer l'importance d'un arc dans des processus tels que la transmission d'informations. La complexité pour le calcul de l'intermédiarité de tous les arcs dans ce cas est donné pour $O(mn)$, ou $O(n^2)$ pour un graphe creux.

4.2.2.2.1.2 Intermédiarité du marcheur aléatoire Avec cette définition, c'est le nombre de passages du marcheur aléatoire sur l'arc qui donne la valeur d'intermédiarité : ce marcheur se déplace de sommets en sommets qui sont adjacents, et de manière équiprobable. Finalement le calcul nécessite l'inversion d'une matrice $n \times n$, ce qui induit une complexité de $O(n^3)$, puis moyenner le flux sur toutes les paires de nœuds, ce qui nécessite une complexité de $O(mn^2)$. La complexité finale est donc de $O(n^3)$ pour un graphe creux, et $O((m+n)n^2)$ dans le cas général.

4.2.2.2.1.3 Intermédiarité de flux Ici, le graphe est considéré comme un réseau de résistances, dont les arcs ont des résistances unitaires. Lorsque l'on applique une tension aux bornes de deux sommets, alors un courant est porté par certains arcs. Cela est calculé en résolvant les équations de KIRCHOFF : cette procédure est répétée pour chaque paire de sommet, et finalement l'intermédiarité d'un arc est la moyenne des courants que cet arc fait passer. L'auteur indique qu'il est possible de montrer que cette mesure est équivalente à la précédente, et par conséquent, la complexité est la même.

Quelques limitations de la méthode sont également présentée par l'auteur. D'abord, l'algorithme peut produire des partitions déséquilibrées avec des communautés de taille très différentes. De plus, il ne peut détecter les communautés qui se recoupent : des travaux pour permettre de prendre en compte ce cas sont cités, et la complexité [118] est évaluée à $O(m^3)$ dans le pire cas, et $O(n^3)$ dans le cas d'un graphe creux.

4.2.2.2.2 Autres méthodes L'étude des cycles est une autre approche possible pour détecter les arcs inter-clusters : les communautés sont censées être densément peuplées, donc il parait logique que beaucoup de cycles s'y trouvent, et à l'inverse, les arcs situés entre les communautés ont peu de chance d'appartenir à un cycle. C'est sur cette idée que RADICCHI et al. propose [139] une mesure *edge clustering coefficient* définie de telle sorte que des faibles valeurs indiquent qu'un arc relie des communautés ; c'est à l'origine une généralisation de la notion de coefficient de clustering pour les sommets définis [186] par WATTS et STROGATZ suivant :

$$c_S = \frac{|Triangle(S)|}{|Triangles|}$$

Avec c_S le coefficient de clustering d'un sommet S, $|Triangles|$ le nombre de triangles possibles, et $|Triangle(S)|$ le nombre de triangles qui contiennent le sommet S. La nouvelle mesure est donc définie par :

$$\tilde{C}_{i,j}^{(g)} = \frac{z_{i,j}^{(g)} + 1}{s_{i,j}^{(g)}}$$

Avec i, j sont les extrémités de l'arc, $z_{i,j}^{(g)}$ le nombre de cycles de taille g passant par l'arc i, j et $s_{i,j}^{(g)}$ le nombre de cycles possibles de taille g passant par i, j. L'auteur indique que le fonctionnement de cette méthode est similaire à l'algorithme de NEWMAN et GIRVAN, et donne des résultats de complexité : $O(\frac{m^4}{n^2})$ dans le cas général, et $O(n^2)$ sur un graphe creux, si g est petit (sans plus de précisions). L'auteur précise par ailleurs que l'étape de recalcul devient couteuse en temps si g n'est pas petit, et si g approche deux fois le diamètre du graphe, les cycles deviennent globaux et surtout la complexité devient plus importante que celle de l'algorithme de NEWMAN et GIRVAN. Par ailleurs cette méthode donnera de mauvais résultats dans le cas de graphes présentant peu de cycles, comme les graphes de réseaux sociaux. Une implémentation est proposée sous licence libre GNU GPL [1].

1. http://filrad.homelinux.org/Data/

Une autre mesure existe, qui estime la facilité pour transmettre une information dans le graphe : la centralité de l'information, et mesure plus précisément l'efficacité. Dans ce contexte, elle est définie comme la moyenne des inverses des distances entre toutes les paires de sommets : si des sommets sont proches, la mesure sera « forte ». L'algorithme est également proche de celui de Newman et Girvan, et l'auteur indique que la complexité finale varie entre $O(m^3n)$ dans le cas général et $O(n^4)$ pour un graphe creux, ce qui est bien plus lent.

Une dernière approche basée sur les boucles est présentée : les voisins d'un sommet au sein d'une communauté sont « proches » même en l'absence de sommet, à cause de la densité interne importante d'arcs. Vragović et al. définissent [182] ainsi le coefficient de boucle (*loop coefficient*) comme la moyenne de $\frac{1}{d_{jk/i}}$ (avec $d_{jk/i}$ la longueur du plus court chemin entre j et k si i est retiré du graphe) pour toutes les paires de voisins de i, et les sommets qui jouent un rôle crucial au sein d'une communauté seront visibles par leur grande valeur. La complexité temporelle est donnée pour $O(mn)$, mais l'auteur indique que les résultats sont assez peu fiable par rapport aux autres méthodes.

4.2.2.3 Méthodes basées sur la modularité

La modularité est la fonction de qualité la plus utilisée. L'auteur indique que les méthodes qui seront présentées dans cette section seront orientées vers les techniques rapides qui peuvent être appliquées sur des grands graphes, sans toutefois donner d'information quantitative, mais qui ne trouvent pas d'optimum pour la mesure ; des techniques plus précises mais plus consommatrices de ressources ; d'autres proposant des compromis entre la complexité et la qualité des solutions.

4.2.2.3.1 Optimisation de la modularité Appliquer des techniques d'optimisation pour trouver les meilleures valeurs de la modularité est une des méthodes les plus populaires, et exploite le fait que la mesure de modularité reflète la qualité d'une partition : plus elle est élevée, plus la partition est « bonne ». Ce problème a été prouvé [31] comme NP-complet, il est donc nécessaire de faire appel à des heuristiques pour trouver de bonnes valeurs, même si elles ne sont pas optimales.

4.2.2.3.1.1 Algorithmes gloutons Une partie des heuristiques évoquées concerne les algorithmes gloutons, dont un premier algorithme apprécié est proposé par Newman dans [117]. Il permet d'identifier des communautés avec une complexité de l'ordre de $O((m+n)n)$ dans le cas général et $O(n^2)$ dans le cas d'un graphe creux, et l'auteur indique que cela permet de traiter des graphes contenant jusqu'à 10^5 sommets *avec les capacités de calcul actuelles* (sic), sans donner de point de comparaison. Clauset, Newman et Moore proposent [52] une amélioration de l'approche en supprimant certaines opérations inutiles et en utilisant des structures de données plus appropriées, et l'auteur indique que des graphes contenant 10^6 sommets peuvent être traités. Une version sous licence libre GNU GPL est accessible [2]. Dans [183], Wakita et Tsurumi proposent encore une amélioration

2. http://www.cs.unm.edu/~aaron/research/fastmodularity.htm

permettant de passer à l'analyse de graphe de 10^7 sommets en améliorant la complexité et l'équilibre des dendogrammes générés. Une version gratuite à des fins académiques est proposée, mais sans accès direct au code source sous licence libre. Une autre modification des travaux de Clauset, Newman et Moore est proposée par Schuetz et Caflisch dans [159] qui permet d'améliorer les résultats et d'approcher de plus près le maximum de modularité ; une implémentation sous licence libre GNU GPL est proposée [3].

Une autre approche que les algorithmes gloutons a été introduite [27] par Blondel et al., qui vise les graphes pondérés. L'auteur indique une complexité temporelle assez intéressante, $O(m)$, mais souligne que la limite majeure se situe sur la complexité spatiale (sans donner de repère) : des graphes contenant 10^9 arcs peuvent être analysés *en temps raisonnable* (sic). La valeur maximale de modularité atteinte par cette méthode est meilleure que celle produite par les algorithmes gloutons, mais certains cas peuvent donner des partitions étranges. Le code source de cette implémentation est disponible gratuitement sans précision de licence [4].

L'auteur conclut que les techniques basées sur les algorithmes gloutons ne sont pas aussi précises que d'autres.

4.2.2.3.1.2 Recuit simulé Quelques publications exploitent le recuit simulé pour optimiser la modularité. Les meilleures performances sont obtenues lorsque l'on optimise une bipartition d'un cluster pris comme un graphe isolé. L'auteur indique que cette méthode permet d'atteindre de très près le maximum de la modularité ; cependant la complexité ne peut être calculé tant elle dépend des paramètres choisis. Il indique par ailleurs que cette méthode ne peut pas être exploitée correctement sur des graphes qui dépassent 10^4 sommets.

4.2.2.3.1.3 Extremal optimization Cette méthode d'optimisation vise la même performance qualitative que le recuit simulé en étant plus efficace du point de vue du temps de calcul : elle procède en une optimisation de variables locales qui contribuent chacune à la fonction globale qui est mesurée ; dans le cas de notre graphe, la modularité peut être écrite comme une somme sur tous les sommets, et la modularité locale d'un sommet est la valeur du terme correspondant dans cette somme. L'auteur note la proximité avec l'algorithme de Kernighan-Lin, mais la différence majeure mise en lumière concerne la taille des communautés : ici, elle est déterminée par l'exécution de l'algorithme, elle n'est pas fixée. Cette méthode donne de bonnes estimations de la modularité maximale, avec une complexité de l'ordre de $O(n^2 log(n))$, ce qui représente un bon compromis. L'auteur met en garde cependant pour les grands réseaux avec de nombreuses communautés, où les résultats peuvent être mauvais en matière de partitionnement.

4.2.2.3.1.4 Optimisation spectrale L'utilisation des vecteurs propres d'une matrice spéciale peut servir de base à l'optimisation de la modularité. L'auteur propose quelques résultats avant de rappeler une limite dure qui se retrouve aussi dans la technique de bisection spectrale : cette approche ne fonctionne bien qu'avec des bisections.

3. http://www.biochem-caflisch.uzh.ch/public/5/network-clusterization-algorithm.html
4. https://sites.google.com/site/findcommunities/

Cela signifie qu'au-delà de deux communautés, les résultats sont moins précis. L'auteur mentionne des améliorations [184] dont le temps de calcul est indiqué comme faible, et dont des résultats de complexité sont proposés : $O(n^2 log(n))$, ce qui le rends similaire à l'*extremal optimization* mais avec des résultats plus précis, particulièrement dans le cas de grands graphes (sans indiquer de données chiffrées).

Une autre série de méthodes d'optimisation spectrale de la modularité sont également évoquées [5], permettant d'identifier un nombre fixé d'au plus K clusters, avec des résultats de complexité : $O(K^2 n + Km)$, dans le pire cas ; l'auteur stipule que cela devrait être essentiellement linéaire suivant le nombre de sommets si le graphe est creux et $K << n$. Mais sur de plus grands graphes creux, l'approche peut devenir plutôt lente. Une modification est proposée [156] par Ruan et Zhang : *Kcut* qui permet d'accélérer les calculs sans perdre en précision quant à la valeur de modularité. La complexité est donnée pour $O((n + m)log(K))$, et les résultats numériques montrent une précision comparable à la version d'origine bien que plus faible.

4.2.2.3.1.5 Autres stratégies d'optimisation Agarwal et Kempe utilisent [1] de la programmation linéaire en nombres entiers pour optimiser la valeur de la modularité, et l'auteur propose une formulation du problème. Celui-ci est bien sûr NP-difficile et une relaxation en nombres réels est applicable, après quoi il convient de procéder à une étape d'arrondi, puis éventuellement une optimisation des valeurs avec un algorithme similaire à celui de Kernighan-Lin. Une version quadratique peut aussi servir à obtenir des bisections de graphes avec des hautes modularités, avant d'appliquer des optimisations spectrales. Ces deux méthodes sont cependant limitées par leur complexité et ne sont pas utilisables avec des graphes dépassant 10^4 sommets. Le code source de l'implémentation est disponible [5], sans précision de licence.

L'utilisation de *mean field annealing* [135] est étudiée [94] par Lehmann et Hansen, et l'auteur présente une complexité temporelle de l'ordre de $O((m + n)n)$, et indique de meilleures valeurs de modularité que l'algorithme de Newman mais sur des graphes artificiels utilisés pour évaluer les performances.

Les algorithmes génétiques ont également été utilisés pour optimiser la modularité, et certains résultats approchent ceux des méthodes gloutonnes, dont l'auteur notait le manque de performances. Par ailleurs aucune information n'est fournie quant à la taille des graphes qui sont utilisables avec les algorithmes génétiques.

4.2.2.3.2 Modifications de la modularité La modularité telle que définie jusqu'à présent ne prend pas en compte de poids sur les arcs. L'auteur présente les modifications nécessaires proposées [116] par Newman à inclure dans les différentes équations pour prendre en compte ce cas.

Elle ne prend pas non plus en compte l'orientation des arcs : ceux-ci sont non orientés. Dans [9] et [96] des modifications sont suggérées pour traiter les graphes orientés.

La modularité peut être difficile à généraliser si l'on considère le cas où un arc peut appartenir à plusieurs communautés. L'auteur fait référence à différents travaux [162, 121]

5. http://www-scf.usc.edu/~gaurava/

qui proposent des définitions alternatives, et rapporte les équations correspondantes.

Gaertler, Görke et Wagner proposent [67] une autre définition de la modularité : c'est la différence entre la couverture – le rapport entre le nombre d'arcs dans les clusters et le nombre total d'arcs – d'une partition et sa couverture attendue dans le *null model* ; ils comparent notamment différentes variables et proposent une classification en terme d'efficacité, ainsi qu'une nouvelle fonction performance qui surpasse la modularité standard.

Un nouveau *null model* est proposé [111] par Massen et Doye et est jugé plus proche de la réalité : il impose qu'il ne puisse y avoir de multiples arcs entre deux sommets ni de boucle. La maximisation de la modularité sur ce nouveau modèle produit des partitions avec une taille de cluster plus faible. Ils proposent ensuite une version locale de la modularité. Cette nouvelle définition permet d'obtenir des partitions plus précises sur des graphes de test : réseau social scolaire, réseaux d'interactions entre protéines d'*E. coli*.

Reichardt et Bornholdt montrent [145] que le problème de détection de communauté peut se formuler comme un problème de découverte de l'état fondamental d'un modèle de verre de spin. L'auteur propose un modèle de spin Hamiltonien qui correspond à la détection de communauté, et les partitions sont détectées en cherchant à minimiser l'énergie, et leur nombre peut être adapté en adaptant certains paramètres (γ, variant entre 0 pour produire un seul cluster contenant tous les sommets et ∞ pour un ensemble de clusters avec chacun un sommet).

Une autre généralisation de la modularité a été proposée [8] par Arenas et al. : au lieu de se limiter aux arcs, cette fois la modularité se base sur la présence de motifs. Notamment, au sein d'un cluster, on plusieurs triangles alors que peu seront présents en dehors des communautés. Les auteurs comparent donc la densité de ces motifs par rapport au *null model*. Aucune information de complexité ou de performance de l'approche n'est apportée.

Par la suite, l'auteur évoque les graphes qui sont construits à partir de systèmes réels, et plus particulièrement sur la base de données de corrélation. Celles-ci peuvent être négatives ou positives, ce qui va influer sur le poids des arcs et par conséquent le niveau d'attraction entre les sommets. Ce problème a été formulé par Bansal, Blum et Chawla dans [19]. Les sommets appartenant à une même communauté seraient ainsi liés ensemble par des arcs de poids positif, et liés avec des arcs de poids négatifs à ceux d'autres communautés. Et la meilleure structure est celle qui maximise la somme des valeurs absolues des poids des arcs situés entre les clusters. L'auteur indique que ceci peut s'écrire sous la forme d'une mesure de modularité et propose quelques résultats tirés d'autres articles, sans toutefois fournir d'informations quant à la qualité de cette approche.

Le modèle de modularité s'applique mal aux graphes bipartites qui se retrouvent dans différents systèmes : réseaux de nourriture, collaborations scientifiques, collaborations artistiques, etc., et des auteurs proposent [20, 21] une définition mieux adaptée. Cette modularité peut s'appliquer aux graphes orientés qui ne sont pas bipartites. L'auteur ne donne pas de retour quant à la qualité de cette méthode. Une autre alternative adaptée au cas bipartite est également proposée dans [20, 21] propose une nouvelle matrice *null* constituée de blocs :

$$P = \begin{bmatrix} O_{p\times p} & \tilde{P}_{p\times p} \\ \tilde{P}^T_{p\times p} & O_{p\times p} \end{bmatrix}$$

Avec O des matrices carrées nulles, et $\tilde{P}_{ij}$ le nombre d'arcs attendus entre i et j dans le *null model*. Cette matrice est accompagnée d'une matrice d'adjacence $\mathbf{A}$, et l'auteur indique que la maximum de la modularité est obtenu par le calcul de $B = A - P$. L'approche est accompagnée d'une technique d'optimisation, *Bipartite Recursively Induced Modules (BRIM)*. Si cet algorithme ne sait pas correctement calculer le nombre de clusters, en revanche il converge rapidement ($O(m)$) pour un nombre de clusters c donné. Le nombre d'étapes de convergence, par contre, reste à déterminer.

4.2.2.3.3 Limites de la modularité L'auteur propose dans cette section d'étudier les limites de la modularité, notamment pour être en mesure de correctement qualifier ses cas d'utilisation.

Une première limite étudiée est la valeur maximale de modularité : l'auteur rappelle qu'elle doit être supérieure à 0, le cas d'égalité correspondant à une seule communauté comprenant tous les sommets. Mais surtout, une valeur importante n'implique pas nécessairement qu'il y a une structure de communauté : [145, 74] ont montré des exemples de graphes aléatoires avec des partitions ayant une valeur de modularité importante. L'origine de ce comportement est à chercher dans la distribution des arcs dans le graphe : les fluctuations vont déterminer la concentration de liaisons qui font former des communautés apparentes sans en être réellement. Les statistiques peuvent venir à notre secours, en calculant un score basé sur le maximum de la modularité (Q_{max}), puis en calculant une moyenne ($< Q >_{NM}$) de celle-ci pour plusieurs alternatives du *null model* correspondant, ainsi que l'écart-type σ_Q^{NM} :

$$z = \frac{Q_{max} - < Q >_{NM}}{\sigma_Q^{NM}}$$

Si $z >> 1$, alors Q_{max} indique une structure de communauté forte. L'auteur note cependant que cette méthode n'est pas sans problème : des faux positifs et des faux négatifs existent. De plus la distribution du maximum ne suit pas une loi Gaussienne, et on ne peut donc utiliser z pour calculer l'importance par rapport à une distribution Gaussienne.

La détection de communautés dans un graphe qui est trop creux peut poser des difficultés avec la méthode exploitant la modularité, comme le rapporte l'auteur citant les travaux de Reichardt et Bornholdt qui étudient [146, 143] la modularité sur des graphes aléatoires.

L'auteur évoque enfin un autre problème fondamental de cette approche : sa capacité à détecter de *bonnes* partitions. Notamment, si un graphe a une structure de communautés clairement établie, alors la modularité devrait le refléter. Ainsi, si des sous graphes sont trop petits par rapport à la taille totale du graphe, alors, par construction – notamment le *null model* –, l'approche de la modularité aura du mal à les détecter : c'est le problème de la résolution. L'auteur précise que sur une partition de modularité maximale, des clusters d'un degré total de l'ordre de $\sqrt{m}$ ou moins ne peuvent être distingués : a priori, il n'est

pas possible de savoir s'ils forment une seule communauté ou plusieurs petites. Cette limitation est gênante en pratique, plusieurs auteurs ayant des cas de communautés réelles avec des différences de tailles importantes. L'origine est tracée dans la définition du modèle nul utilisé pour calculer la modularité : dans celui-ci, il est considéré que n'importe quel sommet peut interagir (avoir un ou plusieurs arcs) avec n'importe quel autre. D'autres approches sont également touchées, même si elles ne reposent pas sur le modèle nul évoqué, telle que la densité de modularité. Une solution proposée est d'introduire un paramètre qui permet de contrôler la résolution. Une approche récursive est également envisageable, mais l'auteur note qu'elle n'est pas fiable.

Un dernier résultat [69] de Good, Montjoye et Clauset est présenté : il s'agit d'une analyse du paysage des modularités. Celle-ci montre que le nombre de partitions est exponentiel et que leur modularité est très proche du maximum global : cela explique la facilité à trouver de bonnes modularité, mais cela confirme la difficulté à trouver la meilleure.

4.2.2.4 Approches spectrales

Après avoir introduit la technique d'analyse spectrale précédemment, l'auteur propose d'étudier différents algorithmes développés dans la littérature et qui visent plus particulièrement le clustering dans des graphes utilisés par les physiciens.

Des premiers travaux ont exploité les vecteurs propres de la matrice de transfert M_t, avec $a_{...}$ les éléments de la matrice d'adjacence et d_k le degré du sommet k, définie comme :

$$M_t = \begin{pmatrix} \frac{a_{1,1}}{d_1} & \frac{a_{2,1}}{d_1} & \cdots & \cdots & \frac{a_{n-1,1}}{d_1} & \frac{a_{n,1}}{d_1} \\ \frac{a_{1,2}}{d_2} & \frac{a_{2,2}}{d_2} & \cdots & \cdots & \frac{a_{n-1,2}}{d_2} & \frac{a_{n,2}}{d_2} \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ \frac{a_{1,i}}{d_i} & \frac{a_{2,i}}{d_i} & \cdots & \cdots & \frac{a_{n-1,i}}{d_i} & \frac{a_{n,i}}{d_i} \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ \frac{a_{1,n-1}}{d_{n-1}} & \frac{a_{2,n-1}}{d_{n-1}} & \cdots & \cdots & \frac{a_{n-1,n-1}}{d_{n-1}} & \frac{a_{n,n-1}}{d_{n-1}} \\ \frac{a_{1,n}}{d_n} & \frac{a_{2,n}}{d_n} & \cdots & \cdots & \frac{a_{n-1,n}}{d_n} & \frac{a_{n,n}}{d_n} \end{pmatrix}^T$$

Cette matrice sert de base temporelle pour le problème du marcheur aléatoire dans un graphe. La notion de ratio de participation est introduite pour estimer la participation de sommets à un vecteur propre et est donnée par X_α :

$$X_\alpha = \Big[\sum_{i=1}^{n} (c_i^\alpha)^4 \Big]^{-1}$$

Avec c^α un vecteur propre de la transposée de M_t, et c_i^α le flux sortant du sommet

i correspondant au mode propre α. Des résultats quant à cette valeur de participation sont évoqués, notamment qu'elle se corrèle avec la taille du cluster. De plus sa pertinence statistique peut être évaluée en comparant cette valeur avec celle obtenue sur un graphe aléatoire correspondant.

D'autres auteurs exploitent les vecteurs propres de la matrice Laplacienne et correspond à l'idée déjà présentée en section 4.2.2.1.4 : utiliser les valeurs des vecteurs propres comme coordonnées des sommets pour les placer dans un espace, et les communautés apparaissent comme des groupes de points assez compacts et bien séparés les uns des autres. Dans [57], Donetti et Muñoz utilisent du partitionnement hiérarchique pour extraire et regrouper les points, en imposant la contrainte de ne regrouper que les paires de cluster ayant au moins un arc les reliant dans le graphe d'origine, puis retiennent la partition avec la plus grande modularité. Ils utilisent la distance Euclidienne et la distance angulaire pour mesurer la similarité. Les meilleurs résultats sont obtenus sur des clusters avec une connectivité complète.

Alves propose [2] d'exploiter les propriétés d'un réseau électrique correspondant au graphe, avec des arcs de résistance unitaire, et utilise les valeurs et vecteurs propres de la matrice Laplacienne. Les conductances sont utilisées pour calculer les probabilités pour le marcheur aléatoire, puis une méthode hiérarchique est appliquée pour grouper les sommets. Cette approche peut être étendue au cas des graphes pondérés, mais l'algorithme est lent.

La matrice stochastique de droite, donnée par R ci-après, est utilisée [40] par Capocci et al. car ayant des propriétés similaires à la matrice Laplacienne.

$$R = \begin{pmatrix} \frac{a_{1,1}}{d_1} & \frac{a_{2,1}}{d_1} & \cdots & \cdots & \frac{a_{n-1,1}}{d_1} & \frac{a_{n,1}}{d_1} \\ \frac{a_{1,2}}{d_2} & \frac{a_{2,2}}{d_2} & \cdots & \cdots & \frac{a_{n-1,2}}{d_2} & \frac{a_{n,2}}{d_2} \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ \frac{a_{1,i}}{d_i} & \frac{a_{2,i}}{d_i} & \cdots & \cdots & \frac{a_{n-1,i}}{d_i} & \frac{a_{n,i}}{d_i} \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ \frac{a_{1,n-1}}{d_{n-1}} & \frac{a_{2,n-1}}{d_{n-1}} & \cdots & \cdots & \frac{a_{n-1,n-1}}{d_{n-1}} & \frac{a_{n,n-1}}{d_{n-1}} \\ \frac{a_{1,n}}{d_n} & \frac{a_{2,n}}{d_n} & \cdots & \cdots & \frac{a_{n-1,n}}{d_n} & \frac{a_{n,n}}{d_n} \end{pmatrix}$$

Notamment, si le graphe a g composantes connexes alors les g plus grandes valeurs propres sont égales à 1 et les vecteurs propres qui appartiennent aux mêmes communautés sont égaux en valeurs. L'auteur relève que cette méthode est applicable aux cas des graphes orientés et pondérés, et bien qu'elle fournisse des estimations des similarités des sommets elle ne permet pas de déduire des partitions du graphe qui soient correctement définies.

Une approche de bisection récursive est proposée [191] par Yang et Liu. Les communautés sont définies comme des sous-graphes dont le degré externe de chaque sommet n'excède pas le degré interne. Une mesure de centralité, *clustering centrality*, est calculée pour les sommets et sert à construire une matrice ; cette mesure donne la probabilité

pour un marcheur aléatoire qui démarre son parcours sur ce sommet d'atteindre une cible donnée. Celle-ci sera plus élevée si les deux sommets (départ et arrivée) font partie de la même communauté. Ainsi, si une structure de communauté est bien définie dans le graphe, alors les valeurs de centralité pour les sommets qui sont membres d'une même communauté seront proches. La matrice d'adjacence peut être ré-arrangée par ordre non croissant de centralité pour faire apparaitre des blocs. Ceux-ci seront identifiés par une bisection itérative : chaque cluster trouvé est découpé en deux tant que les communautés ainsi créées restent des communautés au sens fort. L'auteur évalue la complexité à $O(Kt(n \times log(n+m)))$, avec K le nombre de clusters dans la partition finale et t le nombre moyen d'itérations pour le calcul de la centralité – qui est indépendant de la taille du graphe –, et qui donne donc $O(n \times log(n))$ sur des graphes creux. La plus grosse limitation de cette méthode relevée par l'auteur est qu'elle repose sur des communautés définies au sens fort.

4.2.2.5 Approches dynamiques

L'auteur détaille ici des algorithmes dynamiques parmi trois familles : études des interactions spin-spin, marcheur aléatoire et synchronisation.

4.2.2.5.1 Interactions spin-spin Le modèle de Potts est un des plus populaire et décrit un système de spins pouvant être dans q états différents. Les interactions sont ferro-magnétiques (et favorisent l'alignement des spins) et à une température nulle tous les spins sont dans le même état. Il est donc possible de rechercher des communautés en transformant un graphe en un système de Potts (les variables de spin correspondant aux sommets, et les interactions se produisant entre les spins voisins) en exploitant le fait qu'il y a beaucoup plus d'interactions au sein des communautés qu'entre elles. Reichardt et Bornholdt exploitent [144] ce principe pour détecter les communautés du graphe correspondant à un modèle de q-Potts à une température nulle et avec des interactions entre les voisins proches. Ils utilisent l'énergie donnée par l'équation du modèle Hamiltonien et effectuent une optimisation de celle-ci par une méthode de recuit simulé en partant d'un état initial où les spins sont assignés au hasard aux sommets, et le nombre d'états q est très haut (sans plus de précision). L'auteur note que la procédure est très rapide et n'est pas dépendante de la valeur de q tant qu'il est assez élevé, et permet d'identifier les sommets partagés entre les communautés. De plus il est possible de traiter le cas des graphes pondérés en faisant en sorte que le couplage des spins soit dépendant du poids des arcs.

Le modèle *Ferromagnetic Random Field Isin Model (FRFIM)* est utilisé [166] par Son, Jeong et Noh et est appliqué pour un graphe pondéré. Le comportement du modèle dépend du choix du champ magnétique ; Son, Jeong et Noh le mettent à zéro pour tous les sommets sauf deux nommés s et t, qui est infini et de signe opposé pour chacun. Ainsi, si s et t sont des sommets centraux de communautés, alors ils vont imposer leur état de spin à leurs voisins de communauté. La résolution de l'équation pour trouver l'état d'énergie minimum est équivalente au problème de flot maximum/coupes minimales. Après quelques considérations sur le processus de détection des clusters, l'auteur donne la complexité de la méthode, $O(n^{2+\theta})$ avec $\theta \approx 1.2$, ce qui limite donc la mise en pratique à des graphes

composés de quelques milliers de sommets. L'auteur note que la complexité peut être réduite à $O(n^{\theta})$ si l'on sait d'avance choisir correctement s et t en sachant quels sont les sommets importants d'une communauté.

4.2.2.5.2 Marcheur aléatoire Zhou utilise [200] cette approche pour calculer les distances entre des paires de sommets (i, j) : d_{ij} est définie comme le nombre moyen d'arcs que le marcheur aléatoire va rencontrer pour atteindre j en partant de i ; il introduit deux types *d'attracteurs*, l'un local l'autre global. Un attracteur global d'un sommet i sera le plus proche sommet de i en matière de distance (n'importe quel sommet), un attracteur local de i sera son plus proche voisin. Et deux types de communautés sont définies par rapport à ces attracteurs : un sommet i sera dans la même communauté que ses attracteurs et tous les autres sommets pour lesquels il est un attracteur. Les communautés doivent être des sous-graphes minimums. L'auteur indique que l'application à des cas classiques de la littérature donne de bonnes partitions. L'association entre un sommet i et un attracteur j peut être pondérée par une probabilité proportionnelle à $e^{-\beta d_{ij}}$, β étant une sorte d'inverse de la température. Par la suite, dans [201] le même Zhou utilise une mesure de dissimilarité basée sur la distance définie précédemment. Un processus par découpage permet d'obtenir différentes partitions et les communautés détectées sont valides sur les exemples testés (club de Karaté, etc.). Dans les deux cas, la complexité est de l'ordre de $O(n^3)$, à cause du calcul de la matrice de distances qui nécessite de résoudre autant d'équations linéaires qu'il n'y a de sommets.

Des marcheurs aléatoires biaisés sont utilisés [202] par Zhou et Lipowsky : le biais provient de la sélection des cibles, puisque le marcheur aléatoire va préférer les sommets qui partagent un grand nombre de voisins avec le sommet de départ. Une mesure est définie pour donner la proximité d'une paire de sommets par rapport aux autres. Une procédure, *NetWalk*, permet de détecter les communautés en utilisant une approche hiérarchique cumulative. Les résultats montrent que la méthode donne de bonnes performances, comparables à l'approche Newman-Girvan ; la complexité temporelle est donnée pour $O(n^3)$ mais un calcul de l'index de proximité entre deux sommets en utilisant un voisinage proche du graphe est une bonne approximation qui permet un gain de complexité conséquent, bien que non documenté par l'auteur.

Pons et Latapy proposent [137] une nouvelle mesure de distance entre les sommets qui se base sur des marches aléatoires : elle est calculée à partir des probabilités qu'un marcheur aléatoire se déplace d'un sommet à un autre dans un nombre d'étapes fixé ; ce nombre soit être assez grand pour explorer suffisamment le graphe mais pas trop. Trouver une valeur optimale pour ce paramètre est un problème non trivial. Une approche hiérarchique additive est appliquée ensuite pour regrouper les sommets et former les communautés, et la meilleure partition est sélectionnée par la mesure de modularité. La complexité est donnée pour $O(n^2 d)$ sur un graphe creux, d étant la profondeur du dendogramme pour la sélection des partitions ; en pratique, d est souvent faible et la complexité peut être ramenée à $O(n^2 log(n))$. Le code source est proposé [6] sous licence libre GNU GPL.

Hu et al. présentent [79] une méthode basée sur un processus de signalisation entre les sommets, qui rappelle la diffusion : initialement, un sommet s a une valeur de signal

6. http://www-rp.lip6.fr/~latapy/PP/walktrap.html

unitaire et tous les autres sont à 0. À la première étape, la valeur de signal est transférée à ses voisins. Puis les sommets transmettent à leur tour leur valeur de signal à leurs voisins. Ce processus est répété pour T itérations. On construit un vecteur u_s représentant le sommet source s, et dont la i-ième composante est donnée par la valeur du signe au sommet i normalisé par la quantité totale de signal. Ces étapes sont répétées en prenant pour source chaque sommet, et le vecteur u_s représente la capacité d'influence d'un sommet sur le reste du graphe ; des sommets d'une même communauté vont avoir une influence proche. Le regroupement final est opéré par une méthode de k-means floue. La complexité est donnée pour $O(T(< k > +1)n^2)$, avec $< k >$ le degré moyen du graphe. Trouver une valeur optimale pour le nombre d'itérations est non trivial.

Delvenne, Yaliraki et Barahona introduisent [56] la notion de persistance d'un marcheur aléatoire dans le temps, comme une fonction de qualité : un cluster est persistant par rapport à une marche aléatoire avec t étapes si la probabilité que le marcheur s'échappe du cluster avant t étapes est faible. Ces probabilités sont calculées à partir d'une matrice d'autocovariance. Ils définissent également la stabilité du clustering $r(t; H)$, liée à la précédente matrice, et l'objectif est de trouver la partition qui maximise cette stabilité. L'auteur présente différents résultats intermédiaires liés à $r()$ sur des cas limites ($t = 0$, $t = 1$, $t \to \infty$), et souligne que cette définition est assez générale pour regrouper plusieurs définitions précédentes et permet donc d'unifier le milieu.

L'utilisation de chaînes de Markov est également une pratique documentée de la littérature : E, Li et Vanden-Eijnden proposent [62] une méthode dans laquelle la meilleure partition parmi k clusters est telle que la chaîne de Markov décrivant une marche aléatoire sur le méta-graphe – i.e., le graphe dont les sommets sont les cluters du graphe d'origine – donne la meilleure approximation de la marche aléatoire sur le graphe entier. La qualité de l'approximation est mesurée par la distance entre les parties gauches des matrices correspondant aux chaînes de Markov ; on cherche à minimiser cette distance avec un algorithme de k-means. Plusieurs (l) exécutions sont effectuées avec des conditions initiales différentes, et comparées pour retenir la meilleure. Finalement la complexité est de $O(tlk(n+m))$ avec t le nombre d'étapes pour converger. L'auteur relève que l'application de cette méthode sur des exemples classiques de la littérature tels le club de karaté donne de bons résultats.

Le dernier algorithme présenté par l'auteur repose également sur les chaînes de Markov : *Markov Cluster Algorithm (MCL)* et a été proposé [58] par Dongen. Cette approche simule une diffusion de flot dans le graphe. Après quelques étapes décrites par l'auteur, une matrice présentant de bonnes propriétés est obtenue : c'est une sorte de matrice d'adjacence, ses éléments étant soit à 0 soit à 1. Mais de plus le graphe correspondant à cette matrice est déconnecté et ses composantes connexes correspondent aux communautés dans le graphe d'origine. De plus la méthode est simple à implémenter, ce qui explique en partie son succès (dans le domaine de la bioinformatique). La complexité dans le cas général est donnée pour $O(n^3)$ et baisse à $O(nk^2)$ pour les graphes creux. L'auteur note cependant qu'un paramètre utilisé pendant les premières étapes du processus impacte fortement la partition finalement obtenue ; plusieurs valeurs d'α vont donc engendrer des partitions différentes, et il n'y a pas de bon critère pour les séparer. Enfin, le code source de l'algorithme est disponible [7] sous licence libre GNU GPL.

7. http://micans.org/mcl/

4.2.2.5.3 Synchronisation L'utilisation de synchronisation pour détecter des communautés repose sur des propriétés temporelles : dans un état synchronisé, tous les éléments du système sont dans un état identique ou proche à chaque instant. L'idée est donc d'attacher à chaque sommet du graphe un oscillateur (modèle de Kuramoto) et d'étudier le temps de stabilisation : celui-ci sera assez court au sein des communautés, alors qu'il sera plus important à l'échelle du graphe complet puisque l'on attendra la synchronisation de tous les sommets. Ainsi, ceux d'une même communauté vont avoir un temps de synchronisation proche et l'étude de ce paramètre permet d'identifier les communautés. La première utilisation [7] de cette méthode est l'œuvre de Arenas, Díaz-Guilera et Pérez-Vicente. De long plateaux montrent une forte structure de communauté corrélée par une valeur de modularité élevée sur des graphes sont la distribution du degré est homogène ; ces résultats ne sont plus valides lorsque cette distribution est fortement hétérogène. L'apparition de ces plateaux à des instants différents donne des indices sur la structure hiérarchique du graphe. Les auteurs Arenas, Díaz-Guilera et Pérez-Vicente ont également montré que les différentes échelles révélées par le mécanisme de synchronisation correspondent aux groupes de valeurs propres de la matrice Laplacienne associée au graphe.

Boccaletti et al. proposent [28] une autre méthode également basée sur la synchronisation, et utilisant une variante des oscillateurs précédents, le modèle *Opinon Changing Rate (OPR)*. L'interaction entre deux sommets adjacents est pondérée par un terme proportionnel à une puissance (négative) de la proximité de l'arc entre les sommets, avec un exposant α. Les équations régissant l'évolution du modèle sont résolues en faisant décroître la valeur α, en partant d'une situation initiale où le système est complètement synchronisé ($\alpha = 0$). Le graphe va se scinder en clusters d'éléments synchronisés, et différentes partitions seront produites pour différentes valeurs de α. Une fois de plus, c'est celle avec la modularité la plus élevée qui sera retenue comme la plus pertinente. Les évaluations de cette méthode sur les benchmarks de la littérature montrent de bons résultats pratiques, et l'auteur indique une astuce permettant d'obtenir des partitions avec une meilleure modularité. Enfin la complexité est donnée pour $O(mn)$ dans le cas général et $O(n^2)$ sur les graphes creux.

Li et al. proposent [98] une méthode basée sur la synchronisation et permettant de découvrir des communautés qui se recouvrent.

L'auteur conclut enfin que dans le cas de communautés de taille disparates, les approches par synchronisation n'ont pas été montrées comme fiables, cela reste à étudier.

4.2.2.6 Méthodes statistiques

Dans cette section l'auteur présente une utilisation des statistiques pour effectuer de la détection de communauté, en partant d'un ensemble d'observations et d'hypothèses sur un modèle. Si cet ensemble est un graphe alors le modèle – qui suit les hypothèses de connexion entre les sommets – doit correspondre à la topologie du graphe étudié. Les méthodes étudiées sont principalement basées sur l'inférence Bayésienne, mais le blockmodeling, la sélection de modèle et la théorie de l'information seront également mis à profit.

4.2.2.6.1 Modèles génératifs L'inférence Bayésienne consiste à estimer la probabilité qu'une hypothèse soit vraie, et utilise deux éléments pour ça : d'abord, la preuve D obtenue par des observations du système, et ensuite le modèle statistique paramétré par θ. Le but est de déterminer les choix de ce paramètre qui maximise la probabilité $P(\theta|D)$. Le théorème de Bayes permet de calculer celle-ci mais il repose sur une intégrale complexe à calculer ; de plus le choix de la distribution de probabilité $P(\theta)$ n'est pas évident. Les modèles génératifs diffèrent les uns des autres par le choix du modèle et par leur manière de traiter ces deux problèmes.

L'application d'une approche Bayésienne pour la détection de communautés dans un graphe nécessite de considérer la structure du graphe (matrice d'adjacence ou de poids) comme la preuve. La classification des sommets en groupe est considéré comme une information cachée, et c'est ce que l'on souhaite découvrir. C'est la base de la majorité des publications qui sont analysées, et où l'on cherche essentiellement à maximiser la probabilité $P(D|\theta)$ que le modèle soit cohérent avec la structure observée.

Une première utilisation est proposée [76] par Hastings où le modèle choisi est *planted partition model* : n sommets sont assignés à q groupes, ceux du même groupe sont liés avec une probabilité p_{in} alors que ceux d'autres groupes sont liés avec une probabilité p_{out} ; si $p_{in} > p_{out}$ alors le graphe modèle a une structure de communauté. Un ensemble d'étiquettes q_i permet d'indiquer la classification des sommets, et il est possible de calculer la probabilité $p(q_i)$, pour un graphe donné, qu'une classification est bonne suivant le modèle. L'auteur note que la maximisation de cette probabilité revient en fait à minimiser l'équation Hamiltonienne d'un modèle de Potts avec des interactions courtes et longues. Finalement sur un graphe creux, la complexité est de l'ordre de $O(nlog^{\alpha}(n))$ avec α un paramètre à évaluer numériquement. De plus, si p_{in} et p_{out} sont également des paramètres qui sont en général inconnus, ils peuvent être pris arbitrairement et les mauvais choix peuvent être corrigés.

Une approche similaire est proposée [120] par Newman et Leicht avec un modèle *mixture* et une technique *expectation-maximization*. Initialement, le graphe orienté est composé de n sommets, appartenant à c classes. L'auteur introduit une mesure de *log*-vraisemblance de laquelle il retire deux équations qui peuvent être résolutions en itérant jusqu'à convergence, en prenant un ensemble de conditions initiales bien choisi. Cette approche est rapide et peut traiter des graphes composés de 10^6 sommet. Si une application aux graphes non orientés est assez simple, le passage au cas de graphes pondérés ne l'est pas tant. L'auteur notre un avantage de cette méthode : il n'est pas nécessaire d'avoir d'indication préalable sur la structure, et celle permet de détecter beaucoup d'autres types de structures que simplement des communautés. La principale limitation est de devoir donner le nombre c de groupes à l'initialisation. Des travaux [114] ont montré que cette approche permet également de classifier les sommets suivant leur degré d'influence sur les autres, et ainsi d'identifier ceux qui sont importants pour la structure du groupe et sa stabilité.

Cette approche a également été appliquée [178] par Vazquez pour un problème de stratification de population où les populations animales et leurs attributs sont représentés par des hypergraphes. Il propose notamment un critère permettant de décider le nombre optimal de clusters, en choisissant le nombre $\overline{c}$ dont la solution a la plus grande similarité

avec les solutions obtenues avec différentes valeurs de c. Par la suite, il montre [177] que l'utilisation de la technique *Variational Bayes* pour maximiser la vraisemblance de la classification permet d'améliorer les résultats.

Les approches sont mises en échec sur des instances avec des graphes orientés bipartites. Ramasco et Mungan proposent [141] une modification qui permet de passer outre cette limitation. Une approche similaire a été [148] étudiée par Ren et al. Ces approches ont donné de bons résultats.

Copic, Jackson et Kirman travaillent sur une modification [54] de cette approche en introduisant une matrice de taille S dont les éléments s_{ij} indiquent la force maximale de l'interaction entre les sommets i et j ; par exemple un graphe non pondéré aura toutes ses valeurs égales à 1, et la probabilité que le graphe dissimule une structure de communauté correspond à l'expression donnée par Hastings [76]. Les auteurs utilisent cette expression comme fonction de qualité afin de calculer un ordre entre les différentes partitions. Ils montrent que cet ordre satisfait différentes propriétés (que n'importe quel ordre digne de ce nom se doit de satisfaire) et proposent un algorithme permettant de trouver le maximum de vraisemblance d'une partition. L'auteur conclu sur l'importance de ces travaux pour améliorer la qualité de la définition et une formulation plus rigoureuse du problème de détection de communauté.

Une autre alternative est proposée [197] par Zanghi, Ambroise et Miele : ils utilisent le modèle *planted partition* pour représenter le graphe contenant des communautés, la méthode de classification par vraisemblance avec un algorithme d'*exceptation-maximization*. Leur algorithme est exécuté pour un nombre fixé de clusters, q, mais contrairement aux méthodes précédentes, ici, il est possible d'exécuter pour plusieurs q puis de sélectionner la solution qui maximise l'*Integrated Classification Likelihood*. La complexité est donnée pour $O(n^2)$.

Sur la même trajectoire, Hofman et Wiggins proposent une approche Bayésienne générale [77] : ils exploitent encore une fois le modèle *planted partition*, mais en modifiant les probabilités utilisées, puis cherchent à maximiser la probabilité conditionnelle $P(K|A)$, K étant le nombre de clusters et A une matrice d'adjacence. Cette optimisation est réalisée avec la technique *Variational Bayes* qui permet d'approximer cette probabilité. L'auteur note une complexité temporelle de $O(n^\alpha)$ avec α ayant été évalué numériquement sur des graphes synthétiques à $\alpha = 1.44$, mais par ailleurs la réelle limite se situe sur la complexité spatiale (non donnée). Les résultats sont considérés comme bons et surtout la méthode ne nécessite pas un nombre de clusters en entrée, mais le détermine en sortie. Le code source est mis à disposition [8] sous licence GNU GPL.

4.2.2.6.2 Blockmodeling Cette méthode classique des statistiques et des réseaux sociaux permet de décomposer un graphe en classe d'équivalence de sommets qui partagent des propriétés communes. Deux définitions sont habituellement retenues pour les classes d'équivalences : l'équivalence structurelle, où les sommets sont équivalents s'ils partagent les mêmes voisins ; l'équivalence régulière où les sommets d'une classe ont des schémas de connexion vers d'autres sommets similaires. L'équivalence structurelle implique l'équivalence régulière, mais l'inverse n'est pas vrai. L'auteur rappelle qu'une présentation exhaus-

8. https://github.com/jhofman/vbmod_python

tive de la méthode est hors du champ du document et renvoie le lecteur vers une référence sur le sujet. Par la suite, il se concentre sur des travaux de la littérature qui exploitent cette méthode.

Reichardt et White mettent à profit [147] le blockmodeling à partir d'un graphe orienté de n sommets et m arcs. Un ensemble d'étiquettes σ permet de classifier le graphe, avec $\sigma_i = 1, 2, \ldots, q$ la classe du sommet i. Le blockmodel correspondant est donné par la matrice d'adjacence $q \times q$ B définie par $B_{q_1,q_2} = 1$ si des arcs entre les classes q_1 et q_2 sont autorisés, 0 sinon. L'objectif devient d'arriver à trouver la classification σ et la matrice B qui approxime le mieux la matrice d'adjacence du graphe A. Les auteurs proposent également une fonction de qualité qui permet d'estimer la qualité de l'approximation. Cette fonction $\mathcal{Q}^*(\sigma)$ est optimisée (maximisée) par recuit simulé ; le maximum absolu Q_{max} est atteint quand q correspond au nombre q^* de classes d'équivalence structurelle du graphe. En pratique, le nombre optimal de classes est déterminé en comparant le rapport $\frac{Q^*(q)}{Q_{max}}$ avec celui obtenu pour le modèle *null*. Des communautés recouvrantes peuvent être détectées, et cette méthode peut être facilement étendue au cas des graphes pondérés.

4.2.2.6.3 Sélection de modèle Cette approche essaye de trouver des modèles qui sont simultanément simple et permettent de bien décrire un processus ou un système. La structure d'un graphe peut se considérer comme une description compressée permettant d'approximer les informations que contient la matrice d'adjacence. En exploitant cette vision, Rosvall et Bergstrom ont proposé [153] une méthode où le graphe avec les communautés est représenté par une version synthétique, Y, qui est échangée entre un signaleur et un receveur qui va essayer de déduire la structure du graphe d'origine X à partir de celle-ci. Ainsi, la meilleure partition est celle où Y contient le plus d'informations sur X, et une fonction $H(X|Y)$ permet de calculer l'information conditionnelle de X étant donné Y, que l'on cherchera à minimiser. Le principe de longueur de description minimale, *Minimum Description Length (MDL)*, propose une solution à ce problème et l'optimisation est effectuée par un recuit simulé. Cela rends la méthode lente et ne permet pas d'envisager de traiter des graphes dépassant 10^4 sommets. L'auteur indique que des approches plus efficaces peuvent être envisagées au risque de réduire la qualité des solutions ; cependant cette méthode semble plus performante (sur les benchmarks de la littérature) que l'optimisation de la modularité, notamment dans le cas où les communautés sont de tailles différentes. Les mêmes auteurs proposent [154] une amélioration sur leur technique et l'objectif est toujours de compresser l'information nécessaire pour décrire le graphe. Le marcheur aléatoire est mis à contribution pour la diffusion de l'information. Un codage de Huffman est utilisé pour nommer les sommets, le but étant de compacter l'information (un même nom de sommet pourra être utilisé tant que les sommets sont dans des communautés différentes). La détection de communautés devient un problème de codage : trouver la partition qui propose la description de taille minimale pour une marche aléatoire infinie. La minimisation est effectuée par un algorithme glouton couplé à un recuit simulé. Dans une publication ultérieure [155], une technique d'algorithme glouton rapide est utilisée, et une *probabilité de téléportation* θ est introduite pour garantir l'équiprobabilité. Cette approche s'applique à la fois sur des graphes orientés ou non. Le

code source est mis à disposition [9] gratuitement pour une utilisation académique, et est disponible pour les différentes publications précitées.

Le même principe de description de taille minimale a été mis en place [41] par Chakrabarti pour transformer la matrice d'adjacence en une matrice bloc diagonale et en essayant d'optimiser le compromis entre un nombre limité de blocs (pour une bonne compression de la topologie du graphe), et des blocs homogènes (pour une description compacte de leur structure). Ceci est représenté par un coût total d'encodage T qui contient le nombre total de sommets du graphe, le nombre de blocs et le nombre de sommets et d'arc dans chacun. Son optimisation part d'un graphe où tout est dans un seul cluster, puis à chaque étape on effectue une bipartition du cluster avec la plus grosse entropie par sommet. Quand T ne peut plus être diminué on obtient k^* clusters et la complexité est donnée pour $O(I(k^2)^2 m)$ avec I le nombre d'itérations pour la convergence (indiqué comme étant généralement ≤ 20). L'auteur en conclut que la méthode peut s'appliquer à de grands graphes sans préciser d'ordre de grandeur.

4.2.2.6.4 Théorie de l'information L'utilisation de la théorie de l'information est quelque peu similaire au cas précédent, il s'agit de faire de la compression de données en quelque sorte. Ziv, Middendorf et Wiggins exploitent [203] cette approche basée sur le principe *information bottleneck method* : celui-ci repose sur une mesure importante, l'information mutuelle $I(X,Y)$ de deux variables aléatoires X et Y, et est définie par :

$$I(X,Y) = \sum_x \sum_y P(x,y) log(\frac{P(x,y)}{P(x)P(y)})$$

Avec $P(x)$ la probabilité que $X = x$ et $P(x,y)$ la probabilité jointe, i.e., $X = x$ et $Y = y$. $I(X,Y)$ indique combien nous pouvons découvrir de X en connaissant Y et inversement. Soient X la variable d'entrée, Z celle spécifiant la partition et Y la variable codant l'information que l'on veut conserver, le but est de minimiser l'information mutuelle entre X et Z avec la contrainte que les informations à propos de Y que l'on extrait de Z soient pertinentes. Pour le problème de détection de communauté, une question qui se pose est le choix de la variable Y ; Ziv, Middendorf et Wiggins proposent d'adopter l'information structurelle encodée pendant le processus de diffusion dans le graphe. Ils introduisent également le concept de modularité de réseau, une définition alternative de la modularité qui caractérise le graphe dans son entièreté contrairement à la définition classique. L'auteur n'indique pas de résultat quant à la qualité de la méthode.

4.2.2.7 Autres méthodes

L'auteur présente dans cette section des algorithmes qui ne correspondaient pas aux catégories précédentes.

Une première méthode *simple et rapide* (sic) est proposée [140] par Raghavan, Albert et Kumara et s'intitule *propagation d'étiquette*. Le principe décrit est le suivant : initialement, tous les sommets se voient attribués une même étiquette. À chaque itération,

9. http://www.tp.umu.se/~rosvall/code.html

un balayage a lieu sur tous les sommets en suivant un ordre aléatoire séquentiel et chaque sommet prend l'étiquette qui est majoritairement portée par ses voisins. En l'absence de majorité unique, l'une des étiquettes majoritaires est tirée au hasard. Les étiquettes vont ainsi se propager dans le graphe, et la majorité va disparaitre, seules resteront les dominantes, et le processus s'arrête quand chaque sommet a l'étiquette majoritaire de ses voisins ; et les communautés sont les groupes identifiés par la même étiquette. Plusieurs solutions peuvent être proposées par la méthode. Les tests sur des graphes réels montrent que les différentes partitions alternatives sont similaires. Les auteurs proposent également d'utiliser l'ensemble des étiquettes d'un même sommet dans différentes partitions pour améliorer les résultats. Un avantage important de cette approche est qu'elle ne nécessite pas de connaître a priori le nombre ni la taille des clusters. De plus la complexité est en $O(m)$ et le nombre d'étapes de convergence est très faiblement lié à la taille du graphe. Tibély et Kertész ont par ailleurs montré [174] que cette approche revient à trouver un minimum local d'énergie sur un modèle de *Potts* à température nulle. Plusieurs auteurs proposent des améliorations pour éviter des effets de bords gênants de l'algorithme tels la présence d'une grosse communauté entourée de plein de petites. Finalement, l'approche donne de bons résultats et est encourageante.

L'approche *L-shell method* est proposée [10] par Bagrow et Bollt pour trouver l'appartenance à une communauté de n'importe quel sommet. Les communautés sont définies localement à partir d'un critère lié au nombre d'arcs entrant et sortant d'un groupe de sommets. Grâce à sa nature locale cette méthode est rapide mais la limite forte est qu'elle ne fonctionne bien que dans le cas où le sommet d'origine est suffisamment équidistant de la frontière de sa communauté. Les auteurs proposent la création d'une matrice d'appartenance M, dont les éléments M_{ij} valent 1 si le sommet j appartient à la communauté du sommet i, pour passer outre cette limitation. Cette matrice peut être réarrangée par des permutations de lignes et de colonnes suivant leur distance mutuelle (définie comme étant le nombre d'entrées dont les éléments diffèrent) ; finalement, si le graphe a une structure de communauté, la matrice réarrangée l'exposera sous la forme de bloc. Cette méthode permet d'identifier des communautés recouvrantes.

Une méthode similaire, *bridge bounding* est présentée [134] par Papadopoulos et al., la différence étant que la croissance se fait autour d'un sommet jusqu'à ce qu'il touche la frontière des arcs. Ils peuvent être trouvés grâce à divers mesures déjà présentées. Un problème important est qu'il est rare qu'il y ait une distinction claire dans la distribution de ces valeurs, et donc qu'il revient à l'utilisateur de déterminer une limite. L'auteur indique que les meilleurs résultats de la méthode apparaissent avec une combinaison d'une somme pondérée des coefficients de clustering d'arc sur un voisinage de cet arc, et possède une complexité évaluée à $O(< k >^2 m + < k > n)$ avec $< k >$ le degré moyen du graphe.

Clauset effectue [49] une découverte locale des communautés avec une technique d'optimisation gloutonne maximisant la valeur de la mesure de modularité locale. Pour une communauté $\mathcal{C}$, sa frontière $\mathcal{B}$ est définie comme l'ensemble des sommets ayant au moins un voisin hors de $\mathcal{C}$, et la modularité locale R est le rapport du nombre d'arcs ayant leurs deux extrémités dans $\mathcal{C}$ avec ceux ayant au moins une extrémité dans $\mathcal{C}$. L'optimisation gloutonne est de complexité $O(n_c^2 < k >)$ avec $< k >$ le degré moyen du graphe.

Eckmann et Moses utilisent [59] également un critère local pour détecter les com-

munautés, en se basant sur le coefficient de clustering d'un sommet : de nombreux arcs impliquent de nombreuses boucles a sein d'une communauté, et ainsi, ses sommets ont plus de chances d'avoir un coefficient élevé. Ils introduisent les outils pour placer le graphe dans un espace géométrique grâce auquel les communautés se distinguent par leur courbure. Les auteurs ont appliqué leur méthode sur un graphe du Web construit de telle sorte que les sommets soient des pages et les arcs sont les liens entre ces pages. Ils montrent que les communautés sont les pages parlant de sujets identiques.

L'utilisation d'approximation de graphe est documentée [29] par Bo et al. pour détecter des communautés (mais pas seulement). Les auteurs introduisent un graphe prototype de communauté dont la structure de communauté ne fait aucun doute, et le problème devient de déterminer quel est le graphe prototype qui approxime le mieux l'original, la qualité étant mesuré par la distance calculée entre les matrices respectives. Trois algorithmes sont proposés en variation de la méthode *Community Learning by Graph Approximation (CLGA)* : les deux premiers visent à diviser le graphe en communautés recouvrantes (ou non) ; le troisième ajoute une contrainte imposant des groupes de tailles similaires. La complexité pour les trois est donnée pour $O(tn^2k)$, avec t le nombre d'itérations avant convergence, et k le nombre de groupes (à donner en entrée).

L'utilisation d'une approche basée sur un réseau de résistances électriques est exploitée [189] par Wu et Huberman : le graphe est transformé en un réseau électrique, les arcs sont de résistance unitaire, et les différences de potentiel sont exploitées au travers des équations de Kirchoff. Leur résolution exacte est trop consommatrice de temps, mais de bonnes approximations existent en temps linéaire sur des graphes creux, avec une complexité $O(nlog(n))$. D'autres travaux sont cités se basant sur cette méthode, et les résultats sont encourageants. La limite de cette approche est qu'il est nécessaire de connaître à l'avance le nombre de clusters.

Ohkubo et Tanaka observent [123] que le volume – défini comme le rapport entre le nombre de sommets et la densité d'arcs dans la communauté – est un facteur de détection de celles-ci : le volume doit être faible. Et ils posent l'hypothèse que V_{total}, la somme des volumes, est une référence fiable pour détecter la qualité d'une partition. Leur méthode repose sur une minimisation de V_{total} par un recuit simulé. L'auteur ne donne pas d'informations quant à la qualité de cette approche.

Zarei et Samani remarquent [198] qu'un graphe et son complément permettent d'identifier les communautés : il suffit de chercher les anticommunautés dans le graphe complémentaire. Ils proposent une approche spectrale (basée sur les vecteurs propres, donc), qui donne de bons résultats comparés aux autres approches spectrales sur les benchmarks de la littérature. L'auteur remarque cependant que l'échantillon de graphes pour les tests est plutôt petit. De plus, sur des grands graphes creux, le complémentaire sera très dense et ne laissera pas apparaitre de structure de communauté claire.

Une approche d'évolution dynamique du simplex est étudiée [72] par Gudkov et al. où le graphe est représenté comme un ensemble de points dans un espace multidimensionnel ($n-1$ dimensions). Les points sont déplacés par les forces (attractives ou répulsives suivant les cas) appliquées par les autres points. Les communautés se distinguent par le peu de liens présents entre elles ; cependant si elles sont trop liées ensemble, les points dans l'espace ne seront pas clairement séparés et elles seront mal identifiées. Les auteurs définissent

les communautés comme des groupes suffisamment proches (avec une valeur limite), la distance étant calculée entre chaque paire de sommets. Cette valeur limite permet de paramétrer la recherche et de faire apparaitre des communautés à différentes résolutions. Le cœur de l'algorithme consiste en la résolution d'équations différentielles décrivant la dynamique dans un milieu visqueux ; la complexité est de $O(n^2)$. Aucune information quant à la pertinence de cette approche n'est fournie.

La dernière méthode proposée [3] se décompose en deux étapes : Anand et al. appliquent d'abord une approche permettant de rapidement décomposer le graphe en sous graphes, puis ils appliquent la technique *Clique Percolation*, décrite dans la sous-section 4.2.2.8.1 qui suit, pour identifier les communautés.

4.2.2.8 Détection de communautés recouvrantes

Les techniques présentées précédemment ne pouvaient détecter que des communautés strictement distinctes ; dans la réalité, il peut y avoir des sommets qui appartiennent à plusieurs communautés.

4.2.2.8.1 Clique percolation La méthode *Clique Percolation* a été proposée [133] par Palla et al. et repose sur le fait qu'au sein d'une communauté, les arcs internes vont avoir tendance à former des cliques. Ils introduisent trois concepts nécessaires pour implémenter leur idée :

- Adjacence : deux k-cliques sont adjacentes si elles partagent $k-1$ sommets ;
- Connectivité : deux k-cliques sont connectées si elles font partie d'une chaîne de k-clique ;
- Communauté k-clique : c'est le plus grand sous-graphe obtenu par l'union d'une k-clique et de toutes les k-cliques qui y sont connectées.

Le processus commence en cherchant des cliques maximales, et en construisant une matrice O de recouvrement des cliques : matrice $n_c \times n_c$ avec n_c le nombre de cliques et O_{ij} est le nombre de sommets partagés entre les cliques i et j. Pour trouver les k-cliques, il suffit de conserver les entrées de O qui sont supérieurs ou égaux à $k-1$, mettre les autres à 0, puis en extraire les composantes connexes. Les auteurs indiquent que dans la réalité des graphes creux de 10^5 sommets, bien que dans le cas général le problème de détection de clique soit de complexité temporelle exponentielle à la taille du graphe. Cette méthode permet de distinguer clairement les graphes avec une structure de communauté des graphes aléatoires. De plus les auteurs montrent que les communautés trouvées grâce à leur méthode ne proviennent pas de variations aléatoires. Une implémentation [10] est proposée sous une licence non libre et restrictive ; le code source n'est pas disponible.

Des modifications ont été proposées pour prendre en charge les graphes orientés, pondérés et bipartites. Palla et al. traitent le cas des graphes orientés en définissant des k-cliques orientées : un graphe complet avec k sommets tels qu'il existe un ordre entre ceux-ci et chaque arc part d'un sommet de rang supérieur au sommet de destination. L'ordre est calculé à partir du degré sortant restreint, qui correspond au rapport entre le nombre d'arcs sortants sur degré total. Le cas des graphes pondérés est traité [63] par

10. http://cfinder.org/

Farkas et al. en mettant une limite sur les poids puis en traitant le graphe comme étant non pondéré. Lehmann, Schwartz et Hansen étendent [95] l'approche pour les graphes bipartites en définissant une *biclique*. Trouver ces N_c bicliques est un problème NP-complet (le nombre de bicliques croît exponentiellement avec la taille du graphe) et l'algorithme proposé est de complexité $O(N_c^2)$, cependant, sur un graphe creux, la complexité tombe à $O(m)$ avec m le nombre d'arcs.

Une version séquentielle de la méthode de percolation de clique est proposée [86] par Kumpula et al. : les k-cliques sont détectées par une insertion séquentielle d'arcs en partant d'un graphe vide ; à chaque fois on vérifie si l'arc ajouté génère de nouvelles k-cliques en cherchant des $(k-2)$-cliques dans le voisinage de l'arc ajouté. La complexité finale est linéaire par rapport au nombre de k-cliques du graphe ; mais en pratique l'approche est plus rapide que l'implémentation originale. De plus l'auteur relève un autre avantage quant à la prise en charge des graphes pondérés : en insérant les arcs par poids décroissant, il est possible en une seule exécution d'avoir la structure de communauté pour toutes les valeurs de poids.

Une limite de cette méthode est qu'elle fait l'hypothèse que le graphe contient un grand nombre de cliques, et elle peut donc échouer à découvrir les communautés lorsqu'elles sont peu nombreuses. Au contraire, sur certaines instances la méthode peut donner des résultats triviaux non pertinents. L'auteur note un problème plus fondamental : ce sont des sous-graphes contenant beaucoup de cliques qui sont découverts, pas forcément des communautés. Sur les graphes de la vie réelle, de plus, de nombreux sommets sont ignorés (les feuilles notamment). Enfin, les critères tels que k ou les limites pour les graphes pondérés sont difficiles à définir voire totalement arbitraires.

4.2.2.8.2 Autres approches Baumes et al. proposent [23] une méthode détectant des communautés recouvrantes : une communauté est un sous-graphe qui optimise localement la valeur d'une fonction W (une mesure liée à la densité d'arcs au sein du cluster), et différents sous-ensembles qui se recouvrent peuvent être des optimums locaux. Deux heuristiques sont proposées pour trouver tous les clusters : *Iterative Scan (IS)* et *Rank Removal (RaRe)*, de complexité $O(n^2)$ sur des graphes creux. Les meilleures performances sont atteintes en utilisant *IS* pour améliorer les résultats obtenus avec *RaRe*. Les auteurs ont proposé une amélioration qui réduit le temps de calcul d'un ordre de grandeur sur les graphes creux, pour une qualité comparable des solutions.

Une approche combinant une correspondance spectrale, un clustering flou et l'optimisation d'une fonction de qualité est présentée [199] par Zhang, Wang et Zhang. L'algorithme comporte trois phases importantes : d'abord, placer les sommets dans un espace Euclidien, ensuite, regrouper les points des sommets en n_c (variant entre 2 et $K-1$) clusters, et enfin, maximiser une fonction de modularité sur les couvertures trouvées précédemment. La méthode a une complexité majorée par le calcul des vecteurs propres pour la partie spectrale, $O(K^2n + Km)$, qui sera linéaire suivant n si le graphe est creux et que $K << n$.

Zhang, Wang et Zhang utilisent [199] la similarité des sommets et la minimisation d'une fonction. La méthode peut s'appliquer à des graphes pondérés et la minimisation est un problème d'optimisation qui est résolu avec une méthode itérative basée sur le

gradient. Pour un nombre fixé de clusters n_c la complexité est de $O(n^2 n_c h)$ avec h le nombre d'itérations pour la convergence ; cela limite son application à des graphes assez petits. Si n_c n'est pas connu, la meilleure couverture retenue est celle qui maximise une valeur de modularité. Une implémentation, différente de celle utilisée pour l'article mais qui doit produire les mêmes résultats, est disponible [11] sous licence GNU GPL.

La dernière méthode présentée [71] est proposée par Gregory et fonctionne en trois étapes : d'abord une transformation (*Peacok*) du graphe où l'on retire les sommets recouvrants, puis, une technique de détection de communautés est appliquée et enfin la partition obtenue est mise en correspondance avec une couverture en replaçant les sommets par ceux du graphe d'origine. Une mesure qualifiant la capacité de scission d'un sommet, *split betweeness*, est utilisée pour la première partie des opérations. La complexité globale provient principalement de l'étape de transformation : $O(n^3)$. Une approximation locale est proposée, de complexité $O(nlog(n))$ ce qui rends l'approche intéressante. L'auteur ne propose pas de retour quant à la qualité de la méthode.

4.2.2.9 Communautés hiérarchisées et méthodes multirésolutions

La structure du graphe peut être visible à différentes échelles, et en l'absence d'informations, il est intéressant d'avoir des méthodes de détection qui sachent travailler à différentes échelles. Parfois, également, les communautés peuvent être distinctes mais imbriquées, ce qui justifie les méthodes hiérarchiques.

4.2.2.9.1 Méthodes multirésolutions La résolution, dans ces méthodes, est souvent un paramètre modifiable. L'approche basée sur un modèle de verre de spin évoquée dans le paragraphe 4.2.2.3.2 par Reichardt et Bornholdt repose, par exemple, sur un paramètre γ.

Dans [136], Pons se base sur l'optimisation de fonctions de qualité multi-échelle, dont la modularité multi-échelle. Pour évaluer la pertinence d'une partition, il propose également une fonction chargée de cette évaluation et dépendante de l'échelle.

Une modification de la modularité est étudiée [6] par Arenas, Fernández et Gómez : modifier le calcul en faisant participer également les sommets dans la valeur de densité des arcs du cluster via l'ajout d'une boucle r à chaque sommet. La modularité maximum est ensuite calculée grâce à une optimisation extrémale et une recherche tabou. L'étude du rapport entre le nombre de clusters et r permet d'identifier des plateaux : ce sont les communautés ; la taille du plateau indique la sensibilité de la partition vis-à-vis des modifications de r. Cette approche est lente, notamment à cause du nombre de valeurs de r à évaluer. Si, de plus, la modularité maximum est calculée avec des approches consommatrices, alors il est difficile d'espérer dépasser quelques centaines de sommets. L'auteur souligne que la parallélisation sur les différentes valeurs de r est triviale.

La détection d'une structure hiérarchique et de communautés recouvrantes est proposée dans [89] par Lancichinetti, Radicchi et Ramasco. Le cœur de la méthode repose sur l'optimisation d'une fonction de *fitness* qui estime la force d'un cluster. Si celle-ci peut être arbitraire, les auteurs propose une fonction de compromis entre le degré interne et le

11. https://github.com/ntamas/fuzzyclust

degré total du cluster. L'optimisation part d'un cluster avec un seul sommet sélectionné arbitrairement. Puis on y ajoute et retire des voisins tant que la valeur de *fitness* augmente. Quand le maximum est atteint, le cluster est fermé, un autre sommet choisi au hasard parmi ceux encore disponibles et la procédure recommence. L'aspect recouvrement est géré lors de la construction des clusters, puisqu'un sommet peut être ajouté à plusieurs. La complexité sur des graphes creux est donnée pour $O(n^{\beta})$ avec $\beta \approx 2$ pour des petites résolutions (sans indication supplémentaire) et $\beta \approx 1$ si la résolution est plus grande, le pire cas est en $O(n^2 log(n))$.

L'utilisation d'un modèle de spin de Potts est considérée [151] par Ronhovde et Nussinov, la principale différence dans leur équation Hamiltonienne est l'absence de terme se référant au modèle nul. Le modèle effectue une rétribution énergétique entre paires de sommets dans la même communauté : les arcs entre sommets sont rétribués positivement, ceux manquant sont pénalisés ; un paramètre γ permet d'ajuster le compromis. L'optimisation vise à minimiser l'énergie en faisant est échanges de sommets. Ils introduisent ensuite [152] un nouveau critère pour estimer la stabilité des partitions en calculant leur similarité avec un γ donné et des conditions initiales différentes. Ce critère est utile pour estimer la pertinence d'une partition, quel que soit l'algorithme retenu.

L'auteur termine cette partie en précisant que le problème général des méthodes multirésolutions est d'estimer la stabilité des partitions sur de grands graphes.

4.2.2.9.2 Méthodes hiérarchiques Les approches hiérarchiques sont assez récentes dans la littérature, et il n'y a aucune garantie que la hiérarchie récupérée corresponde exactement à celle du graphe. La manière naturelle d'aborder le problème est le clustering hiérarchique, qui a été présenté dans le paragraphe 4.2.2.1.2.

Dans [157] les auteurs Sales-Pardo et al. utilisent la matrice de similarité calculée entre tous les sommets pour déterminer la structure hiérarchique. La mesure de similarité utilisée est l'affinité des nœuds, qui se base sur la modularité de Newman-Girvan ; l'affinité entre deux sommets, c'est la fréquence à laquelle ils coexistent dans la même communauté dans les partitions qui correspondent à des optimums locaux de la modularité, i.e., l'ensemble des partitions telles que la modularité est stable, pour laquelle on ne peut plus trouver de permutations, de fusion ou de découpage de cluster qui améliorent la modularité. Le calcul d'un score *z-score* permet d'avoir une indication de la pertinence de la structure hiérarchique ainsi découverte : plus celui-ci est important, plus la pertinence est élevée. La seconde partie de la procédure n'est effectuée que si la structure hiérarchique a été montrée pertinente. Il s'agit de mettre la matrice d'affinité sous une forme bloc diagonale via l'optimisation (minimisation) d'une fonction coût correspondant à la distance moyenne entre les sommets connectés de la diagonale : les blocs sont les communautés. La méthode n'est pas rapide mais donne de bons résultats sur des graphes générés et pour des graphes réels de la littérature.

Les graphes aléatoires hiérarchiques sont introduits [50, 51] par Clauset, Moore et Newman, Clauset, Moore et Newman comme un dendogramme $\mathcal{D}$ accompagné d'un ensemble de probabilités p_r associées à ses nœuds internes. Ils définissent un ancêtre d'un sommet i comme n'importe quel nœud interne du dendogramme que l'on peut rencontrer en remontant de la feuille i vers l'origine, et la probabilité que deux sommets i et j soient

liés est donnée par la probabilité p_r du plus petit ancêtre commun entre i et j. Les auteurs recherchent donc un modèle qui corresponde le mieux à la topologie du graphe, grâce à de l'inférence Bayésienne. L'ensemble statistique des graphes aléatoires hiérarchiques correspondant à un graphe $\mathcal{G}$ peut être défini en associant à chaque modèle $(\mathcal{D}, \overline{p_r}$ une probabilité proportionnelle à la vraisemblance maximale $\mathcal{L}(\mathcal{D})$, et il peut être énuméré par une méthode de Monte-Carlo avec une chaîne de Markov. La complexité est donnée pour $O(n^2)$ en moyenne, mais l'auteur note qu'elle peut être (beaucoup) plus élevée, et des graphes de quelques milliers de sommets ont pu être analysés. Les tests sur des graphes réels montrent de bons résultats. Enfin leur approche propose un ensemble de hiérarchies possibles associées à des probabilités, ce qui permet de couvrir plusieurs cas, puisque plusieurs structures peuvent être valides pour une topologie de graphe donnée. L'exploitation de la structure de communauté est par contre notée comme plus difficile, d'autant plus de l'utilisation d'un dendogramme qui rends impossible le calcul d'un ordre des partitions par rapport à leur pertinence. Le code source différentes implémentations est disponible [12] sous licence libre GNU GPL.

4.2.3 Confrontation au monde réel

Fortunato propose de terminer son état de l'art en posant un regard plus appuyé sur les cas « réels », et plus particulièrement, d'analyser quelles sont les propriétés des vraies communautés, qui sera présenté dans la sous-sous section 4.2.3.1. Par la suite, dans la section 4.2.3.2, nous nous intéresserons à son analyse sur des réseaux réels (biologiques, sociaux, etc.), afin d'amener les éléments de comparaison par rapport à notre cas.

4.2.3.1 Quelques propriétés de vraies communautés

Une première question à laquelle il est nécessaire de répondre est de savoir si, en général, les communautés au sein d'un graphe sont d'une taille similaire ou pas. L'analyse de la littérature qui est proposée montre qu'il ne semble pas y avoir de taille caractéristiques et que des petites communautés coexistent avec de très grandes : un exemple donné est construit à partir des données des achats sur le site marchand *Amazon.com*.

Une analyse systématique de communautés dans des grands réseaux réels est proposée [97] par Leskovec et al., avec pour objectif principal d'estimer la qualité des communautés de tailles différentes ; la conductance est la mesure retenue pour mesurer la qualité. Des valeurs faibles vont indiquer un cluster de bonne qualité. Pour chaque réseau les auteurs proposent un *network community profile plot (NCPP)* qui graphe le score de conductance minimale pour les sous-graphes. Ils observent que cette courbe a une forme caractéristique : elle décroît jusqu'aux environ des sous-graphes avec environ 100 sommets, puis elle croît de manière monotone ; et en déduisent que les communautés sont bien définies uniquement quand elles restent assez petites. Une analyse d'un réseau social avec une structure de communauté bien connue, le réseau de bloggeurs *LiveJournal*, montre le même comportement. L'auteur note que cette limite située vers 100 sommet serait ainsi cohérente avec la limite conjecturée par *Dunbar* : 150 personnes au maximum peuvent travailler ensemble. Et de conclure que ces résultats pourraient être liés à la conductance, justifiant de les valider

12. http://tuvalu.santafe.edu/~aaronc/hierarchy/

avec une approche alternative. Cependant, cela met en lumière l'importance de valider la pertinence des clusters.

Une classification du rôle des sommets dans une communauté pour en déterminer leurs propriétés a été commencée [73] par Guimerà et Amaral : ce rôle va dépendre de deux valeurs. La première est le *degré au sein du module* z_i, la seconde est *le ratio de participation* P_i. De grandes valeurs de z indiqueront que le sommet a plus de voisins dans sa communauté que tous les autres sommets de celle-ci ; ceux dont le score est > 2.5 sont classés comme des *hubs*. Des valeurs de P proches de 1 indiquent que les voisins du sommet sont distribués uniformément au sein de toutes les communautés. À partir de ces deux indices, les auteurs définissent 7 rôles pour les sommets :

- Les sommets *non-hubs* peuvent être :
  - Ultra-périphérique, pour $P \approx 0$
  - Périphérique, pour $P < 0.625$
  - Connecteurs, pour $0.625 < P < 0.8$
  - Sommets orphelins, pour $P > 0.8$
- Les sommets qui sont des *hubs* peuvent être :
  - *Hubs* provinciaux, pour $P <\approx 3$
  - *Hubs* connecteurs, pour $0.3 < P < 0.75$
  - *Hubs* orphelins, pour $P > 0.75$

Dans le cas où les communautés sont recouvrantes, il est possible d'étudier d'autres propriétés statistiques : distributions des recouvrements, appartenance des sommets. Cette étude n'a pas permis de faire ressortir de caractéristique claire.

4.2.3.2 Application à des graphes réels

L'identification de communautés au sein de graphes est, dans la littérature actuelle, principalement tournée vers la mise au point de nouveaux algorithmes plus que dans la détection et l'application à des graphes provenant de différents domaines. L'auteur insiste sur le fait que cette présentation n'est pas exhaustive de la littérature, et qu'elle est assez principalement centrée sur les réseaux sociaux et la biologie.

4.2.3.2.1 Réseaux biologiques La modélisation sous forme de graphe dans le contexte de la biologie, notamment depuis les avancées liée à la génétique : réseaux d'interactions entre protéines, réseaux de gènes régulateurs, réseaux métaboliques. Ces réseaux sont organisés sous forme de modules qui s'associent suivant les fonctions : les protéines ont tendance à l'associer pour former deux types de modules cellulaires (les complexes de protéines et les modules fonctionnels). L'identification de ces modules est une pièce fondamentale pour comprendre l'organisation et la dynamique des cellules.

Rives et Galitski étudie [150] l'organisation modulaire des protéines donnent sa forme de filament à une levure (*saccharomyces cerevisiae*). Les communautés ont été détectées avec une méthode hiérarchique, et les protéines interagissent principalement avec les membres de leur communauté. Les arcs entre les modules sont des points de communication importants. Spirin et Mirny travaillent [167] également sur une levure et identifient (détection de clique, clustering superparamagnétique, optimisation de la densité d'arcs) les complexes de protéines ainsi que les modules fonctionnels. Chen et Yuan

détectent [45] également des modules fonctionnel, mais en ajoutant des informations de poids (permettant d'avoir une structure modulaire plus régulière). De plus, en étudiant les modules auxquelles des fonctions inconnues de gènes appartiennent, ils ont été en mesure de prédire leur rôle.

Les réseaux métaboliques sont également une source d'inspiration. L'auteur cite notamment les travaux [78] de Holme, Huss et Jeong qui reconstitue la hiérarchie métabolique.

La détection de groupes de gènes est étudiée [187] par Wilkinson et Huberman à partir d'une base de données de co-occurrences. Ce type de travaux peut permette d'identifier les liens entre des gènes et des maladies, ou bien des communautés de gènes liées au cancer du côlon permettent d'identifier les fonctions des gènes.

4.2.3.2.2 Réseaux sociaux Les études sur les réseaux de personnes, réseaux sociaux, ont été amplifiées par l'arrivée des communications électroniques (courriels, internet, téléphonie mobile). Ainsi, Blondel et al. étudient [27] les interactions entre les clients d'un opérateur de téléphonie mobile belge. Le graphe contient 2.6 millions de sommets et les arcs correspondent aux communications passées entre chaque utilisateur ; ils sont pondérés suivant la durée de la communication. Les auteurs proposent leur approche (optimisation hiérarchique et rapide de la modularité) pour l'analyse et identifient des groupes. Notamment, ils retrouvent la dichotomie linguistique propre à la Belgique entre Flamands et Wallons. L'étude [176] des échanges par courriels au sein de *HP Labs* par Tyler, Wilkinson et Huberman montre que les communautés correspondent à l'organisation de la société en départements et aux groupes des projets.

Traud et al. utilisent [175] des données anonymes de *Facebook* et étudient les relations entre des étudiants qui sont liés sur le site. Une variante de l'optimisation spectrale de la modularité puis l'application d'une méthode similaire à Kernighan-Lin permet de détecter les communautés. Un objectif de l'étude était de pouvoir déduire les relations à partir des informations en ligne et hors lignes des étudiants. Ils trouvent des communautés construites par années, par dortoir, suivant les universités. La taille des groupes a été étudiée [195] par Yuta, Ono et Fujiwara à partir des données du plus grand réseau social japonais, *mixi.jp* et montrent que les groupes sont soit petits soit larges, mais qu'il y a assez peu de tailles intermédiaires.

Sur le modèle des études de collaborations entre chercheurs, Gleiser et Danon ont regardé [68] les collaborations artistiques entre musiciens de jazz et détectent des communautés caractérisés par les origines ethniques et géographiques, ces dernières s'expliquant par la localisation des studios d'enregistrement.

4.2.3.2.3 Autres réseaux Dans leur étude [154], Rosvall et Bergstrom montrent la forme des collaborations scientifiques à partir d'un échantillon de 6 000 revues : elle correspond à un « U », les sciences sociales et l'ingénierie de part et d'autre de ce « U », le centre étant composé d'une chaîne regroupant la médecine, la biologie, la chimie et la physique. Des données en provenance du site web *eBay* allemand ont servies [143] de base pour Reichardt et Bornholdt : le graphe est constitué des acheteurs potentiels (sommets) et un arc existe entre deux sommets si les deux clients ont exprimé un intérêt pour le même objet. Finalement, 85% des acheteurs étaient classés dans quelques catégories

principales correspondant plus ou moins à leurs centres d'intérêts. Jin, Parkes et Wolfe ont utilisé [80] des données similaires pour pouvoir suggérer des produits alternatifs.

4.2.4 Conclusion

L'état de l'art proposé par Fortunato nous permet d'avoir une idée des possibilités quant à la détection de communauté. Dans sa conclusion, l'auteur note que ce problème est loin d'être résolu dans le cas général : le développement de ces techniques et outils a été plutôt chaotique, et il manque toujours de son point de vue un cadre théorique précis et clair, notamment sur ce que la détection de communautés est censée faire. L'absence de cadre explique en partie l'explosion de propositions d'algorithmes. Cela a également un impact direct pour nous, que l'auteur évoque un peu plus loin : quelqu'un qui, comme nous, cherche à détecter des clusters ou des communautés dans un graphe précis, aura des difficultés à trouver quelle approche utiliser. Ceci explique également que les gens exploitent la technique la plus « en vue », et utilisent ce dont ils ont entendu parlé, ou que l'on leur a conseillé. Cela étant, la méthode parfaite n'existe pas.

Grâce à cet état de l'art, nous avons donc un échantillon de méthodes ainsi que les algorithmes les implémentant, avec les différentes évolutions apportées, et souvent même les sources disponibles. Dans la section suivante 4.3 nous exploiterons ces retours pour identifier les paramètres importants dans le graphe, que nous avons déjà analysé dans la section 4.5, afin d'appliquer certaines méthodes.

4.3 Méthodes applicables

Après avoir présenté l'état de l'art sur la thématique de la détection de communauté dans la section précédente, nous allons maintenant détailler les méthodes qui nous intéressent, et notamment mettre en exergue les conditions sur la topologie du graphe.

4.3.1 Caractéristiques importantes

Dans cette première sous-section, et en nous basant sur l'état de l'art [66] de Fortunato, nous allons résumer les caractéristiques importantes sur les graphes qui vont impacter le choix des méthodes que nous pourront appliquer.

4.3.1.1 Importance de la densité du graphe

Une première caractéristique qui est souvent mentionnée est la densité du graphe. Dans de nombreux cas l'impact est au moins documenté sur la complexité des algorithmes : celle-ci est « acceptable » sur les graphes creux, i.e., peu denses, alors qu'elle devient trop importante dans le cas général.

Dans son introduction, Fortunato précise qu'un graphe sera considéré comme creux si la quantité d'arcs est du même ordre de grandeur que le nombre de sommets. Cette première définition ne correspond pas réellement à notre cas. L'auteur indique cependant

Auteur(s)	Réf.	Cas général	Graphe creux	$\|V\|$
Blondel et al. (2008)	[27]	$O(m)$	-	10^9
Clauset, Newman et Moore (2004)	[52]	$O(md \times log(n))$	$O(n^2 log(n))$	10^6
Newman et Leicht (2007)	[120]	-	-	10^6
Newman (2004)	[117]	$O((m+n)n)$	$O(n^2)$	10^5
Palla et al. (2005)	[133]	-	-	10^5
Agarwal et Kempe (2008)	[1]	NP-difficile	-	10^4
Newman et Girvan (2004)	[119]	$O(n^3)$	$O(m^2 n)$	10^3

Table 4.1 – Synthèse de la relation entre complexité et taille du graphe. Le nombre d'arcs est donné par m, le nombre de sommets est n.

que cette contrainte n'est plus vraie dans le cas où le graphe est pondéré et où la répartition des poids est hétérogène.

4.3.1.2 Taille de l'instance à analyser

Chaque méthode et algorithme a sa complexité, ce qui va contraindre la taille de l'instance qu'il est possible d'analyser en un temps raisonnable. Tout au long de son état de l'art, Fortunato donne des ordres de grandeur de taille de graphe que les méthodes peuvent traiter. Pour une même famille de méthodes, le différentiel peut être important ; ainsi, par exemple, des algorithmes gloutons présentés dans la section 4.2.2.3.1.1 permettent d'analyser des graphes compris entre 10^6 et 10^9 sommets. Cette information n'est cependant pas mentionnée pour de nombreuses méthodes. De plus, seule la complexité temporelle est évoquée, alors que dans le cas de l'algorithme permettant de traiter 10^9 sommets, l'auteur précise que la limite ne se trouve pas sur le temps de calcul, mais sur la mémoire nécessaire, sans donner de point de repère ni de complexité spatiale.

Cependant, grâce aux données exposés dans l'état de l'art, il est possible d'avoir une idée de la relation entre la complexité temporelle et la taille des graphes qui sont analysables. Ces données sont présentées dans le tableau 4.1 ; elles permettent d'avoir un ordre d'idée de la taille des instances que nous pouvons être en droit de traiter suivant la complexité des algorithmes.

Dans le tableau 4.2 nous proposons une liste plus complète des complexités basée sur les données proposées dans l'état de l'art. Une liste similaire est proposée dans l'état de l'art de Fortunato sous la forme de deux tableaux qui sont tirés d'études antérieures. Quand cela est possible, la complexité est précisée pour le cas général et pour le cas des graphes creux. Nous indiquons également si les auteurs ont proposé une implémentation « de référence », en vue de la contrainte exposée dans la section 4.4.1.3.

4.3.1.3 Hiérarchie et structure du graphe

La topologie du graphe a un impact sur les méthodes qui peuvent être exploitées : la présence de communautés hiérarchiques, le recouvrement potentiel de ces communautés

		Complexité		
Auteur(s)	Réf.	Cas général	Graphe creux	Code dispo.
Algorithmes gloutons				
Blondel et al. (2008)	[27]	$O(m)$	-	✓
Clauset, Newman et Moore (2004)	[52]	$O(md \times log(n))$	$O(n^2 log(n))$	✓
Clauset (2005)	[49]	$O(n_c^2 < k >)$	-	
Newman (2004)	[117]	$O((m+n)n)$	$O(n^2)$	
Décomposition				
Rattigan, Maier et Jensen (2006)	[142]	$O(m)$	-	
Vragović et al. (2006)	[182]	$O(mn)$	-	
Radicchi et al. (2004)	[139]	$O(\frac{m^4}{n^2})$	$O(n^2)$	✓
Optimisation spectrale				
Wu et Huberman (2004)	[189]	$O(nlog(n))$	-	
Ruan et Zhang (2007)	[156]	$O((n+m)log(K))$	-	
Yang et Liu (2008)	[191]	$O(Kt(n \times log(n+m)))$	$O(n \times log(n))$	
And (2005)	[5]	$O(K^2 n + Km)$	-	
Recouvrement, hiérarchie				
Gregory (2009)	[71]	$O(n^3)$, $O(nlog(n))$	-	
Clauset, Moore et Newman (2008)	[50]	-	$> O(n^2)$	✓
Statistique				
Hofman et Wiggins (2008)	[77]	$O(n^\alpha)$, $\alpha \approx 1.44$	-	✓
Zanghi, Ambroise et Miele (2008)	[197]	$O(n^2)$	-	
Hastings (2006)	[76]	$O(nlog^\alpha(n))$	-	
Chakrabarti (2004)	[41]	$O(I(k^2)^2 m)$, $I < 20$	-	
Percolation de cliques				
Lehmann, Schwartz et Hansen (2008)	[95]	$O(N_c^2)$	$O(m)$	
Palla et al. (2005)	[133]	-	-	✓
Marcheur aléatoire				
Pons et Latapy (2005)	[137]	$O(n^2 d)$	$O(n^2 log(n))$	✓
Hu et al. (2008)	[79]	$O(T(< k > +1)n^2)$	-	
Dongen (2000)	[58]	$O(n^3)$	$O(nk^2)$	✓
E, Li et Vanden-Eijnden (2008)	[62]	$O(tlk(n+m))$	-	
Zhou (2003)	[201]	$O(n^3)$	-	
Synchronisation				
Boccaletti et al. (2007)	[28]	$O(mn)$	$O(n^2)$	
Optimisation (autres)				
Lehmann et Hansen (2007)	[94]	$O((m+n)n)$	-	
Lancichinetti, Radicchi et Ramasco (2010)	[89]	$O(n^2 log(n))$	$O(n^\beta)$, [a]	
Girvan-Newman				
Newman (2005)	[118]	$O((m+n)n^2)$	$O(n^3)$	
Newman et Girvan (2004)	[119]	$O(m^3 n)$	$O(n^4)$	
Autres				
Raghavan, Albert et Kumara (2007)	[140]	$O(m)$	-	
Papadopoulos et al. (2009)	[134]	$O(< k >^2 m + < k > n)$	-	
Bo et al. (2007)	[29]	$O(tn^2 k)$	-	
Gudkov et al. (2008)	[72]	$O(n^2)$	-	
Zhang, Wang et Zhang (2007)	[199]	$O(K^2 n + Km)$	-	✓
Zhang, Wang et Zhang (2007)	[199]	$O(n^2 n_c h)$	-	✓
Son, Jeong et Noh (2006)	[166]	$O(n^{2+\theta})$, $\theta \approx 1.2$	-	

Table 4.2 – Complexité des algorithmes proposés dans Fortunato. Le nombre d'arcs est donné par m, le nombre de sommets est n.

a. *beta* ≈ 2 pour de faibles résolutions, 1 sinon

vont limiter les algorithmes. L'auteur consacre des sections (4.2.2.9.2, 4.2.2.8) respectivement dédiées à présenter ceux qui sont applicables sur ce type de graphe. Cela implique qu'il est nécessaire de connaître, le cas échéant, à l'avance, la structure de ce que l'on souhaite détecter. En pratique, il n'y a aucune raison que cette contrainte soit un problème. La résolution est également un paramètre qui influe, une section (4.2.2.9.1) y est consacrée.

La présentation de l'application à différents cas réels, dans la section 4.2.3.2 montre plus précisément ce point : par exemple, en biologie, la structure hiérarchique est connue *a priori*, c'est-à-dire que l'on sait que les communautés recherchées doivent avoir une telle structure. L'utilisation de l'optimisation spectrale n'est envisageable que si le graphe contient une bisection, comme indiqué dans le paragraphe 4.2.2.3.1.4.

4.3.1.4 Nombre de communautés

La découverte de communauté n'implique pas nécessairement que la quantité de communautés soit inconnue : dans l'état de l'art présenté par Fortunato, deux familles de méthodes et d'algorithmes se distinguent clairement. D'abord, ceux pour lesquels le nombre de clusters à construire est un paramètre, une donnée qui sera *a priori* connue à l'avance. L'autre famille se chargera de donner cette quantité *a posteriori*.

Ce critère distinctif peut paraitre limitant au premier abord, l'auteur émettant souvent un avis négatif sur les méthodes présentant la caractéristique *a priori*, et il l'est dans certains cas. Mais nous serons intéressés dans notre étude à pouvoir contrôler le nombre de communautés (en fait, leur taille), ce qui rends au contraire l'existence de cette caractéristique très utile.

4.3.2 Caractéristiques de nos graphes

La section précédente 4.3.1 nous a permis d'établir les caractéristiques qui sont importantes à étudier dans le graphe et leur impact quant à la sélection des méthodes de détection de communauté applicables. À partir des résultats détaillés dans la section 3.4 nous proposons ici de faire le croisement entre ces deux sources d'informations afin de justifier l'étude de certaines méthodes. La sélection sera opérée dans la section 4.4.2.

4.3.2.1 Importance de la densité du graphe

Dans la section 4.3.1.1, nous avons relevé, à partir de l'état de l'art de Fortunato, l'impact de la densité du graphe sur les différentes méthodes d'analyse.

La section 3.4.2 documente l'occurrence des symboles, et montre que l'écrasante majorité a une fréquence faible et proche. Les symboles étant nos arcs, si le graphe est modifié de sorte à ce que chaque sommet ne soit relié que par un arc de poids correspondant au nombre d'arcs précédemment existants entre ces deux sommets, alors la distribution des poids devrait être en notre faveur. Ceci peut être vérifié en regardant le rapport du nombre d'arcs (chacun étant originellement de poids unitaire, cela correspond bien au poids total) sur le nombre de sommets par sous-répertoire ; une fois éliminés certains sous-répertoires non pertinents, la moyenne de ce rapport est à ≈ 13.6 quand l'écart-type est de ≈ 5.7.

La densité de nos graphes devrait donc être compatible avec les différentes méthodes de détection de communautés qui reposent sur des graphes pondérés et/ou hiérarchiques.

4.3.2.2 Taille de l'instance à analyser

La section 4.3.1.2 documente les contraintes des différentes méthodes quant à la taille des instances, et propose notamment un tableau (4.1) permettant de faire le lien entre la complexité des algorithmes, rappelée dans le tableau 4.2 et la taille de l'instance. Suivant cette complexité, la taille traitable varie entre 10^3 et 10^9 sommets, les publications référencées ne faisant pas mention de la complexité spatiale, bien qu'il soit noté son importance dans certain cas précis.

L'étude menée sur les différents graphes que nous avons produits nous a permis de chiffrer la taille des instances que nous allons traiter, et les chiffres sont proposés dans la section A.1.2. Ils nous permettent d'établir qu'entre l'instance **defconfig** et **allyesconfig**, la quantité de sommets dans les graphes est respectivement de l'ordre de 2 000 à 60 000. De plus, le suivi dans le temps montre une évolution suffisamment lente pour que les méthodes d'analyses ne soient pas rapidement limitées : entre le noyau 3.0 (juillet 2011) et 3.9 (avril 2013), l'inflation est de moins de 7 000 sommets pour **allyesconfig**, soit une moyenne de 700 sommets pour chaque nouvelle version. Une complexité de $O(n^2)$ pour le cas d'un graphe creux permet de traiter de l'ordre de 10^5 sommets, cela nous permet d'estimer, à partir du tableau 4.2, que les deux tiers environ des méthodes seront applicables dans notre cas ; surtout, une méthode dont la complexité est de l'ordre de $O(n^3)$ commencera à présenter des limites dans notre cas, particulièrement sur les instances **allyesconfig**.

4.3.2.3 Hiérarchie et structure du graphe

La structure et la hiérarchie du graphe a un impact sur les méthodes applicables et leurs résultats potentiels ; notamment, si celle-ci est hiérarchique et/ou peut avoir différents niveaux de « rendu », il est nécessaire d'appliquer des algorithmes capables de prendre en compte et de découvrir la hiérarchie à ces différents niveaux. Le graphe étant construit à partir des fichiers objets du noyau, et ceux-ci étants organisés physiquement suivant des « modules » correspondant à leur rôle au sein du noyau, il était attendu qu'une structure hiérarchique se retrouve dans le graphe. Nous avons proposé une visualisation sous la forme de « cartes de chaleur » dans la section 3.4.6, dans l'optique d'avoir une vue globale et un peu plus générale, justement, des interactions entre les différents sous-répertoires, pour pouvoir vérifier et valider ou non cette hypothèse.

L'analyse a été menée à la fois sur l'ensemble du noyau, en prenant le premier niveau de l'arborescence, dans la section A.6.1 ; et également sur des répertoires plus précis tels que `drivers/` dans la section A.6.2, `fs/` dans la section A.6.3, `kernel/` dans la section A.6.4 ou encore `net/` dans la section A.6.5. À chaque niveau d'analyse, nous avons pu observer à la fois des interactions entre les différents répertoires contenus, mais surtout, l'importance de la première bissectrice, qui révèle que les plus fortes interactions ont lieu à l'intérieur de chacun des modules. Ceci indique donc une forte structure hiérarchique, qui justifiera l'utilisation de méthodes d'analyses adaptées, et permet déjà de conjecturer que la répartition physique, l'organisation des sous-répertoires, reste correctement adaptée à

l'utilisation qui est faite par le code.

4.3.2.4 Nombre de communautés

Certaines approches nécessitent de contraindre et de définir le nombre de communautés que l'on souhaite obtenir. Ceci sera particulièrement utile pour l'un des objectifs poursuivi, et détaillé ci-après dans la section 4.4.1.2. Nous avons un graphe qui, dans le cas de l'instance **allyesconfig**, contient près de 60 000 nœuds. Nous proposons d'évaluer les performances avec des communautés de taille différentes :

- 10 communautés, composées de 6 000 nœuds chacune environ ;
- 100 communautés, composées de 600 nœuds chacune environ ;
- 1000 communautés, composées de 60 nœuds chacune environ ;
- 10 000 communautés, composées de 6 nœuds chacune environ.

4.4 Méthodologie d'étude

Après avoir présenté l'état de l'art de la détection de communautés dans les graphes, puis indiqué les critères pertinents et opéré une section sur les méthodes qui sont pertinentes dans le cadre de notre problème, il convient de procéder à leur évaluation pratique. Le but de cette section est d'introduire cette étude en présentant la méthodologie retenue pour effectuer ce travail. Les résultats seront proposés et analysés dans la section 4.5 suivante.

4.4.1 Objectifs et méthode de l'analyse

L'objectif de cette analyse est d'évaluer la pertinence des méthodes de l'état de l'art sur notre instance pour répondre à deux objectifs, et dans l'hypothèse où aucune méthode évaluée ne parvienne à fournir des informations utiles, nous serons amenés à identifier les problèmes et proposer des solutions. Comme indiqué dans l'introduction (4.1), les cas que nous voulons traiter sont :

- Étude de l'adéquation de l'arborescence des sources du noyau par rapport à l'utilisation effective des symboles, détaillé ci-après (4.4.1.1) ;
- Création de communautés de tailles adaptées (contrôlables) aux méthodes d'analyse statique et permettant de qualifier les liens entre communautés, détaillé ci-après (4.4.1.2).

Nous présenterons dans ces deux sous-sections à la fois le rôle de l'objectif ainsi que la manière de l'évaluer.

4.4.1.1 Adéquation de l'arborescence

Nous voulons vérifier que le code source est organisé, physiquement, suivant une hiérarchie qui reste pertinente par rapport à l'utilisation qui en est faite. Cette vérification se justifie d'abord d'un point de vue maintenance du code : celui-ci a été organisé en suivant une certaine logique, mais les évolutions successives peuvent la rendre caduque. S'assurer

que l'organisation du code « colle » à son utilisation permet de simplifier la vie des mainteneurs, par exemple ceux qui sont chargés de garantir la pérennité de certains noyaux dans le temps (*Long-Term Stable*).

Si au contraire l'organisation ne correspond pas à l'utilisation qui est faite du code source, cela va non seulement complexifier la vie des mainteneurs, mais peut également être une indication qu'il est temps de revoir l'organisation.

4.4.1.1.1 Critère d'évaluation Pour évaluer la qualité des résultats des différentes méthodes, nous proposons de comparer la hiérarchie obtenue avec celle du système de fichier. Plus elles seront différentes, moins la méthode aura été efficace. Il faut également éviter un effet de bord provenant de la comparaison : le critère doit permettre de déterminer si le résultat est utilisable ou non.

Par construction de notre graphe, les sommets possèdent une étiquette contenant le chemin complet dans le système de fichier. Nous pouvons donc facilement calculer, communauté par communauté, le taux de classement t_C d'une communauté C en comptant le nombre de feuilles différentes de l'arborescence : au plus les différents nœuds seront homogènes en matière de répertoire, au plus ce taux tendra vers 0.0.

$$t_C = \frac{\sum x_R}{|C|}$$

Avec x_R une variable binaire définie par :

$$x_R = \begin{cases} 1 & \exists v \in C \text{ tel que } R \in l_v \\ 0 & \text{sinon} \end{cases}$$

Et v un sommet de la communauté C, l_v l'étiquette portée par v contenant le répertoire, R un répertoire. $R \in l_v$ exprime le fait que le sommet v appartienne au répertoire R.

4.4.1.2 Communautés contraintes

Nous voulons exploiter la détection de communautés pour construire des sous-ensembles du noyau qui soient analysables par des outils de vérification statique. L'état de l'art a montré que même si ces outils s'améliorent, ils restent limités par le nombre d'états à analyser. L'idée ici est donc de construire des sous-ensembles de taille contrôlable pour qu'elle puisse correspondre à la capacité d'analyse, et donc les interconnexions peuvent être clairement spécifiées, identifiées et éventuellement optimisées. Notre but serait d'avoir des interfaces entre communautés qui soient les plus réduites possibles.

4.4.1.2.1 Critère d'évaluation La capacité à construire les communautés d'une taille demandée sera un premier critère d'évaluation de la pertinence de la méthode. Un second critère sera la taille des interconnexions. Pour estimer la taille des interconnexions, nous comptons le taux t_R d'arcs parmi l'ensemble de tous les arcs E qui relient des communautés.

$$t_R = \frac{\sum_{e \in E} c_e}{|E|}$$

Avec c_e une variable binaire définie par :

$$c_e = \begin{cases} 1 & \text{si l'arc } e \text{ a ses extrémités dans deux communautés} \\ 0 & \text{sinon} \end{cases}$$

4.4.1.3 Disponibilité du code source

Parmi les nombreuses méthodes et algorithmes proposés dans l'état de l'art de Fortunato, de nombreux auteurs proposent une implémentation « de référence » de leur algorithme ainsi que les jeux de données qu'ils ont utilisé pour leurs différentes publications. Afin de s'assurer des biais provenant d'une mauvaise implémentation, nous nous efforcerons d'utiliser les logiciels proposés, et ce d'autant plus qu'à quelques exceptions près, tous sont proposés sous une licence libre et sont portables. Les méthodes et algorithmes ne proposant pas d'implémentation disponible seront donc au maximum évitées.

4.4.2 Méthodes retenues

Les différentes sections précédentes nous ont permis d'établir une relation empirique à partir des données de Fortunato entre complexité et taille des instances, présentée dans le tableau 4.1. Nous avons ensuite proposé une liste classée par méthode et par complexité des différents algorithmes proposés et dont la complexité est/ou le code source sont connus, présentée dans le tableau 4.2, ce qui nous permet de construire un nouveau tableau 4.3 qui expose les algorithmes pour lesquels une implémentation est proposée, et dont pour la plupart nous connaissons la complexité. Cette liste regroupe également les méthodes pour lesquelles nous n'avons pas de complexité, mais leur présentation lors dans l'état de l'art de Fortunato indique des performances souvent correctes ; surtout, l'implémentation est disponible. Les domaines et les approches couverts par cette liste sont assez diversifiés, nous allons donc exploiter toutes ces approches dans notre comparaison.

Comme indiqué dans la section 4.4.1, cette étude poursuit deux objectifs, les différents algorithmes à comparer vont donc être séparés suivant ces objectifs suivant leurs caractéristiques. Le choix est effectué en fonction de la complexité, après évaluation des temps de calcul sur une méthode de complexité « moyenne » (Pons et Latapy) dont les instances ont un temps de calcul qui devient suffisamment important :

– Adéquation de l'arborescence :

1. Blondel et al. [27], une approche gloutonne et rapide ;
2. Clauset, Moore et Newman [50] permet la détection hiérarchique ;
3. Hofman et Wiggins [77] approche statistique ;
4. Pons et Latapy [137] approche hiérarchique avec un marcheur aléatoire.
5. Radicchi et al. [139], utilisation de la notion de cohésion interne/externe ;

Auteur(s)	Réf.	Complexité
Blondel et al. (2008)	[27]	$O(m)$
Clauset, Newman et Moore (2004)	[52]	$O(md \times log(n))$
Hofman et Wiggins (2008)	[77]	$O(n^{\alpha}), \alpha \approx 1.44$
Pons et Latapy (2005)	[137]	$O(n^2 d)$
Radicchi et al. (2004)	[139]	$O(\frac{m^4}{n^2})$
Dongen (2000)	[58]	$O(n^3)$
Zhang, Wang et Zhang (2007)	[199]	$O(K^2 n + Km)$, $O(n^2 n_c h)$
Clauset, Moore et Newman (2008)	[50]	$> O(n^2)$
Schuetz et Caflisch (2008)	[159]	
Agarwal et Kempe (2008)	[1]	
Rosvall, Axelsson et Bergstrom (2009)	[155]	
Palla et al. (2005)	[133]	

Table 4.3 – Liste des algorithmes proposant une implémentation

- Communautés contraintes : les approches par partitionnement de graphe, présentées dans en 4.2.2.1.1 ainsi que le regroupement par partitionnement présenté en 4.2.2.1.3 permettent de contraindre la taille et/ou le nombre des communautés.
  - Zhang, Wang et Zhang [199] permet de la détection de communautés recouvrantes, en fixant le nombre de clusters à l'avance ;
  - La méthode proposée par Hofman et Wiggins permettant de définir des valeurs de `K` lors de l'exécution, nous allons également retenir les résultats pour ce besoin.

Cependant, étant donné que peu d'algorithmes proposent une implémentation avec la possibilité de fixer une taille ou un nombre de communautés, nous observerons également les résultats des autres méthodes sur les critères de cet objectif.

4.4.3 Processus de comparaison

Nous allons effectuer la comparaison sur un sous-ensemble des graphes disponibles, pour nos deux instances (**defconfig** et **allyesconfig**). L'étude de ceux-ci ayant permis de documenter l'évolution temporelle, nous allons nous limiter à appliquer la comparaison sur les versions 3.0 et 3.9. La démarche du reste de l'étude sera la suivante :

- Traduction du graphe dans le format attendu par l'outil d'analyse ;
- Exécution de l'outil d'analyse, suivi du temps de calcul et de la consommation mémoire ;
- Traduction inverse du graphe ;
- Calcul du critère suivant l'objectif choisi.

Ce processus est détaillé dans la figure 4.1 : cela permet d'illustrer l'enchaînement des différentes étapes et de visualiser les opérations effectuées.

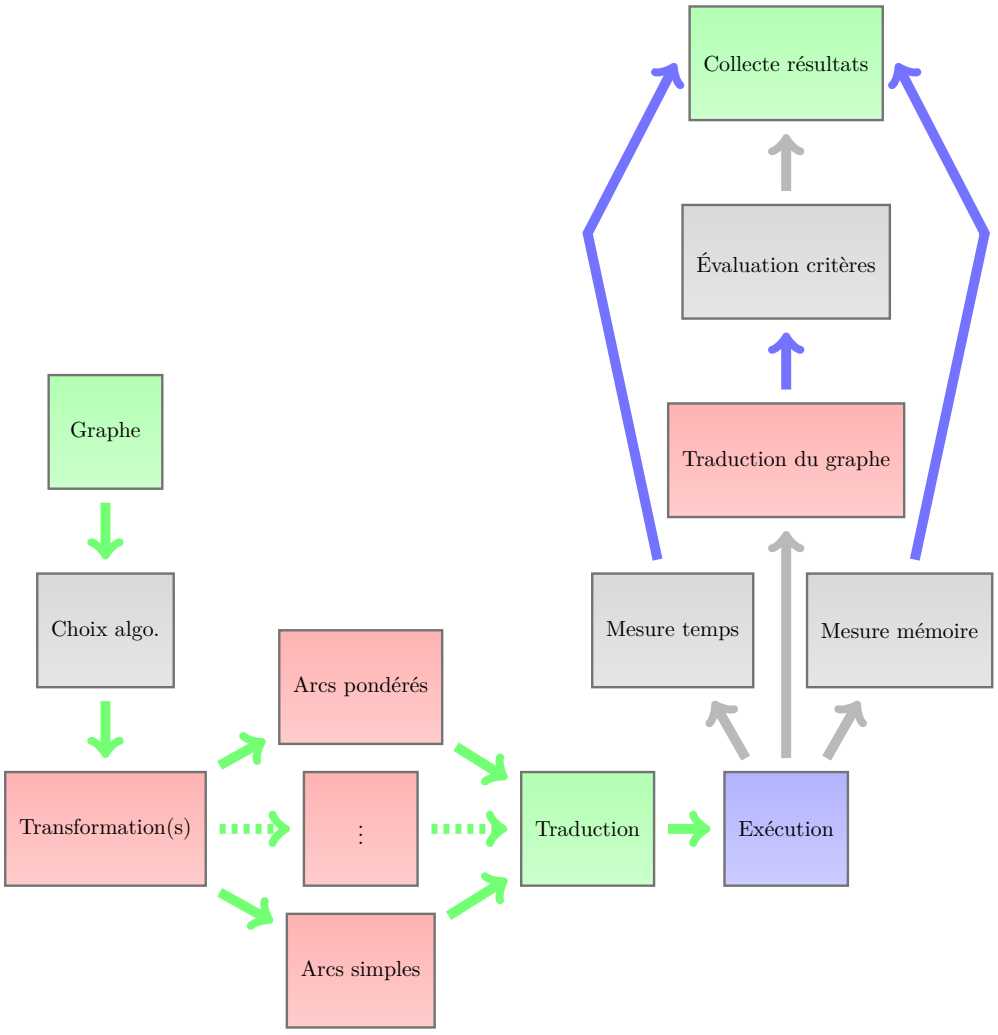


FIGURE 4.1 – Processus de comparaison des différentes méthodes de détection de communautés

4.5 Résultats et analyse

Nous avons proposé une méthodologie pour évaluer les performances de différents algorithmes et méthodes de détection de communauté proposés dans l'état de l'art et qui paraissaient pertinents pour notre cas. Les résultats bruts et détaillés sont proposés dans l'annexe B, et nous présentons dans cette section un résumé des points importants.

4.5.1 Blondel et al. (2008)

La profondeur de l'analyse est contrôlable dans l'implémentation de cette approche hiérarchique, et un mode automatique est proposé. Les niveaux ainsi détectés varient entre 3 et 4 sans avoir d'impact notable sur les résultats, mais l'utilisation de niveau moindre montre des changements. L'étude des tailles et de la concentration des sous-répertoires est illustrée par le graphique B.1a. Les communautés détectées sont de taille variant entre 140 et 450 et le taux de concentration des sous-répertoires évolue entre 0.17 et 0.13 ; l'instance **allyesconfig** présente les meilleurs résultats. La mesure du nombre de communautés détectées, proposée dans le graphique B.2, présente des valeurs cohérentes avec le premier niveau d'arborescence du système de fichier (≈ 13 pour **defconfig** et ≈ 22 pour **allyesconfig**), certaines communautés appartiennent à plusieurs sous-répertoires. C'est également sur cette mesure que nous constatons l'impact du paramètre de l'implémentation, les niveaux de profondeur 2 et 1 détectant respectivement 36 et 360 communautés. Si le niveau 2 ne présente *a priori* pas de lien direct avec la structure du système de fichier, le niveau 1 correspond à un niveau de profondeur supplémentaire dans le système de fichier. Les interconnexions sont assez nombreuses, la mesure évoluant entre 0.52 et 0.60, comme présenté en figure B.3. Les graphiques dans la figure B.4 documentent le temps de calcul et la consommation mémoire : la méthode étant la moins complexe, elle est efficace, et le temps de calcul est de moins d'une seconde pour une consommation de 75MiO, dans les pires cas.

4.5.2 Clauset, Newman et Moore (2004)

L'implémentation n'admet pas de paramètre permettant de contrôler son exécution. La taille des communautés qui sont détectées, visibles dans le graphique B.5a varie en moyenne entre 30 et 60 mais des écarts importants dans les données existent. Le taux de concentration documenté dans le graphique B.5b indique une grande mixité (variant entre 0.80 et 0.825 suivant l'instance). Nous ne recouvrons pas une structure de communautés correspondant au système de fichier. Le nombre de communautés détectées est par contre parfaitement stable, comme documenté dans le graphique B.6 : les variations que nous avons observées précédemment se font donc par déplacement de nœuds entre communautés. Les interconnexions sont, contrairement à la méthode précédente, plus intéressante : le graphique B.7 montre que cette mesure évolue autour de 0.39 à la fois pour **defconfig** et **allyesconfig**. L'utilisation des ressources est raisonnable, la mémoire étant utilisée au maximum à 75MiO comme dans la méthode précédente ; le temps de calcul lui est plus important et passe à une centaine de secondes dans le pire cas (**allyesconfig**) en restant à moins d'une seconde pour les cas favorables (**defconfig**).

4.5.3 Hofman et Wiggins (2008)

Cette implémentation admet trois paramètres contrôlant son exécution (nombre de communautés, nombre de redémarrages, tolérance à la variation). Après diverses évaluations nous les avons fixés respectivement : variation entre 10 et 900 par pas de 10 pour le nombre de communautés, 2 redémarrages et tolérance par défaut de 0.01. Les 256 itérations que nous effectuons nous permettent d'évaluer la variabilité quant aux redémarrages. La méthode détecte des communautés composées de 2 nœuds dans le cas **defconfig** et 11 dans le cas **allyesconfig**, comme indiqué dans le graphique B.9a. Le taux de concentration documenté dans le graphique B.9b est très élevé, au-delà de 0.98, ce qui indique que les communautés détectées ne correspondent pas du tout à des structures proches de l'arborescence du système de fichier. Les tailles des communautés détectées sont intéressantes pour l'approche de découpage du noyau en sous-ensembles. Le nombre de communautés (graphique B.10) varie entre 790 (**defconfig**) et 900 (**allyesconfig**). Les arcs entre communautés sont très nombreux, le taux lisible dans le graphique B.11 étant au-delà de 0.99. La méthode montre sa complexité avec un temps de calcul de plusieurs heures pour **allyesconfig** et plus de 4GiO de mémoire consommée (graphiques B.12a et B.12b).

4.5.4 Pons et Latapy (2005)

Cette implémentation de détection de communauté est paramétrable par la profondeur du chemin aléatoire qu'il est possible de parcourir : la valeur par défaut est de 4, nous l'avons fait varier entre 1 et 256. L'influence de ce paramètre est très nettement visible. Le premier graphique B.13a documente la taille des communautés, et nous observons une évolution passant par une croissance, puis un plateau et enfin une baisse. Un phénomène similaire s'observe sur les autres graphiques, notamment celui du taux de concentration (B.13b). La combinaison de ces deux paramètres permet de délimiter un intervalle de valeurs du chemin aléatoire pertinent : entre 60 et 80, nous observons des tailles de communautés et des taux de concentration pertinents, en rapport avec la structure du système de fichier. Les courbes des instances **defconfig** et **allyesconfig** se croisent également sur cet intervalle dans le graphique B.14. Il en va de même pour la taille des intercommunautés documentée dans le graphique B.15, où ce même intervalle de 60 à 80 correspond à des minimums atteints. La consommation mémoire n'est pas influencée par ce paramètre et reste constante, alors que le temps de calcul lui est situé entre 500 et 1 000 secondes pour cet intervalle pour **allyesconfig**. Cette méthode semble donc capable de recouvrer une structure de communautés correspondante au système de fichier sous-jacent au moins sur un intervalle du paramètre compris entre 60 et 80. Des valeurs de paramètres plus proches de zéro donneraient des communautés adaptées à l'autre analyse ciblée proposée.

4.5.5 Radicchi et al. (2004)

Le seul paramètre contrôlant l'implémentation joue sur la méthode appliquée, 3^{e} ou 4^{e} ordre : seul le premier cas a été étudié, les temps de calcul sont trop importants. La taille des communautés (B.17a) évolue entre 6 nœuds pour **defconfig** et 17 pour **allyesconfig**. Le taux de concentration (B.17b) se situe autour de 0.80 ce qui indique une faible corrélation

avec le système de fichier. Le nombre de communautés, documenté dans le graphique B.18, est stable autour d'une centaine dans le cas **defconfig** et un millier pour **allyesconfig**. L'étude des interconnexions entre les communautés, dont le graphique est disponible en en figure B.19 porte un enseignement différent : dans le cas **defconfig**, celui-ci est plutôt faible (0.2) et devient plus important avec **allyesconfig** (0.6). La complexité de la méthode est visible avec les temps de calcul (14 000 secondes dans le cas **allyesconfig**) et la mémoire consommée (plus de 4GiO avec cette même instance).

4.5.6 Zhang, Wang et Zhang (2007)

Cette implémentation accepte trois paramètres contrôlant la construction des communautés, à la fois directement la taille à atteindre, et indirectement, la fréquence à laquelle un nœud sera retiré d'une communauté (par défaut, aucune suppression) ainsi que le seuil d'appartenance. Les résultats obtenus sont loin de nos espérances. Le graphique B.21a qui documente la taille des communautés détectées montre une moyenne de quelques centaines (**defconfig**) à quelques milliers (**allyesconfig**). Cette moyenne cache des disparités très fortes dans la constitution des communautés. Le taux de concentration que nous observons (graphique B.21b) est concentré autour de 0.2 mais est très variable, avec des pics à plus de 0.5. Une communauté de 47 nœuds présente ainsi un taux de 0.68, traduisant un regroupement de fichiers n'appartenant principalement pas au même sous-répertoire. Le nombre de communautés détectées est proposé dans la figure B.22 et varie peu, entre 2 et 5. Le paramètre contrôlant cette quantité s'est révélé sans influence dans nos essais, les communautés supplémentaires étant vides. Les interconnexions entre les communautés (B.23) sont également instables, variant entre 0.01 et 0.5, quelle que soit l'instance. Enfin les temps de calcul de cette méthode sont corrélés avec les résultats précédents, et visibles dans les graphiques B.24a et B.24b. La consommation mémoire, dans le second graphique, est stable et dépend du graphe traité. Elle varie entre moins de 200MiO pour **defconfig** à près de 5GiO pour **allyesconfig**. Le temps de calcul est très variable, et se situe soit autour de quelques centaines de secondes soit au-delà de 15 000 secondes.

4.6 Conclusion

Dans la section précédente, nous avons proposé une première analyse pour les résultats de chaque méthode ; nous allons maintenant revenir sur les besoins qui ont justifié l'étude de ces différentes approches, et faire le point sur leur efficacité éventuelle. Les différents objectifs ont été présentés dans la section 4.4.1 : vérification de l'adéquation de l'arborescence des sources quant à l'utilisation effective du code source ; découpage du noyau en sous-ensembles de taille définie, contrôlable et permettant d'en qualifier des liens. Dans une première sous-section 4.6.1, nous commenterons les résultats quant à la détection de l'arborescence du système, puis, dans une seconde sous-section 4.6.2 nous faisons la synthèse quant au problème découpage contrôlé.

4.6.1 Analyse de l'arborescence

Cet axe a été défini dans la section 4.4.1.1, avec la mesure nous permettant d'évaluer les méthodes : selon sa définition, une méthode qui nous indiquera une bonne correspondance entre les communautés détectées et la structure du système de fichier aura tendance à minimiser la valeur de ce critère.

La première méthode Blondel et al. (2008) que nous avons évaluée propose un bon résultat allant dans ce sens : le taux de concentration est assez faible, entre 0.13 et 0.17. Le nombre de communautés détectées correspond au premier niveau d'arborescence cohérent : avec **defconfig**, de l'ordre de 14 ; avec **allyesconfig**, de l'ordre de 23. Les temps de calculs sont très intéressants, puisque la plus grosse instance est traitée en moins d'une seconde, et en consommant assez peu de mémoire.

Avec l'approche suivante, Clauset, Newman et Moore (2004), nous constatons cette fois un taux de concentration particulièrement plus élevé, situé entre 0.80 et 0.825. Aucune différence notable n'est observée entre les instances **defconfig** et **allyesconfig**. Le nombre de communautés qui sont détectées est par contre très stable et implique une relative petitesse de celles-ci : environ 27 nœuds pour **defconfig**, et 49 pour **allyesconfig**. Les temps de calculs sont raisonnables : de quelques secondes dans le premier cas, à une centaine de secondes dans le second.

La méthode Hofman et Wiggins (2008) présente des communautés très petites (entre 2 et 3 nœuds sur **defconfig**, un peu moins de 11 en moyenne sur **allyesconfig**) et le taux de concentration est particulièrement élevé, au-delà de 0.98. Cette approche détecte de nombreuses communautés petites, mais qui ne correspondent pas à l'arborescence du système de fichier. De plus, les temps de calculs sont très élevés, 2 000 secondes pour les instances **defconfig** et 12 000 secondes pour **allyesconfig**.

Avec Pons et Latapy (2005) les résultats obtenus montrent un comportement intéressant, puisque suivant la profondeur autorisée, nous sommes en mesure de recouvrir des communautés de tailles diverses. Surtout, le taux de concentration obtenus sur cette approche et tant avec des instances **defconfig** que **allyesconfig** atteint des valeurs faibles, indiquant une bonne concentration de nœuds d'un même sous-répertoires. Les temps de calcul, particulièrement pour les graphes importants, sont élevés mais restent acceptables pour la profondeur de recherche donnant des résultats intéressants, en étant autour de 500 à 600 secondes.

La dernière méthode que nous avons évaluée pour ce besoin est Radicchi et al. (2004). Le taux de concentration de celle-ci est élevé, autour de 0.80 quel que soit le cas considéré (**defconfig** ou **allyesconfig**). Les communautés détectées sont nombreuses et petites (6 nœuds pour le premier cas, 17 pour le second). Ces valeurs n'indiquent clairement pas la détection de l'arborescence du système de fichier. Les temps de calculs sont acceptables pour le cas **defconfig**, mais explosent à plus de 14 000 secondes pour **allyesconfig** ; la mémoire consommée est également importante.

4.6.1.1 Synthèse

Pour ce premier axe d'analyse, nous pouvons conclure qu'une partie des méthodes proposées est en mesure de recouvrer une arborescence proche de celle du système de fichier d'origine, à des niveaux de profondeur différents : Blondel et al. (2008) et Pons et Latapy (2005).

4.6.2 Communautés contraintes

Cet axe a été défini dans la section 4.4.1.2, et les critères principaux pour l'analyse sont la capacité à construire des communautés d'une taille donnée, en minimisant l'interface entre celles-ci. La seule méthode étudiée exclusivement pour cet objectif est Zhang, Wang et Zhang (2007). Ses résultats ne sont pas du tout à la hauteur de nos attentes, puisqu'elle n'a pas été en mesure de construire des communautés d'une taille donnée.

Certaines des méthodes que nous avons utilisées pour l'analyse de l'arborescence se prêtent mieux à cette décomposition. Elles ne permettent pas de contrôler la taille des communautés, mais les résultats présentés sont plus exploitables. Par exemple, celles qui sont créées par Clauset, Newman et Moore (2004) varient entre 27 et 49 nœuds. L'utilisation de Hofman et Wiggins (2008) permet de construire des communautés d'une taille plus petite encore, de l'ordre de 3 à 11 nœuds suivant les instances. Enfin Radicchi et al. (2004) construit également des communautés du même ordre de grandeur, variant d'environ 5 nœuds avec **defconfig** à environ 17 avec **allyesconfig**. Pour ces trois méthodes, nous rappelons que le taux de concentration est supérieur à 0.80 ; cela indique que les communautés qui sont créées ne contiennent que peu de nœuds appartenant au même sous-répertoire.

Les intercommunautés des résultats de Clauset, Newman et Moore (2004) donnent un taux de l'ordre de 0.39, ce qui est intéressant dans l'optique de cette analyse. Au contraire, la méthode Hofman et Wiggins (2008) expose sur ce critère une valeur très forte, au-delà de 0.99. Radicchi et al. (2004) donne des résultats un peu plus intéressants sur ce critère, la valeur étant autour de 0.6 dans le pire des cas, **allyesconfig**, et 0.2 avec **defconfig**.

4.6.2.1 Synthèse

Pour ce second axe d'analyse, la méthode que nous espérions exploiter se révèle contre-performante. Par contre l'étude des résultats des méthodes précédentes fait ressortir que nous pourrions faire usage de Clauset, Newman et Moore (2004) et Radicchi et al. (2004) pour ce type de besoins, leurs résultats étant plus proches de nos contraintes.

Chapitre 5

Application et intégration

5.1 Introduction

5.1.1 Mageia et Mandriva

La distribution Mageia est une version dérivée de Mandriva, dont le point de départ se situe au cours de l'année 2010. Plusieurs aspects les différencient, à la fois sur le plan technique mais également quant à l'organisation et les objectifs. Mageia a fait le choix de la continuité technologique dans son évolution alors que Mandriva a mis à profit cette divergence pour introduire de nouveaux outils. Une différence majeure reste le gestionnaire de paquets, Mandriva ayant fait le choix d'utiliser RPM5. Cette base est utilisée par les équipes brésiliennes pour construire un système d'exploitation de bureau vendu aux constructeurs d'ordinateurs. Pour son offre serveur, Mandriva a fait le choix au contraire de se baser sur la distribution Mageia dont la gouvernance est entre les mains de sa communauté alors que celle produite par le Brésil est entièrement entre les mains de la société Mandriva.

5.1.2 Objectifs

Nous présentons dans ce chapitre la démarche permettant de répondre au besoin exprimé au début de ces travaux : intégrer de la vérification de code permettant de garantir des propriétés à destination des utilisateurs dans la distribution. Les résultats qui ont été obtenus sont documentés. Une analyse des besoins est proposée dans la section 5.2.1 pour documenter l'état général de la distribution Mageia : quels sont les paquets les plus importants, quels sont les langages utilisés et comment sont-ils utilisés, quelle est l'ampleur des modifications apportées par la distribution. Dans une seconde section 5.2.2 nous présentons une étude comparative du taux de fautes détectées par un outil d'analyse (Undertaker) sur différentes variantes de noyaux : version originale proposée par les développeurs, version mise à disposition par certaines distributions (Debian, Mandriva, OpenSUSE). Par la suite dans la section 5.3 un premier prototype proposant l'intégration de différents outils d'analyse dans le système de construction de la distribution est étudié. Ces outils utilisables sont présentés dans la section 5.4.

5.2 Analyse des besoins

Dans cette première section, nous allons détailler l'étude préliminaire proposée en 1.3 : il s'agit de documenter le comportement des différentes versions de Mageia, 1, 3 et 4. L'objectif est d'avoir une documentation de l'état actuel et de l'évolution de la distribution, pour définir les besoins des outils que nous proposerons par la suite. La version 2 n'a pas été traitée par manque de ressources fiables et facilement disponibles : obtenir certaines informations détaillées (mainteneurs des paquets par exemple) était trivial avec la version 1, et possible sur les versions 3 et 4, suite à des changements organisationnels.

5.2.1 Étude des paquets Mageia

La compréhension de la base de code va nous permettre, par la suite, d'orienter notre étude pour effectuer efficacement l'analyse de code dans la distribution. La gestion des dépendances entre les paquets n'est pas de notre ressort, nous ne nous y intéresserons donc pas : il s'agit de problèmes différents. Seul le code source nous intéresse, ce qui n'exclut pas que nous ayons besoin des sources des paquets : certains logiciels sont modifiés par la distribution, et nous voulons que ces modifications soient appliquées de manière à obtenir le code source exactement utilisé lors de la construction des paquets. Avoir des éléments de compréhension sur le code source utilisé permet également de faciliter et d'assainir le travail de maintenance.

Les objectifs de cette étude sont donc :

1. Connaître le volume d'utilisation des langages au sein de la distribution, ce qui nous permettra par la suite d'avoir une estimation de l'intérêt d'un outil d'analyse suivant les langages pris en charge ;
2. Connaître la quantité de code présente pour chaque langage, pour être capable de cerner les besoins en matière de volume de données à traiter pour effectuer l'analyse ;
3. Documenter la complexité de maintenance, pour être capable d'identifier les points faibles et limiter les coûts de maintenance ;

Les informations sur les langages sont à la fois utiles pour les mainteneurs, puisque cela leur permet d'avoir une idée des compétences nécessaires. Cela peut être utile également pour sélectionner des outils d'analyse, qui sont souvent dépendant du langage : plus un outil sera utilisable sur une base de code grande, plus il sera facilement « rentable ». Nous chercherons donc dans la distribution un ou des paquets qui auraient les caractéristiques suivantes :

- Important, voire critique, ce qui permet de donner une forte valeur ajoutée à sa validation ;
- Dont le code suit de bons standards de codage, pour éviter des biais liés à une piètre qualité de code ;
- Une base de code suffisamment grande, pour nous « assurer » de pouvoir y trouver des problèmes ;
- Maintenu correctement, pour avoir la possibilité d'échanger avec ses développeurs en cas de soucis ;

Cette analyse sera menée sur différentes versions de la distribution Mageia : la première version correspond à la base de réflexion et de choix pour nos travaux, la suite de cette étude sur la distribution vise un objectif un peu différent puisqu'il s'agit maintenant de documenter l'évolution des mesures qui avaient été effectuées. Il est important de garder à l'esprit que la version de Mageia 4 étudiée était très mouvante, et malgré des mises à jour fréquentes des données, celle-ci peuvent rapidement diverger de l'état de la distribution finale.

5.2.1.1 Mesures retenues

Pour répondre aux objectifs préalablement définis en vue de pouvoir sélectionner des paquets pertinents et de définir la suite de l'étude, nous proposons les définitions des mesures que nous effectuons sur la distribution.

5.2.1.1.1 Quantité de correctifs Pour chacun des paquets de la distribution, nous allons simplement compter le nombre de correctifs, c'est-à-dire, le nombre de fichiers contenant des différences à appliquer lors de la construction du paquet. La maintenance d'un grand nombre de correctifs est lourde et coûteuse.

5.2.1.1.2 Quantité de fichiers modifiés Nous évaluons également, pour chaque paquet, le nombre de fichiers dans les sources du logiciel original qui sont touchés par les différents correctifs. En général, plus un correctif touche de fichiers différents, plus la maintenance devient complexe et consommatrice de ressources.

5.2.1.1.3 Quantité de modifications Un autre facteur de complexité pour la maintenance concerne la taille des modifications à appliquer. Les correctifs contiennent une série de fichiers sources à modifier, et pour chacun, propose les lignes à supprimer et celles à ajouter. Nous allons donc compter, et agréger au niveau du paquet, la quantité de modification pour chacun en séparant les insertions et les suppressions de code.

5.2.1.1.4 Effort de maintenance Comme évoqué précédemment, la quantité de changements a un impact sur la maintenance des paquets. Chacun peut avoir un nombre de mainteneurs différent, en général 0 ou 1. Nous proposons d'estimer l'effort E_P nécessaire pour faire effectuer la maintenance d'un paquet P comme le rapport entre la quantité de correctifs de ce paquet, et le nombre de mainteneurs :

$$E_P = \frac{|correctifs(P)|}{|maint(P)| + 1}$$

Avec $maint(P)$ l'ensemble des mainteneurs du paquet P, et $correctifs(P)$ l'ensemble des correctifs qui sont appliqués pour P. Dans l'hypothèse où un paquet n'a pas de mainteneur, nous ajoutons 1 pour ne pas diviser par zéro.

Cette manière de mesurer présente des biais, notamment, nous faisons deux hypothèses assez fortes : d'abord, tous les correctifs se valent en matière de complexité, et également

tous les mainteneurs sont équivalents. Modéliser plus finement serait possible, notamment en exploitant les résultats des autres mesures sur les correctifs. Cependant, cela nécessitera d'étudier plus précisément et en détail la complexité des correctifs. D'autre part, la quantité est un indicateur intéressant mais c'est jamais parfait, il n'est pas rare qu'un correctif de deux lignes soit complexe parce qu'il interagit avec beaucoup d'autres composants, par exemple.

Connaître le nombre de mainteneurs sur un paquet et pour une version de la distribution n'est pas trivial : les outils du système de construction ont une liste des mainteneurs, mais celle-ci n'a pas d'historique. Il n'est donc pas possible à partir de cette liste de remonter dans le temps. Pour répondre à nos besoins, nous nous sommes donc appuyés sur le dépôt Subversion du projet Mageia : un mainteneur est un utilisateur qui aura effectué des modifications sur les sources du paquet. En exploitant les outils de requêtage Subversion pour obtenir la liste des modifications sur l'intervalle de dates situé entre deux versions majeurs, nous sommes en mesure d'avoir cette liste de manière plus ou moins fidèle. Les robots de gestion (deux, `umeabot` et `schedbot`) sont exclus automatiquement.

5.2.1.1.5 Volume de code par langage Cette mesure s'appuie sur les résultats fournis par l'outil SLOCCount. Le volume de code pour chacun des langages est agrégé sur la totalité de la distribution, ce qui permet de montrer l'importance de chacun. Il faut garder à l'esprit que les lignes de codes ne sont pas égales, chaque langage ayant sa propre sémantique. Le volume permet par contre de se faire une assez bonne idée de la quantité de travail nécessaire pour la maintenance, et surtout, l'impact potentiel d'outils d'analyse de code qui sont limités à certains langages.

5.2.1.1.6 Utilisation des langages par les paquets En exploitant d'une autre manière les données fournies par SLOCCount, nous effectuons un comptage de l'utilisation des langages par chaque paquet. Le but est de montrer les langages qui sont les plus utilisés, sans considération du volume : la maintenance des paquets en est affectée. Comme nous travaillons au niveau des sources, ce comptage inclut également les langages utilisés dans les différents outils de construction des logiciels : par exemple, les projets utilisant les Autotools reposent massivement sur `sh`. Le système de construction faisant partie du logiciel, et étant également sujet à modifications (correctifs, intégration), cela est acceptable.

5.2.1.2 Quantité de code et langages utilisés

Le premier résultat est proposé dans la figure 5.1, et correspond à la mesure explicitée dans le paragraphe 5.2.1.1.5. Environ la moitié des langages présents dans la distribution a un volume de code supérieur à 1 000 000 lignes, et seuls 5 langages, sur un total de 5, dépassent 10 000 000 de lignes de code : `C`, `C++`, `Shell`, `Java` et `Perl`. Le langage `Python` est quasiment à égalité avec ce dernier, mais légèrement en dessous.

Si ce premier ensemble des langages les plus présents en matière de volume de code n'est pas particulièrement surprenant, la suite des données l'est un peu plus. À la suite quasi-immédiate de `Python`, nous observons la présence de `C#`, puis de l'assembleur, suivi

par PHP. Si le volume de ce dernier n'est pas complètement surprenant, environ 5 000 000 lignes de code, la présence de C# et d'assembleur dans des quantités supérieurs (autour de 6 000 000 à 7 500 000 lignes) peut surprendre :

- Pour C#, cela reste un langage plutôt orienté pour la plateforme Microsoft .Net, et même s'il est standardisé ECMA le monde du logiciel libre ne lui est pas complètement acquis. Le voir ainsi avec un volume de code plus important que PHP montre une présence finalement non négligeable.
- Pour l'assembleur, c'est un langage qui est souvent considéré comme peu utilisé, parce que complexe (étant proche de la machine), et loin des besoins actuels (productivité, maintenance). Nous visualisons ainsi qu'il est finalement encore plutôt utilisé, même si cela reste limité à des cas bien particuliers.

Il convient de noter un biais sur les valeurs absolues : certains paquets sources sont identifiés comme différents (parce que leur nom est différent), mais partagent un code source très proche s'il n'est pas commun. C'est le cas, notamment, de différentes variantes de noyau : `kernel`, `kernel-xen`, `kernel-linus`, `kernel-tmb`, `kernel-vserver`. Cela conduit donc, en partie, à une inflation de la l'importance de certains langages. Cependant, ces codes mêmes s'ils sont proches, ne sont que très rarement identiques.

Le code écrit en C# est réparti principalement dans les paquets suivants :

- Moonlight, une implémentation libre de Silverlight qui permet une certaine interopérabilité. Cet outil représente à lui seul une base de 2.6M de lignes de code ;
- L'interpréteur Mono, utilisé pour exécuter les programmes, et qui présente un volume similaire à celui de Moonlight ;
- Monodevelop, un environnement de développement intégré adapté à mono, qui représente un peu plus de 700 000 lignes de code ;
- Du code liant KDE 4 à Mono compose le paquet `kdebindings4` et contient de l'ordre de 160 000 lignes de code ;
- La traîne est ensuite composée à la fois d'autres bibliothèques de liaison (connecteur MySQL, `libgoogle`, etc.) et de logiciels écrits en Mono (F-Spot [1], Banshee [2], etc.).

Pour l'assembleur, les utilisateurs se répartissent ainsi :

- Gromacs, un outil de simulation du comportement des protéines, lipides, etc., avec presque 800 000 lignes de code assembleur ;
- Wine, le logiciel permettant l'utilisation des programmes Windows sous Linux est également un gros utilisateur d'assembleur, mais moins que Gromacs avec 500 000 lignes de code ;
- Le navigateur Chromium arrive ensuite avec plusieurs variantes (stables, non stables, etc.) situées entre 350 000 et 260 000 lignes d'assembleur. Il est intéressant de noter que les versions stables, donc plus anciennes, contiennent plus de code assembleur que les futures versions : une factorisation est probablement en cours ;
- Les outils GNU binutils contiennent également une bonne part d'assembleur, variant entre 340 000 lignes, et 320 000 lignes dans le cas de la version MingW32 destinée à permettre la compilation croisée à destination de Windows ;
- Finalement, les différentes versions du noyau Linux ainsi que la GLibC arrivent

1. Gestionnaire de bibliothèque photographique
2. Gestionnaire de bibliothèque audio et lecteur

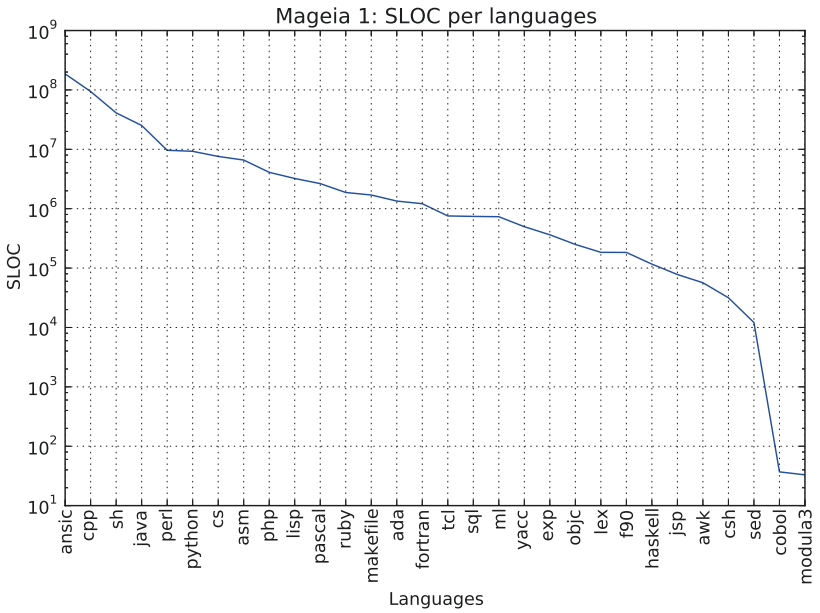


FIGURE 5.1 – Volume de code suivant les langages de programmation utilisés dans les paquets de Mageia, version 1

bien après, avec respectivement un volume de code assembleur compris autour de 230 000 et 175 000 lignes.

Enfin, par curiosité, nous regardons la population des utilisateurs de code `C`, dont environ une trentaine présente un volume de code dépassant le million de lignes de `C` :

- Le noyau LINUX, dans ses différentes version, est majoritaire en matière de quantité de code `C`, entre l'instance XEN pourvue d'un peu plus de 7 000 000 lignes de code, et les instances `kernel-linus`, `kernel-vserver` et `kernel-tmb` qui varient entre 9 300 000 et 9 150 000 lignes de code ;
- WINE est le suivant, avec un peu plus de $4.6M$ lignes dans sa version embarquant le moteur GECKO, et environ $2.2M$ sans ;
- Le lecteur multimédia XBMC contient aussi plus de 2 300 000 lignes de code ;
- Les différentes instances de CHROMIUM sont également situés dans un intervalle compris entre 2 100 000 et 2 500 000 lignes de code `C` ;
- GCC et WIRESHARK sont constitués d'environ 2 000 000 lignes de codes chacun.

La figure 5.2 illustre l'utilisation de chaque langage par les paquets, indépendamment de la quantité de code. Alors que dans la figure 5.1, qui dépend de la quantité, le classement des 10 premiers langages était : `C`, `C++`, `Shell`, `Java`, `Perl`, `Python`, `C#`, `Assembleur`, `PHP`, `Lisp` ; en matière de nombre d'utilisateurs, il devient : `Perl`, `Shell`, `C`, `Make`, `C++`, `Python`, `PHP`, `Java`, `Ruby`, `Yacc`. L'ordre change donc beaucoup, et cela nous permet de déduire :

- Les langages `C`, `C++`, `Shell`, `Perl`, `Java`, `Python` et `PHP` sont à la fois les plus utilisés et les plus importants en volume de code ;
- Certains langages sont peu utilisés en volume, mais très présent dans toute la distribution, comme SED qui est juste après YACC dans la figure 5.2 alors qu'il est quasiment dernier dans 5.1.

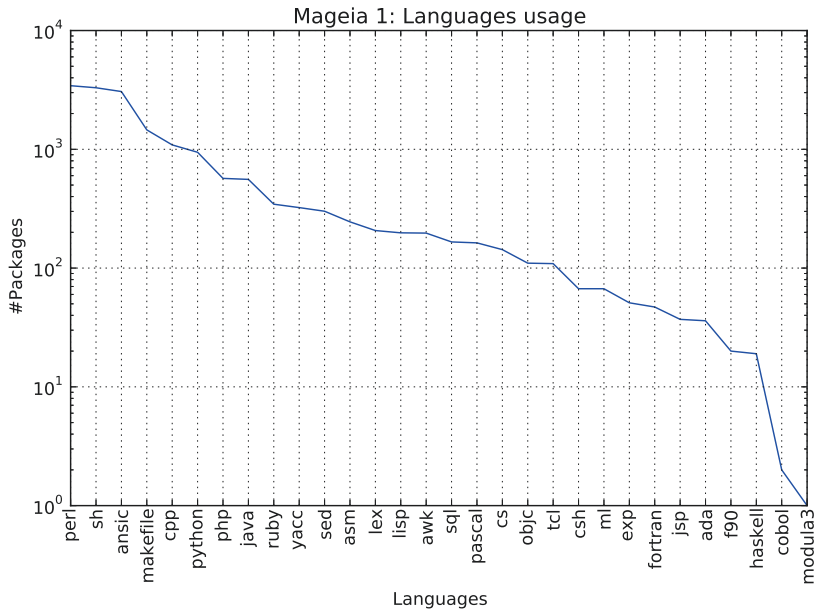
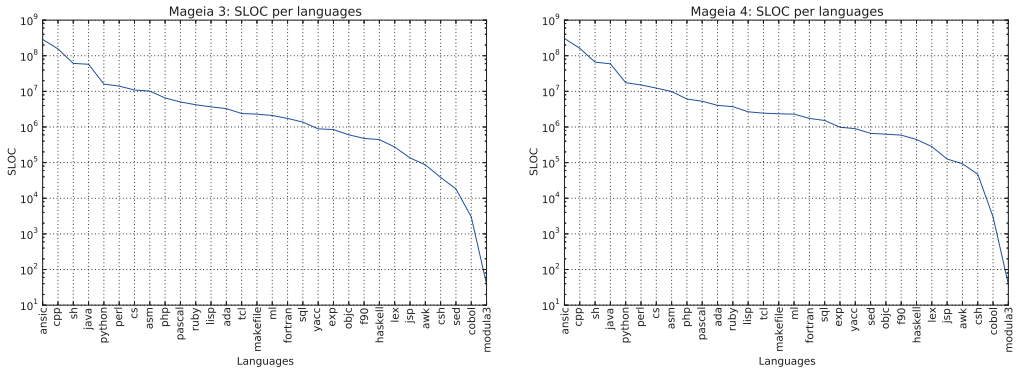


FIGURE 5.2 – Langages de programmation utilisés dans les paquets de Mageia, version 1

Enfin, COBOL et MODULA3 ont la particularité d'être particulièrement peu représentés dans les deux cas : respectivement, deux paquets (`colorer-take5` et `sloccount`) et un seul paquet (`a2ps`) sont utilisateurs.

La première série de figures que nous pouvons comparer concerne le nombre de lignes de code par langages dans les paquets. Les données pour Mageia 3 et 4 sont proposés respectivement dans les figures 5.3a et 5.3a ; celles pour Mageia 1 sont en figure 5.1.

La première constatation qu'il est possible de faire concerne l'ordre des langages : il évolue, mais à la marge. Quelques langages deviennent plus importants, et d'autres moins. C'est le cas, notamment, de PERL qui dans Mageia 1 était devant PYTHON, d'une courte avance. Le passage à Mageia 3 et 4 montre une inversion de cet ordre. Le langage LISP, qui était entre PHP et PASCAL est maintenant relégué derrière RUBY ; il en est de même pour ADA, MAKEFILE et TCL qui s'échangent leurs positions. Enfin l'ordre des langages pour cette première visualisation n'évolue pas pour les derniers ; par contre, la quantité de code COBOL fait un saut assez important, passant d'un ordre de grandeur de 10^1 lignes de code à 10^3. Cela s'explique d'abord par un doublement du nombre de paquets ayant du code dans ce langage, Mageia 1 en avait deux, Mageia 3 en a maintenant quatre. Surtout, parmi ces paquets, l'un est un module PYTHON pour la coloration syntaxique de code, et il contribue à hauteur de près de 3 000 lignes de code identifiées comme du COBOL. Quelques inversions sont visibles également entre Mageia 3 et Mageia 4, par exemple le volume de code ADA qui passe devant celui de RUBY. La quantité de code identifié comme le langage SED fait un bon assez surprenant entre ces deux versions, gagnant presque plus d'un ordre de grandeur en passant de 10^4 à presque 10^6 : d'abord, nous passons de 422 paquets identifiés comme contenant ce langage dans Mageia 3 à 430 dans Mageia 4. Ce bond s'explique en fait par l'arrivée dans cette nouvelle version de deux paquets, NCL et



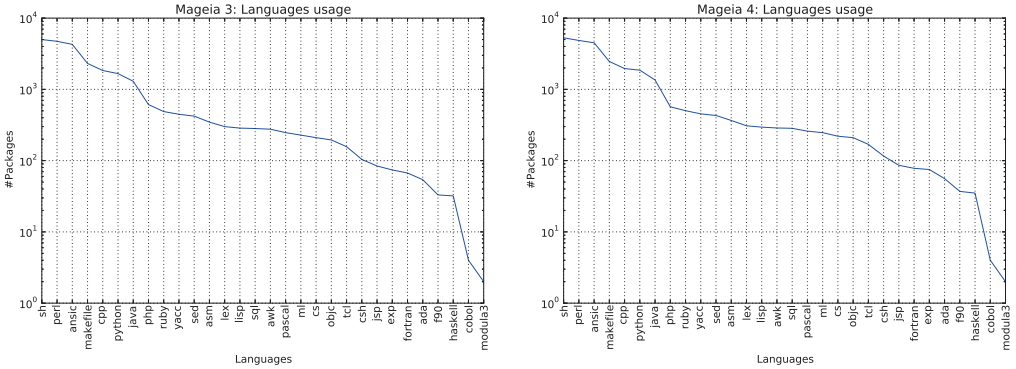
(a) Mageia 3 (b) Mageia 4

FIGURE 5.3 – Volume de code suivant les langages de programmation utilisés dans les paquets de Mageia, versions 3 et 4

STUFF qui contribuent respectivement un peu plus de 11 000 et 630 000 lignes de code SED.

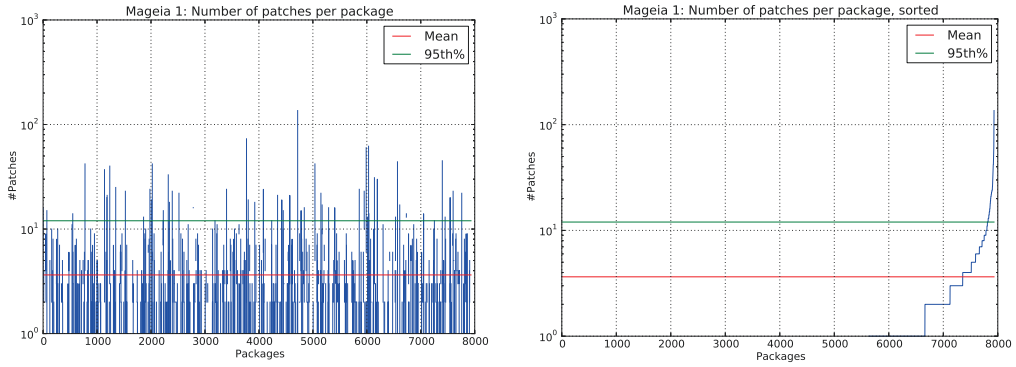
Ensuite, en comparant les échelles, nous pouvons constater une augmentation générale du volume de code dans la distribution. Cela s'explique en partie par l'inflation de paquets, leur nombre évoluant d'un peu plus de 7 200 dans Mageia 1 à plus de 10 000 dans Mageia 3, puis un peu plus de 11 200 dans Mageia 4. Et nous passons donc d'environ 400 000 000 lignes de code dans la première version à presque 650 000 000 dans Mageia 3, puis plus de 680 000 000 dans Mageia 4. Les taux d'évolution entre le nombre de paquets et la quantité de code source sont proches, et l'on peut affirmer qu'ils se corrèlent plutôt bien : ce taux est d'environ 1.49 pour les paquets entre Mageia 1 et 3, et est à 1.62 pour le volume de code source ; attention cependant, ces données incluent bien sûr Mageia 2.

Nous pouvons ensuite comparer l'évolution de l'utilisation des langages par les paquets, documenté pour Mageia 3 et 4 dans les figures respectives 5.4a et 5.4b, et pour Mageia 1 dans 5.2. La perte d'influence de PERL se confirme : celui-ci était le premier langage en terme d'utilisateur, et il cède sa place à partir de Mageia 3 pour le langage SH. La différence entre ces deux compétiteurs est plutôt ténue, ce qui n'est pas le cas de l'inversion qui se produit entre JAVA et PHP : la version 3 de Mageia montre une augmentation non négligeable du nombre d'utilisateurs du premier ; le nombre de paquets double, passant de 559 à 1299. L'étude des différences entre ces deux versions ne met en lumière aucun travail particulièrement important de redécoupage des paquets qui pourrait expliquer trivialement ce genre d'augmentation importante ; il s'agit donc du résultat de l'importation et de l'amélioration massive de la disponibilité d'outils de l'écosystème JAVA dans Mageia. Une évolution similaire est visible pour PYTHON. L'écosystème de la distribution évolue, c'est le résultat d'un mélange du choix des contributeurs, et des besoins auxquels il est nécessaire de répondre. Par exemple, une partie de l'augmentation de la base de code PYTHON est lié au support simultané des versions 2 et 3 du langage pour certains modules.



(a) Mageia 3 (b) Mageia 4

FIGURE 5.4 – Langages de programmation utilisés dans les paquets de Mageia, versions 3 et 4



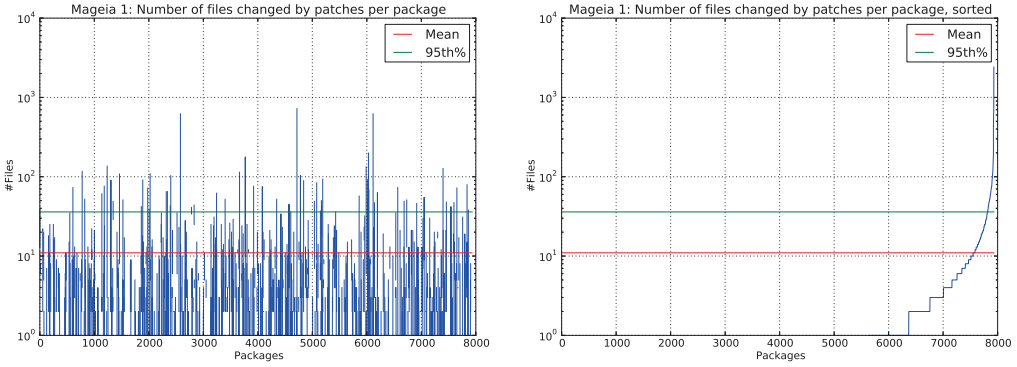
(a) Par paquet (b) Par taille

FIGURE 5.5 – Volume de correctifs dans les paquets de Mageia, version 1

5.2.1.3 Quantité de correctifs et modifications

Nous allons maintenant étudier quantitativement les modifications qui sont appliquées lors de la construction des paquets. D'abord, regardons simplement la quantité de correctifs qui sont appliqués, c'est-à-dire, le nombre de fichiers qui contiennent des changements. Les données sont proposées dans la figure 5.5a en utilisant comme abscisse les identifiants des différents paquets, et dans la figure 5.5b où le tri est effectué par taille de paquet. La seconde visualisation permet de se faire une bonne idée de la situation : plus de 6 500 paquets sources sur environ 7 900 ne contiennent pas la moindre modification à appliquer. On retrouve une distribution proche de celle de Pareto, avec quelques paquets qui concentrent le plus gros volume de correctifs, et la majorité qui n'en contient pas ou très peu.

Dans les figures 5.6a et 5.6b nous exposons la quantité de fichiers qui sont touchés par les correctifs de chaque paquet, comme défini précédemment dans la sous-section 5.2.1.1.2. La figure 5.6a présente ce résultat en n'effectuant aucun tri particulier, les paquets sont



(a) Par paquet

(b) Par taille

FIGURE 5.6 – Quantité de fichiers modifiés par les correctifs dans les paquets de Mageia, version 1

donc dans l'ordre alphabétique ; alors que la figure 5.6b propose de faire le tri sur la quantité de fichiers modifiés, ce qui permet de mieux lire les résultats. Immédiatement, nous observons que la tendance générale est plutôt similaire à celle décrite pour le cas précédent : une répartition où quelques paquets concentrent la majorité des modifications, et où l'immense majorité n'est pas ou peu touchée. Quelques paquets dépassent plusieurs centaines de fichiers impactés, la liste de ceux en ayant plus de 100 est donnée dans le tableau 5.1. Nous observons ainsi que parmi les paquets les plus modifiés en matière de nombre de fichiers impactés, les différentes versions du noyau sont particulièrement bien placées, accompagnées de `gdb` et `ncurses`.

La dernière mesure liée aux correctifs appliqués concerne le nombre de changements eux-mêmes est présentée dans la figure 5.7 : les points verts correspondent aux nombres de lignes de code ajoutées, et les points rouges à celles retirées. La valeur moyenne du nombre de lignes insérées est de 220, et 90 pour les suppressions de code. Cela s'observe bien sur le graphique, la densité de points étant supérieure en dessous de l'axe correspondant à la moyenne. Les modifications qui sont appliquées sur les logiciels de la distribution sont donc, plutôt, en général, des ajouts de code. La moyenne pouvant être influencée par des valeurs anormalement élevée, et comme le graphique montre que quelques paquets ont ce comportement, nous regardons également la valeur du 95^{e} centile du nombre d'insertions et de suppressions, qui est respectivement de 104.2 et 43.0 ; dans 95% des paquets qui contiennent des modifications, on ajoute un peu plus d'une centaine de lignes de code et l'on en retire une quarantaine.

Le second ensemble de résultats que nous allons étudier pour Mageia 3 et 4 concerne le volume et l'importance des correctifs qui sont appliqués sur la distribution. La comparaison avec Mageia 1 permettra de déterminer la complexité future de la maintenance.

En premier lieu, les figures 5.8a et 5.8b présentent le volume de correctifs dans les versions de Mageia 3 et respectivement 4. Ces données sont triées par volume croissant. Les courbes pour Mageia 1 sont proposées dans la figure 5.5. On retrouve dans les versions 3 et 4 la distribution déjà observée dans la version 1, à savoir qu'une majorité de paquets

Paquet	Quantité de fichiers	Paquet	Quantité de fichiers
`bash`	100	`grub`	136
`mailman`	103	`bacula`	141
`bsd-games`	104	`pulseaudio`	178
`apt`	108	`autofs`	199
`php`	109	`perl-Gtk2`	205
`mplayerplugin`	114	`vdr`	212
`qt3`	115	`initscripts`	260
`libreoffice`	117	`kernel-tmb`	497
`net-tools`	119	`kernel`	621
`mutt`	122	`kernel-linus`	621
`gcc`	125	`ncurses`	676
`netpbm`	126	`gdb`	722
`dietlibc`	127	`kernel-vserver`	1040
`john`	133	`kernel-xen`	2420
`hplip`	134		

TABLE 5.1 – Liste des paquets de Mageia 1 ayant plus de 100 fichiers modifiés

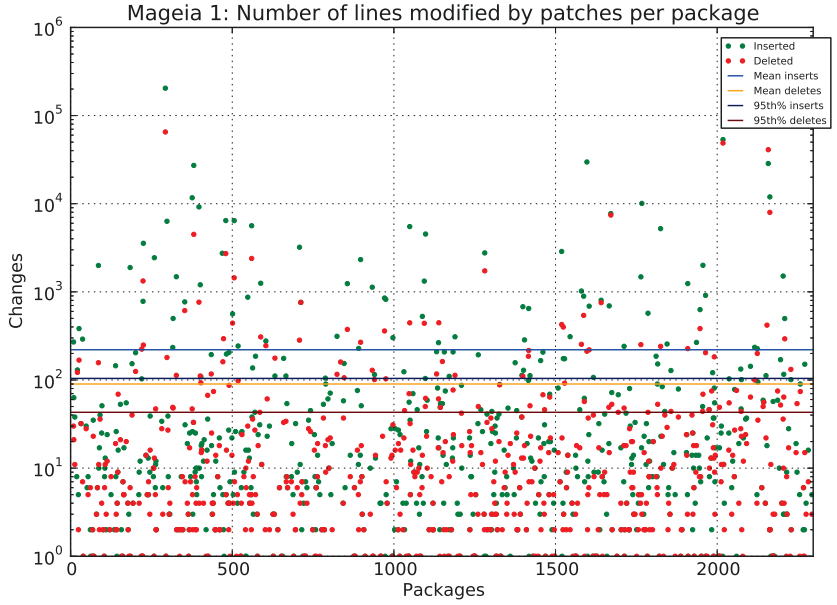
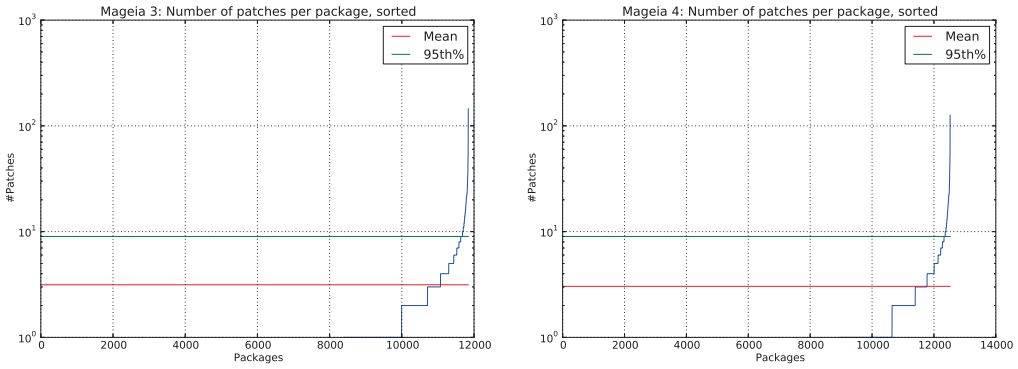


FIGURE 5.7 – Volume de modifications sur les paquets de Mageia, version 1



(a) Mageia 3 (b) Mageia 4

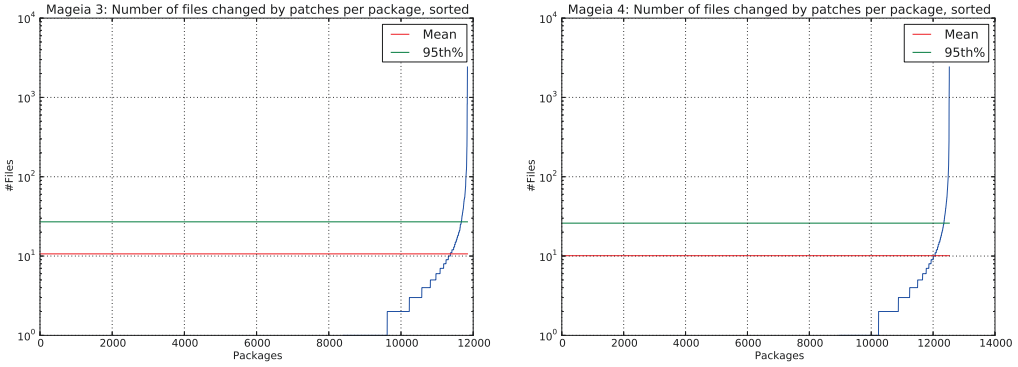
FIGURE 5.8 – Volume de correctifs dans les paquets de Mageia, versions 3 et 4

n'a pas de correctifs, ou très peu. Un glissement vers la droite de la courbe est visible. Le début de l'ascension de la courbe correspond aux paquets qui ont un seul correctif. Si ce début se déplace vers la droite, cela indique que la quantité de paquets qui n'ont pas de correctifs a augmenté. La distribution Mageia 1 comprenait un peu moins de 8 000 paquets, la version 3 est à un peu moins de 12 000. Plus précisément, nous avons 5 635 paquets sans correctifs sur 7 933 pour Mageia 3, et 8 391 sur 11 843 pour Mageia 4. Nous passons donc de 2 298 paquets avec des correctifs à 3 452.

Une légère baisse peut se lire sur les valeurs de moyenne ainsi que sur le nombre de paquets représentant le volume de 95%. Ces valeurs sont calculées sur l'ensemble des paquets qui ont au moins un correctif. Cette baisse légère traduit donc une amélioration : moins de correctifs à maintenir.

Nous passons à un niveau de précision supérieur avec l'analyse de la quantité de fichiers qui sont modifiés par les correctifs. Les graphiques sont disponibles pour les versions 3 et 4 de Mageia dans les figures 5.9a et respectivement 5.9b. La comparaison avec Mageia 1 peut être faite grâce à la figure 5.6. Dans ces nouvelles versions, la répartition de la quantité de fichiers modifiés est bien sûr toujours liée à la quantité de correctifs : la forme de la courbe est similaire. La tendance à la baisse de la quantité de modifications est confirmée sur ces graphiques : les valeurs de moyenne et de 95^{e} centile sont toujours plus faibles, entre les passages de la version 1 à 3, puis 3 à 4.

Les tableaux 5.2 et 5.3 proposent la liste des paquets dont le nombre de fichiers modifiés dépasse 100. Les données pour Mageia 1 sont disponibles dans le tableau 5.1. Ces listes permettent de visualiser l'évolution plus précisément. Ainsi, nous pouvons remarquer qu'entre Mageia 3 et 4, le paquet `bash` a vu sa quantité de correctifs augmenter, passant de 100 à 116, alors qu'entre les versions 1 et 3 il était resté stable. D'autres paquets se sont améliorés, comme `ncurses` dont les modifications impactaient plus de 600 fichiers, et qui passe à 237 dans Mageia 3. Plus tard, en version 4, ce paquet repasse à 870. Les différentes versions du noyau sont plutôt stables, avec quelques variations. Le paquet `gdb` chute de manière régulière, passant ainsi de 722 à 718 puis à 594 fichiers modifiés. L'outil SYSTEMD, était très peu modifié dans Mageia 1, avec seulement trois fichiers ; il explose en



(a) Mageia 3

(b) Mageia 4

FIGURE 5.9 – Quantité de fichiers modifiés par les correctifs dans les paquets de Mageia, versions 3 et 4

Paquet	Quantité de fichiers	Paquet	Quantité de fichiers
bash	100	dietlibc	174
FreeSOLID	103	castor	197
bsd-games	104	saoimage	197
ice	104	glassfish-toplink-essentials	207
jtds	110	systemd	217
chromium-browser-unstable	112	ncurses	237
netbeans	116	zoneminder	238
apt	117	open-iscsi	262
eclipse	118	insight	288
net-tools	120	java-1.7.0-openjdk	338
qt3	121	pacemaker	653
tycho	121	gdb	718
codeblocks	122	gcc3.3	766
jboss-as	122	kernel	828
mutt	124	kernel-linus	828
cups	128	kernel-tmb	828
netpbm	133	kernel-vserver	1117
vdr	136	kernel-rt	1213
grub	141	selinux-policy	1417
mythtv	157	sdcc	1664
valgrind	172	kernel-xen	2420

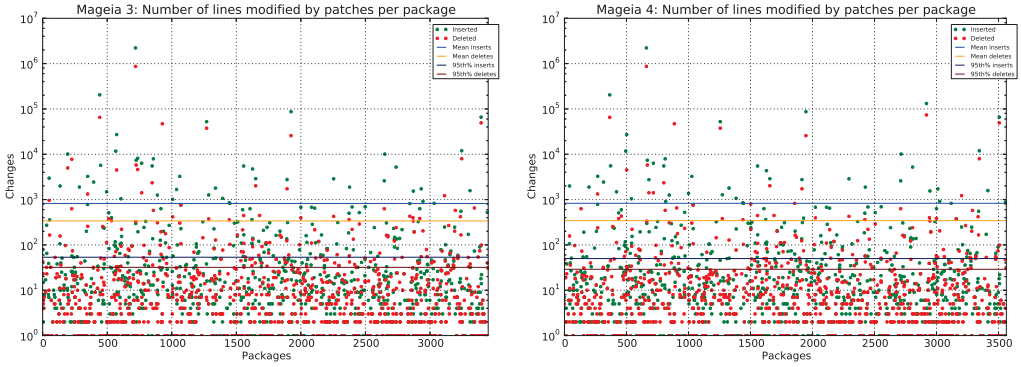
TABLE 5.2 – Liste des paquets de Mageia 3 ayant plus de 100 fichiers modifiés

version 3 avec 217 fichiers, puis la version 4 commence à réduire l'importance des correctifs en retombant à 195.

L'avant-dernier niveau d'analyse que nous proposons est celui du nombres de lignes modifiées par paquet : ajouts et suppressions. Ces données sont proposées dans les figures 5.10a et 5.10b respectivement pour Mageia 3 et 4 ; et en figure 5.7 pour Mageia 1. Le premier constat que nous pouvons faire concerne l'échelle entre les versions 1 et 3 : un ordre de grandeur supplémentaire est présent. Cela indique que certains paquets contiennent des correctifs encore plus importants en quantité de modifications dans Mageia 3 par rapport à Mageia 1. La moyenne des insertions et de suppressions de code augmente, quasiment d'un ordre de grandeur également. Dans le même temps, cependant, le 95^{e} centile des insertions diminue ; cela indique que pour 95% des paquets, la quantité de lignes de codes qui sont ajoutées diminue, et incidemment, à la vue de l'augmentation de la moyenne, l'évolution se fait sur quelques paquets et est importante. Les suppressions de ligne de code suivent une tendance similaire, même si les valeurs absolues sont plus faibles ; c'est cohérent avec

Paquet	Quantité de fichiers	Paquet	Quantité de fichiers
FreeSOLID	103	saoimage	197
bsd-games	104	perl-Gtk2	204
ice	104	glassfish-toplink-essentials	207
jtds	110	open-iscsi	262
chromium-browser-unstable	112	kernel	274
bash	116	kernel-linus	274
tycho	117	insight	288
apt	118	valgrind	402
net-tools	121	gdb	594
qt3	121	pacemaker	653
jboss-as	122	gcc3.3	766
mutt	124	ncurses	870
netpbm	133	kernel-tmb	892
vdr	136	selinux-policy	1007
grub	141	kernel-rt	1226
dietlibc	174	kernel-vserver	1289
policycoreutils	190	sdcc	1664
systemd	195	kernel-xen	2420
castor	197		

TABLE 5.3 – Liste des paquets de Mageia 4 ayant plus de 100 fichiers modifiés



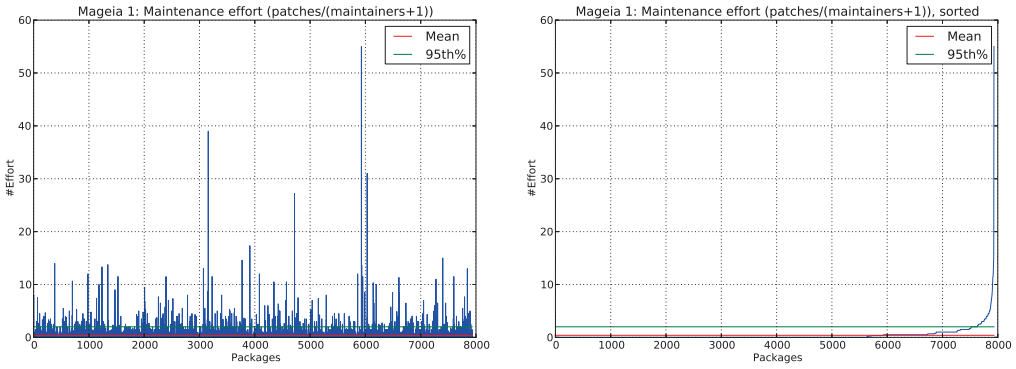
(a) Mageia 3 (b) Mageia 4

FIGURE 5.10 – Volume de modifications sur les paquets de Mageia, versions 3 et 4

ce que nous avons déjà observé sur Mageia 1 : les correctifs appliqués sur le code ont plutôt tendance à ajouter des lignes qu'à en supprimer. L'évolution entre Mageia 3 et Mageia 4 est similaire, mais très inférieure en valeur absolue. La diminution du nombre de modifications appliquées aux paquets est une bonne nouvelle du point de vue de la maintenance de la distribution.

5.2.1.4 Effort de maintenance

La dernière mesure présentée est l'effort de maintenance, dont la définition et les limites ont été décrits dans le paragraphe 5.2.1.1.4. Il est présenté dans la figure 5.11. Ce graphique nous apprends que l'effort de maintenance suit une courbe similaire à celles du nombre de correctifs (figure 5.5) et du nombre de fichiers (figure 5.6) : c'est logique, l'effort est calculé en partie à partir de ces paramètres. La valeur moyenne de cet effort est de l'ordre de 0.43, mais pour 95% des paquets, c'est une valeur de 2.0 qui est présente ; cela correspond, par exemple, à 4 correctifs pour 1 mainteneur. Les valeurs brutes ne sont pas l'intérêt principal de cette mesure ; nous voyons surtout un phénomène souvent ressenti par les



(a) Par paquet

(b) Par taille

FIGURE 5.11 – Estimation de l'effort de maintenance sur les paquets de Mageia, version 1

contributeurs de la distribution : l'effort à fournir sur les paquets est inégal, et certain sont particulièrement mal servis. La production de ces données permet ainsi de cibler les paquets qu'il convient d'améliorer : réduction du nombre de correctifs (en faisant remonter les changements aux développeurs), augmentation du nombre de mainteneurs, pour se répartir la tâche. Enfin cela nous permettra de renouveler l'analyse avec les différentes versions de Mageia et d'avoir un point de comparaison.

Les paquets qui présentent le plus gros niveau d'effort sont proposés dans le tableau 5.4, où nous indiquons également le nombre de correctifs du paquet. Ainsi, les paquets `net-tools` et `pulseaudio` ont un nombre de correctifs assez similaires, respectivement 78 et 73 ; cependant, le second a plus de mainteneur, puisqu'ils sont au nombre de 4 pour Mageia 1, le paquet `net-tools` n'ayant qu'un seul mainteneur. Un phénomène identique s'observe avec les paquets `kernel-xen` et `gdb` : ce dernier a nettement plus de correctifs, mais du fait d'un nombre accru de contributeurs, sont effort de maintenance est moindre.

Un dernier constat qu'il est possible de tirer de ces graphiques et de ce tableau concerne la population des logiciels : parmi ceux nécessitant le plus de maintenance, il y a tant des composants bas niveau (noyau, débogeur, `libc`) que des outils de haut niveau tels (enregistreur multimédia numérique, jeu).

Dans cette série de graphiques nous présentons l'effort de maintenance estimé pour les distributions Mageia 3 et 4 sur l'ensemble de leurs paquets ; les figures 5.12a et 5.12b sont respectivement les graphiques des versions 3 et 4. Nous pouvons d'abord remarquer que la forme générale des graphiques proposés rejoint celle qui est visible pour Mageia 1, en figure 5.11, et rappelle qu'une petite partie des paquets concentrent l'effort. Cela est d'autant plus visible en regardant les droites donnant la moyenne ainsi que le 95^{e} centile : ces valeurs évoluent respectivement de 0.39 et 2.0 pour Mageia 1 à 0.45 et 2.0 pour Mageia 3, ce qui représente une appréciation de 15% pour la moyenne de l'effort de maintenance entre ces deux versions. Pour Mageia 4, cette moyenne passe à 0.61 et le 95^{e} centile passe à 3.0, pourtant l'accroissement de celle-ci à 33%.

La distribution Mageia 1 a une petite particularité : il s'agit de la première version après la séparation d'avec Mandriva. Ceci implique qu'elle est composée d'un ensemble

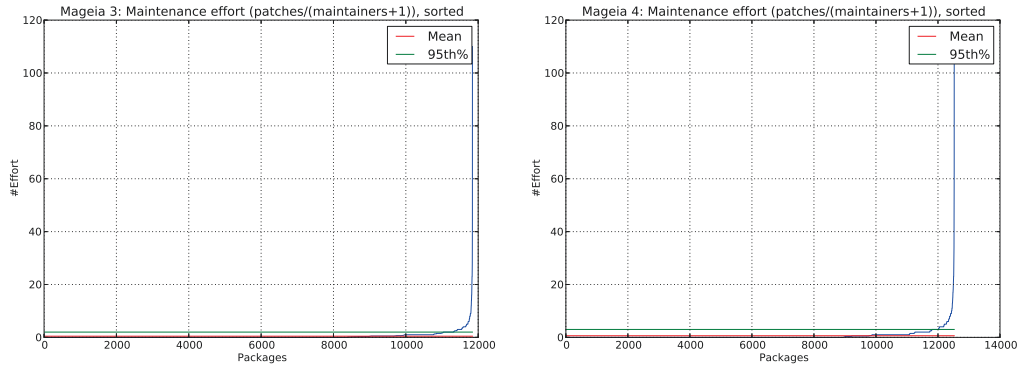
Paquet	Patches	Effort
`kernel-xen`	110	55.0
`net-tools`	78	39.0
`autofs`	62	31.0
`gdb`	136	27.2
`vdr`	52	17.3
`dietlibc`	45	15.0
`util-linux-ng`	30	15.0
`qt3`	42	14.0
`pulseaudio`	73	14.6
`stepmania`	14	14.0

TABLE 5.4 – Liste des 10 paquets de Mageia 1 ayant le plus gros effort de maintenance

plus restreint de paquets que les versions suivantes, et surtout, que tous ont eu un mainteneur pour réaliser l'importation et la première version. Pour Mageia 3 ces mêmes paquets peuvent n'avoir été que recompilé automatiquement par les robots, sans travail d'un mainteneur ; cela peut arriver pour des logiciels qui ne sont pas modifiés localement, et qui n'ont pas connu d'évolution majeure de la part de leur développeur. Le calcul étant effectué entre Mageia 2 et Mageia 3, et ces deux versions n'étant pas séparées de plus d'une année – elles sont respectivement sorties fin mai 2012 et mi-juin 2013 –, nombre de paquets peuvent être dans ce cas. Cela implique que si Mageia 1 ne présente aucun paquet n'ayant pas au moins un mainteneur, pour les versions 3 et 4 ce n'est pas vrai. Ainsi, 4532 paquets sur 11842 (38%) n'ont pas eu de mainteneur pour Mageia 3, et ces chiffres passent à 7516 sur 12525 (60%) pour Mageia 4. Ces chiffres nous permettent de relativiser quelque peu l'augmentation que nous observons pour la moyenne de l'effort de maintenance : elle croît, mais de manière moindre que la quantité de paquets n'ayant pas de mainteneur.

Le 95^{e} centile nous indique l'effort de maintenance pour la population « majoritaire ». S'il reste stable entre Mageia 1 et 3, cela nous indique que la masse des paquets qu'il représente n'a pas subi d'augmentation de l'effort nécessaire ; l'augmentation à 3.0 avec Mageia 4 traduit, par contre, la hausse du nombre de paquets sans mainteneur.

Nous proposons dans le tableau 5.5 la liste des 10 paquets qui présentent le plus fort taux de maintenance pour les distributions Mageia 3 et 4. Notons d'abord que le pire paquet reste `kernel-xen`, qui contient plus d'une centaine de correctifs mais qui n'a, sur les versions 3 et 4, aucun mainteneur ; XEN étant par essence un gros ensemble de modifications à appliquer sur le noyau, ce n'est pas forcément un problème très important, puisque d'une part les développeurs se fixent sur quelques versions du noyau, donc la mise à jour est plus réduite côté distribution, et ensuite, car depuis quelque temps, ses développeurs visent à intégrer leurs correctifs dans le noyau officiel. Entre Mageia 3 et 4 nous constatons, par contre, l'apparition de trois paquets qui peuvent être plus inquiétant : `grub`, `gdb` et `systemd`. Pour le premier, il s'agit d'un artefact provenant du passage à `grub2` dans la version 4 de la distribution : l'effort de maintenance s'est déporté sur celui-ci, qui en plus repose sur beaucoup moins de correctifs (passant de 42 à 14). Le paquet `gdb`, par contre, voit son nombre de mainteneurs baisser, passant de 7 à 1. Cela s'explique



(a) Mageia 3

(b) Mageia 4

FIGURE 5.12 – Estimation de l'effort de maintenance sur les paquets de Mageia, versions 3 et 4

Mageia 3			Mageia 4		
Paquet	Patches	Effort	Paquet	Patches	Effort
`tcp_wrappers`	23	23.0	`x11-driver-video-sisimedia`	29	29.0
`vnc`	23	23.0	`libvmime07`	34	34.0
`dkms`	24	24.0	`rsh`	34	34.0
`net-tools`	79	26.3	`net-tools`	80	40.0
`util-linux-ng`	27	27.0	`systemd`	80	40.0
`jboss-as`	28	28.0	`grub`	42	42.0
`libvmime07`	34	34.0	`vdsm`	44	44.0
`rsh`	34	34.0	`gcc3.3`	46	46.0
`vdsm`	44	44.0	`gdb`	126	63.0
`kernel-xen`	110	110.0	`kernel-xen`	110	110.0

TABLE 5.5 – Liste des 10 paquets de Mageia 3 et 4 ayant le plus gros effort de maintenance

probablement par le saut de version important entre Mageia 1 et 3, puisque `gdb` est passé de la 7.1 à la 7.5.1, alors que la version Mageia 4 accueille la 7.6. Enfin, l'intégration de `systemd` expose un phénomène similaire, Mageia 2 et 3 ayant eu beaucoup de mainteneurs en action sur ce paquet alors que pour la version 4, un seul a été nécessaire. La première version de Mageia incluait ce composant dans sa version 18, alors que Mageia 3 se base sur la version 195, et Mageia 4 sur la 207. La version 208 est arrivée quelques jours après la construction de ces résultats, et montre une amélioration notable : la plupart des correctifs ont été supprimés, et il n'en reste que 4 ; la quasi-intégralité des modifications proviennent ainsi de rétroportages, devenus inutiles avec la montée en version. Nous pouvons enfin noter que les paquets présents dans cette liste ont 0 ou 1 mainteneur, et que le taux de maintenance de ces 10 paquets augmente entre Mageia 3 et 4, la borne minimale passant de 23.0 à 29.0.

5.2.2 Étude comparative de noyaux

Dans l'introduction, nous avons établis lors de l'étude préalable que le paquet du noyau Linux était un bon candidat pour servir à mettre en place des outils de vérification de code au sein de la distribution. Dans cette sous-section, nous allons effectuer une étude comparative des erreurs entre différents types de noyaux, pour éclairer et documenter la situation actuelle. Nous commencerons par présenter les objectifs de cette étude dans la partie 5.2.2.1 puis les noyaux qui ont été retenus en 5.2.2.2. L'outil utilisé pour effectuer la détection de ces problèmes est Undertaker, dont nous rappelons les caractéristiques dans 5.2.2.3. Les résultats sont présentés et commentés avant de conclure. Une première version de cette étude a été communiquée [100] dans le cadre de la conférence *Ottawa Linux Symposium 2011*, et les différents retours nous ont permis d'affiner notre démarche.

5.2.2.1 Objectifs visés

Nous avons pu documenter la présence de correctifs dans des quantités non négligeables sur les différentes sources du noyau Linux dans la distribution Mageia. Une question qui se pose pour pouvoir effectuer des analyses, est le rôle de ces correctifs : nous savons empiriquement qu'il s'agit souvent d'ajouts de pilotes et de rétro-portages de corrections. Nous proposons ici d'avoir une analyse méthodique et exploitant un des outils de l'état de l'art quant à la vérification de code. Nous allons donc chercher à étudier les différences en matière de quantité de problèmes détectés sur le code source du noyau, entre la version « *vanilla* » et le code utilisé dans différentes distributions.

5.2.2.2 Noyaux et distributions

Pour cette comparaison, nous proposons d'étudier le code de différentes distributions ainsi que le noyau officiel : Mandriva, OpenSUSE et Debian. Le choix de ces distributions permet de balayer un spectre un peu large de besoins, et de pratiques de gestion de paquets ainsi que de modifications : Debian est souvent considéré comme étant une distribution appliquant beaucoup de correctifs. Un autre critère pour l'analyse a été la facilité de la disponibilité des sources du noyau modifié, prêt à être compilé : ces trois distributions proposent des dépôts de sources qui sont librement accessibles, et obtenir les sources modifiées prêtes à compiler est également simple.

Pour cette évaluation, un intervalle de versions du noyau devait être décidé : nous avons restreint au plus petit commun dénominateur de ce qui était disponible entre les différentes distributions sélectionnées. Étant donné l'hétérogénéité de gestion, cet ensemble n'est pas parfait, mais correspond environ aux versions 2.6.30 à 2.6.38 :

- Pour le noyau *vanilla*, l'intervalle est 2.6.32 à 2.6.38 ;
- Pour Debian, seul les noyaux 2.6.37-1 et 2.6.37-2 étaient disponibles ;
- Pour Mandriva, seule la version 2.6.33.7 était disponible ;
- OpenSUSE proposait un ensemble de versions très variés, allant de 2.6.30 au 2.6.37.

5.2.2.3 Undertaker, validation de configuration

Cet outil est présenté en détail dans la section 2.10 : il permet de valider la cohérence entre le modèle de configuration `Kconfig` du noyau et son implémentation dans le code source avec les directives `#ifdef` du préprocesseur. Nous retenons cet outil pour plusieurs raisons : d'abord, au moment de réaliser cette étude, l'intégration de Coccinelle n'était pas encore réalisée, alors que celle d'Undertaker était déjà fonctionnelle ; ensuite, cela permet d'évaluer la pertinence, l'utilisation et les résultats d'un autre outil d'analyse. Pour estimer la portée de ce type de problèmes, il convient de rappeler un constat qui a motivé les auteurs de l'outil : pendant environ six mois, le code du noyau Linux en charge de la gestion du branchement à chaud des processeurs était cassé, rendant ainsi inopérant l'ajout ou la suppression. L'origine du problème était dans le changement de nom d'une option Kconfig, dont les répercussions dans le code préprocesseur du noyau n'ont pas été correctement effectuées : une erreur de frappe a rendu le code inopérant.

Nous documentons à la fois l'évolution dans le temps du taux d'erreurs identifiées pour chacune des distributions, pour donner une idée de la qualité du code qui va atteindre l'utilisateur final. Nous documentons aussi les tendances générales qu'il est possible d'observer au sein d'une distribution. Enfin, nous pouvons faire des comparaisons en matière de « *qualité* » noyau sur les différentes distributions, en comparant ce taux. Il ne s'agit pas de dire que le travail d'une communauté est meilleur ou non, celles-ci répondants à des besoins différents et n'ayant pas les mêmes moyens ; de plus, de par son fonctionnement, Undertaker risque non seulement de détecter des faux positifs, mais il va faire l'analyse, par exemple, pour toutes les architectures supportées. Une option de configuration invalide posera des problèmes pour *certaines* personnes, ses utilisateurs. Par exemple, la distribution Debian fonctionne sur plus d'une dizaine d'architectures matérielles, là où Mandriva et OpenSUSE se limitent, grossièrement, à trois, `i686`, `amd64` et `armv7`. Un problème détecté sur `MIPS` sera donc plus gênant pour la première distribution.

Enfin, cette analyse se borne à un aspect quantitatif, nous n'allons pas chercher dans le détail à écarter les faux positifs, ni les problèmes qui ne sont pas pertinents pour les différentes distributions. De plus, les analyses sont effectuées en se basant sur l'arborescence du système de fichier comme cela a été précédemment fait dans le chapitre 3 ; ainsi nous définissons le taux de faute d'un sous-répertoire :

$$\text{taux}_{dir} = \frac{|Fautes_{dir}|}{|SLOC_{dir}|}$$

Avec *dir* le répertoire considéré, $Fautes_{dir}$ l'ensemble des problèmes qui ont été détectés pour ce répertoire, et $SLOC_{dir}$ la quantité de lignes de code détectées par Sloccount. Plus ce taux tends vers 0, meilleure sera la qualité. Pour des facilités de lecture, dans les graphiques qui suivront, ce taux sera donné par millier de lignes de code.

Le calcul de la différence entre deux taux taux_X et taux_Y, est défini par :

$$\text{diff}_{X,Y} = \text{taux}_X - \text{taux}_Y$$

Et il nous permet d'effectuer une comparaison. Ainsi, si $\text{diff}_{X,Y} > 0$, alors nous pouvons conclure que Y présente une meilleure qualité que X.

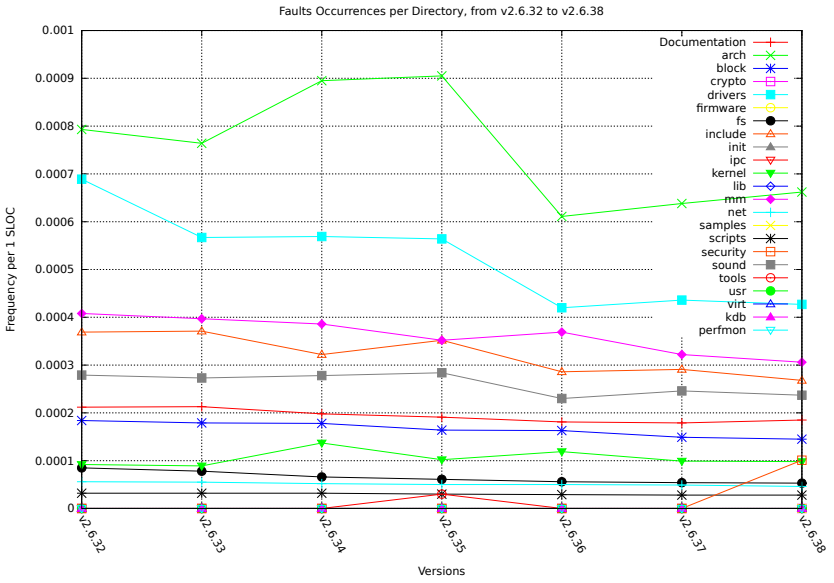


FIGURE 5.13 – Taux de faute sur le noyau, versions 2.6.32 à 2.6.38

5.2.2.4 Résultats

Nous allons maintenant présenter les résultats des analyses pour les différentes combinaisons retenues. Nous commençons par le noyau LINUX original dans le paragraphe 5.2.2.4.1.

5.2.2.4.1 Noyau *vanilla* Ce premier graphique visible en figure 5.13 nous permet de suivre l'évolution du taux d'erreur sur les versions 2.6.32 à 2.6.38. Dans [132, p. 13, Figure 9], PALIX et al. étudient le taux d'erreur avec COCCINELLE par sous-répertoire ; on peut constater que dans leur étude, `arch` présente un taux décroissant mais supérieur à `drivers`, lui-même décroissant. Notre graphique montre un comportement similaire, même s'il diffère en valeur absolue.

La tendance générale que nous pouvons dégager à partir de ce premier graphique, et en considérant l'intervalle de version limité que nous étudions, est une baisse du taux d'erreur, i.e., une augmentation de la qualité. La vitesse de ce changement est, par contre, différente entre les sous-répertoires : `drivers` et `arch` diminuent plus rapidement que tous les autres, même si le cas de ce dernier est moins clair, les régressions semblant fréquentes. Seul le répertoire `security` montre une évolution inverse, sur les instances du noyau 2.6.35 et 2.6.38, avec un taux en hausse.

5.2.2.4.2 Noyau Mandriva et améliorations L'étude des noyaux MANDRIVA se limite, de par les données disponibles au moment de sa réalisation, à étudier l'évolution sur deux versions, seulement. Les résultats sont proposés dans la figure 5.14. Une augmentation du taux de faute est très légèrement perceptible sur les sous-répertoires `arch`, `mm` et `include`, mais la variation est trop faible pour être pertinente. Nous notons également une

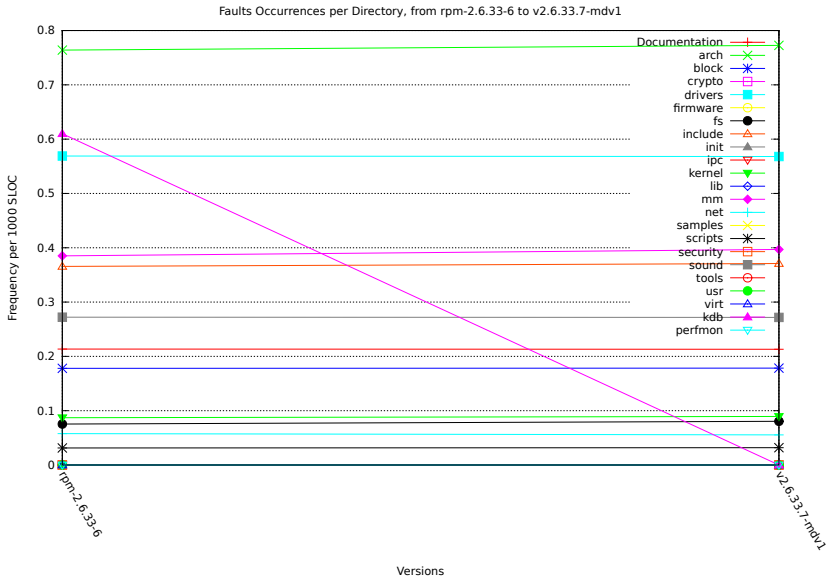


FIGURE 5.14 – Taux de faute sur le noyau MANDRIVA, versions 2.6.33.6 à 2.6.33.7

chute très importante, puisque la nouvelle version a un taux de 0 sur le répertoire `kdb`. Ce répertoire correspond au débogeur en espace noyau. Cette chute n'est pas spécifique à MANDRIVA, nous l'observons également sur le noyau OPENSUSE ci-après.

Dans un second graphique présenté en figure 5.15, nous proposons d'effectuer une comparaison, sur la version 2.6.33.7 du noyau, entre la version distribuée par les développeurs et la version modifiée par MANDRIVA. Globalement, nous pouvons conclure que cette version maintenue par la distribution présente moins d'erreurs qu'à l'origine, le nombre de répertoires présentant une amélioration du taux de faute étant de 5, alors que seulement deux montrent une diminution ; l'amélioration la plus importante se situe dans `drivers`, alors que les régressions sont concentrées d'abord dans `fs` puis dans `arch`. Nous pouvons noter que les améliorations concernent également les répertoires `include`, `mm` et `sound`.

5.2.2.4.3 Noyau OpenSUSE La distribution OPENSUSE propose un intervalle de versions du noyau plus étoffé que DEBIAN et MANDRIVA, qui est étudié dans le graphique 5.16. Celui-ci couvre les versions 2.6.30 à 2.6.37.1, soit une période allant de juin 2009 à janvier 2011.

Plusieurs tendances peuvent être dégagées à partir de ce graphique. La première concerne les répertoires qui concentrent le plus gros taux de faute, qui restent `arch` et `drivers`. Nous retrouvons l'intrus `kdb` et observons sa disparition lors du passage des noyaux 2.6.34 à 2.6.36. Sur ce graphique, nous pouvons aussi observer un effet de plateaux : les variations de taux d'erreurs au sein d'une même version « majeure » publiée par les développeurs sont faibles. Cela s'observe bien sur les répertoires `arch`, `drivers` et `kernel` :

– Pour `arch`, un premier plateau est identifiable sur les versions 2.6.30, un second sur la version 2.6.31, puis un troisième sur le noyau 2.6.32, suivi par le 2.6.33. Ces

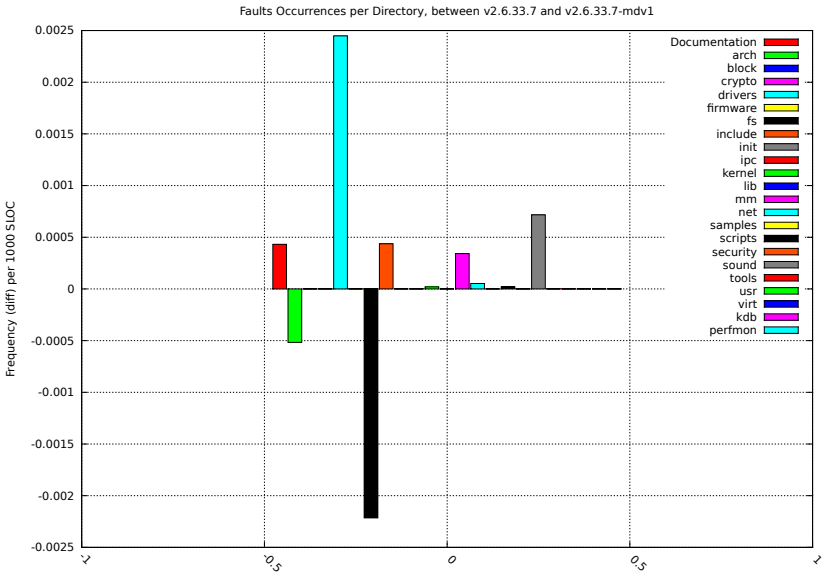


FIGURE 5.15 – Comparaison du taux de faute sur le noyau 2.6.33.7 entre la version *vanilla* et MANDRIVA

quatre plateaux montrent une baisse constante qui est renversée par l'arrivée du noyau 2.6.34 dont le taux de fautes régresse au niveau de la version 2.6.30. Enfin, l'arrivée du noyau 2.6.37 améliore notablement le taux de fautes sur ce répertoire et présente un dernier plateau identifiable ;

– Le répertoire `drivers` présente les mêmes ruptures de taux définissant les mêmes plateaux, sauf sur la période du noyau 2.6.33, où contrairement au répertoire `arch` qui baissait, ici aucune baisse ne peut être observée ;
– Parmi les particularités sur les plateaux que nous pouvons observer sur le répertoire `kernel`, notons d'abord que les césures n'ont pas lieu au même moment que pour les deux répertoires précédents ; ainsi, si le passage du noyau 2.6.30 au 2.6.31 est l'occasion de la création d'un premier plateau, celui-ci ne sera altéré en un autre plateau qu'au milieu de la série des noyaux 2.6.32, contrairement aux répertoires `arch` et `kernel`, qui évoluaient dès la première nouvelle version introduite. De même, aucun plateau distinct n'apparait lors du passage au noyau 2.6.33 ; par contre nous pouvons observer la création de trois plateaux pour les passages 2.6.33 à 2.6.34, 2.6.34 à 2.6.36 et 2.6.36 à 2.6.37, là où aucune différence n'était visible entre le 2.6.36 et 2.6.37.

Enfin, nous pouvons également tirer de ce graphique que les répertoires les plus sûrs quant aux problèmes identifiables par UNDERTAKER sont `kernel`, `fs` et `net`.

5.2.2.4.4 Amélioration 2.6.37 par Debian Nous proposons dans la figure 5.17 d'étudier l'impact des modifications appliquées sur le code source utilisé pour construire le noyau 2.6.37-2 utilisé par DEBIAN. La comparaison est effectuée suivant le calcul décrit préalablement en 5.2.2.3 ; l'évolution est calculée dans le sens 2.6.37 *vanilla* → 2.6.37-2

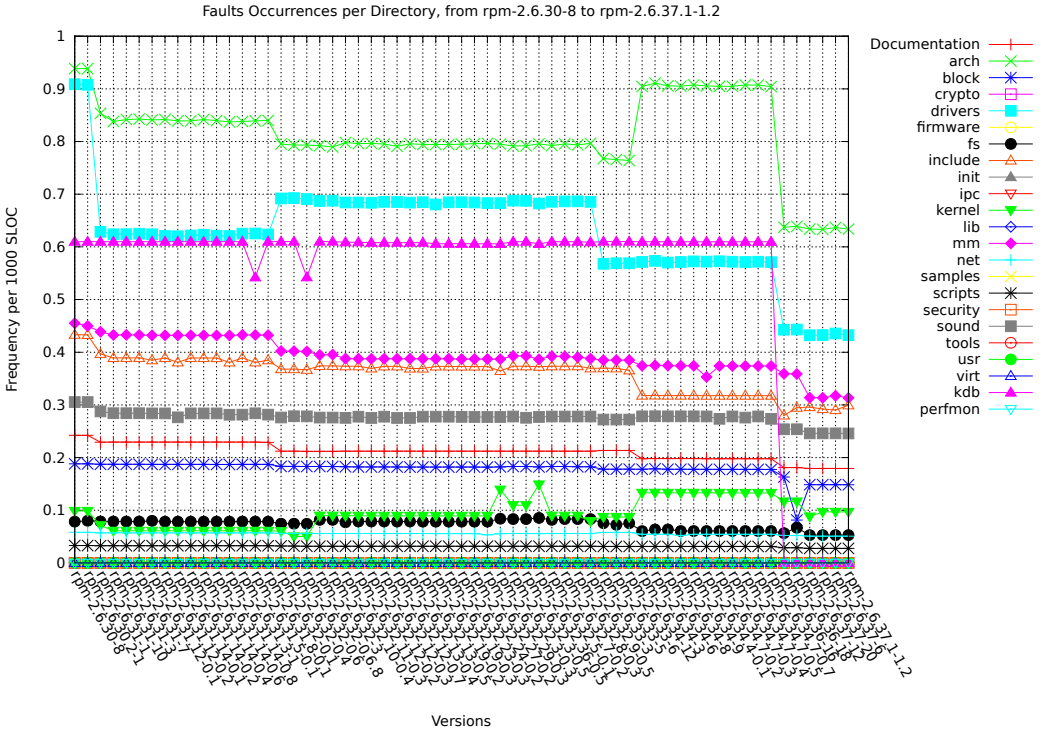


FIGURE 5.16 – Taux de faute sur le noyau OPENSUSE, versions 2.6.30 à 2.6.37.1

DEBIAN. Sur ce graphique, nous pouvons observer que la majorité des différences non nulles sont positives, cela indique que le noyau DEBIAN présente **moins** d'erreurs détectées par UNDERTAKER. Seuls deux répertoires présentent des régressions, `include` et `net`, alors que ceux ayant le plus de corrections sont `mm`, `arch` puis `drivers` avant `kernel`. Les améliorations sur `block` et `fs` et `scripts` sont visibles, mais plus limitées.

Nous pouvons également remarquer que l'ordre de grandeurs des améliorations est de 10^-3 pour 10^3 lignes de code, ce qui rapporté à la taille totale du noyau – de l'ordre de 9×10^6 – nous amène à des corrections de problèmes d'une quantité acceptable pour des modifications appliquées par une distribution, puisque cela fait environ une dizaine de corrections pour le cas `mm`.

Nous étudions ensuite dans la figure 5.18 l'évolution entre deux versions du noyau chez DEBIAN, 2.6.37-1 et 2.6.37-2 ; cela nous permet de mettre en lumière le travail effectué par les mainteneurs de cette distribution. Le répertoire `kernel` présente une différence négative qui montre la correction de problèmes en son sein. Nous pouvons aussi remarquer ce phénomène sur le répertoire `drivers`, à une échelle moindre. Nous notons également qu'une nouvelle version apporte certes des corrections, mais également des régressions ; ainsi les répertoires `include`, principalement, et aussi dans une moindre mesure `arch` et `mm` montrent une augmentation du taux de faute. La nouvelle version contient des problèmes par rapport à l'ancienne, sur ceux-là.

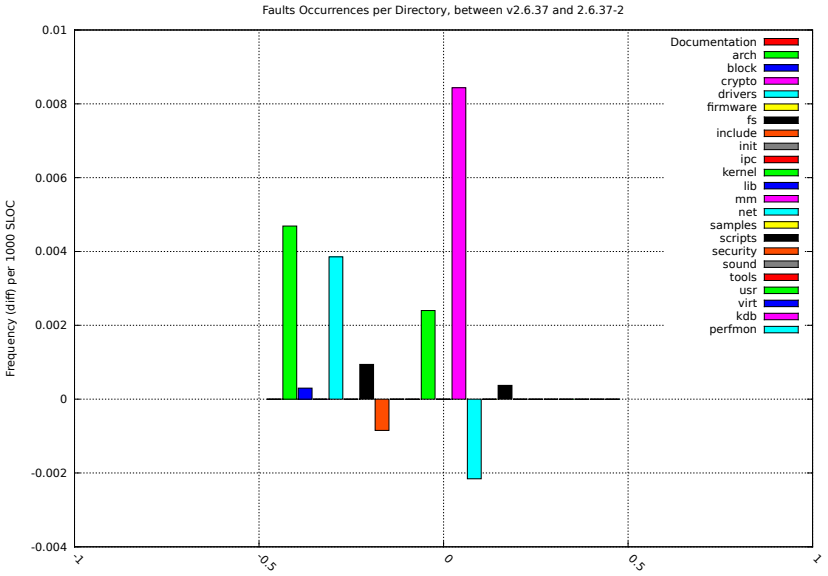


FIGURE 5.17 – Comparaison du taux de fautes entre le noyau *vanilla* 2.6.37 et le noyau DEBIAN 2.6.37-2

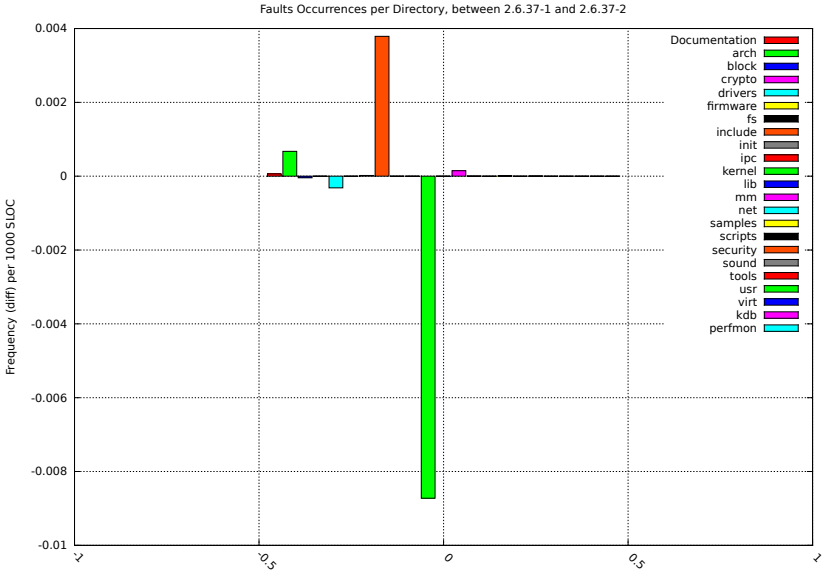


FIGURE 5.18 – Comparaison du taux de fautes entre les noyaux DEBIAN 2.6.37-1 et 2.6.37-2

5.2.2.5 Conclusion

Cette étude des différentes versions et variantes de noyaux avait plusieurs objectifs : d'abord, évaluer l'utilisation de l'outil UNDERTAKER en vue de sa possible intégration dans un ensemble de vérifications plus complet ; ensuite, étudier et documenter l'état des noyaux MANDRIVA ; enfin, nous cherchons également à voir l'impact des modifications appliquées par les distributions par rapport au noyau *vanilla*. Les résultats présentés permettent de répondre à toutes ces questions.

UNDERTAKER fourni des résultats que nous pouvons explorer et exploiter facilement. De plus, nous avons pu remarquer que l'analyse des problèmes identifiés par cet outil en s'intéressant aux sous-répertoires donne des résultats proches mais différents d'autres outils, tels que COCCINELLE. Cela confirme l'intérêt d'avoir plusieurs points de vues différents. Son utilisation sur les noyaux MANDRIVA nous a permis de documenter l'état actuel de la distribution, et particulièrement par rapport au reste de l'écosystème représenté par DEBIAN et OPENSUSE. Si la version proposée par MANDRIVA contient, globalement, moins de problèmes identifiables par UNDERTAKER, nous pouvons noter un taux non négligeable de régressions sur le répertoire `fs` documenté dans la figure 5.15 qui indique des problèmes potentiels.

Ces résultats nous permettent de voir que dans toutes les distributions, les versions qui sont proposées après le travail des mainteneurs contiennent **moins** de problèmes qui sont identifiables avec UNDERTAKER ; nous observons aussi que toutes les distributions, lors de l'application des modifications locales, semblent introduire des régressions par rapport à la version *vanilla*.

L'ordre de grandeur du taux d'erreur est similaire entre toutes les distributions, et reste assez proche des valeurs que nous pouvons observer sur le noyau d'origine. Travailler à détecter un maximum de ces problèmes préalablement à la mise à disposition des paquets aux utilisateurs permettrait donc d'avoir un avantage sur les autres distributions.

5.3 Propositions et implémentation

Dans la section précédente, nous avons présenté l'état actuel de la distribution MANDRIVA, et mis en lumière à la fois la présence de modifications parfois importantes sur les paquets, et leurs conséquences sur les problèmes identifiables dans le code source avec l'exemple du noyau LINUX. Ces résultats ainsi que ceux présentés dans l'état de l'art proposé en 2 nous conduisent à proposer un prototype d'outil pour permettre d'améliorer la qualité de la distribution. Nous présentons d'abord les besoins précisément dans la sous-section 5.3.1, puis nous proposons une architecture en 5.3.2 dont nous détaillons les différents modules. Un point sur l'état de l'implémentation est proposé en 5.3.3 avant de conclure en présentant les premiers résultats dans la sous-section 5.3.4.

5.3.1 Description des besoins

Nous avons montré dans les sections précédentes que la distribution réalise la jonction entre les développeurs de logiciels et leurs utilisateurs ; elle a donc une responsabilité

importante quant à la qualité de ce qu'elle livre. Par exemple, une équipe est chargée de la sécurité et son travail est de se tenir au courant des failles qui sont découvertes dans les logiciels proposés, pour assurer la sécurité des utilisateurs. Ce rôle central donne de grands pouvoirs, et implique donc une grande responsabilité ; notamment, rien n'est plus désagréables que de subir des régressions : des problèmes qui apparaissent sur une fonctionnalité dans une nouvelle version alors que le fonctionnement était correct dans la version précédente. L'absence de régressions identifiées est l'un des critères les plus importants pour l'acceptation des modifications sur le noyau Linux.

De plus, la distribution applique des modifications dans des proportions variables suivant les paquets ; la majorité n'est pas ou peu adaptée, mais quelques un sont lourdement modifiés. Plusieurs bonnes raisons expliquent cet état de fait : rétroportage de corrections de problèmes identifiés ou de fonctionnalités nécessaires, correctifs de sécurité, adaptation et intégration du paquet à la distribution. Tous ces changements font dériver le code source par rapport à la version originale, et impliquent plus de travail de la part de la distribution : les remontées d'erreurs en provenance des utilisateurs doivent être validées avant d'être remontées aux développeurs, ces erreurs pouvant être liées aux modifications appliquées.

Il est donc nécessaire de pouvoir identifier un maximum de problèmes avant qu'ils n'apparaissent chez l'utilisateur, et il est d'autant plus important que cette détection puisse se faire automatiquement qu'elle est rébarbative et donc plus sujette à des erreurs d'inattention de la part d'humains qui l'effectueraient. Nous voulons donc un outil qui vérifie le plus souvent la qualité des paquets de la distribution, par rapport à des ensembles de problèmes généraux ou déjà connus.

Dans un second temps, une fois un outil fonctionnel sur une distribution, nous souhaitons pouvoir l'étendre à d'autres, afin de proposer une plateforme ouverte orientée sur la qualité et intégrant des sources de données différentes : distributions, mais aussi la possibilité pour les développeurs d'avoir un suivi de leur projet. Des plateformes similaires mais orientées sur l'intégration continue existent, nous pouvons citer Travis [3].

Dans la partie 5.3.1.1 nous présenterons les outils existants et qui sont déjà utilisables mais dont nous ne faisons pas usage, puis dans la partie 5.3.1.2 nous définirons les besoins précis de l'outil que nous réaliserons.

5.3.1.1 Des outils inexploités

Au cours de la réalisation de l'état de l'art proposé en chapitre 2, nous avons croisé plusieurs outils permettant de détecter des problèmes dans une base de code source `C`, dont l'exemple principal est le noyau Linux, et par conséquent d'estimer la qualité de celui-ci. Ces outils sont utilisables, c'est-à-dire qu'ils fonctionnent, qu'ils ont un historique assez important et des développeurs suffisamment réactifs, et qu'ils présentent un taux de faux positifs acceptables. Ces deux derniers points sont primordiaux, puisque nous aurons forcément des cas de faux positifs, et qu'il sera donc nécessaire d'interagir avec les développeurs pour les identifier, et améliorer la technique pour les éviter.

Mais ces outils sont aussi peu voire pas exploités. Un des objectifs principaux dans l'in-

3. http://www.travis-ci.org

tégration de ces outils reste d'abaisser leur barrière à l'entrée, afin de faciliter l'utilisation par les développeurs et les mainteneurs.

5.3.1.1.1 Coccinelle Nous avons présenté en détail les différentes publications autour du projet Coccinelle dans la section 2.6. La première partie, évidente, utilisable est la détection de problème en exploitant les correctifs sémantiques, ce qui permet de les identifier au sein des fonctions : l'analyse inter-procédurale n'est pas possible. Le second axe d'utilisation dont les auteurs font mention est le projet Herodotos, qui propose de faire un suivi temporel des problèmes identifiés.

L'utilisation des correctifs sémantiques et de Coccinelle est principalement effectuée par ses développeurs, sur le noyau. D'autres développeurs qui ne bénéficient pas forcément directement de ces remontées, les mainteneurs des distributions et des versions dont la prise en charge est garantie plusieurs années, pourraient bénéficier fortement de la validation proposée par cet outil. Cependant, sa prise en main nécessite un apprentissage qui est jugé trop coûteux.

La généralisation à d'autres bases de code `C` que le noyau est assez simple, il suffit d'écrire des correctifs sémantiques adaptés ; il est possible de réutiliser une partie de ceux déjà présents qui sont suffisamment génériques.

5.3.1.1.2 Undertaker L'exploitation de cet outil est moins avancée, notamment de par sa plus relative jeunesse, comme les publications proposées dans l'état de l'art en section 2.10. Celui-ci permet de détecter les incohérences entre les options de configuration disponibles et ce qui est implémenté dans le code source, dont les premiers résultats ont été proposés sur le noyau Linux. Nous avons proposé d'étudier son utilisation sur différents cas, permettant à la fois d'évaluer son exploitation et de présenter l'état du noyau dans différentes distributions.

L'application à d'autres bases de code est assez simple, et l'outil peut déjà gérer à l'heure actuelle à la fois du `C` et du `C++`. La complexité réside principalement dans l'écriture d'extracteurs de variabilité.

5.3.1.1.3 GCC MELT Nous avons présenté ce projet dans la section 2.11 de l'état de l'art. Celui-ci permet de modifier le comportement de GCC et de détecter des violations de règles de programmation dans une base de code `C`. L'outil est très jeune, mais il est déjà fonctionnel et est proche de l'approche proposée 2.5 qui consiste à étendre le compilateur avec un ensemble de règles de vérification propre à chaque projet ; la qualité des analyses repose sur ces extensions écrites en Melt.

L'utilisation de ce projet reste expérimentale pour le moment, et les règles de validation à construire : nous pouvons assez rapidement introduire un petit ensemble de règles de base simples, mais qui nous assurent déjà quelques propriétés intéressantes.

5.3.1.2 Intégration continue de la qualité

À partir de l'ensemble de ces constatations, nous pouvons résumer le besoin général pour améliorer la qualité du code dans la distribution : mettre en place un système d'inté-

gration continue de la qualité. L'utilisation d'intégration continue est courante maintenant, et vise à exécuter une liste de tests (unitaires, d'intégration) prédéterminée à chaque modification du code. Sous l'hypothèse que la couverture du code par les tests est suffisante et importante, cette approche permet de déterminer très rapidement les modifications qui apportent des régressions en cassant des tests existants.

Nous proposons donc de mettre en place une approche similaire, mais qui au lieu d'exploiter des jeux de tests unitaires propres à chaque logiciel, effectuera un ensemble de tests avec des outils de qualité logicielle à la recherche de problèmes connus ou communs dans du code source. Les besoins pour mettre en place un système de ce type sont :

- Intégration avec le gestionnaire de source utilisé pour gérer l'empaquetage des logiciels de la distribution ;
- Cadre applicatif permettant de gérer l'exécution des différents outils de qualité ;
- Méthode, outils de collecte et d'archivage des résultats bruts retournés par les outils ;
- Mise en place d'un outil de navigation dans les résultats bruts ;
- Construction d'un tableau de bord synthétique répondant aux besoins de pilotage de la distribution.

5.3.2 Architecture générale

Nous allons présenter l'architecture générale de l'outil d'intégration continue de qualité proposé. Nous commencerons par la présenter sous la forme d'un schéma dans la section 5.3.2.1, puis les différents modules principaux sont détaillés dans les sections respectives 5.3.2.2, 5.3.2.3 et 5.3.2.4.

5.3.2.1 Schéma général

L'architecture proposée est présentée dans la figure 5.19, et présente l'enchaînement des interactions entre les différents composants, qu'ils soient pré-existants ou à construire : les quatre blocs situés en haut du schéma, vert et gris sont des éléments qui existent déjà, et avec lesquels nous allons nous intégrer. Le reste, c'est-à-dire les blocs bleus, orange et rouge correspondent à ce que nous devons ajouter. Les blocs d'une même couleur sont des composants d'un même module, et seront présentés ci-après dans les sections 5.3.2.2, 5.3.2.3 et 5.3.2.4.

Une contrainte importante pour la mise en place de cette architecture est d'abord de pouvoir suivre au plus près l'évolution de la distribution – et donc d'être directement connecté au dépôt de paquets utilisé pour le développement – sans pour autant gêner le travail. Cela justifie l'utilisation d'un crochet asynchrone sur le dépôt Subversion, pour déclencher le processus : l'intégration est ainsi réalisée, le processus sera déclenché à chacune des modifications envoyées sur le dépôt, et l'aspect asynchrone nous garanti que nous ne bloquerons pas le processus. Dans le pire cas, notre système ne sera pas alimenté.

5.3.2.2 Module : exécution des outils

Ce premier module est celui qui est composé des différents blocs de couleur bleue. Il est directement en communication avec le dépôt de paquets, c'est lui qui reçoit les

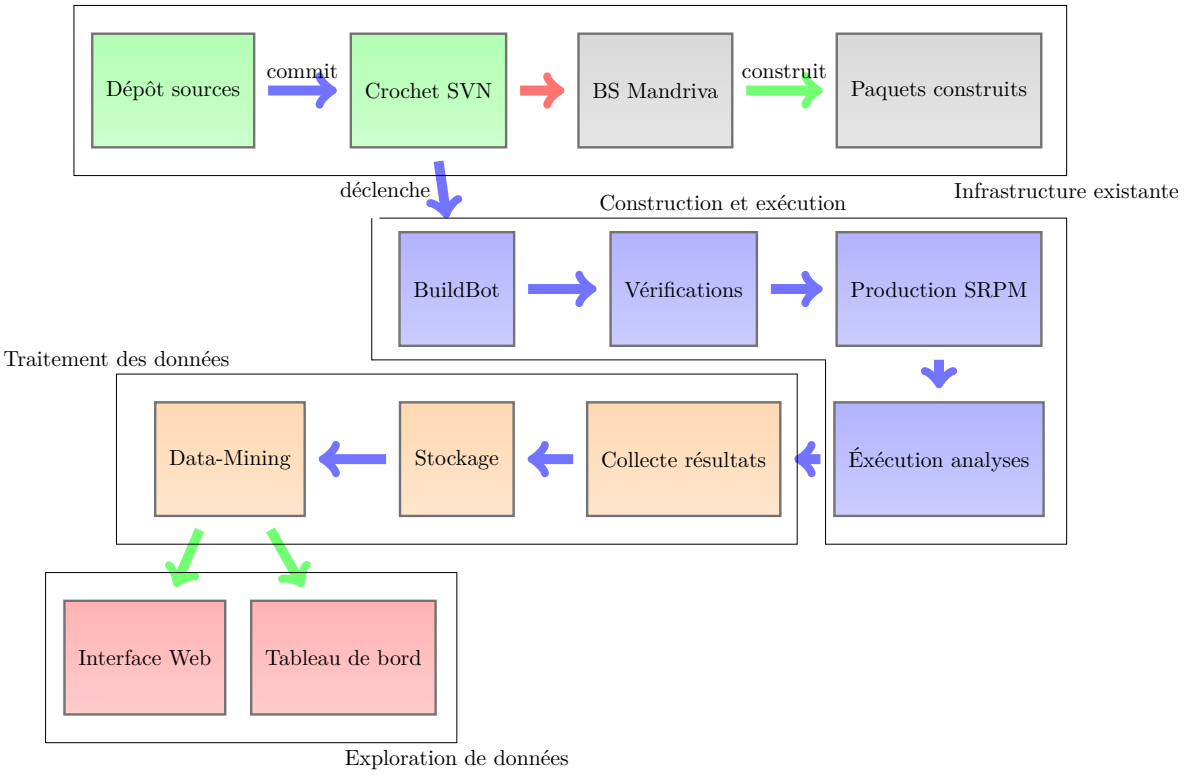


FIGURE 5.19 – Architecture générale pour la plateforme de qualité proposée

notifications asynchrones lors des modifications. Cette notification contient le nécessaire pour que le service ciblé puisse récupérer les sources :

- Adresse du dépôt ;
- Version de la distribution ;
- Nom du paquet source ;
- Branche de travail.

Le prototype d'implémentation proposée utilise BUILDBOT comme service pour l'exécution des différentes tâches. Ce choix se justifie particulièrement par la malléabilité de cet outil, simple à étendre et à adapter à notre cas d'utilisation, par le langage dans lequel il est écrit, PYTHON, qui est exploité dans d'autres projets importants de MANDRIVA, par ses modules déjà existants qui facilitent notre implémentation et par la dynamique de sa communauté. Le schéma le mentionne comme cible du crochet, avant d'indiquer d'autres étapes. Il est important de noter que ces autres étapes se déroulent **au sein** de BUILDBOT. Ce projet permet de mettre en place des outils de construction automatique, c'est-à-dire, prendre un code source puis le compiler. Les différentes étapes sont décrites dans le fichier de configuration qui est un script PYTHON ; ainsi, en écrivant une configuration adaptée et avec l'écosystème nécessaire, il est facile de monter son propre mécanisme de construction. Nous avons donc implémenté les différentes étapes nécessaires à la réalisation du processus :

1. Récupération du code source du paquet à construire depuis le dépôt SUBVERSION ;
2. Détection des outils d'analyse à appliquer sur ce paquet ;
3. Construction du paquet RPM source, permettant d'obtenir le code source avec les modifications appliquées ;
4. Exécution des différents outils d'analyses.

Les données produites au cours de l'exécution des analyses sont extraites de l'environnement de construction, puis stockées temporairement par les différents mécanismes mis à disposition par BUILDBOT, notamment les sorties standard et d'erreur. Si un outil d'analyse produit des fichiers, nous devrons les collecter à la fin de la construction, pour être en mesure de les extraire dans les modules décrits ci-après.

5.3.2.3 Module : exportation, extraction et stockage des résultats

Le rôle de ce module est d'extraire les données, *temporaires*, qui ont été produites lors des différentes étapes dans BUILDBOT. En particulier, pour les analyses que nous exécutons, nous sommes intéressés par les sortie standard et sortie d'erreur standard où les données sont présentées. Nous téléchargeons ces sorties via l'interface de programmation mise à disposition par BUILDBOT, et, pour chaque outil d'analyse, donc mettons en place un module PYTHON chargé du traitement des données. Ainsi les données pertinentes de chacun sont extraites, et le stockage permanent peut être réalisé. Ce stockage à long terme nécessite de pouvoir prendre en charge des données structurées et volumineuses.

Dans un premier temps, nous proposons une implémentation avec une simple base de données relationnelle permettant de stocker les résultats des analyses. Pour être en mesure de tenir des montées en charges importantes, cependant, nous avons étudié l'utilisation

d'outils de traitement massif de données, *BigData*, tel Hadoop qui propose un module Flume permettant de faire du traitement sur des journaux de services. Ce problème se rapproche assez bien du nôtre.

5.3.2.4 Module : interface web d'exploration des données

Ce dernier module permet de faire l'interface avec l'utilisateur, selon deux profils :
- Responsable :
  - État global de la distribution
  - Statistiques sur les paquets
  - Statistiques sur les langages

  Les responsables ont besoin d'avoir une vue globale pour être en mesure de piloter la distribution, c'est-à-dire, gérer l'affectation des développeurs pour la maintenance des paquets, savoir quels sont les besoins pour améliorer la qualité, etc. ;
- Développeur : pour l'aider dans son travail quotidien, le développeur a plutôt besoin d'un niveau de lecture des données plus bas, être en mesure de naviguer dans les résultats précis, avoir un état des erreurs, être capable de les traquer, et de les identifier entre différents paquets.

Ces deux profils justifient donc les deux composants proposés dans le schéma, en couleur rouge : une interface web d'exploration et de navigation dans les données, avec un moteur de recherche ; et un tableau de bord synthétisant des métriques permettant d'expliquer l'état général actuel de la distribution. Par exemple, un développeur sera intéressé à voir les différents correctifs sémantiques appliqués sur le noyau Linux qui sont les plus générateurs d'erreurs.

Les différentes métriques à faire figurer dans un tableau de bord nécessitent une étude un peu plus approfondie, mais une première approche peut consister à reprendre celles que nous avons proposé : volume de code source, quantité de correctifs pour chaque paquet et caractérisation des modifications, évolution du taux de maintenance.

5.3.3 État de l'implémentation

L'implémentation actuelle proposait une preuve de concept fonctionnel et avait pour objectif d'illustrer les besoins et d'y proposer une réponse concrète ; il s'agissait également d'étudier les comportements et les particularités pratiques afin d'avoir un recul nécessaire avant de mettre en place un service plus complet. Dans la section 5.3.3.1 nous décrivons plus en détail l'état précis de l'implémentation, puis dans la section 5.3.3.2 nous présentons un prototype de système de construction pour la distribution réutilisant les briques développées.

5.3.3.1 Prototype fonctionnel

Les différents modules présentés précédemment ont été implémentés, mais à l'état de prototype, pour montrer la faisabilité et la pertinence ; cependant certains composants sont tout de même loin d'un simple prototype. C'est le cas de la majeure partie du code produit et intégré avec Buildbot : nous avons mis en place la chaîne de traitement complète,

depuis la récupération de la notification d'un nouveau paquet jusqu'à la production des sources prêtes à être construites.

L'exécution conditionnelle de différents outils d'analyse est fonctionnelle, et a été validée à la fois sur Coccinelle et sur SLOCCount ; l'approche retenue a été d'introduire un fichier descriptif des analyses à exécuter, à la racine des sources du paquet, `analysis.ini`, au format *clef-valeur* : pour chaque outil d'analyse, une clef avec son nom, et un booléen indiquant l'exécution. Certains outils sont exécutés par défaut, mais peuvent ainsi être désactivé, et l'exemple retenu pour ce cas d'utilisation est SLOCCount. D'autres outils sont eux à activer manuellement, partant du principe que leur applicabilité à n'importe quel code source n'est pas immédiate et qu'il est nécessaire de s'assurer et de mettre en place un jeu de données de test pour leur bonne marche ; c'est le cas de Coccinelle, qui n'a ainsi été activé que pour les paquets sources des différentes variantes du noyau Linux.

Pour le stockage des résultats des analyses, comme déjà indiqué, cela est effectué en deux étapes : d'abord un stockage temporaire dans les archives de l'outil de construction Buildbot, qui permet un accès à distance à ces données via une API REST. Nous avons donc un second outil directement connecté à la base de données utilisée par l'interface web, qui va se connecter à l'instance Buildbot et en récupérer les derniers résultats non encore extraits : cet outil comprend plusieurs petits modules, un pour chaque logiciel d'analyse, qui sont eux en charge d'effectuer le traitement, c'est-à-dire, de parcourir les fichiers journaux produits et d'en extraire les données pertinentes. Cette approche n'est pas forcément la plus efficace en matière de capacité de traitement, mais pour l'échelle de l'évaluation elle permet d'obtenir rapidement de bons résultats.

L'interface web est centrée sur un moteur de recherche permettant d'effectuer facilement des filtrages au sein des résultats sur différents critères qu'il est possible d'étendre et de combiner :

- Nom de paquet ;
- Version ;
- Nom de fichier source ;
- Langage utilisé dans le paquet ;
- Erreur remontée par Coccinelle.

L'aspect tableau de bord est proposé avec une page de détails généraux sur la distribution contenant différentes statistiques similaires aux mesures déjà présentées, notamment la volumétrie des langages. L'analyse des correctifs n'est pas encore implémentée, mais elle pourrait facilement l'être. Ces données sont archivées et peuvent être visualisées avec leur historique, sous la forme de graphique : il est ainsi possible de suivre, pour un paquet, l'évolution de la répartition de l'utilisation des différents langages de programmation. C'est également disponible pour l'ensemble de la distribution.

Outre la mise en place de plus nombreuses métriques et la finition de l'interface web, le point critique pour être en mesure de pouvoir tenir une cadence correcte, l'autre élément important reste le stockage des données après leur extraction depuis Buildbot : si la solution d'une base de données relationnelles parait la meilleure pour la connexion avec l'interface de visualisation et de fouille de données, le stockage à long matière de la totalité de ce qui a été généré, lui, n'est pas ainsi le plus optimal. Plusieurs raisons nous poussent à vouloir effectuer un archivage des résultats bruts : la possibilité de retraiter les données que

nous explorons déjà, la conservation d'informations présentes mais dont nous ne faisons pas l'usage, etc.

5.3.3.2 Système de construction ARM

Par effet de bord, la réalisation de ce prototype nous a permis de mettre en place les briques nécessaires à la construction des paquets binaires : d'une part, nous avons implémenté le nécessaire permettant de créer des paquets RPM source à partir du dépôt Subversion de la distribution, d'autre part, Buildbot propose un module pour exécuter des constructions de paquets binaires à l'aide de l'outil Mock, dont une variante est disponible pour Mandriva. En combinant les deux résultats, nous avons pu mettre en place rapidement et efficacement un prototype de système de construction de distribution, visant plus particulièrement l'architecture ARMv7 sur une tablette *Google Nexus 7*, capable de construire automatiquement les paquets dès qu'ils sont modifiés dans le dépôt source de la distribution et de les mettre à disposition sur un serveur, prêts à être téléchargés et installés comme depuis n'importe quel dépôt de paquet binaires. La mise en place de ce système a permis d'accélérer le travail sur le projet de recherche Solen visant à proposer une liseuse électronique innovante.

5.3.4 Résultats préliminaires

Nous allons évoquer maintenant les premiers résultats que nous avons pu obtenir en utilisant ce prototype pendant une partie du cycle de développement de la distribution Mandriva Business Server, se basant sur Mageia. Un premier essai de réalisation de prototype a été mené dans le cadre d'un stage de 4^{e} année au sein de Mandriva avec Bernadin Namoano. Ces travaux ont permis d'aider à la structuration des besoins pour réaliser un tel outil, en proposant une première architecture logicielle. Ces résultat ont fait l'objet d'une communication [102] dans le cadre de la conférence *Ottawa Linux Symposium 2012*, afin de présenter l'idée et l'approche d'une plateforme de qualité générique.

5.3.4.1 Documentation du code des paquets

L'interface web obtenu permet de documenter de manière constante et reproductible la composition des paquets ainsi que celle de la distribution en général. Nous l'illustrons avec deux copies d'écrans : dans le cas du suivi d'un paquet précis, `drakxtools` avec la figure 5.20 ; puis dans le cas de la distribution avec la figure 5.21.

Ce paquet contient un des composants important de Mageia puisqu'il s'agit d'une suite d'assistants de configuration pour aider à la mise en place des différents outils et services systèmes, ainsi qu'à sa maintenance. La documentation de sa composition nous apprends qu'il contient principalement du code `Perl`, ce qui était attendu, mais également du `sh` et `Makefile`, pour le système de construction. Nous trouvons également quelques traces de langage `C`. Le choix de visualisation effectué permet de ne pas être influencé par la valeur absolue des différents langages : la quantité de code évolue, en permanence. Pour avoir une vue synthétique de la composition des paquets, il est intéressant de rapporter le volume brut de chacun au total du logiciel, afin éventuellement de détecter des évolutions.

FIGURE 5.20 – Illustration du suivi temporel de la composition d'un paquet, avec l'exemple de `drakxtools`

Dans le premier prototype, l'évolution temporelle est limitée pour le moment au cas de la volumétrie des langages, mais il est tout à fait pertinent de vouloir suivre n'importe quelle métrique de chaque paquet. Nous pouvons aussi constater, dans la liste des analyses, que plusieurs versions identiques existent : nous analysons et collectons toutes les instances modifiées depuis le dépôt de code source ; ainsi à chaque changement envoyé par les développeurs, nous regénérons une analyse. Cette stratégie a été retenue temporairement pour l'établissement du prototype, à la fois pour générer un volume de données pertinent et pour proposer un retour rapide aux développeurs.

La seconde capture d'écran, proposée dans la figure 5.21, illustre l'état global de la distribution tel que vu à un instant précis. Il est important de ne pas considérer comme définitives les données documentées dans cette copie d'écran : l'ensemble de la distribution n'a pas été analysé entièrement et plusieurs paquets importants sont en doublons, ce qui fausse les résultats. Il s'agit de montrer une visualisation des données que nous avons proposées dans l'étude préalable et dans sa mise à jour. L'intérêt de cette interface, contrairement aux études réalisées précédemment, réside dans la connexion directe avec le dépôt, via cependant les outils d'analyse, ce qui garantit d'avoir des données le plus à jour possible. Comme, de plus, nous archivons les informations des paquets lorsque nous effectuons l'analyse, nous sommes en mesure d'accéder à l'historique et ainsi de pouvoir naviguer dans le temps, et effectuer des comparaisons de manière plus efficace et rapide. Cette interface est à voir comme le précurseur d'un tableau de bord général de la distribution qui permettrait de suivre son cycle de vie entier, et d'avoir un état global à la fois de la composition de la base de code mais aussi de la quantité de problèmes identifiés par les outils d'analyses, et pourquoi pas de le connecter au système de suivi de problèmes pour

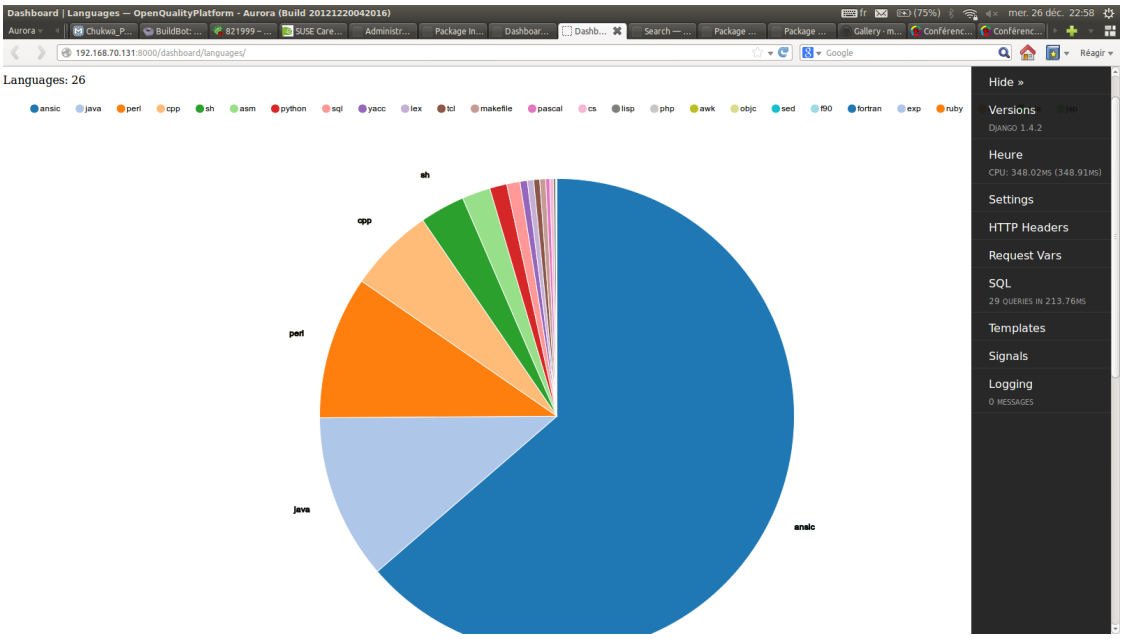


FIGURE 5.21 – Illustration de l'indication de l'état général de la distribution avec la répartition des langages utilisés

compléter la vue.

5.3.4.2 Détection des erreurs fréquentes sur le noyau

La réalisation du stockage des données et du « moteur de recherche » nous permet de fouiller les résultats, et notamment d'avoir un aperçu du résultat de chaque correctif sémantique. Lorsque nous collectons tous les résultats d'une analyse COCCINELLE, nous enregistrons les données correspondant au correctif sémantique :

- Titre du correctif ;
- Fichier source `.cocci` ;
- Options à inclure lors de l'exécution de COCCINELLE ;
- Rapports liés : instances d'erreur du correctif.

Ces rapports liés sont les endroits, dans des paquets, où le correctif sémantique a détecté *quelque chose*. Nous proposons de l'illustrer avec la capture d'écran 5.22 que nous allons expliciter. Le correctif concerné détecte l'utilisation d'une construction obsolète dans le code source du noyau : une séquence de fonctions à appeler a été remplacé et simplifié par un seul appel, ce qui facilite l'écriture et la maintenance du code. Sur l'instance retenue, ce problème a été identifié 21 fois : pour chacune des occurrences, nous indiquons le fichier source concerné, le paquet, et la position dans le fichier (ligne, intervalle de colonnes).

FIGURE 5.22 – Illustration d'un correctif sémantique Coccinelle tel que visualisé avec les occurrences

5.4 Améliorations et extensions

Dans cette section, nous allons exposer les résultats de travaux exploratoires menés dans le but d'étendre un système de vérification de code automatisé sur une distribution. Nous allons d'abord nous intéresser au projet GCC MELT en rappelant et en détaillant son fonctionnement dans la section 5.4.1 ainsi que son exploitation dans le cadre de notre objectif dans la section 5.4.2. Nous évoquerons également un projet qui propose de s'attaquer au problème de la validation sur les structures de données dynamiques dans la partie 5.4.3. Puis nous ferons le point sur les derniers développements de la communauté sur la thématique de vérification de code et de qualité à l'échelle d'une distribution dans la section 5.4.4.

5.4.1 Présentation de GCC MELT

Les différentes publications relatives à GCC MELT sont proposées dans la section 2.11 de l'état de l'art, et dont une rapide présentation a été proposée en 2.3.2. MELT est donc un langage dédié, inspiré de *List*, qui permet de simplifier l'écriture de greffons pour le compilateur GCC. Le compilateur est divisé en trois parties fondamentales : *front-end*, *middle-end* et *back-end*. La première et la dernière sont responsables de la transformation du code source en une représentation intermédiaire *IR* propre à GCC puis de traduire celle-ci en code machine. La partie *middle-end* travaille sur une représentation interne indépendante des langages d'entrée et de sortie. C'est sur celle-ci que les optimisations et les vérifications sont effectuées, par toute une série de *passes* qui s'enchaînent. Une particularité de GCC par rapport à son compétiteur libre principal CLANG, c'est la quantité

et la relative petite taille des différentes passes, ainsi que la difficulté de prédictibilité de l'ordre et des passes qui seront appliquées à un code source.

MELT permet à l'utilisateur d'entrer dans la représentation intermédiaire en définissant où se connecter grâce au référencement d'une passe. Une fois « connecté », l'utilisateur peut manipuler à son bon vouloir les structures internes de la représentation du code : vérification, modification. Les vérifications peuvent être d'ordre très général, mais la cible privilégiée reste les projets ayant un code source important et des contraintes en terme d'intégrité qui ne peuvent être exprimées habituellement ou facilement. Les modifications qu'il est possible d'apporter au code de cette manière n'ont, théoriquement, pas de limite, et MELT peut être vu comme un outil de prototypage rapide pour évaluer de nouvelles passes d'optimisation à réaliser dans GCC.

L'expressivité et les constructions de base du langage permettent d'être rapidement efficace pour manipuler les structures internes : le code n'est jamais qu'un arbre, et exprimer simplement la reconnaissance des motifs dans ce cas est très utile. Le pattern-matching proposé permet de réaliser cette opération. Pour toutes ces raisons, il parait naturel de souhaiter exploiter sa puissance pour améliorer la qualité générale de la distribution.

5.4.2 Exploitation de MELT

Les possibilités offertes par cet outil sont particulièrement étendues, mais dans notre cas elles se bornent surtout à faire de la vérification. Un avantage de MELT est son fonctionnement en tant que greffon de GCC : cela nous permet de l'exécuter **pendant** les phases de compilation, ce qui facilite l'intégration dans l'écosystème. Pour pouvoir exploiter cet outil, deux phases sont nécessaires : d'abord, comprendre son fonctionnement et celui de ses règles ; ensuite, le faire fonctionner avec la version de GCC utilisée dans la distribution.

Cette seconde condition ne nécessite qu'un travail d'empaquetage en lien avec le développeur afin de remonter les différents problèmes identifiés et améliorer le code. L'autre condition, par contre, a nécessité l'encadrement d'un projet étudiant mené à bien par Pierre Vittet qui a ainsi explicité le fonctionnement et comment tirer parti de la puissance de cet outil. Il a donné naissance au projet Talpo, présenté ci-après. La prise en main de MELT par Pierre Vittet a permis la réalisation de deux communications pour participer à la dissémination technologique : [181] dans le cadre des *Rencontres Mondiales du Logiciel Libre 2011* afin de toucher un public de contributeurs à différents logiciels libres large ; [168] dans le cadre du *GNU Hackers Meeting in Paris 2011* pour cibler plus précisément des contributeurs GNU.

Nous proposons dans le listage 5.1 un exemple de code qui vise à identifier un problème ne se manifestant que sur certaines plateformes matérielles, notamment ARM : c'est un problème courant et connu [4] de longue date, mais qui pourtant persistait dans un pilote du noyau Linux. La déclaration d'un type `char` sans indiquer son signe le rend indéterminé et le compilateur va choisir suivant la méthode qui sera la plus efficace sur la plateforme matérielle. Le problème avec ce pilote survenait lors de la manipulation de cette variable où une opération binaire avec risque de débordement est effectuée : `x |= ~0x0f;`. Sur

4. http://www.arm.linux.org.uk/docs/faqs/signedchar.php

une plateforme AMD64, cette variable `x` est utilisée comme étant signée, alors que sur ARM elle est utilisée en étant non signée. La valeur qui était lue préalablement depuis le périphérique, couplée à cette opération binaire, engendrait le dépassement de capacité, ce qui sur ARM générait une valeur positive alors que le développeur s'attendait à une valeur négative.

```
;; -*- Lisp -*-
;; file char.melt

;;      (?(gimple_assign_cast ?lhs ?rhs)
;;        (debug "lhs=" lhs " treetype=" (tree_type lhs))
;;        (if (==t (tree_type lhs) tr_char)
;;          (debug "coucou")
;;    (walk_use_def_chain_depth_first
;;      (lambda (val :tree tr :gimple gi)
;;        (debug "lambda: val=" val
;;               " tr=" tr
;;         " gi=" gi)
;;             :true
;;            )
;;      'toto lhs
;;    )
;;        )
;;        (debug "rhs=" rhs)
;;      )

(definstance ref_char_type_tree class_reference)
(definstance ref_signedchar_type_tree class_reference)
(definstance ref_unsignedchar_type_tree class_reference)

;; our gimple walker function just matches types
(defun char_gimple_walker (pass :gimple g)
  (let ( (:tree tr_char (tree_char_type_node))
         (:tree tr_signedchar (tree_signed_char_type_node))
         (:tree tr_unsignedchar (tree_unsigned_char_type_node))
   )
    (debug "char_gimple_walker g=" g)
    (match
     g
     (?(gimple_assign_binaryop ?lhs ?rhs1 ?rhs2 ?opcode)
       (let ( (:tree tr_type_lhs (tree_type lhs)) )
         (when (or
          (==t tr_type_lhs tr_char)
              (==t tr_type_lhs tr_signedchar)
          (==t tr_type_lhs tr_unsignedchar)
         )
            (debug "binaryop"
                   " lhs=" lhs
              " rhs1=" rhs1
              " rhs2=" rhs2
              " opcode=" opcode
            )
      (match
        rhs1
        (?(tree_ssa_name ?tvar ?tvalue ?vers ?gdef)
```

```
            (debug "tree_ssa_name"
                       "␣tvar=" tvar
                  "␣tvalue=" tvalue
                 "␣vers=" vers
                 "␣gdef=" gdef
                )
    (let ( (la (list)) )
              (walk_use_def_chain_depth_first
                 (lambda (val :tree tr :gimple gi)
                   (debug "lambda␣rhs1:␣val=" val
                          "␣tr=" tr
                     "␣gi=" gi)
                  (list_append la (make_tree discr_tree tr))
                            :true
                           )
                   'toto rhs1
                  )
    (debug "la=" la)
              (walk_use_def_chain_depth_first
                 (lambda (val :tree tr :gimple gi)
                   (debug "lambda␣lhs:␣val=" val
                          "␣tr=" tr
                     "␣gi=" gi)
                          :true
                        )
                   'toto lhs
                  )
    )
         )
         (?_
            (void))
       )
     )
          )
       )
     (?_
       (debug "g=" g)
     )
     )
  )
  (return :true g)       ;continue the walk on other gimples
)

(defun signedchar_pass_exec (pass)
  (with_cfun_decl
   ()
   (:tree cfundecl)
   (debug "char_pass_exec␣cfundecl=" cfundecl "␣pass=" pass)
   ;; that pass may be called several times for a given cfun, so we
   ;; register the cfun in a pass-global map
   (let ( (passmap (get_field :gccpass_data pass))
    )
     (if (maptree_get passmap cfundecl)
   (return))
     (maptree_put passmap cfundecl :true))
   ;; every basic block of the current cfun is interesting
```

```
   (each_bb_current_fun
    ()
    (:basic_block bb)
    (debug "char_pass_exec␣bb=" bb)
    (let (
    (:long bbix (basicblock_index bb))
    (:gimple_seq bbgs (gimple_seq_of_basic_block bb))
    )
      (debug "char_pass_exec␣before␣walking␣bbgs=" bbgs)
      (walk_gimple_seq_unique_tree pass bbgs
           char_gimple_walker
           ())
      (debug "char_pass_exec␣after␣walking␣bbgs=" bbgs)

      ))
   (debug "char_pass_exec␣end␣cfundecl=" cfundecl)
   ))

(defun signedchar_start_all_passes  ()
  (let ( (:tree chartypetree (tree_char_type_node))
         (:tree signedchartypetree (tree_signed_char_type_node))
         (:tree unsignedchartypetree (tree_unsigned_char_type_node))
   (chartypetreeval (make_tree discr_tree chartypetree))
   (signedchartypetreeval (make_tree discr_tree signedchartypetree))
   (unsignedchartypetreeval (make_tree discr_tree unsignedchartypetree))
   ;; (:tree ptrchartree (tree_pointer_type chartypetree))
   ;; (ptrchartreeval (make_tree discr_tree ptrchartree))
   ;; (:tree ptrinttree  (tree_pointer_type inttypetree))
       )
  (set_ref ref_char_type_tree chartypetreeval)
  (set_ref ref_signedchar_type_tree signedchartypetreeval)
  (set_ref ref_unsignedchar_type_tree unsignedchartypetreeval)
  (debug "chartypetree=" chartypetree)
  (debug "signedchartypetree=" signedchartypetree)
  (debug "unsignedchartypetree=" unsignedchartypetree)
  )
)

;; command and MELT mode for our signedchar typechecking analysis
(defun signedchar_docmd  (cmd moduldata)
  (debug "signedchar_docmd␣cmd=" cmd)
  ;; pour les headers
  ;; (register_finish_decl_hook_first signedchar_finish_decl_hook)
  (let (
  ;; for some reason, at this point, char_type_node is NULL in C
  (signedcharpass (instance class_gcc_gimple_pass
        :named_name '"signedchar_pass"
        ;; a map to register the visited cfuns
        :gccpass_data (make_maptree discr_map_trees 100)
        :gccpass_exec signedchar_pass_exec))
   )
    ;; at this early stage, (tree_char_type_node) gives a null tree. We
       need
    ;; to register a hook before all passes to initialize
    (register_all_passes_start_hook_first signedchar_start_all_passes)
    (install_melt_gcc_pass signedcharpass "before" "release_ssa" 0)
```

```
    ;; (install_melt_gcc_pass signedcharpass "after" "phiopt" 0)
    (debug "signedchar_docmd installed signedcharpass=" signedcharpass)
    (return :true)
    ))

(definstance signedchar_mode
  class_melt_mode
  :named_name '"signedchar"
  :meltmode_help '"type check signed/unsigned char for ARM, http://www.arm.
      linux.org.uk/docs/faqs/signedchar.php"
  :meltmode_fun signedchar_docmd
)
(install_melt_mode signedchar_mode)

;; eof signedchar.melt
```

Listing 5.1 – Exemple de code GCC MELT détectant des cas potentiels de risque de problème avec la manipulation de bits sur certains types de données pour la plateforme ARM

5.4.2.1 Projet Talpo

Ce projet fait suite aux travaux réalisés pendant le projet de fin d'étude de Pierre Vittet portant sur la prise en main de GCC MELT. Étant parvenu à un niveau de maitrise suffisant, il a jeté les bases d'une sorte de bibliothèque de tests de base pour la couverture de la `libc` et de certains appels systèmes : par exemple, lors d'un appel à la fonction `malloc()`, l'utilisateur est censé tester la valeur de retour, et si celle-ci est à `NULL` alors l'allocation n'a pas réussi et la mémoire n'est pas utilisable. Il est donc intéressant de vouloir tester que la valeur de retour de cette fonction est bien vérifiée par un test d'égalité. Intuitivement, ce besoin se généralise facilement à d'autres cas, et nous souhaitons pouvoir demander à vérifier le test de valeur de retour d'une fonction arbitraire.

5.4.2.2 Intégration dans un *buildsystem*

Une fois MELT fonctionnel, son intégration dans la construction de la distribution nécessite d'être en mesure de pouvoir modifier le compilateur utilisé ; cela se fait généralement par des variables d'environnement exportées. Un rapide état des lieux montre qu'une part non négligeable des sources des paquets ne respecte pas ces conventions. Il convient aussi de définir un ensemble de règles de codage que nous souhaitons appliquer au sein de la distribution. La réutilisation du projet Talpo serait une façon intelligente de réaliser cette intégration.

5.4.3 Validation de structures de données dynamiques avec Forester

Dans l'état de l'art nous avons vu qu'un problème souvent exposé mais non traité est celui des structures de données dynamiques : leur vérification est complexe à cause de l'arithmétique des pointeurs. Des auteurs proposent une approche dédiée à partir de forêts d'automates, et les différentes publications liées ont été présentées dans la section 2.13.3.2.

Leurs travaux sont disponibles à titre expérimental sous la forme, eux aussi, d'un greffon au compilateur GCC. Afin de pouvoir intégrer cet outil dans notre système, des contraintes similaires à celles de MELT existent, et il est nécessaire de qualifier son état d'avancement, et de documenter son utilisation. Un projet de fin d'étude a été confié à Cédric Vernou pour réaliser ces différentes tâches, et a conclu à un état encore beaucoup trop préliminaire de l'outil pour qu'il soit utilisable, en l'état, malgré une bonne collaboration de ses auteurs. Dans le cadre de ce projet de fin d'étude, il a été demandé à Cédric Vernou de produire une documentation et des exemples d'utilisation de la bibliothèque. Ces exemples ont été complétés et sont partagés [180] sous licence libre. L'expérience accumulée au cours du projet (difficultés, bogues rencontrés) et la documentation de l'utilisation ont été reversés aux auteurs originaux.

5.4.4 Qualité : avancées de la communauté

Parallèlement à la réalisation de notre prototype, différents acteurs de la communauté du logiciel libre se sont lancés, aussi, dans des démarches similaires. Nous avons d'abord la distribution Fedora qui a proposé un format de description standard des erreurs rapportées par les outils de vérification, que nous présentons en 5.4.4.1, et qui est notamment exploité dans le cadre d'un projet chez Debian présenté dans la section 5.4.4.2.

5.4.4.1 Format de description d'erreurs pour les outils d'analyse, analyse statique chez Fedora

Récemment, au moins à partir de décembre 2012 [5] [6], la distribution Fedora a lancé des travaux autour de l'analyse statique. Une contribution importante à l'heure actuelle est la proposition Firehose [7] : il s'agit d'une spécification proposée pour décrire les erreurs rapportées par les outils d'analyse. Le but est d'avoir un format simple et extensible permettant de remonter depuis les outils les erreurs détectées de manière standard. À plus long terme, et comme indiqué dans l'annonce sur la liste de diffusion de Fedora, l'objectif est clairement de mettre en place un système similaire à ce que nous avons proposé pour Mandriva.

Les caractéristiques principales de ce format sont :
- Proposition de standard commun, reposant sur XML (syntaxe *RelaxNG*) ;
- Localisation des notifications dans le code source : fichier, ligne, colonne, ... ;
- Classification des notifications : erreur ou avertissement, identifiant de problème de sécurité, adresse URL de description précise ;

5. confere l'historique de la page du Wiki https://fedoraproject.org/w/index.php?title=StaticAnalysis&action=history
6. confere le message sur la liste de diffusion https://lists.fedoraproject.org/pipermail/devel/2012-December/175232.html
7. https://github.com/fedora-static-analysis/firehose

5.4.4.2 Intégration continue d'outils de qualité logicielle chez Debian

À la suite de l'initiative lancée par Fedora et évoquée précédemment, des développeurs Debian ont travaillé sur la mise en place d'un outil similaire pour l'écosystème de la distribution. Ces travaux s'intègrent dans un cadre général d'amélioration de la qualité des paquets Debian et incluent également la reconstruction complète de la distribution avec un autre compilateur que GCC, LLVM/CLang : cela permet d'identifier des problèmes passés sous silence avec le premier, de bénéficier d'outils d'analyse et de vérification qui ne sont présents que dans le second, et aussi de s'assurer que tous les paquets peuvent être reconstruits avec un autre compilateur. Ces travaux reposent d'une part sur l'utilisation du format de description proposé par Fedora et que nous avons évoqué dans la section précédente, mais aussi exploitent des outils de l'état de l'art que nous avons évoqué : notamment, Coccinelle.

La mise en place de l'interface web a été réalisée par Matthieu Caneill, au cours d'un stage au sein de l'IRILL, et dans le cadre plus général d'une politique de vérification et d'amélioration de qualité appliquée sur Debian par l'institut, notamment au travers du projet Debil [8]. Son rapport [39] nous éclaire sur les motivations : développer une application permettant de présenter les résultats des analyses à plusieurs publics :

- Développeurs originaux, ceux qui écrivent le code intégré par les distributions. Ils sont intéressés à avoir un retour sur la qualité de leur code, à les aider à identifier les problèmes potentiels ;
- Développeurs Debian, qui sont responsables de la production et de la maintenance des paquets logiciels ; ils travaillent avec la base de code obtenue des développeurs originaux et peuvent avoir besoin d'y adjoindre des modifications locales. L'analyse du code leur permet de simplifier leur travail de maintenance et de leur apporter une assurance supplémentaire sur les différentes modifications effectuées ;
- Développeurs d'outils d'analyse, qui obtiennent ainsi une couverture importante pour leur outil, permettant d'identifier de nouveaux problèmes, d'améliorer la détection des faux positifs.

Deux composants ont été développés au cours du stage : une application de visualisation en ligne du code source des paquets, *Debsources* ; une interface graphique web à la bibliothèque et au format *Firehose*. Les caractéristiques et contraintes principales données de ces deux applications sont : recherche de paquets, navigation au sein des versions et du code source ; interface de programmation JSON. L'interface est fonctionnelle et permet d'avoir un début d'état des lieux de la distribution. L'utilisation de Coccinelle a été retenue comme illustration de la mise en application.

Par ailleurs, dans son rapport de stage, l'auteur présente aussi les conséquences de l'annonce publique, sur des canaux de diffusion de Debian et sur plusieurs sites web gravitant autour de la communauté, de la disponibilité de ses outils : en quelques heures, plusieurs dizaines de milliers de requêtes étaient envoyées, et la moyenne journalière s'établit à 6 000 une fois l'effet d'annonce passé.

Nous constatons donc que les travaux de la communauté autour de Debian rejoignent de très près les nôtre : le constat est très proche, les solutions proposées correspondent

8. http://debil.debian.net

aux mêmes besoins et à des contraintes similaires ; par exemple, nous avons aussi retenu d'utiliser Coccinelle comme illustration. Son déploiement effectif diffère un peu, puisque nous nous limitons à l'appliquer au noyau Linux en réutilisant l'infrastructure déjà présente, là où les développeurs Debian ont décidé d'extraire un sous-ensemble de règles de Coccinelle et de les appliquer sur plus de paquets.

Chapitre 6

Conclusion

Nous allons à présent revenir sur les différents objectifs posés au cours de ces travaux, et faire le point pour chacun sur nos apports, contributions ainsi que sur les résultats que nous présentons.

6.1 Vérification de code noyau

Lors de la réalisation de cet état de l'art, nous avions pour objectif de documenter les techniques et les outils existants permettant de faire de la vérification sur le code source qui puissent s'appliquer à la distribution Mandriva.

6.1.1 Documentation de l'état de l'art

La constitution de cet état de l'art permet à la distribution d'avoir connaissance non seulement des techniques qui sont exploitées par les différents outils existant, mais surtout de leurs limites. Nous avons pu voir que l'utilisation de model-checking sur un code source volumineux est encore problématique et nous proposons d'exploiter des techniques de détection de communautés sur un graphe construit à partir des sources du noyau afin d'être en mesure de pouvoir le découper de manière pertinente.

6.1.2 Outils utilisables

Au travers de cet état de l'art, nous avons également documenté l'existence de projets libres qui chacun visent à améliorer la qualité du code source utilisé tous les jours par la distribution. Nos axes de recherche nous ont permis de nous focaliser principalement sur le code du noyau Linux, les outils utilisables sont donc taillés et pensés pour celui-ci. L'étude de la distribution en elle-même nous permet de valider l'utilisation de ces outils pour une partie importante du reste des paquets, ceux qui partagent le langage commun que nous analysons, le `C`.

Un résultat important réside dans le constat, obtenu après avoir étudié l'état de l'art et avoir échangé avec des experts (mainteneurs du noyau Linux, employés Mandriva tra-

vaillant à la maintenance de la distribution), de l'existence des outils de validation et de vérification et de leur connaissance par ces acteurs. Cette connaissance, malheureusement, ne se transforme pas automatiquement en utilisation. Un outil inutilisé par ses utilisateurs destinataires est inutile.

6.2 Cohérence du code source

L'établissement de l'état de l'art de l'utilisation de model-checking sur des noyaux a mis en lumière, entre-autres, la proéminence toujours présente du problème de l'explosion combinatoire : le nombre d'états à analyser reste toujours un problème. Nous avons proposé d'évaluer la faisabilité d'une approche exploitant des techniques de détection de communautés pour construire des ensembles de taille connues voire contrôlables et dont les interactions entre eux seraient également pleinement identifiés.

6.2.1 Graphe du noyau

Nous avons mis en place et évalué une représentation du code source du noyau en vue de son découpage et de son analyse par des techniques de détection de communautés : un graphe orienté dont les sommets correspondent aux fichiers objets obtenus lors de la compilation, et dont les arrêtes sont composées des symboles utilisés (import, export) entre ces unités de compilation. L'étude de cette première représentation du noyau a permis d'établir quelques résultats notamment sur la composition et la taille du graphe. Une méthode de visualisation claire des liens est proposée avec les cartes de chaleur qui permettent de repérer les points chauds d'utilisation.

6.2.2 Documentation de l'état de l'art

Nous proposons une lecture de l'état de l'art [66] de Fortunato suivant notre problème afin de savoir quelles sont les approches déjà existantes que nous pouvons appliquer en espérant de bons résultats et comprendre leurs limites pour proposer des améliorations. Une dizaine d'algorithmes proposant des implémentations disponibles ont ainsi été comparés sur notre cas selon des critères définis et adaptés au problème.

6.2.3 Arborescence du noyau

L'étude de l'arborescence du noyau a été proposée à la suite de l'étude du graphe et comme une application directe des résultats de détection de communauté orientée vers les besoins des mainteneurs et des développeurs : il s'agit de les aider à s'assurer et vérifier que la répartition du code source en arborescence est cohérente avec l'utilisation qui est faite par ce même code.

Les résultats de l'utilisation des différentes méthodes de détection de communautés sélectionnées dans l'état de l'art montrent qu'une partie de ces méthodes permet de mettre en évidence des communautés qui correspondent à différents niveaux de profondeur de l'arborescence du système de fichier.

6.2.4 Communautés contraintes

L'autre axe d'analyse de la détection de communautés appliquée au noyau visait à permettre la construction d'ensembles contrôlables en taille et en interconnexions dans le but d'améliorer les outils d'analyse statique. Les résultats de la méthode que nous avons pu essayer suite à l'étude de l'état de l'art ne sont pas probant : elle n'est pas en mesure de détecter le nombre de communautés que nous souhaitons, et le nombre de communautés qui sont détectées reste toujours le même. Certaines des méthodes que nous avons évaluées dans le cadre de la validation de la cohérence du code source se sont montrées plus performantes sur ce plan et sont capables de détecter un nombre de communautés beaucoup plus intéressant dans le cadre de ce type d'analyses : celles qui sont ainsi construites varient de 2-3 nœuds à une vingtaine, suivant les algorithmes et les instances étudiées. Cependant, ces approches ne prennent pas en charge la contrainte que nous voulons imposer, qui est de pouvoir contrôler leur nombre et leur taille.

6.3 Contributions pour Mandriva/Mageia

La distribution Mageia sert à présent de base à Mandriva Business Server, il est donc légitime que nous fassions le point sur les apports et contributions vis-à-vis de celle-ci.

6.3.1 Documentation des méthodes et outils de validation de code source

Au travers de l'état de l'art que nous avons réalisé et constitué, les développeurs de la distribution peuvent bénéficier de notre retour d'expérience, et notamment avoir les bases et les références nécessaires pour rapidement comprendre, découvrir et prendre en main efficacement les techniques et les outils de détection de bogues et de vérification de code que nous intégrons.

6.3.2 Intégration d'outils d'analyse dans la distribution

Nous avons proposé et réalisé l'intégration au sein de la distribution de différents outils de l'état de l'art permettant de vérifier du code et détecter des problèmes. Cela concerne notamment Coccinelle et Undertaker. Leur mise à disposition dans la distribution, couplée à la documentation de leur utilisation et à la justification du besoin permet de s'assurer que ces outils continuerons à être maintenu et à évoluer dans la distribution.

6.3.3 Intégration d'outils d'analyse et qualité dans le système de construction

Une fois intégrés dans la distribution les différents outils d'analyse proposés, il devient possible d'en faire plus facilement usage dans le système de construction : ils peuvent être installés comme n'importe quel paquet, donc sans avoir à intervenir spécifiquement sur l'infrastructure. Nous avons proposé un système d'analyse automatisé reposant sur

ces outils, s'exécutant en parallèle du système de construction et étant *branché* sur celui-ci. Ceci nous permet d'introduire progressivement l'utilisation et la prise en main à la fois des outils et surtout l'interprétation de leurs résultats auprès des développeurs de la distribution.

6.4 Perspectives d'évolution

Nous allons maintenant faire un point sur les évolutions attendues à la suite de ces travaux, qui concernent principalement la mise en application et la prise en main par les utilisateurs finaux.

6.4.1 Intégration et extension de Talpo

Les travaux autour de GCC et plus précisément MELT visaient à mettre en place une bibliothèque standard de tests et de vérification que nous pourrions inclure dans les différentes étapes de construction de la distribution en intégrant un ensemble de règles construites à partir de l'expérience : connaissance des experts (mainteneurs et développeurs), analyse de la typologie des problèmes rencontrés (via les outils de suivis de bogues).

Une partie de ce travail a été initié à la fin de son projet de fin d'étude par Pierre Vittet en fondant le projet Talpo : une preuve de faisabilité techniquement avancée est fournie, il conviendrait de passer à une industrialisation de la solution. Ceci impliquera de pérenniser le greffon GCC MELT, qui représente la limite principale de cette approche.

6.4.2 Application générale de Coccinelle

Coccinelle est un autre outil, à l'origine destiné à appliquer des transformations en masse sur une base de code, et maintenant exploité pour décrire et détecter des problèmes. De par son objectif, il se rapproche de GCC MELT et donc de Talpo, cependant, la mise en œuvre diffère et Coccinelle est implémenté comme outil indépendant. Un jeu de règles de vérification existe déjà et est intégré au sein du noyau Linux, pour vérifier le code quant à des problèmes courants.

Plusieurs travaux ont déjà été menés pour mettre en œuvre Coccinelle sur d'autres bases de code, et aucune difficulté technique ne s'y oppose, à part l'écriture des correctifs sémantiques. Intégrer ce type de vérification lors de la construction des paquets permettrait d'améliorer globalement la qualité du code en traquant les problèmes déjà connus mais là où personne ne pense à aller voir. De plus, comme dans le cas de Talpo, l'effort bénéficie à la fois à la distribution et aux projets.

Cette généralisation ne se limiterait, dans un premier temps, qu'à la base de code écrite en langage `C` ; bien que conséquence comme nous l'avons documenté précédemment, elle est cependant loin de représenter la totalité ni la majorité du code de la distribution.

6.4.3 Détection de communautés

L'étude de l'utilisation d'outils de l'état de l'art de détection de communautés appliqués sur un graphe du noyau Linux a mis en lumière, d'abord, que certaines méthodes permettent effectivement de détecter des communautés proches de ce que le système de fichier regroupe ; mais surtout, nous avons constaté que la possibilité de détecter des communautés d'une taille définie ne fonctionne pas : la seule méthode correspondant à ce besoin et proposant une implémentation donne de très mauvais résultats. Une première perspective serait donc de travailler sur ce sujet afin d'évaluer d'autres approches de l'état de l'art qui ne proposeraient pas d'implémentation. Dans un second temps, il convient d'exploiter également les résultats obtenus sur la thématique de la vérification de l'arborescence : par exemple, la mise en place de métriques pour assurer un suivi dans le temps et éventuellement alerte en cas de divergence.

Par ailleurs, la construction du graphe permet de connaître les liens entre les différents modules ; un approfondissement de ces travaux avec une mise en correspondance du code source permettrait de travailler sur un outil d'aide à la maintenance à destination des développeurs. Notons que cette approche n'est pas dépendante de la manière de construire le graphe, et simplement une analyse du code source suffirait.

6.4.4 Intégration et extension d'Undertaker

Quelques logiciels permettent une configuration complexe à la construction, c'est notamment le cas du noyau, pour lequel Undertaker a été écrit. Rien ne s'oppose à son application à d'autres bases de code, d'autant plus que contrairement à GCC MELT et Coccinelle, la base de code en langage `C++` est accessible. La limitation principale de l'utilisation plus générale de cet outil provient simplement de son objectif : la validation de la cohérence entre les options de configurations disponibles et leur implémentation. L'effort nécessaire pour l'adapter à plusieurs projets n'est peut-être pas totalement pertinent, si ceux-ci sont assez petits et simples à tester.

6.4.5 Plateforme de qualité ouverte : « Open Quality Platform »

L'intégration d'outils de qualité au sein d'un système de construction est particulièrement utile pour simplifier la vie des mainteneurs de celle-ci. En généralisant l'application à la totalité de la distribution, il parait naturelle d'étendre encore plus la portée de l'utilisation et d'envisager la mise en place d'une plateforme de qualité, qui viserait à aider n'importe qui à faire valider son code source selon des règles communément admises et automatique. Des services similaires existent déjà pour la mise à disposition d'infrastructures de tests unitaires, que ce soit sous forme hébergée ou installée. Étendre le concept dans un outil spécialisé serait la suite logique du prototype que nous avons mis en place et qui vise à servir de démonstrateur de cette approche.

Annexes

Annexe A

Analyse des graphes des noyaux

Cette annexe présente le détail des résultats de l'étude des graphes construits à partir du code source du noyau. Dans la section 3.3, un ensemble de mesures qui nous paraissent pertinentes ont été définies. Nous reprenons chacune et rappelons succinctement leur rôle avant de documenter les résultats obtenus. Les instances étudiées étant séparées entre les deux cas : **defconfig** et **allyesconfig**. Nous présentons les éléments importants résultants de cette étude dans la section 3.4. Ces résultats détaillés ont servis pour la réalisation des communications [103, 99].

A.1 Taille de la base de code

Dans cette section nous allons présenter des premiers résultats d'analyse de la topologie du noyau : d'abord, en regardant la quantité de code disponible telle que rapportée par l'outil `SLOCCount` ; puis en regardant le graphe, en comptant les arcs et les sommets.

A.1.1 Quantité de code

L'analyse avec `SLOCCount` vise simplement à donner une idée de la quantité de code qui constitue le noyau. La granularité restera globale dans le paragraphe A.1.1.1 et permettra surtout de comparer l'évolution dans le temps avec les versions 3.0 (juin 2011) à 3.9 (avril 2013). De plus, par définition, la comparaison entre les instances **defconfig** et **allyesconfig** n'a pas de sens, puisque le comptage effectué dans les deux cas se portera sur les mêmes fichiers sources, les résultats seront donc strictement identiques. Par la suite, dans le paragraphe A.1.1.2 une vue éclatée sur certains sous-répertoires avec l'évolution dans le temps permettra d'avoir plus de précisions.

A.1.1.1 Aperçu sur tout le noyau

Les résultats sont présentés dans le graphique disponible en figure A.1. Il convient dans un premier temps de constater que le noyau Linux, dans ces versions, contient de l'ordre de 10 millions de lignes de codes : la version 3.0 comporte de l'ordre de 9.6 millions de

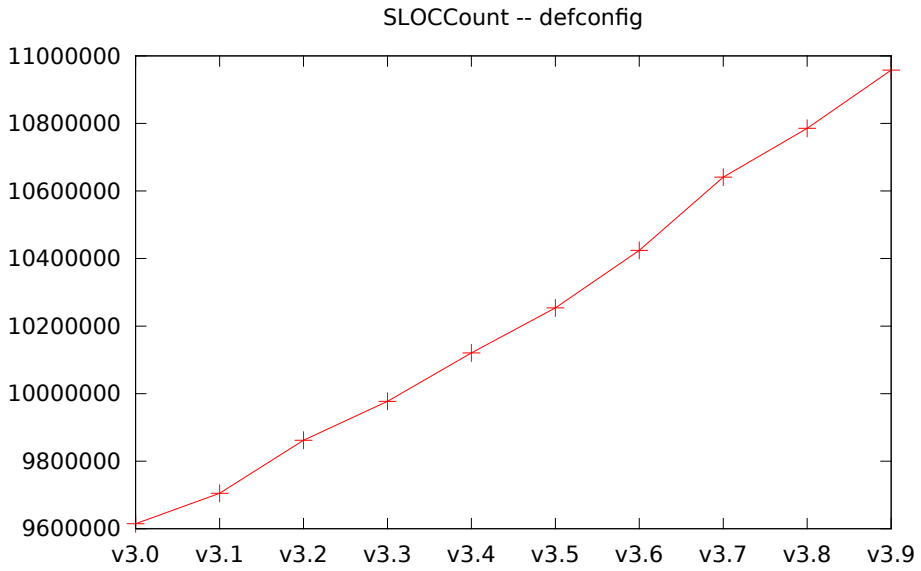


FIGURE A.1 – Évolution de la quantité de code

lignes de code, alors que la version 3.9 en contient 10.95 millions. La pente de la courbe, également, montre que l'inflation est constante.

A.1.1.2 Détail par sous-répertoire

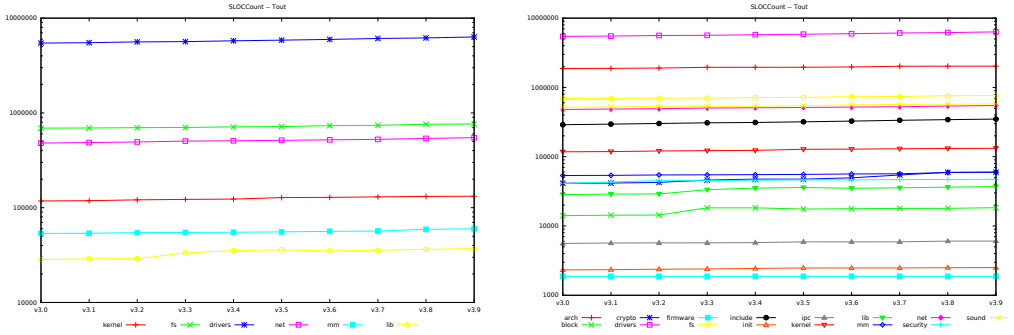
Après avoir vu globalement la quantité de code présente dans le noyau, le graphique A.2a propose un détail sur certains sous-répertoires, ceux qui seront analysés par la suite dans la section A.2, à savoir : `kernel/`, `fs/`, `drivers/`, `net/`, `mm/` et `lib/`. Une vision globale avec tous les sous-répertoires est proposée dans le graphique A.2b. Une première observation permet de constater, notamment dans ce dernier graphique, que la moitié des sous-répertoires contiennent moins de 100 000 lignes de code. Et seulement deux, `drivers/` et `kernel/` dépassent le million.

Une autre information intéressante fournie par ces deux graphiques est qu'aucun sous-répertoire ne voit sa dimension diminuer sur l'intervalle de versions sélectionné.

A.1.2 Taille du graphe

À présent nous nous intéressons à quantifier le nombre de nœuds et d'arcs dans chaque graphe construit, ainsi qu'à l'influence des sous-répertoires sur ces chiffres ? Des graphiques sont proposés en figures A.3a (sous-ensemble de répertoires, instance **defconfig**), A.3b (sous-ensemble de répertoires, instance **allyesconfig**), A.3c (totalité des répertoires, instance **defconfig**), A.3d (totalité des répertoires, instance **allyesconfig**).

Dans un premier temps, la comparaison entre l'instance **defconfig** et **allyesconfig** sur le sous-ensemble de répertoires sélectionné met en lumière l'effet de l'activation de cette option : les échelles, que ce soit pour les sommets ou les arcs, augmentent ; celle des



(a) Sous-ensemble (b) Totalité

FIGURE A.2 – Évolution de la quantité de code. (échelle log.)

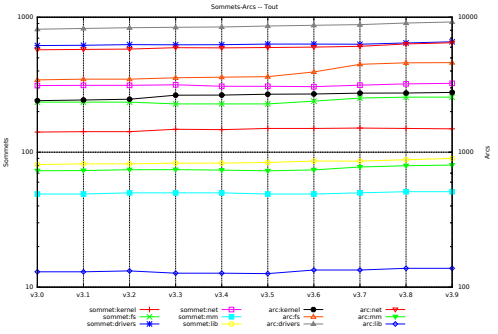
sommets est multipliée par 10 alors que celle des arcs est multipliée par 100. Cependant, cette hausse n'est pas uniforme : le sous-répertoire `kernel/` reste assez stable en matière de sommets – passant d'environ 150 à environ 200 – mais croît en terme d'arcs, le nombre de ceux-ci étant multiplié par 2. L'exemple du répertoire `fs/` est par contre plus parlant, puisque le nombre de sommets se retrouve multiplié par 2 alors qu'il y a 10 fois plus d'arcs dans l'instance **allyesconfig**. De plus, toujours sur ce répertoire, il est possible de constater à l'instant des versions 3.5 à 3.7 une augmentation nette du nombre d'arcs et de sommets, qui ne se produit que dans l'instance **defconfig**.

Le sous-répertoire `drivers/` montre un comportement similaire : alors que l'instance **defconfig** contient de l'ordre de 600 sommets et 8500 arcs, le passage à l'instance **allyesconfig** implique une augmentation d'un facteur 10 dans les deux cas. L'influence de l'instance est bien moins marquée dans le cas du sous-répertoire `init/`, celui-ci ne présente aucune évolution : un seul changement notable, une chute du nombre de symboles dans l'instance **defconfig** à partir du noyau 3.7. Ce comportement confirme que l'activation de l'instance **allyesconfig** a des conséquences qui dépendent, principalement, des pilotes et des sous-systèmes utilisés.

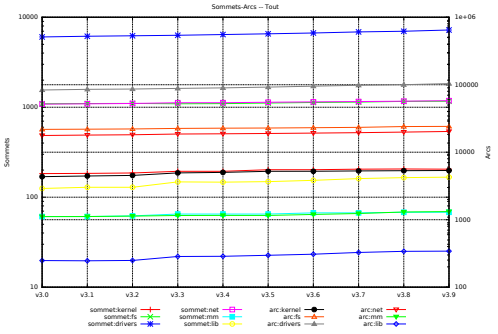
Le sous-répertoire `lib/` est intéressant car il montre une nette progression du nombre de sommets et d'arcs sur l'intervalle de versions sélectionné, mais uniquement dans l'instance **allyesconfig** : l'instance **defconfig** ne passe que de 81 sommets à 90 quand la première voit une progression de 125 sommets à 167.

Ces graphiques montrent également, à l'instar des résultats avec `SLOCCount` présentés auparavant, une tendance constante à la croissance du noyau. Le graphe complet correspondant au noyau dans son instance **defconfig** comporte de l'ordre 1900 sommets au total, variant de 1858 dans la version 3.0 à 1981 dans la version 3.9. Sur la même instance, et sur le même intervalle de versions, le nombre d'arcs évolue de 52354 à 58702. Dans le cas de l'instance **allyesconfig**, ces chiffres sont respectivement multipliés par environ 6 pour le nombre de sommets, et environ 7 pour le nombre d'arcs : ainsi la version 3.9 présente plus de 400000 arcs à analyser.

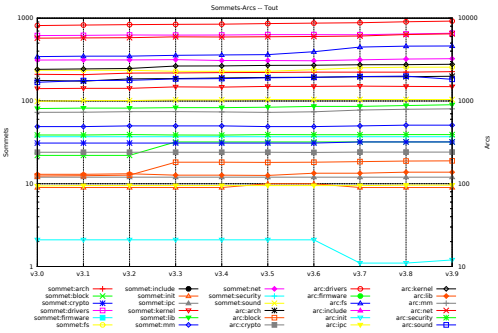
Les prochaines sous-sections, en particulier A.3.1, permettront de déduire un peu plus d'informations sur ces répertoires en étudiant la densité du graphe en leur sein. Au plus



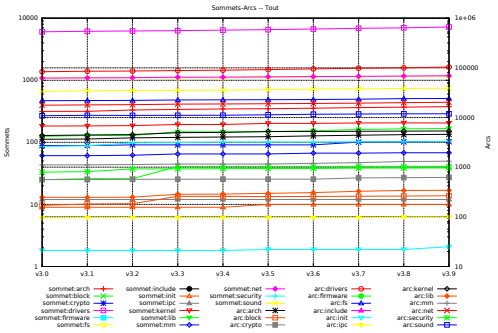
(a) Sous-ensemble, **defconfig**



(b) Sous-ensemble, **allyesconfig**

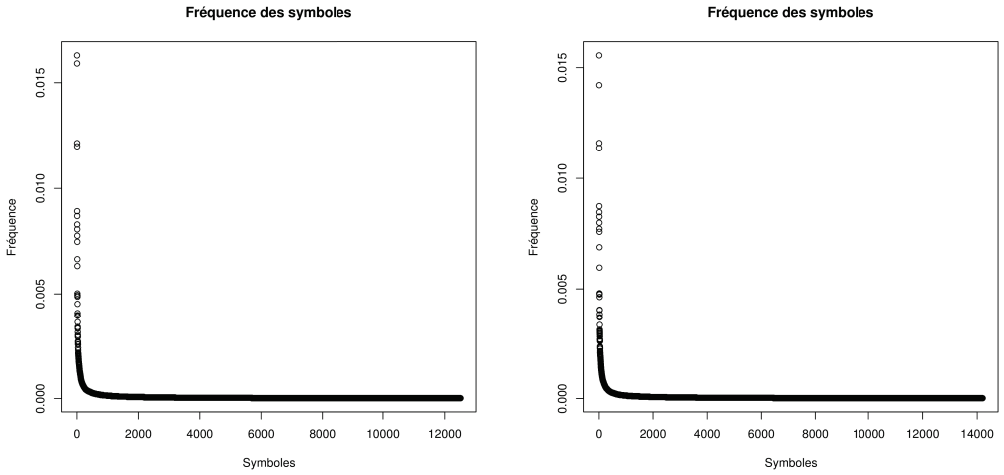


(c) Totalité, **defconfig**



(d) Totalité, **allyesconfig**

FIGURE A.3 – Évolution de la taille du graphe. (échelle log.)



(a) Noyau v3.0 (b) Noyau v3.9

FIGURE A.4 – Distribution des symboles sur l'instance **defconfig**

le graphe sera dense, au plus ses éléments seront interdépendants.

A.2 Résultat : occurrence des symboles

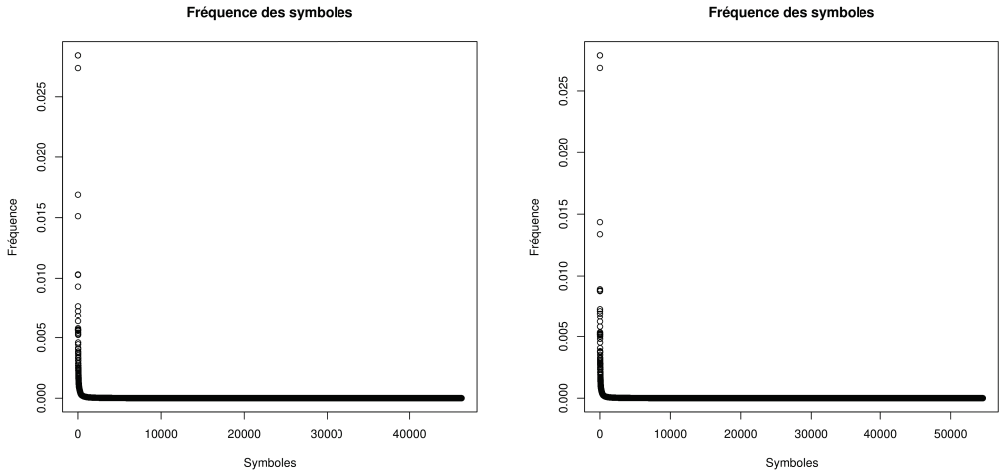
Dans cette sous-section, nous présentons une analyse sur les symboles présents dans le graphe. Dans un premier temps, nous nous intéresserons à étudier la distribution des symboles suivant leur fréquence d'apparition. Par la suite, nous regarderons les symboles qui sont les plus usités. L'analyse des plus gros exportateurs et/ou importateurs de symboles sera faite dans la sous-section A.5.2.

A.2.1 Distribution des symboles

La distribution des symboles a été étudiée avec les versions 3.0 à 3.9 du noyau Linux, et les graphiques pour certaines sont disponibles ci-après. Notamment, pour l'instance **defconfig**, le noyau 3.0 est visible en figure A.4a et le 3.9 est visible en figure A.4b. Sur ces deux graphiques, les symboles sont présents en abscisse (sans leur nom, seulement un identifiant numérique) et la fréquence est en ordonnée, calculée en prenant le nombre d'arc apparaissant dans le graphe divisé par la totalité des arcs du graphe. Deux différences « majeurs » marquent l'évolution temporelle entre le noyau 3.0 et le 3.9 : le nombre de symboles augmente, et ceux qui sont les plus fréquents voient leur fréquence changer légèrement. La distribution reste très similaire, et surtout, il est nettement visible qu'un petit nombre de symboles apparaissent beaucoup plus que les autres.

La comparaison avec l'instance **allyesconfig** dont le noyau 3.0 est visible en figure A.5a et le 3.9 est visible en figure A.5b confirme ces tendances :

– Les fréquences pour les symboles les plus usités sont similaires



(a) Noyau v3.0 (b) Noyau v3.9

FIGURE A.5 – Distribution des symboles sur l'instance **allyesconfig**

– Dans le temps, le nombre de symboles augmente
– Quelques symboles concentrent les usages

Les symboles les plus utilisés seront analysés dans la section suivante, A.2.2.

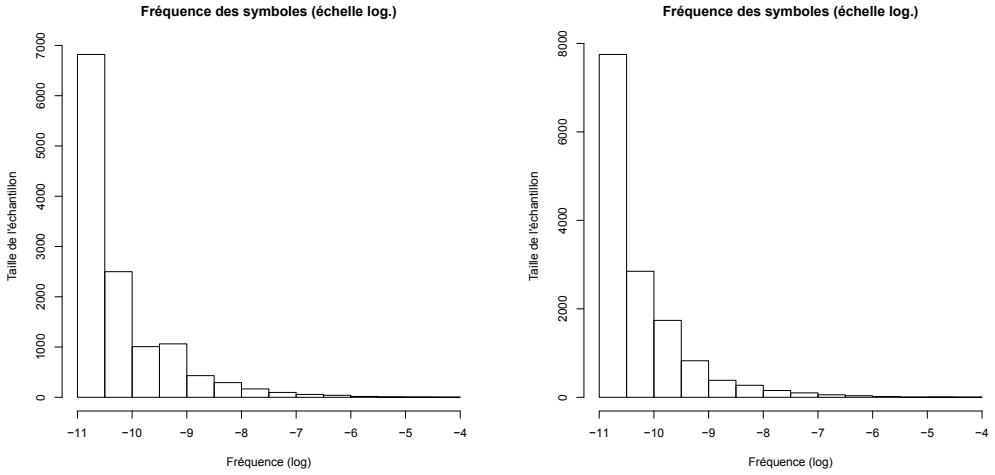
Une autre visualisation de la distribution de l'utilisation des symboles est proposée également dans les figures A.6a, A.6b, A.7a et A.7b sous la forme d'un histogramme avec une échelle logarithmique. La fréquence donnée en abscisse est celle définie ci-après au paragraphe A.2.2.1, et ses valeurs étant assez faibles et proches de zéro, cela explique le $log()$ négatif. Sur l'instance **defconfig**, ces graphiques montrent une répartition similaire, bien que le nombre de symboles soit croissant, puisque les fréquences les plus faibles regroupent une écrasante majorité des symboles : sur un total, de plus de 12500, respectivement 14200, symboles environ pour le noyau 3.0, respectivement 3.9, ceux dont la fréquence est $<= -10$ sur l'histogramme représentent une population de 9500 membres, respectivement 1000 membres environ.

Une comparaison entre les versions 3.0 et 3.9 du noyau, grâce aux graphiques A.6a et A.7a permet de constater, là encore, l'effet de l'activation de tous les modules et pilotes : l'échelle logarithmique pour l'abscisse passe d'une borne inférieur de -11 et descend en dessous de -12 : l'importance relative des symboles diminue fasse à leur quantité.

A.2.2 Symboles les plus utilisés

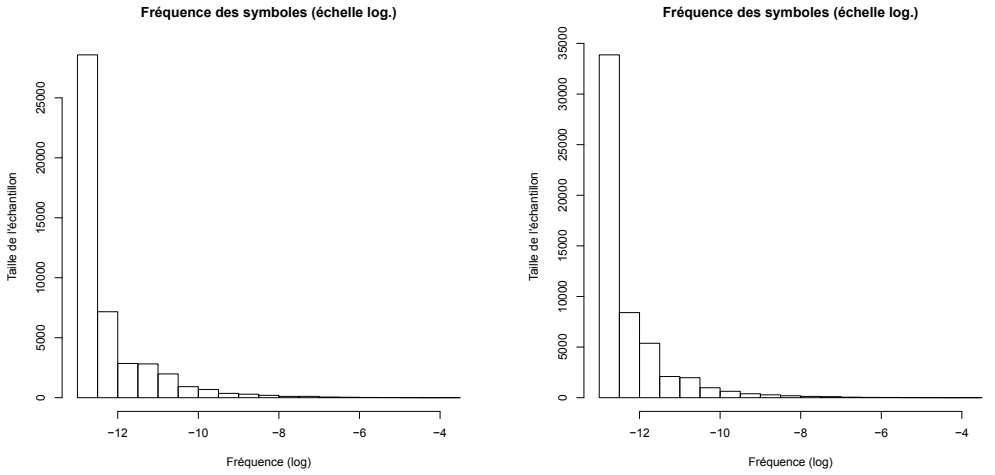
A.2.2.1 Définition de la fréquence

Précédemment, dans la section A.2.1, nous avons constaté une distribution des symboles intéressante. Maintenant nous nous intéressons à étudier quels sont ces symboles les plus courants. Dans les tableaux A.2a, A.2b, A.3a et A.3b sont regroupés les résultats pour les instances **defconfig** et **allyesconfig** des noyaux 3.0 et 3.9. Seuls les 20 premiers



(a) Noyau v3.0 (b) Noyau v3.9

FIGURE A.6 – Histogramme des symboles sur l'instance **defconfig**, échelle logarithmique



(a) Noyau v3.0 (b) Noyau v3.9

FIGURE A.7 – Histogramme des symboles sur l'instance **allyesconfig**, échelle logarithmique

Affichage	Mémoire	Verrouillage
printk()	kfree()	mutex_unlock()
sprintf()	kmalloc_caches()	mutex_lock()
	kmem_cache_alloc_trace()	_raw_spin_lock()
	__kmalloc()	_raw_spin_unlock_irqrestore()
	memcpy()	_raw_spin_lock_irqsave()
	_copy_to_user()	
	_copy_from_user()	

TABLE A.1 – Grandes familles de symboles, avec quelques exemples

symboles d'un point de vue fréquence sont considérés. Par définition, la fréquence d'un symbole S dans un graphe $G = (V, E)$ est :

$$f_S = \sum_{e \in E} \frac{x_e^S}{|E|}$$

Avec, $x_e^S = 1$ si l'arc e porte le symbole S, 0. La somme de ces fréquences est donc par définition de 1.

A.2.2.2 Analyse de la fréquence des meilleurs symboles

La dernière ligne correspond à la somme des fréquences de ce « Top 20 des symboles », et permet d'avoir une indication de l'importance de ceux-ci par rapport à la globalité. Sur cette donnée, il est possible de comparer, dans un premier temps, les deux versions 3.0 et 3.9 du noyau, respectivement dans les tableaux A.2a et A.3a ou A.2b et A.3b, et de constater une légère baisse d'importance à la fois sur les instances **defconfig** et **allyesconfig**, de l'ordre de 0.006 points. Cette différence pointe l'inflation du nombre de symboles.

Un second résultat peut être tiré de la comparaison entre les instances **defconfig** et **allyesconfig** grâce à ce total, dans les tableaux A.2a et A.2b : la fréquence cumulée des 20 premiers symboles montre une différence de l'ordre de 5% en faveur de **allyesconfig**. Cette tendance se retrouve quelle que soit la version du noyau considérée, comme cela est visible pour le noyau 3.9 dans les tableaux A.3a et A.3b. Cela signifie que cette liste de symboles se retrouve plus présente dans la seconde instance, laquelle représente principalement l'ajout de tous les pilotes et modules possibles du noyau.

A.2.2.3 Analyse des symboles les plus utilisés

La comparaison des différents tableaux A.2a, A.2b, A.3a et A.3b est également intéressante si du point de vue des symboles eux-mêmes. Le premier constat est flagrant : les symboles les plus utilisés sont plus ou moins les mêmes, à quelques variations (sur l'ordre lié à la fréquence) près. Il est possible, de plus, de constituer plusieurs familles. Celles-ci sont regroupées dans le tableau A.1.

Symbole	Fréquence
kfree	0.016293
printk	0.015911
kmalloc_caches	0.012129
kmem_cache_alloc_trace	0.011976
mutex_unlock	0.00892
mutex_lock	0.008691
current_task	0.00829
__kmalloc	0.008061
_cond_resched	0.007755
_raw_spin_lock	0.007468
warn_slowpath_null	0.006647
memcpy	0.006322
_raw_spin_unlock_irqrestore	0.005023
jiffies	0.004947
kernel_stack	0.004928
_raw_spin_lock_irqsave	0.004871
sprintf	0.004527
_copy_to_user	0.004088
strlen	0.003992
_copy_from_user	0.003954
- Total -	0.154792

(a) Noyau v3.0, **defconfig**

Symbole	Fréquence
__gcov_merge_add	0.028411
__gcov_init	0.028411
mcount	0.027367
printk	0.016886
kfree	0.01512
kmalloc_caches	0.010324
kmem_cache_alloc_trace	0.010283
__stack_chk_fail	0.009319
__kmalloc	0.007697
mutex_unlock	0.007295
mutex_lock_nested	0.006878
dev_get_drvdata	0.006366
__raw_spin_lock_init	0.005769
_raw_spin_unlock	0.005668
_raw_spin_unlock_irqrestore	0.005662
_raw_spin_lock	0.005633
_raw_spin_lock_irqsave	0.00563
warn_slowpath_null	0.005603
dev_set_drvdata	0.005547
jiffies	0.005402
- Total -	0.219271

(b) Noyau v3.0, **allyesconfig**

TABLE A.2 – Utilisation des symboles – Noyau 3.0 – Top 20

Ces trois familles de fonctions se retrouvent dans les instances **defconfig** et **allyesconfig** ; cela étant, la seconde instance présente quelques spécificités : les symboles `GCOV` prennent la tête du classement, et les primitives de manipulation de données propres aux pilotes apparaissent (`dev_get_drvdata()` et `dev_set_drvdata()`), ce qui confirme l'impact des pilotes dans cette instance. `GCOV` est un outil de `GCC` qui permet de tester la couverture du code du programme. La présence de ces symboles est un artefact qui provient du choix de tout activer dans cette instance ; l'option `CONFIG_GCOV_KERNEL` est activée, `GCOV` est donc rajouté. Il en est de même pour les symboles `__fentry__()` (pour le noyau 3.9 dans le tableau A.3b) ou `mcount()` (pour le noyau 3.0 dans le tableau A.2b) qui proviennent de l'activation de certaines options du module `FTrace`, qui permet de faire du traçage au sein du noyau (quelles fonctions sont appelées, changements de contextes, durée du masquage des interruptions, etc.).

Ce résultat amène donc à tempérer les conclusions du paragraphe A.2.2.2 où nous indiquions une inflation de l'ordre de 5% en faveur de l'instance **allyesconfig** : elle est due à l'effet conjugué de `GCOV` et `FTrace`. Ces deux options modifient la fréquence cumulée de $0.028411 * 2 + 0.027367 = 0.084189$ (pour le noyau 3.0 dans le tableau A.2b, les valeurs sont sensiblement proches sur le cas 3.9 dans le tableau A.3b). La fréquence cumulée totale corrigée devient donc : $0.219271 - 0.084189 = 0.135082$, reflétant ainsi que le noyau est plus « large » dans cette instance.

A.2.3 Symboles par sous-répertoires

La section A.2.2 nous a permis d'obtenir un premier aperçu de la distribution des symboles, et de voir quels sont ceux qui sont les plus utilisés par la totalité du noyau Linux, et notamment si l'on retrouve les trois familles identifiées précédemment dans le paragraphe A.2.2.3 : affichage, gestion mémoire et primitives de verrouillage. Maintenant nous allons regarder plus en détails l'utilisation des symboles, mais par sous-répertoire ; plus précisément, certains répertoires « bien choisis » :

Symbole	Fréquence
kfree	0.015553
printk	0.014207
kmalloc_caches	0.011584
kmem_cache_alloc_trace	0.011362
mutex_unlock	0.008739
mutex_lock	0.008466
warn_slowpath_null	0.008262
current_task	0.00799
_cond_resched	0.007717
__kmalloc	0.007581
_raw_spin_lock	0.006882
memcpy	0.005962
kernel_stack	0.004804
jiffies	0.004787
_raw_spin_unlock_irqrestore	0.004753
_raw_spin_lock_irqsave	0.004634
sprintf	0.004054
snprintf	0.004037
strlen	0.00385
_copy_to_user	0.003714
- Total -	0.148939

(a) Noyau v3.9, **defconfig**

Symbole	Fréquence
__gcov_merge_add	0.027876
__gcov_init	0.027876
__fentry__	0.026862
printk	0.014353
kfree	0.013376
__stack_chk_fail	0.008929
kmalloc_caches	0.008827
kmem_cache_alloc_trace	0.008779
mutex_unlock	0.007316
__kmalloc	0.007149
mutex_lock_nested	0.006937
dev_get_drvdata	0.006628
warn_slowpath_null	0.006214
dev_set_drvdata	0.005793
__raw_spin_lock_init	0.005337
dev_err	0.005315
_raw_spin_unlock_irqrestore	0.005227
_raw_spin_lock_irqsave	0.0052
_raw_spin_unlock	0.005138
__list_add	0.005128
- Total -	0.208262

(b) Noyau v3.9, **allyesconfig**

TABLE A.3 – Utilisation des symboles – Noyau 3.9 – Top 20

– `kernel/` : ce répertoire contient des éléments de base du noyau ;
– `fs/` : ce répertoire concerne tous les systèmes de fichiers ;
– `drivers/` : ce répertoire contient une grosse partie des pilotes de périphériques
– `net/` : ce répertoire contient à la fois les piles réseaux et des pilotes réseaux ;
– `mm/` : ce répertoire concerne la gestion du sous-système mémoire du noyau ;
– `lib/` : ce répertoire contient des bibliothèques mises à disposition du reste du noyau ;

Les tableaux présentés sont construits de la même manière que dans la section précédente, en se limitant au sous-répertoire analysé ; ainsi, la fréquence est calculée sur l'ensemble des symboles qui vivent dans ce sous-répertoire.

A.2.3.1 Analyse du noyau 3.0, defconfig

Dans un premier temps, intéressons-nous à l'instance **defconfig**, dont les résultats par sous-répertoires sont disponibles dans les tableaux A.4a, A.4b, A.4c, A.4d, A.4e et A.4f pour le noyau dans sa version 3.0.

Un premier résultat montre la disparité de distribution des symboles : en ne retenant que le top 20 des plus utilisés, on constate que certains répertoires montrent une fréquence cumulée sur cette liste bien différente. Ainsi, les répertoires `fs/` (A.4b), `net/` (A.4d) et `drivers/` (A.4c) sont à un total compris entre 0.128 et 0.143, là où le répertoire `kernel/` (A.4a) est à 0.32 et les répertoires `mm/` (A.4e) et `lib/` (A.4f) dépassent un total de 0.50. Ce résultat confirme que les répertoires `fs/`, `drivers/` et `net/` possèdent une plus grande diversité de symboles. Les grandes lignes de ces résultats se retrouvent également sur l'instance **allyesconfig**, comme cela est visible dans les tableaux A.5a, A.5b, A.5c, A.5d, A.5e et A.5f, même si certaines valeurs évoluent (notamment sur les sous-répertoires `kernel/` et `mm/`).

A.2.3.2 Symboles utilisés, noyau 3.0, defconfig

Les résultats exposés dans la section précédente A.2.2 montraient trois familles de symboles parmi ceux qui sont les plus utilisés : affichage, gestion de la mémoire et primitives de verrouillage. Après avoir, dans le paragraphe précédent, regardé l'utilisation quantitative des symboles par sous-répertoires, nous allons identifier leurs familles.

Dans le cas des répertoires `mm/` (A.4e) et `lib/` (A.4f), constatons :

- Pour `mm/`, ce sont principalement des symboles de la famille correspondant à la gestion mémoire, ce qui est plutôt cohérent avec l'objectif de ce répertoire. De plus, il est à noter qu'entre l'instance **defconfig** (A.4e) et **allyesconfig** (A.5e), la fréquence cumulée de ces symboles est importante, étant de 0.522 et 0.719 respectivement.
- Pour `lib/`, le constat est similaire, dans le sens où l'on trouve énormément de symboles liés à la gestion de chaînes de caractères (telles `sprintf()`, `strcpy()` ou encore `snprintf()`, `strcmp()` et `simple_strtoul()`). Le constat évolue un peu avec l'instance **allyesconfig** (A.5f) qui fait apparaître des symboles provenant des options utilisées pour l'analyse et la vérification des problèmes (par exemple, pour les accès `DMA` avec `debogue_dma_free_coherent()` et `debogue_dma_alloc_coherent()`). Surtout, cette dernière instance montre une présence accrue de symboles en liens avec la gestion des listes chaînées (verrouillage `__raw_spin_lock_init()`, ajout et suppression d'éléments avec `__list_add()` et `list_del()`). Il convient donc de distinguer les familles suivant l'instance ; pour l'instance **defconfig**, les familles identifiables sont les symboles liés à la manipulation de chaînes de caractères ainsi que ceux pour gérer les compteurs de référence. Dans le cas de l'instance **allyesconfig**, les familles identifiées permettent de gérer les listes chaînées ainsi que les chaînes de caractères.
- Pour `kernel/`, le symbole le plus utilisé reste `printk()` (de la famille en charge de l'affichage), mais la famille la plus représentée concerne la gestion des verrous, que ce soit les mutex (`mutex_lock()`, `mutex_unlock()`), ou les verrous tournants. Ceci se vérifie dans l'instance **allyesconfig**, si l'on omet les symboles présents à des fins d'analyse et de correction de problèmes, on retrouve les mêmes familles.
- Pour `fs/`, les familles de symboles liées à l'affichage (`printk()`) et à la manipulation des primitives de verrouillage se retrouvent, mais elles sont accompagnées également de la famille dédiée à la gestion de la mémoire. Une famille spécifique au sous-système dédié aux systèmes de fichier fait son apparition, au travers des symboles `iput()`, `fput()`, `dput()` ou encore `seq_printf()`, `seq_read()`, etc. : il s'agit de fonctions manipulant des structures propres aux systèmes de fichier (*inodes*, *dentries*, ...).
- Pour `drivers/`, la gestion de la mémoire, les primitives de verrouillage ainsi que celles d'affichage forment les trois familles habituelles que l'on retrouve. Une nouvelle famille fait son apparition, cependant, dans ce sous-répertoire, avec des symboles comme `dev_get_drvdata()`, `dev_set_drvdata()`, `dev_err()`, `_dev_info()` : celle de la manipulation des structures de données liées aux pilotes.
- Pour `net/`, parmi les familles « classiques » nous pouvons retrouver la gestion de la mémoire ainsi que les primitives de verrouillage, et l'affichage (dont fait partie le symbole `net_ratelimit()`). La spécificité de ce sous-répertoire se manifeste par les symboles liés à la gestion des tampons réseaux (symboles « `skb` », pour « *Socket*

Buffers ») comme `kfree_skb()`, `skb_put()`, `skb_copy_bits()`.

Alors que la section A.2.2 proposait une vue assez globale sur l'utilisation des symboles dans la totalité du noyau, nous venons de voir que, si l'on s'intéresse plus précisément à certains sous-répertoires, l'utilisation devient différente, et montre à la fois des similitudes et des différences :

- Certains sous-répertoires n'utilisent que les familles « générales », comme `kernel/` ;
- Certains n'utilisent que des familles non présente au niveau « général », comme `lib/` ;
- Enfin, d'autres utilisent les familles « générales » couplées aux spécificités liées à leur rôle, comme `net/` ou `fs/`.

A.2.3.3 Symboles utilisés, noyau 3.0, allyesconfig

Maintenant nous allons regarder de plus près l'instance **allyesconfig**, dont les résultats par sous-répertoires sont disponibles dans les tableaux A.5a, A.5b, A.5c, A.5d, A.5e et A.5f pour le noyau dans sa version 3.0.

Quelques premières conclusions étaient déjà disponibles quant à cette instance dans la partie A.2.3.2, nous allons approfondir. Un changement assez important concerne l'apparition de symboles tels que `__gcov_init()` ou `__gcov_merge_add()` : comme indiqué dans le paragraphe A.2.2.3, la génération de ces symboles est due à l'activation d'outils de vérification lors de l'instance **allyesconfig**. Comme également indiqué dans le paragraphe précédent, on observe une variation – à la fois à la hausse comme à la baisse – des fréquences cumulées sur les différents sous-répertoires : notamment, `kernel/` dont l'inflation est documentée dans le paragraphe précédent.

Le sous-répertoire `lib/` ne voit aucune variation substantielle entre les deux instances, bien que les symboles de `GCOV` soient présents. C'est le seul répertoire à présenter cette absence : tous les autres voient leur fréquence cumulée évoluer : `kernel/`, `drivers/`, `net/`, `mm/` et `lib/`. Dans le cas de ces deux derniers, il est à noter que les symboles `GCOV` ne sont pas présents, mais que d'autres symboles ayant une utilisation similaire arrivent dans le top 20.

A.2.3.4 Symboles utilisés, noyau 3.9, defconfig

Après avoir étudié les symboles des sous-répertoires pour le noyau 3.0 dans les deux instances disponibles, regardons le cas **defconfig**, dont les résultats par sous-répertoires sont disponibles dans les tableaux A.6a, A.6b, A.6c, A.6d, A.6e et A.6f pour le noyau dans sa version 3.9.

Globalement, les résultats de chaque sous-répertoire sont similaires : quelques variations quant aux fréquences sont visibles, mais elles demeurent assez limitées, et elles sont du même ordre sur tous les sous-répertoires : cela s'explique assez facilement par l'accroissement de la taille du noyau. L'ordre des symboles (et les familles représentées) reste également équivalent.

- Pour `mm/`, comme dans l'instance **defconfig**, les symboles que l'on retrouve sont principalement ceux qui sont liés à la gestion de la mémoire. La seule variation notable est l'apparition d'un symbole `vmalloc()` combinée à la disparition du symbole

Symbole	Fréquence
printk	0.052223
mutex_unlock	0.029701
mutex_lock	0.028949
_cond_resched	0.026343
_raw_spin_lock	0.025127
warn_slowpath_null	0.023159
_raw_spin_unlock_irqrestore	0.01789
_raw_spin_lock_irqsave	0.017311
schedule	0.010595
__wake_up	0.010421
_raw_spin_lock_irq	0.009785
nr_cpu_ids	0.009611
__mutex_init	0.009495
capable	0.008974
__init_waitqueue_head	0.008742
module_put	0.008106
warn_slowpath_fmt	0.007585
cpu_possible_mask	0.007063
__tracepoint_module_get	0.006832
finish_wait	0.006716
- Total -	0.324629

(a) Répertoire `kernel/`

Symbole	Fréquence
seq_printf	0.014643
seq_read	0.010459
seq_lseek	0.010459
kfree	0.00844
printk	0.008007
iput	0.008007
_raw_spin_lock	0.007935
proc_create_data	0.007574
fput	0.007286
current_task	0.007069
_cond_resched	0.006131
kmalloc_caches	0.005771
kmem_cache_alloc_trace	0.005771
dput	0.005626
mutex_unlock	0.005554
mutex_lock	0.005266
seq_puts	0.005049
d_instantiate	0.004905
__brelse	0.004905
seq_open	0.004761
- Total -	0.14362

(b) Répertoire `fs/`

Symbole	Fréquence
kfree	0.013073
kmalloc_caches	0.010266
kmem_cache_alloc_trace	0.010266
printk	0.010074
mutex_unlock	0.007459
mutex_lock	0.007229
dev_err	0.006344
dev_get_drvdata	0.006306
dev_printk	0.00619
_dev_info	0.006152
__kmalloc	0.006152
acpi_error	0.005998
dev_set_drvdata	0.005114
dev_warn	0.00496
put_device	0.004768
_raw_spin_unlock_irqrestore	0.004268
_raw_spin_lock_irqsave	0.004229
sprintf	0.003883
dev_set_name	0.003845
current_task	0.003845
- Total -	0.130421

(c) Répertoire `drivers/`

Symbole	Fréquence
kfree_skb	0.010817
printk	0.009451
kfree	0.009124
skb_put	0.008687
register_pernet_subsys	0.007375
net_ratelimit	0.007212
init_net	0.007048
__alloc_skb	0.006119
warn_slowpath_null	0.005846
kmalloc_caches	0.005573
kmem_cache_alloc_trace	0.005573
jiffies	0.005518
dst_release	0.005463
__kmalloc	0.005409
__pskb_pull_tail	0.005354
memcpy	0.005135
skb_copy_bits	0.004862
unregister_pernet_subsys	0.004808
_raw_spin_lock	0.004698
_raw_spin_unlock_bh	0.004535
- Total -	0.128606

(d) Répertoire `net/`

Symbole	Fréquence
kfree	0.120173
kmalloc_caches	0.089642
kmem_cache_alloc_trace	0.088387
__kmalloc	0.059668
kmem_cache_free	0.01659
kmem_cache_alloc	0.016311
kmem_cache_create	0.014917
put_page	0.013662
free_pages	0.013244
__get_free_pages	0.010038
kstrdup	0.010038
alloc_pages_current	0.009619
vfree	0.009341
kmem_cache_destroy	0.008086
kmemdup	0.008086
numa_node	0.007389
unlock_page	0.00711
node_states	0.006971
__free_pages	0.006971
vmalloc	0.006134
- Total -	0.522376

(e) Répertoire `mm/`

Symbole	Fréquence
sprintf	0.05498
strlen	0.049459
snprintf	0.047619
strcmp	0.044398
memcmp	0.040948
find_next_bit	0.032436
strcpy	0.024155
simple_strtoul	0.023004
strlcpy	0.020474
strncmp	0.020474
_ctype	0.018634
kref_put	0.016333
strncpy	0.015873
kref_get	0.015413
strchr	0.015183
__sw_hweight64	0.014723
sscanf	0.013803
find_first_bit	0.013343
kref_init	0.012652
kstrtoull	0.010582
- Total -	0.504486

(f) Répertoire `lib/`

TABLE A.4 – Utilisation des symboles par sous-répertoire (`kernel/`, `fs/`, `drivers/`, `net/`, `mm/`, `lib/`) – Noyau 3.0, instance **defconfig**

Symbole	Fréquence
__gcov_merge_add	0.096156
__gcov_init	0.096156
printk	0.056986
__stack_chk_fail	0.031456
mutex_unlock	0.024773
mutex_lock_nested	0.023378
warn_slowpath_null	0.019496
_raw_spin_unlock	0.01931
_raw_spin_unlock_irqrestore	0.0193
_raw_spin_lock	0.019172
_raw_spin_lock_irqsave	0.019172
param_ops_int	0.013826
__mutex_init	0.0125
msleep	0.012431
__wake_up	0.011153
__init_waitqueue_head	0.011085
debug_lockdep_rcu_enabled	0.01078
lockdep_init_map	0.00966
init_timer_key	0.009591
free_irq	0.008992
- Total -	0.525373

(a) Répertoire `kernel/`

Symbole	Fréquence
__gcov_merge_add	0.014658
__gcov_init	0.014658
mcount	0.014618
seq_printf	0.008588
printk	0.008507
iput	0.007228
kfree	0.007188
_raw_spin_lock	0.006353
_raw_spin_unlock	0.006313
seq_read	0.006084
__brelse	0.006084
seq_lseek	0.006084
current_task	0.005801
mutex_unlock	0.004805
mutex_lock_nested	0.004657
__mark_inode_dirty	0.004603
__kmalloc	0.004348
kmalloc_caches	0.004253
kmem_cache_alloc_trace	0.00424
generic_file_llseek	0.004065
- Total -	0.143136

(b) Répertoire `fs/`

Symbole	Fréquence
__gcov_merge_add	0.020397
__gcov_init	0.020397
mcount	0.019893
dev_get_drvdata	0.013662
printk	0.012812
dev_set_drvdata	0.011805
kfree	0.011514
dev_err	0.009798
kmalloc_caches	0.008088
kmem_cache_alloc_trace	0.008088
dev_printk	0.007502
__stack_chk_fail	0.006985
__kmalloc	0.005752
mutex_unlock	0.005543
_dev_info	0.005368
mutex_lock_nested	0.005152
_raw_spin_unlock_irqrestore	0.005128
_raw_spin_lock_irqsave	0.005115
pci_unregister_driver	0.004909
__pci_register_driver	0.004806
- Total -	0.192713

(c) Répertoire `drivers/`

Symbole	Fréquence
__gcov_merge_add	0.016828
__gcov_init	0.016828
mcount	0.016753
skb_put	0.010646
printk	0.010306
kfree_skb	0.008538
kfree	0.007875
__stack_chk_fail	0.00692
warn_slowpath_null	0.00658
consume_skb	0.0058
jiffies	0.005211
skb_pull	0.005128
__alloc_skb	0.005095
free_netdev	0.005053
skb_push	0.004829
net_ratelimit	0.004804
__netif_schedule	0.004755
netpoll_trap	0.004638
dev_alloc_skb	0.004605
kmalloc_caches	0.004597
- Total -	0.155791

(d) Répertoire `net/`

Symbole	Fréquence
kfree	0.182241
kmalloc_caches	0.124516
kmem_cache_alloc_trace	0.123947
__kmalloc	0.092774
might_fault	0.043046
vfree	0.016955
kmem_cache_free	0.014076
kmem_cache_alloc	0.013223
put_page	0.01237
kmem_cache_create	0.012014
vmalloc	0.011694
free_pages	0.010557
kmem_cache_destroy	0.009917
alloc_pages_current	0.008566
kmemdup	0.008318
__get_free_pages	0.007358
kmalloc_order_trace	0.007109
unlock_page	0.007074
__free_pages	0.006825
kstrdup	0.006292
- Total -	0.718871

(e) Répertoire `mm/`

Symbole	Fréquence
__raw_spin_lock_init	0.071038
__list_add	0.064405
sprintf	0.056359
strcpy	0.047298
list_del	0.046319
snprintf	0.045486
memcmp	0.038563
strcmp	0.027799
__list_del_entry	0.025008
strlcpy	0.020369
__dynamic_pr_debug	0.019644
strncmp	0.017542
debug_dma_free_coherent	0.015404
strncpy	0.015259
debug_dma_alloc_coherent	0.015222
simple_strtoul	0.014208
find_next_bit	0.012432
debug_dma_map_page	0.011163
kref_put	0.010909
iowrite32	0.010909
- Total -	0.585336

(f) Répertoire `lib/`

TABLE A.5 – Utilisation des symboles par sous-répertoire (`kernel/`, `fs/`, `drivers/`, `net/`, `mm/`, `lib/`) – Noyau 3.0, instance **allyesconfig**

`___get_page_tail()`.
- Pour `lib/`, trois nouveaux symboles prennent place en queue de peloton : `nla_put()`, `kobject_put()` et `kobject_uevent()`, alors que dans le même temps `kref_put()`, `kref_get()` et `kref_init()` disparaissent : lors du passage du noyau 3.2 au noyau 3.3, les symboles `kref_*()` ont été éliminés.
- Pour `kernel/`, au-delà de la baisse généralisée des fréquences et de quelques alternances entre l'ordre des symboles, il n'y a pas de changement notable par rapport à l'instance **defconfig**.
- Pour `fs/`, le même constat tient : baisse généralisée et uniforme des fréquences, quelques alternances dans l'ordre des symboles.
- Pour `drivers/`, encore les mêmes résultats, à la différence que le symbole `dev_warn()` a disparu (i.e., il est sorti du top 20 des symboles les plus utilisés, mais il reste présent dans le noyau malgré tout) au profit de `drm_ut_debogue_printk()`. L'apparition de ce symbole est intéressante, puisque c'est un des éléments faisant partie du sous-système `DRM` (*Direct Rendering Manager*), utilisé pour fournir l'accélération graphique des pilotes vidéos.
- Pour `net/`, les changements sont plus succincts et se limitent à des modifications dans l'ordre des symboles, ainsi qu'à la baisse des fréquences.

A.2.3.5 Symboles utilisés, noyau 3.9, allyesconfig

Regardons la dernière instance, **allyesconfig**, dont les résultats par sous-répertoires sont disponibles dans les tableaux A.7a, A.7b, A.7c, A.7d, A.7e et A.7f pour le noyau dans sa version 3.9.

L'évolution de l'utilisation des symboles entre l'instance **defconfig** et **allyesconfig** pour le noyau 3.9 est calquée sur l'évolution qui est décrite pour le noyau 3.0. Tout au plus, la seule différence observable, mais qui rejoint ce qui est visible lors de l'évolution du noyau 3.0 à 3.9 dans l'instance **defconfig**, est que les fréquences cumulées des tops 20 sont plus faibles dans l'instance **allyesconfig**.

A.3 Résultat : densité du graphe

Nous allons maintenant étudier la densité du graphe du noyau. La définition de la mesure est donnée en section 3.3.3. Les graphiques A.8a et A.8b proposent la vision de l'évolution de la densité de tout le graphe sur l'intervalle des versions 3.0 à 3.9 respectivement dans les instances **defconfig** et **allyesconfig**. Une analyse au niveau des sous-répertoires est proposée par la suite, dans la section A.3.1.

Dans l'instance **defconfig**, le graphique montre une relative stabilité de la mesure de la densité sur l'intervalle de versions : on peut à la fois constater des hausses, puis des baisses ; mais sur le temps l'évolution est plutôt neutre en définitive. La moyenne des valeurs se situe à 0.01516 avec un écart-type de l'ordre de 8.9×10^{-5}. La comparaison avec l'instance **allyesconfig** montre un tout autre comportement. D'abord, la valeur de la densité est en baisse constante et régulière : elle passe ainsi de 0.0035 à la version 3.0 à 0.00305 avec la version 3.9. La moyenne dans ce cas se situe à 0.0033, soit un facteur de 4.5 par rapport à

Symbole	Fréquence
printk	0.046681
mutex_unlock	0.028829
warn_slowpath_null	0.028316
mutex_lock	0.027906
_cond_resched	0.026111
_raw_spin_lock	0.02293
_raw_spin_unlock_irqrestore	0.016877
_raw_spin_lock_irqsave	0.016415
_raw_spin_lock_irq	0.009952
schedule	0.009849
__wake_up	0.009747
__mutex_init	0.009695
nr_cpu_ids	0.009695
warn_slowpath_fmt	0.009234
__init_waitqueue_head	0.008413
capable	0.007951
module_put	0.007284
cpu_possible_mask	0.007079
init_timer_key	0.00672
cpu_online_mask	0.006617
- Total -	0.316302

(a) Répertoire `kernel/`

Symbole	Fréquence
seq_printf	0.012976
seq_read	0.009703
seq_lseek	0.009586
proc_create_data	0.008066
kfree	0.00719
_raw_spin_lock	0.007131
fput	0.00678
iput	0.00643
current_task	0.006254
_cond_resched	0.005962
printk	0.005495
kmalloc_caches	0.005436
kmem_cache_alloc_trace	0.005436
dput	0.005378
mutex_unlock	0.005378
mutex_lock	0.005144
remove_proc_entry	0.004793
__brelse	0.004676
seq_open	0.004267
single_open	0.00415
- Total -	0.130231

(b) Répertoire `fs/`

Symbole	Fréquence
kfree	0.012703
kmalloc_caches	0.009976
kmem_cache_alloc_trace	0.009976
printk	0.009147
dev_err	0.007594
mutex_unlock	0.007387
mutex_lock	0.007076
_dev_info	0.006662
dev_get_drvdata	0.006386
__kmalloc	0.005834
dev_warn	0.005592
acpi_error	0.005454
dev_set_drvdata	0.004833
dev_printk	0.004833
put_device	0.00466
warn_slowpath_null	0.004556
_raw_spin_unlock_irqrestore	0.004142
_raw_spin_lock_irqsave	0.004108
drm_ut_debug_printk	0.003935
sprintf	0.003832
- Total -	0.128685

(c) Répertoire `drivers/`

Symbole	Fréquence
kfree_skb	0.010455
kfree	0.008812
printk	0.007916
register_pernet_subsys	0.007916
net_ratelimit	0.006671
warn_slowpath_null	0.006422
__alloc_skb	0.005675
unregister_pernet_subsys	0.005576
skb_put	0.005377
init_net	0.005327
jiffies	0.005177
__kmalloc	0.005128
dst_release	0.005078
kmalloc_caches	0.004829
kmem_cache_alloc_trace	0.004829
memcpy	0.004779
__pskb_pull_tail	0.00468
skb_copy_bits	0.00463
_raw_spin_unlock_bh	0.00458
_raw_spin_lock	0.00453
- Total -	0.118385

(d) Répertoire `net/`

Symbole	Fréquence
kfree	0.117985
kmalloc_caches	0.087883
kmem_cache_alloc_trace	0.086097
__kmalloc	0.057781
kmem_cache_free	0.016327
kmem_cache_alloc	0.015689
kmem_cache_create	0.014413
put_page	0.014158
free_pages	0.01199
vfree	0.009949
kstrdup	0.009566
kmemdup	0.009184
__get_free_pages	0.009056
alloc_pages_current	0.008546
kmem_cache_destroy	0.008163
__get_page_tail	0.007653
numa_node	0.007526
unlock_page	0.007143
node_states	0.006505
__free_pages	0.006505
- Total -	0.512117

(e) Répertoire `mm/`

Symbole	Fréquence
sprintf	0.051982
snprintf	0.051549
strlen	0.050249
strcmp	0.043102
memcmp	0.038553
find_next_bit	0.034655
strlcpy	0.024691
strcpy	0.022309
strncmp	0.022092
simple_strtoul	0.017977
_ctype	0.017977
strchr	0.016461
strncpy	0.015378
__sw_hweight64	0.015161
find_first_bit	0.014295
sscanf	0.014295
kstrtoull	0.011696
nla_put	0.011263
kobject_put	0.00953
kobject_uevent	0.008447
- Total -	0.491661

(f) Répertoire `lib/`

TABLE A.6 – Utilisation des symboles par sous-répertoire (`kernel/`, `fs/`, `drivers/`, `net/`, `mm/`, `lib/`) – Noyau 3.9, instance **defconfig**

Symbole	Fréquence
__gcov_merge_add	0.093539
__gcov_init	0.093539
printk	0.04817
__stack_chk_fail	0.029873
mutex_unlock	0.02464
mutex_lock_nested	0.023366
warn_slowpath_null	0.021394
_raw_spin_unlock_irqrestore	0.017697
_raw_spin_lock_irqsave	0.017582
_raw_spin_unlock	0.01736
_raw_spin_lock	0.017196
__mutex_init	0.012308
param_ops_int	0.012299
msleep	0.011585
debug_lockdep_rcu_enabled	0.011412
lockdep_init_map	0.010303
__init_waitqueue_head	0.010295
rcu_is_cpu_idle	0.010286
__wake_up	0.00995
lock_is_held	0.009678
- Total -	0.502473

(a) Répertoire `kernel/`

Symbole	Fréquence
__gcov_merge_add	0.013788
__gcov_init	0.013788
__fentry__	0.013717
seq_printf	0.008302
printk	0.007862
kfree	0.006698
iput	0.005974
seq_read	0.005891
seq_lseek	0.005867
_raw_spin_lock	0.005843
_raw_spin_unlock	0.005819
__brelse	0.005582
current_task	0.00538
mutex_unlock	0.004727
mutex_lock_nested	0.004596
__mark_inode_dirty	0.00424
__kmalloc	0.004157
kmalloc_caches	0.004145
kmem_cache_alloc_trace	0.004145
kernel_stack	0.004121
- Total -	0.134642

(b) Répertoire `fs/`

Symbole	Fréquence
__gcov_merge_add	0.020061
__gcov_init	0.020061
__fentry__	0.019664
dev_get_drvdata	0.013926
dev_set_drvdata	0.01208
dev_err	0.011409
printk	0.010363
kfree	0.010109
kmalloc_caches	0.006922
kmem_cache_alloc_trace	0.006922
__stack_chk_fail	0.006578
_dev_info	0.006105
mutex_unlock	0.005558
__kmalloc	0.005222
mutex_lock_nested	0.005206
_raw_spin_unlock_irqrestore	0.004684
_raw_spin_lock_irqsave	0.004673
dev_warn	0.004639
__raw_spin_lock_init	0.004053
jiffies	0.004042
- Total -	0.182275

(c) Répertoire `drivers/`

Symbole	Fréquence
__gcov_merge_add	0.016176
__gcov_init	0.016176
__fentry__	0.016127
printk	0.009148
skb_put	0.008903
kfree_skb	0.00856
kfree	0.007762
__stack_chk_fail	0.006838
warn_slowpath_null	0.006544
consume_skb	0.00604
__netdev_alloc_skb	0.005649
jiffies	0.004774
skb_pull	0.004767
skb_push	0.004753
__alloc_skb	0.004557
free_netdev	0.004445
__kmalloc	0.004445
kmalloc_caches	0.004403
kmem_cache_alloc_trace	0.004403
kernel_stack	0.004361
- Total -	0.148829

(d) Répertoire `net/`

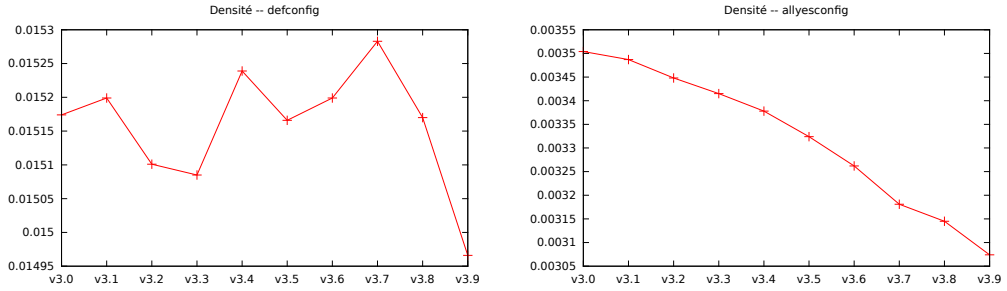
Symbole	Fréquence
kfree	0.173742
kmalloc_caches	0.114742
kmem_cache_alloc_trace	0.114065
__kmalloc	0.092871
might_fault	0.041774
vfree	0.017581
kmem_cache_free	0.014129
kmem_cache_alloc	0.013032
put_page	0.012903
kmem_cache_create	0.012161
vmalloc	0.011194
kmemdup	0.010194
kmem_cache_destroy	0.010065
free_pages	0.009645
alloc_pages_current	0.008323
unlock_page	0.006968
kmalloc_order_trace	0.006903
__free_pages	0.006774
__get_free_pages	0.006548
kstrdup	0.006516
- Total -	0.690129

(e) Répertoire `mm/`

Symbole	Fréquence
__raw_spin_lock_init	0.062786
__list_add	0.060365
sprintf	0.047498
snprintf	0.04461
list_del	0.044289
strcpy	0.037462
memcmp	0.033494
__dynamic_dev_dbg	0.032706
__dynamic_pr_debug	0.031714
__list_del_entry	0.027338
strcmp	0.02512
strlcpy	0.022553
strncmp	0.016134
debug_dma_free_coherent	0.0138
debug_dma_alloc_coherent	0.013742
find_next_bit	0.013392
strncpy	0.012604
iowrite32	0.011291
debug_locks	0.010766
ioread32	0.010737
- Total -	0.5724

(f) Répertoire `lib/`

TABLE A.7 – Utilisation des symboles par sous-répertoire (`kernel/`, `fs/`, `drivers/`, `net/`, `mm/`, `lib/`) – Noyau 3.9, instance **allyesconfig**



(a) Instance **defconfig** (b) Instance **allyesconfig**

FIGURE A.8 – Évolution de la densité générale du graphe.

l'instance **defconfig**. Contrairement à cette dernière, l'instance **allyesconfig** montre une plus grande dispersion des points par rapport à la moyenne comme l'atteste le graphique, l'écart-type étant de l'ordre de 1.5×10^{-4} augmentant d'un facteur 1.7.

La densité est donc beaucoup plus faible dans l'instance **allyesconfig**, comme annoncé lors de la définition de la mesure dans la section 3.3.3 : cela confirme l'ajout au cours du temps de nouveaux composants qui sont peu liés entre eux.

A.3.1 Densité par sous-répertoires

Après avoir étudié la densité sur le graphe dans son intégralité, nous allons nous intéresser à l'étudier en décomposant les sous-répertoires. Le but est de pouvoir identifier des comportements différents.

Un premier résultat que nous pouvons apporter à partir du graphique A.9a concerne le répertoire `mm/` : sa densité est sensiblement supérieure à celle des autres répertoires. Ce constat se retrouve également dans l'instance **allyesconfig** dont le graphique est disponible en figure A.9b. À l'opposé, les sous-répertoires qui sont les plus susceptibles d'accueillir en leur sein des éléments assez indépendants, comme des pilotes, se retrouvent en bas de l'échelle de densité, tels `drivers/`, `fs/`, `net/` ou encore `lib/`.

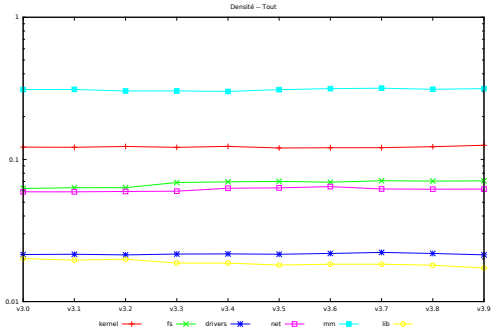
Le passage à l'instance **allyesconfig** permet de montrer un comportement intéressant : d'une part, on observe bien la chute de la densité qui était rapportée précédemment, puisque le bas de l'échelle passe de 0.01 à 0.001. D'autre part, les sous-répertoires `mm/` et `kernel/` n'évoluent pas ou peu lors de ce passage. Le répertoire `lib/`, lui, voit bien sa densité baisser quelque peu – passant de l'ordre de 0.18 à 0.13 – quand trois autres sous-répertoires voient la leur baisser de manière importante : `fs/`, `net/` et `drivers/`. Ce dernier subit une baisse conséquente de l'ordre d'un facteur 10 : passant d'environ 0.022 à 0.0022. Par ailleurs, l'ordre entre `fs/` et `net/` est conservé. Ces derniers sont donc beaucoup plus influencés par l'activation de nombreux éléments optionnels, tels des pilotes (périphériques, systèmes de fichiers).

Dans les figures A.9c et A.9d sont présentés l'évolution de la densité sur la totalité des sous-répertoires dans le cas des instances, respectivement, **defconfig** et **allyesconfig**. Cela permet de comparer le sous-ensemble sélectionné au reste du noyau, et de mettre en

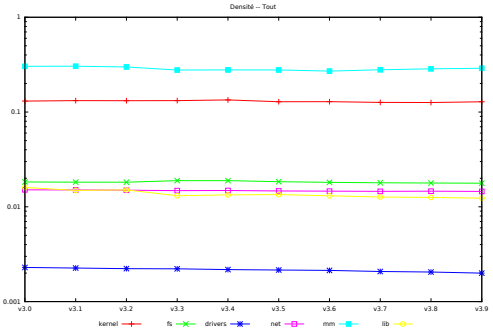
lumière différents faits :

- Indépendamment de l'instance sélectionnée, **defconfig** ou **allyesconfig**, le sous-répertoire `ipc/` présente une densité quasi constante et la plus importante, de l'ordre de 0.73. Ce sous-répertoire contient les primitives pour toutes les opérations de communications inter-processus (*IPC SysV* : files de messages, appels `sysctl`, sémaphores et mémoire partagée), il est donc assez logique que l'évolution soit ténue. De plus, la quantité de code dans ce sous-répertoire est assez limitée, ce qui explique facilement la densité élevée.
- Le sous-répertoire `block/` a également un comportement intéressant : en effet, au passage de la version 3.2 à 3.3, une grosse chute de densité peut être observée, à la fois sur l'instance **defconfig** et sur l'instance **allyesconfig** ; la différence est notable, puisque la densité est quasiment divisée par deux – passant ainsi d'environ 0.28 à 0.15 dans la première instance et environ 0.18 dans la seconde. L'explication de ce comportement est assez simple et se trouve principalement dans un changement [1], qui déplace le code du sous-répertoire `fs/partitions` vers `block/partitions`. L'impact est également visible sur la densité du sous-répertoire `fs/` mais est moins net : passant de 0.63 à 0.70 dans l'instance **defconfig** et de 0.181 à 0.188 dans l'instance **allyesconfig**.
- Un comportement similaire est visible pour le sous-répertoire `init/`, mais en plusieurs étapes : d'abord, au passage du noyau 3.4 à 3.5, puis entre le 3.6 et le 3.7. Une différence majeure, cependant, est que la chute n'est vraiment marquée que dans le cas de l'instance **defconfig**. Elle semble s'expliciter dans un premier temps par un premier changement [2] qui introduit une nouvelle option de configuration dans ce sous-répertoire, option désactivée par défaut et concerne les espaces de nommage. Les évolutions constatées sur l'intervalle du noyau 3.6 au noyau 3.7 semblent s'expliquer par la concurrence de plusieurs changements, dont la mise en place de l'infrastructure de signature des modules noyau qui ajoute plusieurs nouvelles options de construction.
- Le sous-répertoire `sound/` montre une baisse conséquente de sa densité lors du passage à l'instance **allyesconfig**, ce qui s'explique assez facilement par le fait que ce répertoire contient de nombreux pilotes ou modules, en rapport avec le sous-système son.
- Un comportement similaire peut également être observé avec `security/` : la configuration de l'instance **allyesconfig** permet d'activer tous les modules liés à la sécurité, par exemple *SELinux*, *AppArmor*, *SMACK* ou *Tomoyo*. La densité de ce sous-répertoire passe ainsi de 0.29 à 0.09, une valeur assez proche de celle du répertoire `lib/`. Cette chute s'explique car les modules de sécurité sont plutôt très indépendants entre eux, ce qui n'est pas surprenant.

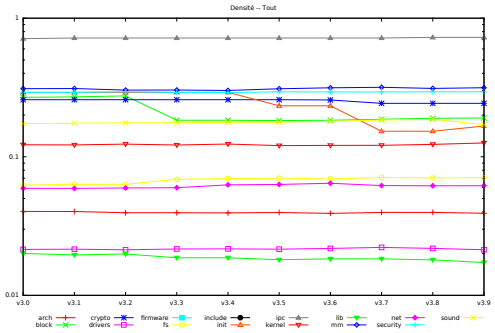
1. http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=9be96f3fd10187f185d84cf878cf032465bcced3
2. http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=e1c972b681bf118fcedb9fe2ed7a73de983aa5ef



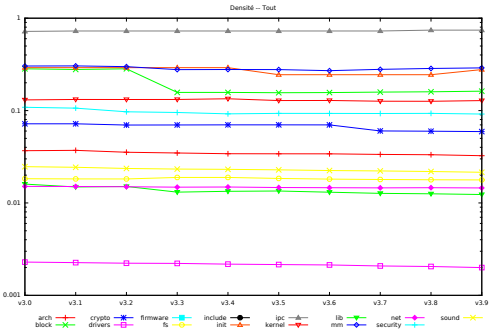
(a) Sous-ensemble, **defconfig**



(b) Sous-ensemble, **allyesconfig**

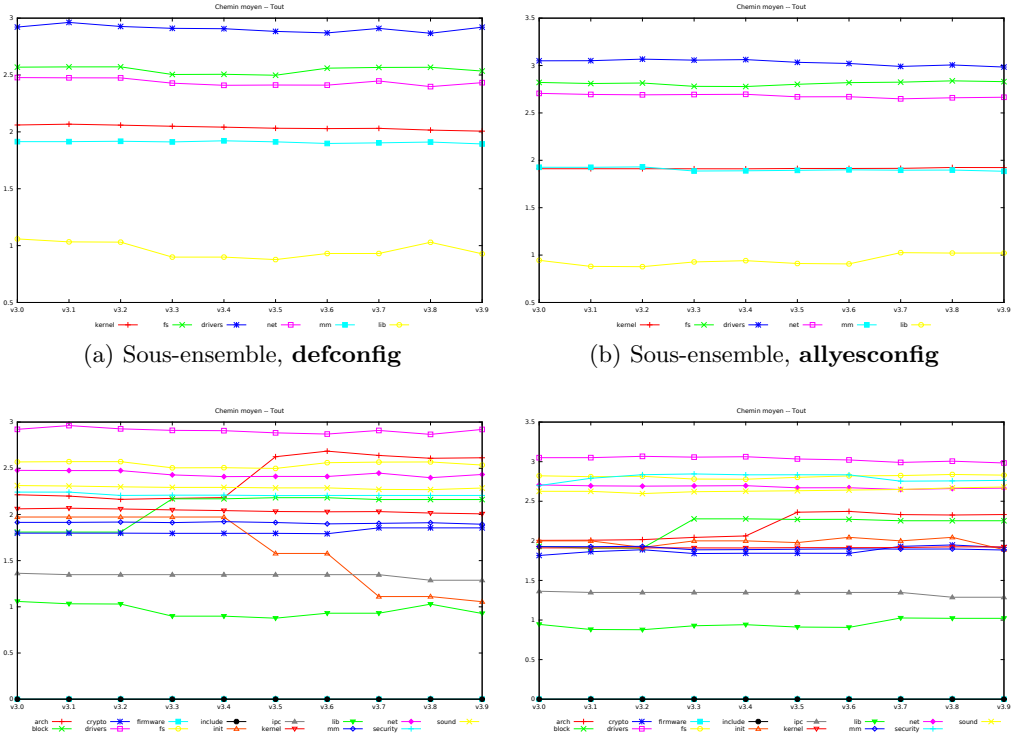


(c) Totalité, **defconfig**



(d) Totalité, **allyesconfig**

FIGURE A.9 – Évolution de la densité du graphe, par sous-répertoire.



(a) Sous-ensemble, **defconfig**

(b) Sous-ensemble, **allyesconfig**

(c) Totalité, **defconfig**

(d) Totalité, **allyesconfig**

FIGURE A.10 – Évolution de la taille moyenne du chemin dans le graphe.

A.4 Résultat : taille moyenne du chemin

Comme indiqué dans la définition de cette mesure donnée en section 3.3.4, la taille moyenne du chemin permet d'avoir une idée de la distribution de la population de nœuds dans le graphe : plus la taille moyenne du chemin sera grande, plus il y aura de niveau de dépendance entre les fichiers objets.

Dans un premier temps, nous étudions rapidement les différents graphiques présentant l'évolution de la mesure, visibles dans les figures A.10a, A.10b, A.10c et A.10d. En considérant que la taille moyenne du chemin est donné par le nombre d'arcs qui le constitue, alors on peut observer que celle-ci évolue entre environ 1 arc au minimum et au plus environ 3 arcs.

A.4.1 Taille moyenne du chemin par sous-répertoire : defconfig

A.4.1.1 Sous-ensemble de répertoires

Sur la totalité de l'intervalle de versions considérées, on peut remarquer dans ce sous-ensemble l'opposition entre les sous-répertoires `drivers/` et `lib/` : le premier présente une taille moyenne de chemin d'environ 2.9 là où le second varie autour de 1.0. Les éléments dans ce dernier sont donc plus proches les uns des autres que dans le répertoire `drivers/`.

Les sous-répertoires `fs/` et `net/`, qui comportent également à la fois des piles et des pilotes, ont une taille moyenne de chemin proche : entre 2.4 et 2.5. Enfin, un dernier groupe est constitué par les sous-répertoires `mm/` et `kernel/`, dont la valeur de taille moyenne de chemin varie autour de 2.0.

Par ailleurs, il convient de constater que sur les versions étudiées, l'évolution de la taille moyenne du chemin est globalement neutre : ni tendance à l'augmentation ou à la diminution n'est clairement visible.

A.4.1.2 Totalité des répertoires

Les résultats pour l'instance **defconfig** sur la totalité des sous-répertoires sont proches : par exemple, on retrouve `lib/` et `ipc/` avec des valeurs assez proches, en bas du classement, ce qui à la vue de leurs rôles respectifs est cohérent. La totalité des autres sous-répertoires, à l'exception notable de `drivers/` se retrouve dans la zone comprise entre 1.8 et 2.5 avec un écart plus ou moins régulier. Alors que l'intégralité des sous-répertoires montrent un comportement dans le temps plutôt stable, trois se distinguent de cette tendance : `block/`, `init/` et `arch/`.

Dans les deux premiers cas, les ajouts de code documentés dans la section A.3.1 suffisent à expliquer les changements abrupts observés respectivement sur les passages de la version 3.2 à 3.3 et sur les passages des versions 3.4 à 3.5 ainsi que 3.6 à 3.7. Un gros changement survenu sur l'architecture *x86* du noyau entre les versions 3.4 et 3.5 concerne la gestion des exceptions, qui a été entièrement revue, et qui a permis la suppression de vieux code ; ceci pourrait expliquer le bond visible sur le graphique, et qui se reproduit sur l'instance **allyesconfig**.

A.4.2 Taille moyenne du chemin par sous-répertoire : allyesconfig

A.4.2.1 Sous-ensemble de répertoires

L'instance **allyesconfig** sur le sous-ensemble de répertoires met en évidence des regroupements intéressants :

- D'abord, le sous-répertoire `lib/` qui reste celui présentant la plus faible valeur de taille moyenne du chemin, oscillant autour de 1.0, confirmant le comportement présenté dans l'instance **defconfig**.
- Ensuite, un groupe constitué des sous-répertoires `mm/` et `kernel/`, dont la valeur de taille moyenne du chemin baisse légèrement par rapport à l'instance **defconfig**, tombant sous 2.0 à 1.9.
- Enfin le dernier groupe, qui est représenté par les sous-répertoires `drivers/`, `net/` et `fs/`, et dont les tailles moyennes de chemin sont supérieures par rapport à l'instance **defconfig**.

Cette tendance au regroupement semble correspondre à la finalité des sous-répertoires : `lib/` est censé abriter un ensemble de bibliothèques plutôt utilisées par le reste du noyau ; `mm/` et `kernel/` mettent en place des sous-systèmes de base ; là où les répertoires `fs/`, `net/` et `drivers/` eux contiennent à la fois des sous-systèmes et des pilotes qui les exploitent.

Dans tous les cas, le graphique montre également qu'à la manière de l'instance précédente, sur la période concernée, la taille moyenne du chemin reste plutôt stable pour chaque sous-répertoire étudié.

A.4.2.2 Totalité des répertoires

Comparons maintenant ces résultats avec la vue sur la totalité des répertoires que propose la figure A.10d. Tout d'abord, les augmentations observées dans le cas de l'instance **defconfig** sur les répertoires `block/` (au passage de la version 3.2 à 3.3) et `arch/` (au passage de la version 3.4 à 3.5) se confirment, alors que celui sur `init/` disparait.

Par ailleurs, le regroupement estimé dans le paragraphe précédent peut se retrouver dans ce graphique :

- Les répertoires `ipc/` et `lib/` sont assez proches, avec une taille moyenne de chemin comprise environ entre 1.0 et 1.3 : ces répertoires fournissent des primitives compactes et réutilisées surtout par d'autres parties du noyau.
- Le second groupe est constitué des éléments dont la valeur de taille moyenne oscille autour de 2.5 : ce sont les répertoires `arch/`, `block/`, `crypto/`, `init/`, `mm/` et `kernel/`.
- Le dernier groupe se compose des répertoires `sound/`, `net/`, `security/`, `fs/` et `drivers/`, ce dernier étant en tête suivant la valeur.

Ce classement et le regroupement que l'on peut faire grâce à cette mesure tendrait à confirmer l'hypothèse émise précédemment : grossièrement, les sous-répertoires comportant des ensembles de bibliothèques assez peu dépendants entre eux ont une taille moyenne de chemin < 1.5, les répertoires qui composent le ciment de base du noyau sont ceux dont la taille moyenne est comprise entre 1.9 et 2.3, et enfin les répertoires correspondants à des sous-systèmes comprenant à la fois des piles applicatives et des pilotes ou des modules qui en font usage seraient ceux situés au-delà de 2.6.

A.5 Résultat : degrés entrant, sortant et total

Nous allons maintenant étudier les résultats de la mesure des degrés au sein du graphe, tels que définis dans la section 3.3.5 : d'abord, en regardant le degré total (entrant et sortant) dans la sous-section A.5.1 puis en sous-section A.5.2 pour le détail de chaque cas. Il est important de garder à l'esprit que le degré mesuré, pour chaque sous-répertoire, correspond à la somme des degrés de tous les nœuds appartenant à ce sous-répertoire : ce qui nous intéresse, par cette analyse, n'est pas tant de comparer les sous-répertoires (cela sera fait de manière plus lisible dans la section A.6), mais plutôt de suivre l'évolution temporelle de la mesure ainsi que de pouvoir examiner l'influence des instances **defconfig** et **allyesconfig**.

De plus, les arcs du graphe correspondent aux symboles qui sont échangés entre les nœuds. Ces symboles peuvent être de plusieurs types : fonctions, variables, etc.

A.5.1 Degré total

Les résultats du degré total sont présentés dans les graphiques visibles en figures A.11a, A.11b, A.11c et A.11d pour les instances **defconfig** et **allyesconfig**. Rapidement, l'influence du passage de la première instance à la seconde est visible grâce à l'échelle des graphiques, qui se retrouve multipliée par environ 10.5, passant d'un maximum de 35 000 (**defconfig** avec `drivers/`) à un maximum légèrement supérieur à 50 000 (**allyesconfig** avec `drivers/`). La borne inférieure, quant à elle, progresse également passant de 4 400 avec le sous-répertoire `lib/` à 28 500 avec `lib/` et `mm/` : la progression de ce sous-répertoire `lib/` est deux fois moindre (multipliée par environ 6.5) que celle du sous-répertoire `drivers/` sur le même échantillon de versions. Le sous-répertoire `drivers/` est celui qui bénéficie le plus du passage de l'instance **defconfig** à **allyesconfig** : l'écart par rapport aux autres sous-répertoires est bien plus important dans cette dernière configuration. Par ailleurs, sur les graphiques du degré total pour l'instance **deconfig** se retrouve le bond pour le sous-répertoire `fs/` entre les versions 3.5 à 3.7.

Les graphiques peuvent donner l'impression que certains sous-répertoires ont une progression beaucoup plus importante que d'autres ; mais c'est un effet de bord lié à l'utilisation du degré de chacun. Il suffit pour s'en convaincre de calculer la progression entre les versions 3.0 et 3.9 :

- Pour `lib/` :
  - **defconfig** : on passe d'un degré de 4 347 à 4 617, soit un facteur d'environ 1.062.
  - **allyesconfig** : on passe d'un degré de 27 591 à 34 275, soit un facteur d'environ 1.242.
- Pour `drivers/` :
  - **defconfig** : on passe d'un degré de 26 008 à 28 970, soit un facteur d'environ 1.114.
  - **allyesconfig** : on passe d'un degré de 290 494 à 353 406, soit un facteur d'environ 1.217.

Les facteurs sont assez proches, et surtout, la progression la plus forte est en réalité enregistrée pour le sous-répertoire `lib/` sur l'intervalle de versions de l'instance **allyesconfig**.

Il est également intéressant d'étudier le degré rapporté au nombre de nœuds (i.e., une mesure proche de la densité) dans les sous-répertoires correspondants, ce qui pour les versions 3.0 et 3.9 nous donne le résultat proposé dans le tableau A.8 pour les sous-répertoires `lib/` et `drivers/`. Grâce à celui-ci, nous pouvons d'abord constater que les sous-répertoires `lib/` et `drivers/` ont un rapport degré sur nœuds qui n'a pas la même évolution dans le temps : dans le premier cas, ce rapport à tendance à diminuer au fur et à mesure des versions du noyau, indiquant que l'apport de nœuds est plus important que l'apport en matière de symboles utilisés ; dans le second cas, le rapport augmente, ce qui montre l'inverse : une augmentation plus importante du degré que du nombre de nœuds. Cela indique que, dans le premier cas, il y a plus de nouveaux fichiers objets alors que dans le second cas, ces fichiers deviennent plus importants puisqu'ils utilisent de plus en plus de symboles.

Par ailleurs, un autre résultat intéressant se trouve dans la comparaison de l'évolution du répertoire `lib/` entre les instances **defconfig** et **allyesconfig**, surtout par rapport au répertoire `drivers/` : l'augmentation du rapport est bien plus faible dans le second cas que

Répertoire	Instance	Nœuds		Degré		Rapport D/N	
		3.0	3.9	3.0	3.9	3.0	3.9
`lib/`	**defconfig**	81	90	4347	4617	**53,667**	**51,3**
`lib/`	**allyesconfig**	125	167	27591	34275	**220,728**	**205,24**
`drivers/`	**defconfig**	617	658	26008	28970	**42,125**	**44,028**
`drivers/`	**allyesconfig**	6040	7213	290494	353406	**48,095**	**48,996**

TABLE A.8 – Comparaison des rapports du degré total par rapport au nombre de nœuds

dans le premier, puisque `lib/` voit son rapport être multiplié par 4 alors que pour `drivers/` le facteur est autour de 1.1. Cela indique que le degré par rapport au nombre de nœuds est assez stable dans le cas du répertoire `drivers/` pour le passage de la première instance à la seconde, alors que l'activation de **allyesconfig** sur `lib/` augmente énormément la quantité de symboles utilisés.

A.5.2 Degrés entrant et sortant

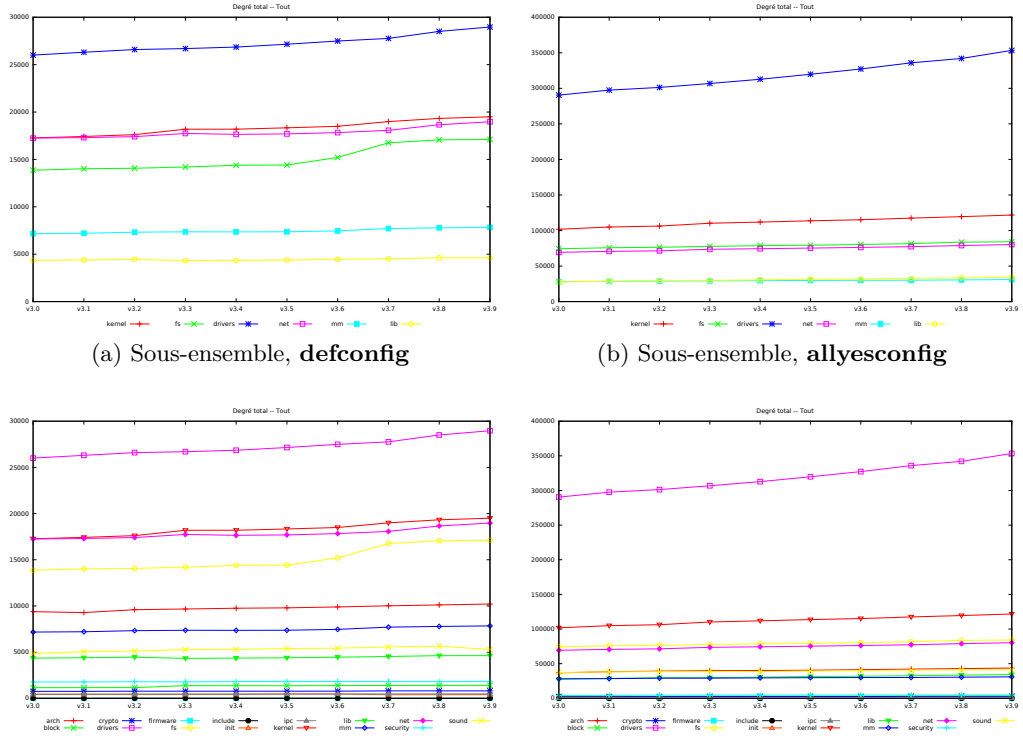
Dans la sous-section précédente, nous avons étudié le degré total des sous-répertoires composant le noyau. Ce degré étant composé du degré entrant et sortant, et chacun ayant une signification particulière, il est intéressant de regarder, également, séparément, leur évolution. Les résultats du degré entrant sont présentés dans les graphiques visibles en figures A.12a, A.12b, A.12c et A.12d pour les instances **defconfig** et **allyesconfig**. Les graphiques pour le cas du degré sortant sont visibles en figures A.13a, A.13b, A.13c et A.13d, également pour ces deux instances.

A.5.2.1 Évolution du degré entrant

Ce degré, comme indiqué en section 3.3.5, correspond au nombre de symboles qui sont *importés* par le module. D'abord, intéressons-nous au sous-répertoire `drivers/`. Son degré entrant varie entre 8802 et 9886 sur l'intervalle des versions 3.0 à 3.9, dans l'instance **defconfig**. Ces chiffres sont multipliés par environ 10 lors du passage à l'instance **allyesconfig** : cette évolution est similaire à celle observée pour le degré total. Mais ceci n'est pas exactement répercuté sur tous les sous-répertoires : `lib/` voit son degré entrant évoluer entre 3945 et 4189 sur l'instance **defconfig** alors que l'autre instance montre une évolution contenue entre 26569 et 32925 ; alors que le degré total était multiplié par 6.5 sur ce sous-répertoire, il est ici multiplié par 7.8 sur les dernières itérations de versions. Le saut de degré qui était observé entre les versions 3.5 et 3.7 pour le sous-répertoire `fs/` est également présent sur la mesure du degré entrant.

Une autre tendance très nette peut être lue sur les graphiques : alors que la mesure du degré total montrait une répartition assez dispersée en valeur absolue, notamment visible sur les graphiques en figures A.11c et A.11d, le degré entrant montre lui une répartition bien plus groupée dans le cas de l'instance **allyesconfig** :

– Un premier groupe est formé par les sous-répertoires `kernel/` et `drivers/`.
– Un second groupe est formé par `mm/`, `lib/`, `fs/`, `arch/` et `net/`.



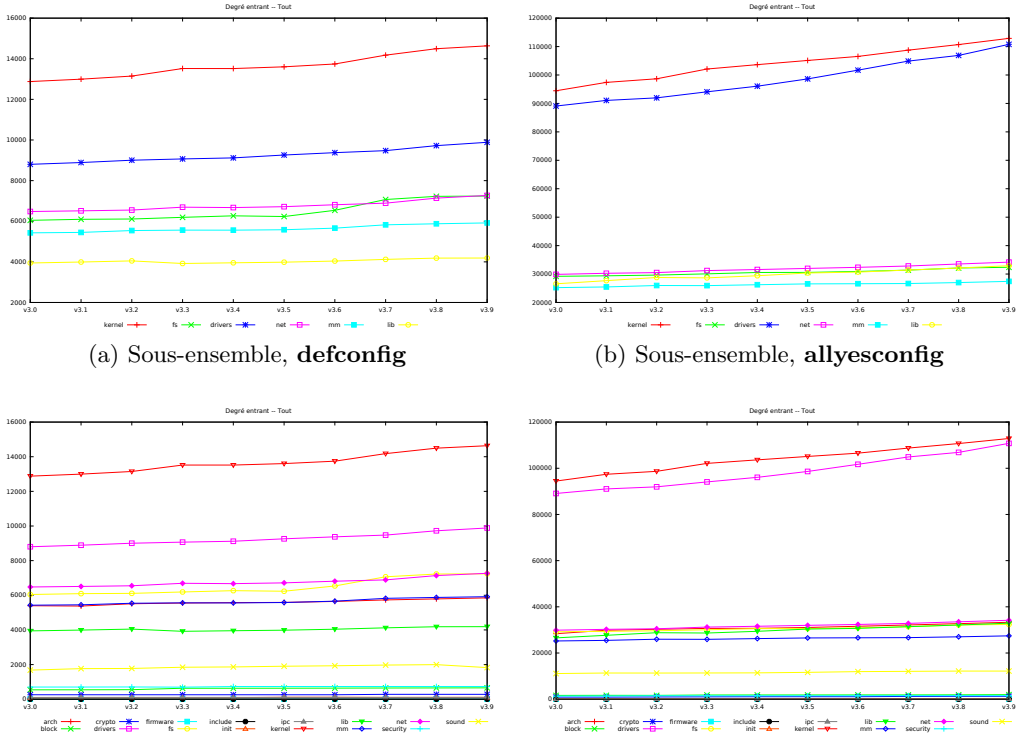
(a) Sous-ensemble, **defconfig**

(b) Sous-ensemble, **allyesconfig**

(c) Totalité, **defconfig**

(d) Totalité, **allyesconfig**

FIGURE A.11 – Évolution du degré total dans le graphe.



(a) Sous-ensemble, **defconfig**

(b) Sous-ensemble, **allyesconfig**

(c) Totalité, **defconfig**

(d) Totalité, **allyesconfig**

FIGURE A.12 – Évolution du degré entrant dans le graphe.

– Le répertoire `sound/` est isolé, avant le dernier groupe où l'on retrouve notamment `block/`, `crypto/`, `security/`.

Par ailleurs, dans le tableau A.9, nous proposons d'étudier le rapport entre le degré entrant et le nombre de nœuds, sur les sous-répertoires `lib/` et `drivers/`, comme cela a été fait précédemment, pour les noyaux 3.0 et 3.9 sur les deux instances. Dans un premier temps, si nous comparons l'évolution de ce rapport au regard de l'évolution de celui présenté dans le tableau A.8, les résultats restent très similaires : pour le sous-répertoire `lib/`, le passage de la version 3.0 à 3.9 se fait avec une baisse du rapport, quelle que soit l'instance ; dans le cas du répertoire `drivers/` l'évolution à la hausse est confirmée. La seule différence notable dans les chiffres provient du fait que le degré total est la somme des degrés entrant et sortant.

A.5.2.2 Évolution du degré sortant

Ce degré, comme indiqué en section 3.3.5, correspond au nombre de symboles qui sont *exportés* par le module. Un premier constat peut être fait sur les graphiques concernant le degré sortant : les échelles, sur l'instance **allyesconfig**, sont bien plus importantes qu'avec le degré entrant, et l'augmentation est de l'ordre d'un facteur 2. Pour l'instance **defconfig**, par contre, elles sont similaires. Les sous-répertoires `lib/` et `mm/` ont un degré sortant bien plus faible que leur degré entrant : cela indique qu'ils exportent assez peu de symboles. Le

Répertoire	Instance	Nœuds		Degré		Rapport De/N	
		3.0	3.9	3.0	3.9	3.0	3.9
`lib/`	**defconfig**	81	90	3945	4189	**48,70**	**46,54**
`lib/`	**allyesconfig**	125	167	26569	32925	**212,55**	**197,16**
`drivers/`	**defconfig**	617	658	8802	9886	**14,27**	**15,02**
`drivers/`	**allyesconfig**	6040	7213	89083	110780	**14,75**	**15,36**

TABLE A.9 – Comparaison des rapports du degré entrant par rapport au nombre de nœuds

Répertoire	Instance	Nœuds		Degré		Rapport Ds/N	
		3.0	3.9	3.0	3.9	3.0	3.9
`lib/`	**defconfig**	81	90	402	428	**4,96**	**4,76**
`lib/`	**allyesconfig**	125	167	1022	1350	**8,18**	**8,08**
`drivers/`	**defconfig**	617	658	17206	19084	**27,89**	**29,00**
`drivers/`	**allyesconfig**	6040	7213	201411	242626	**33,35**	**33,64**

TABLE A.10 – Comparaison des rapports du degré sortant par rapport au nombre de nœuds

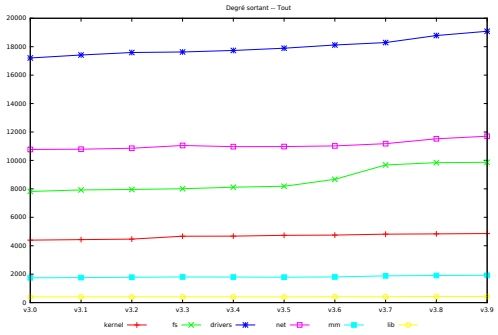
répertoire `kernel/` est dans une situation similaire : alors que son degré entrant était de l'ordre de 14 000 à 16 000, son degré sortant est compris entre 4 000 et 4 500 environ. Le répertoire `fs/` est un peu plus équilibré, puisque l'on passe d'un degré entrant de l'ordre de 6 000 à un degré sortant de l'ordre de 8 000. Par contre, le répertoire `drivers/` voit lui son degré sortant être bien supérieur à celui entrant.

Cette tendance se confirme lorsque l'on étudie l'instance **allyesconfig** : alors que dans le cas du degré entrant les deux sous-répertoires `kernel/` et `drivers/` évoluaient à des niveaux similaires, le degré sortant montre un tout autre visage. Le répertoire `kernel/` évolue entre 7300 et 8800 alors que `drivers/` est entre 200000 et 250000. Ceci s'explique assez facilement par l'activation des différents modules. Le même comportement sur l'instance **allyesconfig** se retrouve d'ailleurs pour les sous-répertoires `fs/` et `net/` qui restent assez proches en matière de degré sortant, avec des valeurs comprises entre 45000 et 52000, respectivement 39000 et 46000. L'incrément en matière de degré est d'ailleurs similaire sur la période des noyaux 3.0 à 3.9.

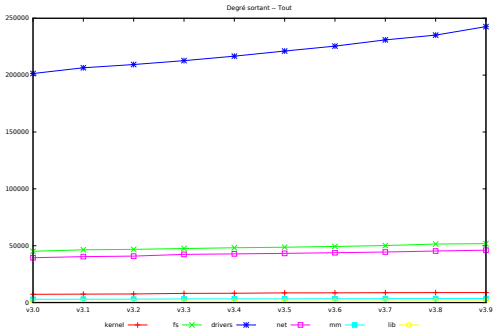
Enfin, le tableau A.10 propose la comparaison du rapport degré sortant sur le nombre de nœuds des sous-répertoires, et les tendances précédemment documentées sur les sous-répertoires `lib/` et `drivers/` sont encore une fois confirmées : dans le premier cas, l'évolution entre les noyaux 3.0 et 3.9 montre une baisse du rapport ; alors que dans le second cas, ce rapport augmente.

A.5.2.3 Rapport des degrés entrant/sortant

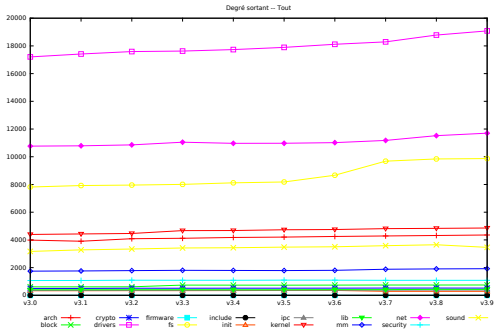
Par ailleurs, afin de mieux étudier chaque sous-répertoire, nous proposons en figure A.11 une série de tableaux présentant le rapport, pour chacun, entre le degré entrant et le degré sortant. Ce calcul permet de comparer les différents sous-répertoires en faisant abstraction de la quantité de code qu'ils contiennent. Les versions 3.0, 3.5 et 3.9 sont



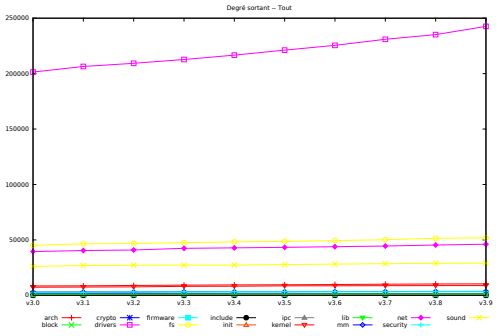
(a) Sous-ensemble, **defconfig**



(b) Sous-ensemble, **allyesconfig**



(c) Totalité, **defconfig**



(d) Totalité, **allyesconfig**

FIGURE A.13 – Évolution du degré sortant dans le graphe.

proposées afin d'avoir une vue de l'évolution temporelle, de même que les instances **defconfig** et **allyesconfig** sont présentes afin de pouvoir constater leur impact. Les répertoires `firmware/` et `include/` ne contiennent pas de fichiers objet, ce qui explique leur rapport égal à 0. Dans cette première instance, nous constatons que seuls quatre sous-répertoires ont un rapport dépassant 1 : `arch/`, `kernel/`, `lib/` et `mm/`. De plus, ceux que nous avons précédemment désigné comme comportant principalement des pilotes et des piles applicatives ont un rapport assez proche : `drivers/`, `fs/`, `net`, `security` et `sound` ; tous sont entre 0.511 (`drivers/` dans la version 3.0) et 0.773 (`fs/` dans la version 3.0). L'évolution temporelle dépend de chaque répertoire, certains étant dans une tendance à la baisse là où d'autres voient une augmentation du rapport : une diminution traduit un renforcement relatif du degré sortant (symboles exportés) alors qu'une hausse correspond à l'inverse à une perte d'influence de ce degré sortant en faveur du degré entrant (symboles importés).

Le passage à l'instance **allyesconfig** montre une variation intéressante du rapport : pour certains répertoires, il est assez proche, alors qu'il augmente beaucoup pour d'autres. Les cinq répertoires pour lesquels l'augmentation est notable sont : `arch/`, `block/`, `kernel/` `lib/` et enfin `mm/` ; le sous-répertoire `kernel/`, par exemple, passe d'un rapport de l'ordre de 3 à presque 13. D'autres répertoires connaissent également une croissance, mais plus nuancée, tels `crypto`, `init/`, `ipc/` et `net/`. Les autres subissent une baisse, peu importante, `drivers/` passant par exemple de 0.518 à 0.465.

A.6 Résultat : carte de chaleur

Les analyses précédentes ont montré que chaque répertoire avait ses caractéristiques, et que celles-ci dépendent, plus ou moins de son rôle dans le code du noyau. La mise en forme des données proposée permet, certes, d'étudier l'évolution temporelle, et de procéder à des comparaisons entre sous-répertoires, mais il reste difficile de voir les liens entre répertoires et de les comparer au sein d'une même instance. Comme indiqué en section 3.3.6, nous proposons maintenant une vision sous la forme de « cartes de chaleur », construites à partir du nombre d'arcs entre chaque sous-répertoire. Dans la section A.6.1, nous présenterons la carte de chaleur du premier niveau de sous-répertoire, celui qui a été utilisé pendant les étapes d'analyses précédentes, avant d'étudier plus en détails les cartes pour les sous-répertoires `drivers/`, `fs/`, `kernel/` et enfin `net/` respectivement dans les sections A.6.2, A.6.3, A.6.4 et A.6.5. Pour chacun, nous présenterons les cartes pour les noyaux 3.0 et 3.9, dans les instances **defconfig** et **allyesconfig**.

A.6.1 Carte de chaleur du noyau complet

Dans cette première série de cartes visibles en figure A.14, nous pouvons relever deux points importants :

– D'abord, la présence de lignes horizontales, plus particulièrement sur les répertoires `mm/`, `lib/`, `kernel/`, `fs/`, `arch/` et en partie sur `drivers/`. Ces lignes dénotent l'utilisation de ces répertoires par une majorité d'autres, la zone colorée étant l'intersection entre deux sous-répertoires : par exemple, pour le répertoire `drivers/` pris en abscisse et `kernel/` pris en ordonnée, le lien qui les unit est clairement mis en

Répertoire	Rapport E/S
arch	1.354328
block	0.882544
crypto	0.519114
drivers	0.511565
firmware	0
fs	0.773215
include	0
init	0.262839
ipc	0.290140
kernel	2.930814
lib	9.813432
mm	3.110601
net	0.601634
security	0.657623
sound	0.530618

(a) Noyau 3.0, **defconfig**

Répertoire	Rapport E/S
arch	1.329369
block	0.865122
crypto	0.500970
drivers	0.517520
firmware	0
fs	0.761794
include	0
init	0.320113
ipc	0.288461
kernel	2.874683
lib	9.912935
mm	3.123670
net	0.611936
security	0.669097
sound	0.547126

(b) Noyau 3.5, **defconfig**

Répertoire	Rapport E/S
arch	1.343742
block	0.877005
crypto	0.543186
drivers	0.518025
firmware	0
fs	0.733508
include	0
init	0.373702
ipc	0.278947
kernel	3.012762
lib	9.787383
mm	3.079084
net	0.621208
security	0.675900
sound	0.527440

(c) Noyau 3.9, **defconfig**

Répertoire	Rapport E/S
arch	3.353149
block	1.732142
crypto	0.633733
drivers	0.442294
firmware	0
fs	0.647367
include	0
init	0.355844
ipc	0.234234
kernel	12.934273
lib	25.997064
mm	8.637889
net	0.757387
security	0.493055
sound	0.424830

(d) Noyau 3.0, **allyesconfig**

Répertoire	Rapport E/S
arch	3.270267
block	1.461965
crypto	0.735175
drivers	0.445907
firmware	0
fs	0.628016
include	0
init	0.442542
ipc	0.219917
kernel	12.268323
lib	25.588879
mm	8.137381
net	0.738848
security	0.456685
sound	0.419218

(e) Noyau 3.5, **allyesconfig**

Répertoire	Rapport E/S
arch	3.215016
block	1.522921
crypto	0.757503
drivers	0.456587
firmware	0
fs	0.624655
include	0
init	0.466992
ipc	0.214711
kernel	12.795081
lib	24.388888
mm	7.688340
net	0.741443
security	0.462731
sound	0.419608

(f) Noyau 3.9, **allyesconfig**

TABLE A.11 – Rapport du degré entrant sur le degré sortant, pour les noyaux 3.0, 3.5 et 3.9, dans les instances **defconfig** et **allyesconfig**, pour les sous-répertoires.

évidence et la couleur est proche du milieu de l'échelle. En inversant la recherche, on constate que le lien entre `kernel/` et `drivers/` est assez faible. Les sous-répertoires `mm/`, `lib/` et `kernel/` sont particulièrement utilisés par les répertoires `drivers/`, `fs/` ainsi que `net/`.

– Ensuite, la première bissectrice est également bien visible. Par construction du graphique, celle-ci indique que le sous-répertoire s'utilise, c'est-à-dire, qu'il dépend de symboles qui sont définis en son sein. Sur le premier graphe, cela montre que les sous-répertoires `drivers/` et `net/` utilisent beaucoup de composants définis en leur sein.

L'évolution temporelle entre le noyau 3.0 et 3.9 sur l'instance **defconfig** montre assez peu de changement, à part un renforcement des liens internes dans les répertoires `fs/` et `net/`.

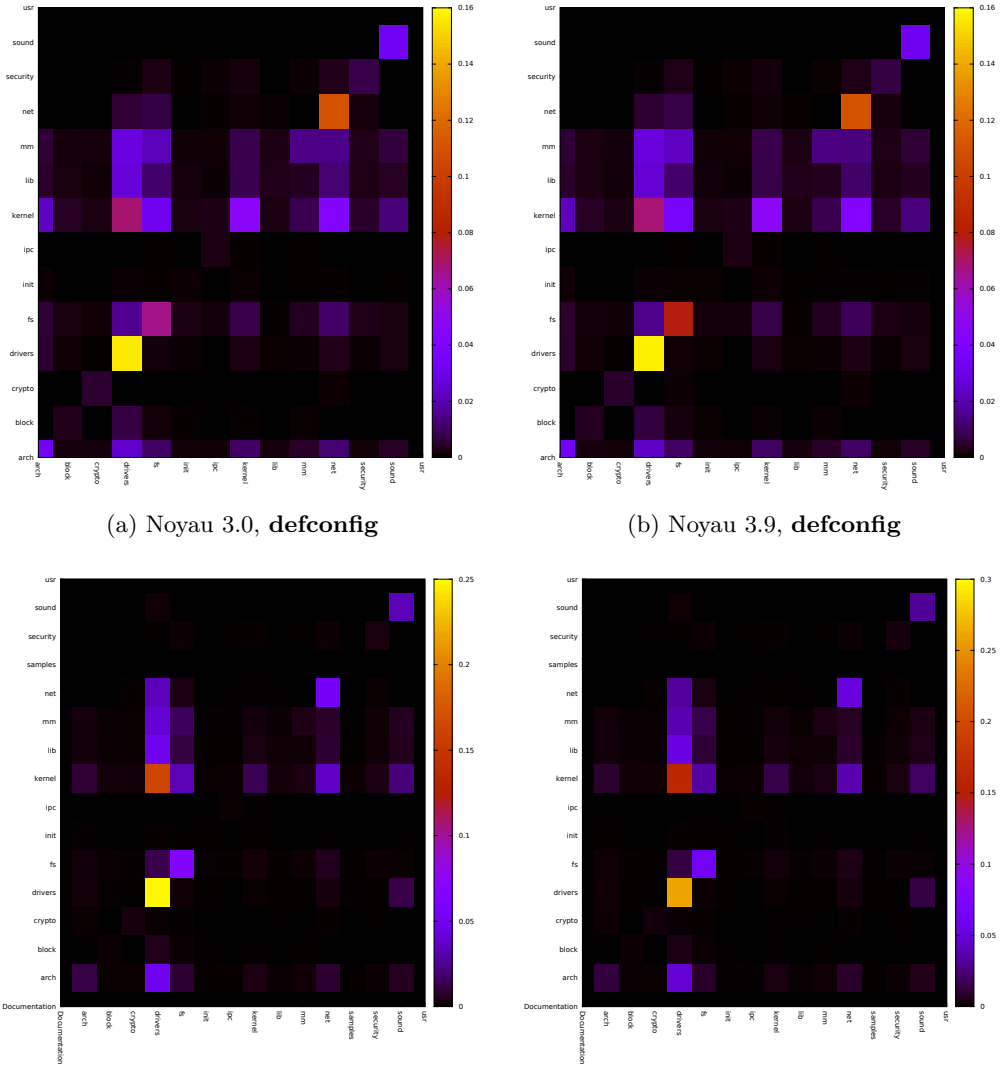
L'instance **allyesconfig** montre un comportement un peu différent. L'échelle de couleur a évolué, le maximum passant à 0.25 au lieu de 0.16, suite à l'augmentation du nombre d'arcs qui a déjà été documentée. La première bissectrice reste nette sur les sous-répertoires majeurs, tels que `drivers/`, `fs/` ou `net/`, mais surtout, les lignes horizontales qui étaient préalablement visibles le sont beaucoup moins, et l'on observe maintenant plutôt des lignes verticales au niveau des répertoires `drivers/`, `fs/`, `net/` et dans une moindre mesure `sound/`. Particulièrement, la première utilise en majorité les répertoires `kernel/`, `lib/`, `mm/` et `net/`. L'utilisation par `drivers/` de `kernel/` et de lui-même progresse, comme l'atteste l'évolution de la couleur à l'intersection de `drivers/` et `kernel/`.

Enfin, si l'on étudie la dernière carte correspondant au noyau 3.9 dans son instance **allyesconfig**, en dehors de l'évolution de l'échelle (passant à 0.30), il n'y a pas de différence majeure par rapport à la version 3.0 : le changement de couleur observé est lié à la croissance de l'échelle.

A.6.2 Carte de chaleur du noyau, sous-répertoire `drivers/`

La section précédente a permis de mettre en lumière le taux important de liens internes, notamment dans le sous-répertoires `drivers/`. Les cartes présentées dans la figure A.15 vont nous permettre de plonger un peu plus dans les détails. Dans un premier temps, l'instance **defconfig** du noyau 3.0 nous montre des caractéristiques similaires à celles déjà observées sur la carte correspondant au noyau en entier, à savoir, la présence de lignes horizontales et l'existence de la première bissectrice. Ces lignes font ressortir principalement deux répertoires, `base/` et `pci/`, qui sont utilisés par bon nombre d'autres répertoires, ce qui ne laisse pas de place au doute quant à leur rôle de pile applicative. Sur la première bissectrice, deux sous-répertoires sortent du lot : `gpu/` et `acpi/` ; ce sont ceux qui s'utilisent le plus, suivi ensuite par `pci/`, `usb/` et `video/`. Le passage au noyau 3.9 ne montre pas de changement visible, au-delà de l'apparition de certains sous-répertoires (`gpio/`), et l'échelle de couleur est restée équivalente.

Avec le passage à l'instance **allyesconfig**, la première surprise provient de la quantité de sous-répertoires qui apparaissent. La seconde concerne l'échelle, qui est divisée par deux, passant de 0.25 à 0.12. La première bissectrice, quant à elle, montre plus de points avec un taux d'utilisation élevé : `net/` est le premier, suivi de `staging/` puis de `media/` et enfin de `gpu/`. Une ligne verticale se forme, également, au niveau du répertoire `staging/` :



(a) Noyau 3.0, **defconfig**

(b) Noyau 3.9, **defconfig**

(c) Noyau 3.0, **allyesconfig**

(d) Noyau 3.9, **allyesconfig**

FIGURE A.14 – Cartes de chaleur des noyaux 3.0 et 3.9, instances **defconfig** et **allyesconfig**, pour le premier niveau de répertoires.

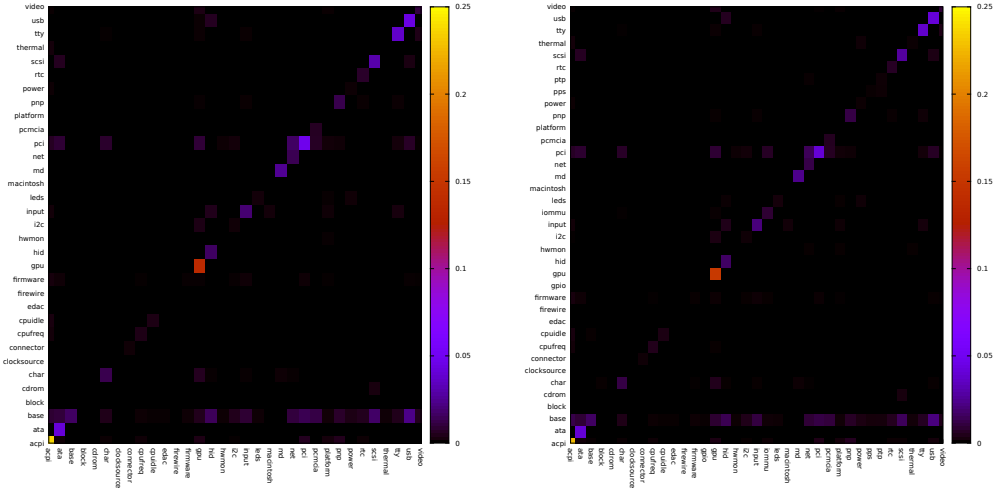
celui-ci contenant de nombreux pilotes de types différents, il parait normal que ceux-là référencent d'autres répertoires. Enfin, cette instance révèle trois lignes horizontales, sur les répertoires `base/`, `pci/` et plus légèrement pour `usb/`. La présence d'une utilisation visible du répertoire `i2c/` par `media/` (pilotes pour la gestion des caméras, des cartes d'acquisition, etc.), `mtd/` (gestion de certains types de périphériques mémoires, principalement les mémoires Flash) et `hwmon/` (surveillance matérielle) s'explique assez bien par l'utilisation courante de ce bus I^2C au sein de ce type de périphériques. Le noyau 3.9 ne montre que quelques évolutions à la marge, tels l'ajout de nouveaux sous-répertoires.

A.6.3 Carte de chaleur du noyau, sous-répertoire `fs/`

Les cartes présentées dans la figure A.16 permettent de plonger au cœur du sous-répertoire `fs/`. L'organisation de celui-ci est un peu différente des autres : la première bissectrice est bien identifiable, mais, et surtout sur l'instance **defconfig**, il est difficile de voir apparaître une ligne horizontale caractéristique comme dans les cas précédents. Pour le noyau 3.0, une grosse utilisation des répertoires `jbd/` et `quota/` est visible, malgré tout, de la part du répertoire `ext3/` pour le noyau 3.0. Ces résultats peuvent paraître surprenants, mais en réalité ils sont logiques : `jbd/` est chargé de la gestion de la journalisation, et `quota/` s'occupe des quotas d'utilisation. Or, sur l'instance analysée, le seul « vrai » système de fichier qui soit activé et qui puisse faire usage de ces fonctionnalités eeset `Ext3`. Les autres sont soit des pseudo-systèmes de fichiers (`sysfs/`, `deboguefs`, `devpts`), des systèmes de fichiers réseaux (`nfs`) ou des implémentations de systèmes de fichiers qui n'ont pas ce type de fonctionnalités (`fat/` et `isofs`). La première bissectrice rapporte un taux d'utilisation locale des symboles assez importante, et sur quasiment tous les sous-répertoires : les deux plus gros utilisateurs sont `nfs/` puis `ext3/`. La version 3.9 amène quelques évolutions, dont la plus notable est le remplacement de `Ext3` par `Ext4` comme système de fichier par défaut, intervenu au cours [3] du développement du noyau 3.6, et le remplacement de `JBD` par `JBD2`. Outre un petit changement de l'échelle, passant de 0.3 à 0.35, les liens au sein du sous-répertoire `ext4/` sur le noyau 3.9 se renforcent, comparé aux liens dans `ext3`.

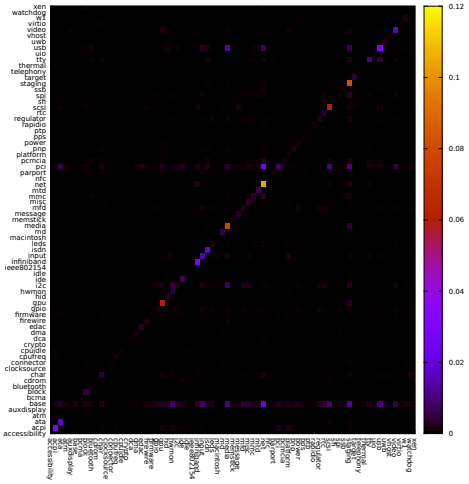
Le passage sur l'instance **allyesconfig** montre, d'abord, la quantité de systèmes de fichiers optionnels. L'échelle diminue, passant à 0.14. Mais la quantité de code utilisé reste principalement au sein des sous-répertoires, comme l'atteste la présence marquée de la première bissectrice. Quelques sous-répertoires correspondent à des composants réutilisés par plusieurs systèmes de fichiers, principalement `proc/` et `nls/`, mais avec un taux assez faible. Le répertoire `quota/` ne reste utilisé principalement que par les systèmes de fichiers `Ext`, `OCFS2` et `ReiserFS`. Il en est de même pour `jbd/` et `jbd2/`, utilisés respectivement par les systèmes de fichiers `Ext3` et `Ext4`. Quelques points chauds sont visibles sur la première bissectrice, dont les principaux se trouvent être `btrfs/` et `ocfs2/`, puis `ext4/`, `gfs2/`, `jffs2/` et enfin `nfs/`. À noter qu'avec le noyau 3.9, le taux au sein du répertoire `nfs/` augmente sensiblement, la bascule se situant à l'instant de la version 3.6 : celui-ci procède à une épuration du code du protocole `NFS`, à l'issue de laquelle seule le code de la version 4 est toujours activée par défaut dans notre configuration.

3. http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=3fe2cb8f9e9486980ff03d7e020781cfdb028ffa

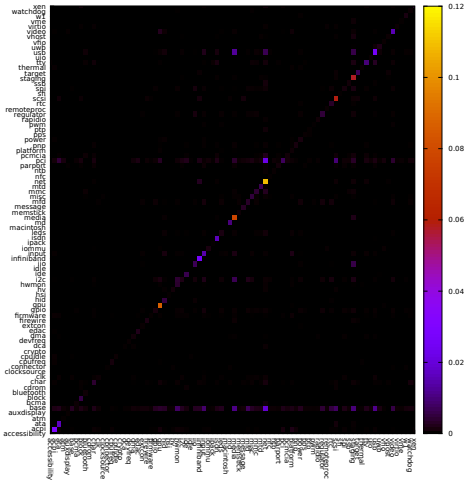


(a) Noyau 3.0, **defconfig**

(b) Noyau 3.9, **defconfig**

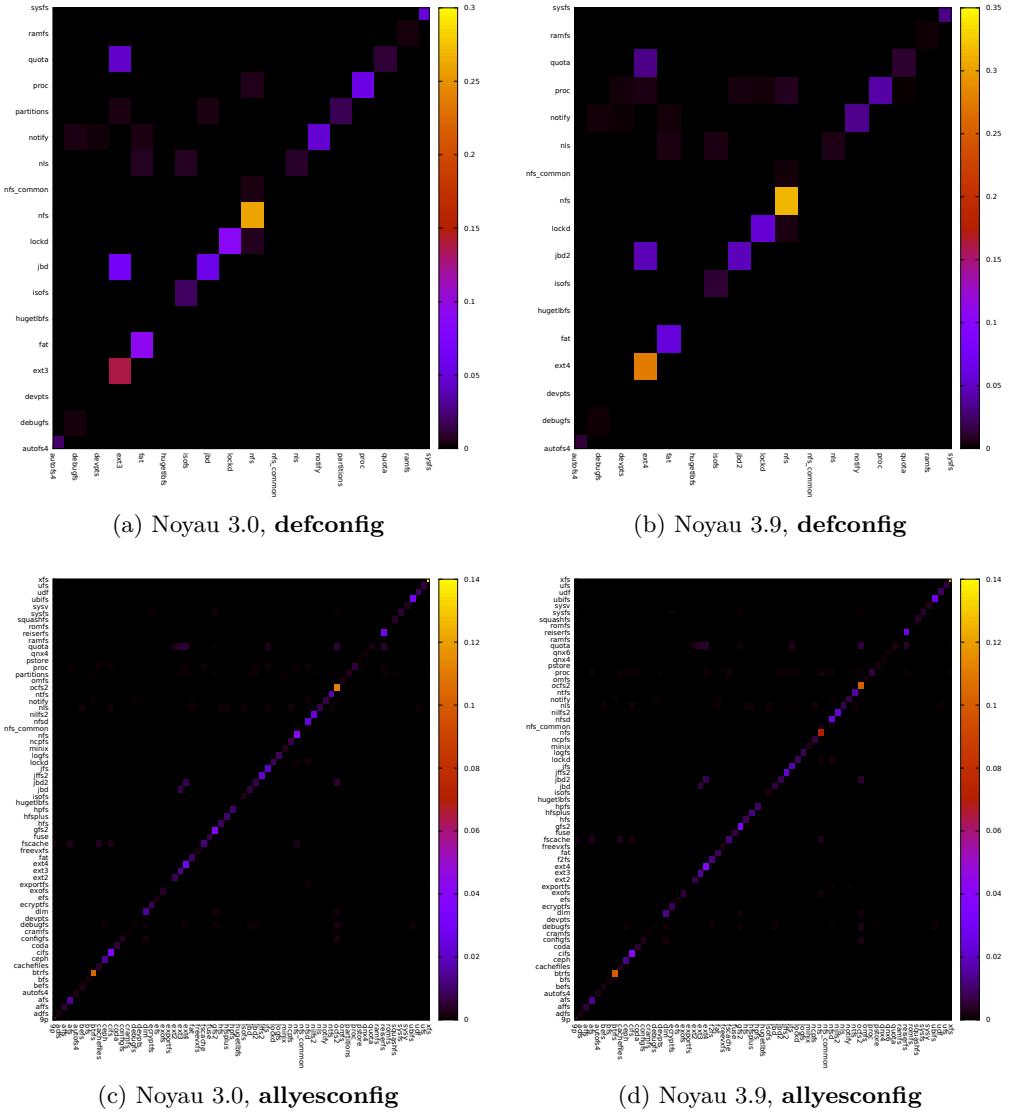


(c) Noyau 3.0, **allyesconfig**



(d) Noyau 3.9, **allyesconfig**

FIGURE A.15 – Cartes de chaleur des noyaux 3.0 et 3.9, instances **defconfig** et **allyesconfig**, pour le sous-répertoire `drivers/`.



(a) Noyau 3.0, **defconfig**

(b) Noyau 3.9, **defconfig**

(c) Noyau 3.0, **allyesconfig**

(d) Noyau 3.9, **allyesconfig**

FIGURE A.16 – Cartes de chaleur des noyaux 3.0 et 3.9, instances **defconfig** et **allyesconfig**, pour le sous-répertoire `fs/`.

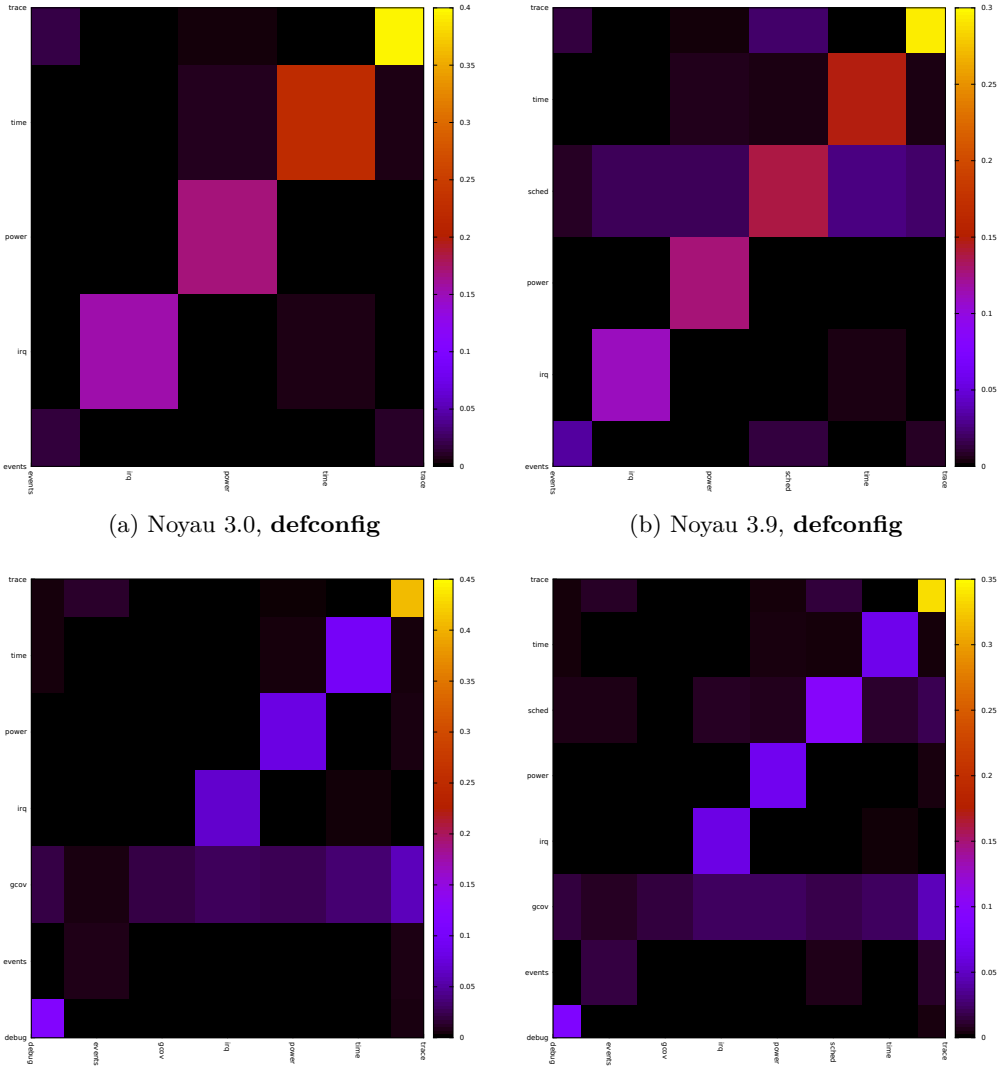
A.6.4 Carte de chaleur du noyau, sous-répertoire `kernel/`

Intéressons-nous maintenant au contenu du répertoire `kernel/`, dont les cartes de chaleur sont visibles en figure A.17 : il contient moins de composants, que ce soit dans l'instance **defconfig** ou dans l'instance **allyesconfig**. La première bissectrice se retrouve, et elle présente des taux assez forts, indiquant une bonne concentration des usages au sein des sous-répertoires qui composent `kernel/` : `trace/` présente le plus important, suivi ensuite par `time/` et enfin `power/` et `irq/`. L'évolution principale avec le passage sur la version 3.9 du noyau concerne l'apparition du sous-répertoire `sched/`, suite à un nettoyage de code effectué au cours du développement de la version 3.2. Ce répertoire est d'autant plus intéressant, qu'avec son arrivée on constate la formation d'une ligne horizontale, qui touche tous les sous-répertoires : cela indique que le code exporté par `sched/` est utilisé par les autres modules. Dans le noyau 3.9, l'ordre des sous-répertoires les plus utilisateurs de symboles en leur sein reste identique à celui du noyau 3.0.

Avec l'instance **allyesconfig**, les résultats précédents se retrouvent : le passage au noyau 3.9 et l'apparition du répertoire `sched/` font apparaitre une ligne horizontale. La quantité de code utilisé ici semble, par contre, moins importante, au regard de l'échelle. Mais l'activation de cette instance montre également un autre élément, qui a déjà été documenté lors de l'analyse des nœuds et des arcs : le répertoire `gcov/`, qui est utilisé par tous les autres sous-répertoires, comme l'atteste la présence de la ligne horizontale. Enfin, si `trace/` reste en tête en matière de quantité de liens utilisés en interne, la répartition entre les autres répertoires est plus égale, quelle que soit la version du noyau étudiée.

A.6.5 Carte de chaleur du noyau, sous-répertoire `net/`

Passons maintenant au répertoire `net/` qui contient différents sous-systèmes réseaux : il s'agit ici d'implémentation de protocoles, et non de pilotes de périphériques réseaux, ces derniers étant dans le répertoire `drivers/net/`. Les cartes de chaleur illustrant ce cas sont visibles en figure A.18. Dans le noyau 3.0 avec l'instance **defconfig**, la lecture de cette carte nous permet d'observer la présence marquée de la première bissectrice : cela indique donc qu'une bonne partie des composants de ce répertoire ont une majorité de liens en leur sein, et plus particulièrement `ipv4/`, `ipv6/`, puis `sunrpc/` (implémentation du protocole `SunRPC`, aussi connu sous le nom de `ONCRPC`, *Open Network Computing Remote Procedure Call*, et qui est notamment utilisé pour `NFS`), `mac80211/` (implémentation du `Wi-Fi`) et enfin `netfilter/` (pare-feu). Outre cette première bissectrice, plusieurs lignes horizontales sont visibles, la première étant située sur le sous-répertoire `core/` et impactant quasiment tous les sous-répertoires, dont les deux plus importants sont `ipv6/` puis `ipv4/` : `core/` implémente de nombreuses structures de base pour le réseau, tels la gestion des périphériques ou la gestion des tampons `skbuff`, qui sont des composants communs à une majorité des protocoles réseaux qui sont implémentés dans le noyau Linux. Une seconde ligne horizontale peut se distinguer sur le répertoire `netlink/` notamment à l'intersection avec `core/`, `ipv4/`, `netfilter/`, `netlabel/` et `wireless/` : `NetLink` implémente une famille de socket réseau permettant un dialogue entre espace utilisateur et espace noyau – en remplacement de l'ancienne méthode basée sur des `IOCTLs` –, et est notamment utilisé par les outils de configuration tels `iproute2` ou la gestion du pare-feu.



(a) Noyau 3.0, **defconfig**

(b) Noyau 3.9, **defconfig**

(c) Noyau 3.0, **allyesconfig**

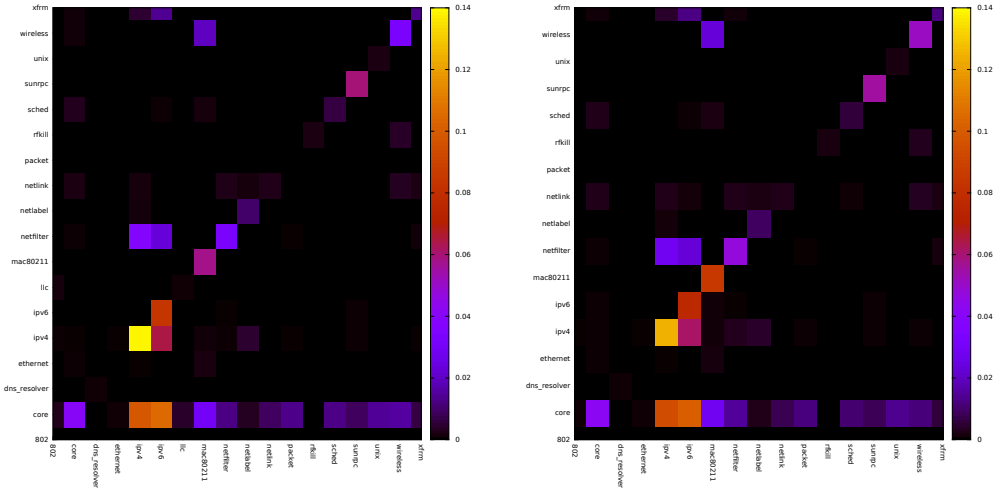
(d) Noyau 3.9, **allyesconfig**

FIGURE A.17 – Cartes de chaleur des noyaux 3.0 et 3.9, instances **defconfig** et **allyesconfig**, pour le sous-répertoire `kernel/`.

Des liens unissent les protocoles `ipv4/` et `ipv6/` avec le code de gestion du pare-feu, `netfiler/`, ces derniers étant utilisateurs du module `netfilter/`. Il est intéressant, par ailleurs, de constater que le taux d'utilisation au sein du répertoire `ipv4/` est plus fort que celui au sein du répertoire `ipv6/`, ce qui pourrait s'expliquer par un protocole plus complexe à implémenter (la simplification des entêtes avec `IPv6` étant un argument fort de son déploiement). On note aussi que la rétro-compatibilité avec `IPv4` est visible sur cette carte, puisque le répertoire `ipv6/` a un lien assez visible d'utilisateur du répertoire `ipv4/`.

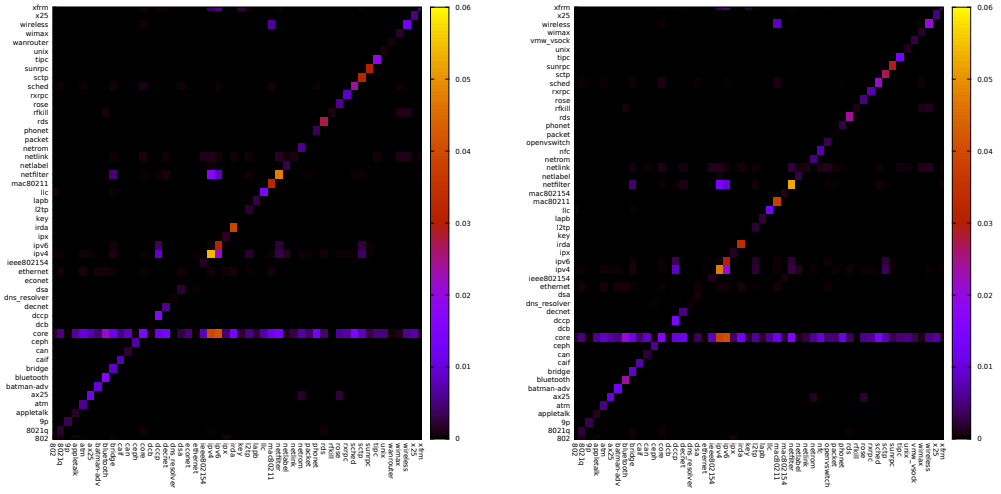
L'évolution temporelle avec le noyau 3.9 ne montre pas de changement dans l'échelle de la carte, par contre, certains répertoires ont une évolution intéressante : si la ligne horizontale de `core/` reste proche de celle observée pour le noyau 3.0, celle de `netlink/` révèle une hausse des contributions d'utilisation de la part des répertoires `ipv4/`, `ipv6/`, `netfilter/` et `netlabel/`. Le sous-répertoire `llc/`, en charge du protocole *Logical Link Control*, disparait, en tout cas dans l'instance **allyesconfig**. Simultanément, le répertoire `mac80211/` voit sa quantité de code interne utilisé croître de manière significative avec cette version 3.9, la bascule ayant eu lieu entre les versions 3.2 et 3.3.

Sur l'instance **allyesconfig**, les noyaux 3.0 et 3.9 se comportent de manière assez similaire là encore : l'échelle, bien que réduite par rapport aux cas de l'instance **defconfig** (passant de 0.14 à 0.06), est similaire entre ces deux versions. La première bissectrice reste similaire sur les sous-répertoires déjà présents dans l'instance précédente, et la ligne horizontale de `core/` est confirmée. L'arrivée du sous-système `ethernet/` montre également une telle ligne, bien qu'elle soit moins marquée. Parmi les nouveaux composants qui ont un fort taux d'utilisation interne, on trouve `irda/` (en charge du support de l'infrarouge) et `sctp/` qui implémente le protocole *SCTP*. L'intersection de ce répertoire avec ceux des protocoles `IPv4` et `IPv6` s'explique assez bien par le fait que c'est un protocole amené à vivre au-dessus de ces derniers, et à côté de protocoles tels que `TCP`. Au cours du temps, le taux diminue sur ce répertoire, comme le montre la carte du noyau 3.9.



(a) Noyau 3.0, **defconfig**

(b) Noyau 3.9, **defconfig**



(c) Noyau 3.0, **allyesconfig**

(d) Noyau 3.9, **allyesconfig**

FIGURE A.18 – Cartes de chaleur des noyaux 3.0 et 3.9, instances **defconfig** et **allyesconfig**, pour le sous-répertoire `net/`.

Annexe B

Analyse des résultats de la détection de communautés

Nous avons introduit respectivement les différentes méthodes que nous estimons pouvoir appliquer dans notre cas et la démarche que nous allons suivre pour les évaluer dans les sections 4.3 et 4.4. Dans la suite de cette annexe nous documentons les résultats détaillés des différentes méthodes sur les graphes pour lesquels nous les avons évaluées. Les éléments importants sont présentés dans la section 4.5. Pour chaque méthode, nous documentons les paramètres influents sur les résultats et le choix retenus pour ces valeurs. Un ensemble de 256 itérations ont été effectuées pour chaque cas, afin soit de comparer l'influence ou de s'assurer des variations liées à l'utilisation d'aléa.

B.1 Résultats pour Blondel et al. (2008)

La méthode proposée par Blondel et al. est hiérarchique, comme cela a déjà été présenté précédemment, elle admet donc un paramètre d'utilisation qui permet de contrôler le niveau de profondeur que nous souhaitons obtenir. La pondération des arcs peut être prise en compte par l'implémentation, nous avons donc extrait et utilisé cette information pour le calcul des mesures de performance. Pour la comparaison effectuée, nous avons choisi de ne retenir que la valeur de niveau la plus élevée obtenue : le nombre de niveaux détecté varie entre 3 et 4, et ces deux derniers présentent dans le cas des instances **defconfig** et **allyesconfig** des similarités très proches avec l'arborescence du système de fichier.

Le premier graphique de résultat que nous pouvons analyser pour cette méthode concerne la taille des communautés détectées par la méthode et est visible en figure B.1a. Des variations sont visibles dans les cas **defconfig** et **allyesconfig**, mais la courbe moyenne des valeurs peut respectivement s'établir autour d'environ 140 et 450 ; de plus les écarts semblent plus faibles dans le premier cas. La pertinence des communautés qui sont détectées peut-être déduite à partir du second graphique qui est proposé en figure B.1b et qui documente le taux de sous-répertoires communs au sein d'une communauté tel que défini dans la section 4.4.1.1 : plus ce taux tends vers 0, plus les composants sont proches entre eux. Nous pouvons constater, tant pour **defconfig** que **allyesconfig**, que les valeurs sont

faibles avec respectivement de l'ordre de 0.17 et 0.13, ce qui dénote à la fois qu'une partie suffisamment importante des composants de chaque communauté correspond au même sous-répertoire ; et surtout que le cas **allyesconfig** présente de meilleures propriétés quant à ce phénomène.

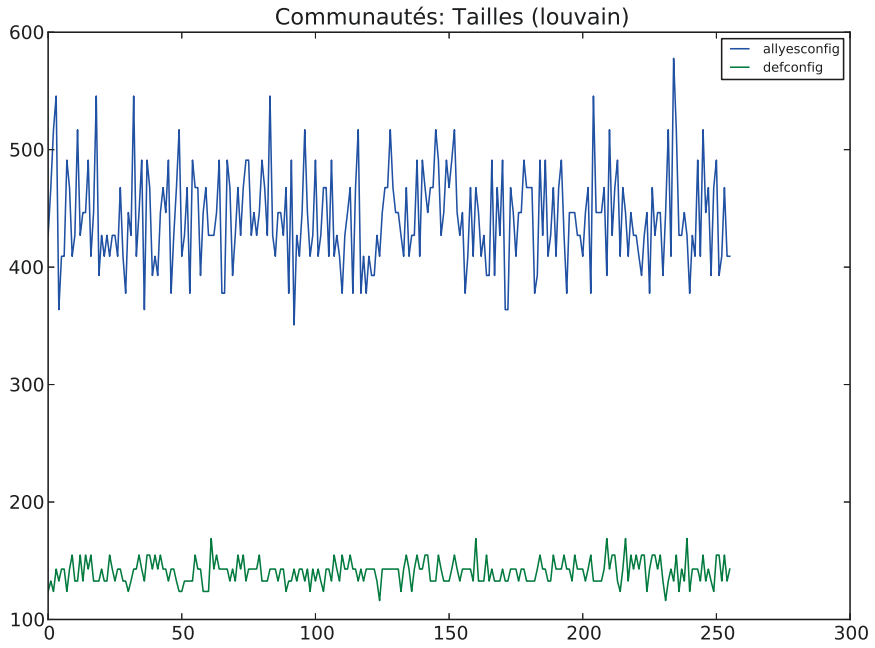
Incidemment, le graphique B.2 documentant la quantité de communautés détectées présente un comportement proche de celui considérant la taille (B.1a) : les valeurs oscillent autour de 13 et respectivement 22 pour les cas **defconfig** et **allyesconfig**. Ici nous constatons donc l'influence du paramètre sélectionnant le niveau hiérarchique retenu pour l'analyse, comme précisé au début de cette section : la détection d'environ 13 communautés dans le cas de la configuration par défaut se corrèle très bien avec le premier niveau de l'arborescence des sources du noyau `Linux`. Un œil attentif aux données collectées au cours des analyses montre que, sur ce cas, et au niveau précédent, la quantité de communautés est de l'ordre de 80 à 90, et cela correspond également à peu près à la population de modules existants dans cette configuration. Le cas de la configuration complète, **defconfig**, montre une variation autour d'environ 22 – toujours alors que nous nous intéressons au plus haut niveau hiérarchique détecté par la méthode –, ce qui est supérieur au premier niveau d'arborescence. Cette quantité supérieure couplée au taux de concentration inférieur que nous avons documenté précédemment montre qu'il existe plusieurs communautés correspondant au même sous-répertoire. Si nous allons voir les données des autres niveaux de détection, nous constatons une évolution croissante par niveau décroissant : 22 (niveau 3), 36 (niveau 2), puis 360 (niveau 1). Au-delà, les nœuds ne forment plus de communautés. Ces niveaux 3 et 2 proposent un nombre de communautés qui ne correspond pas à la réalité de l'arborescence du système de fichier. Par contre, le premier niveau hiérarchique est plus proche : l'arborescence compte, à un niveau équivalent, de l'ordre de 300 répertoires.

La mesure de la taille des interconnexions permet d'avoir une idée des liens entre les communautés et est définie dans la section 4.4.1.2. Pour la méthode étudiée, le graphique est proposé en figure B.3. Nous pouvons d'abord constater que la variabilité des valeurs est plus forte dans le cas de l'instance par défaut, **defconfig** que sur le cas **allyesconfig**, le taux évoluant respectivement autour de 0.52 et 0.60. Malgré les différentes variations locales sur les instances étudiées, la tendance générale est très stable, et il ne semble pas y avoir d'évolution fondamentale du taux.

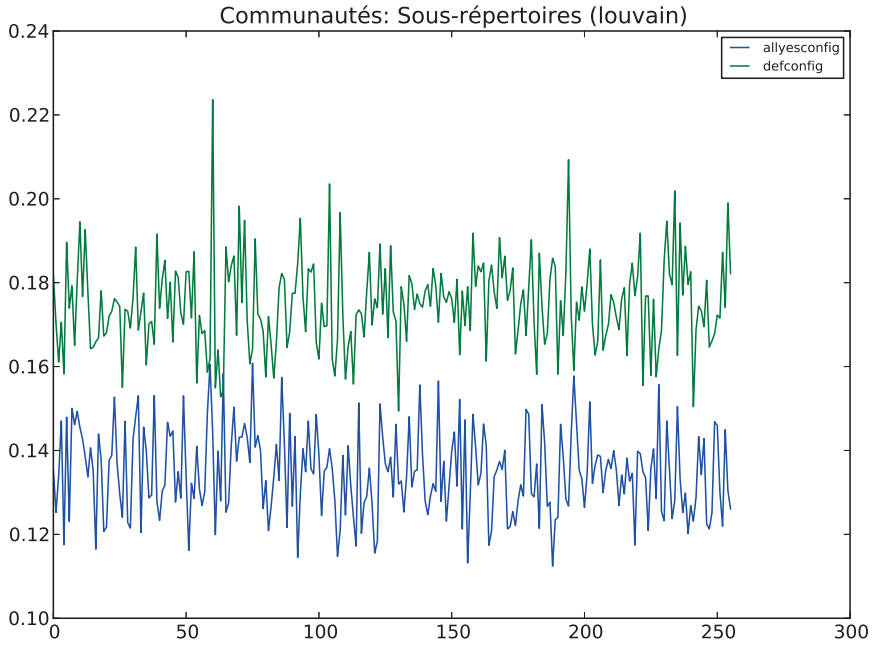
Enfin, les graphiques B.4a et B.4b présentent respectivement le temps de calcul et la consommation mémoire pour les deux cas **defconfig** et **allyesconfig** sur l'ensemble des instances étudiées. Cette méthode propose la complexité la plus faible comme indiqué dans le tableau 4.3, et l'impact sur les temps de calcul est visible : le cas **defconfig** est résolu en moins de 0.1 seconde, et l'instance la plus grosse, **allyesconfig** en moins de 0.5 seconde. Quelques variations sont visibles à la marge, sans influence notable sur le temps de calcul. De même, la consommation mémoire est raisonnable, de l'ordre de 35MiO pour **defconfig** et **allyesconfig** se contente d'environ 75MiO.

B.2 Résultats pour Clauset, Newman et Moore (2004)

La méthode proposée par Clauset, Newman et Moore n'admet pas de paramètre permettant d'adapter son exécution. De plus le graphe qui est accepté en entrée n'est ni



(a) Taille des communautés



(b) Taux de concentration

FIGURE B.1 – Variations des différentes instances de BLONDEL et al. (2008) sur la taille et la concentration des communautés

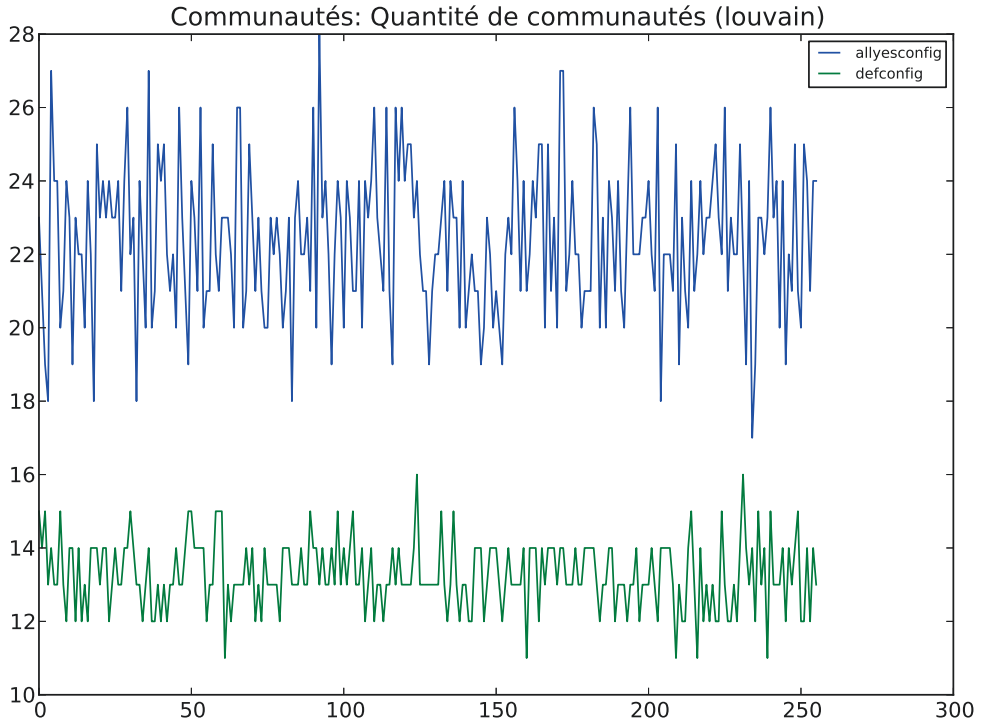


FIGURE B.2 – Variations des différentes instances de BLONDEL et al. (2008) sur le nombre de communautés

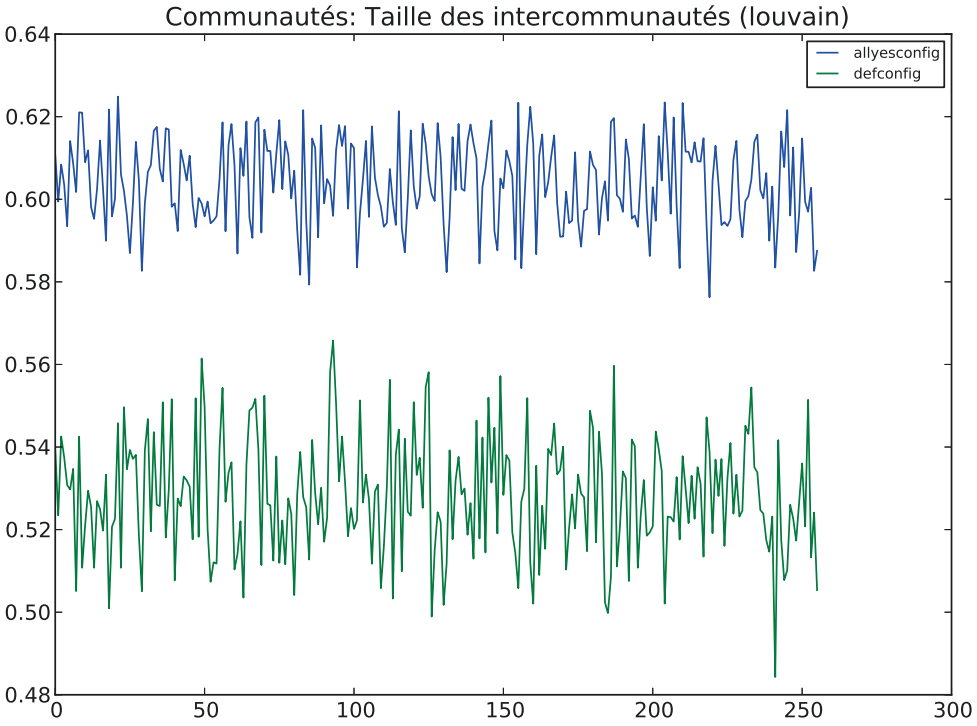
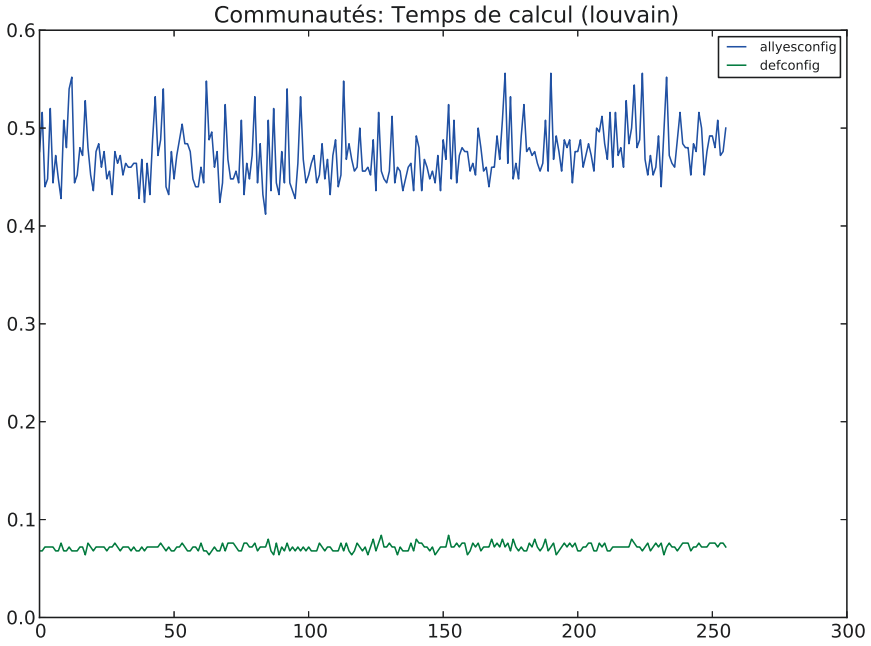
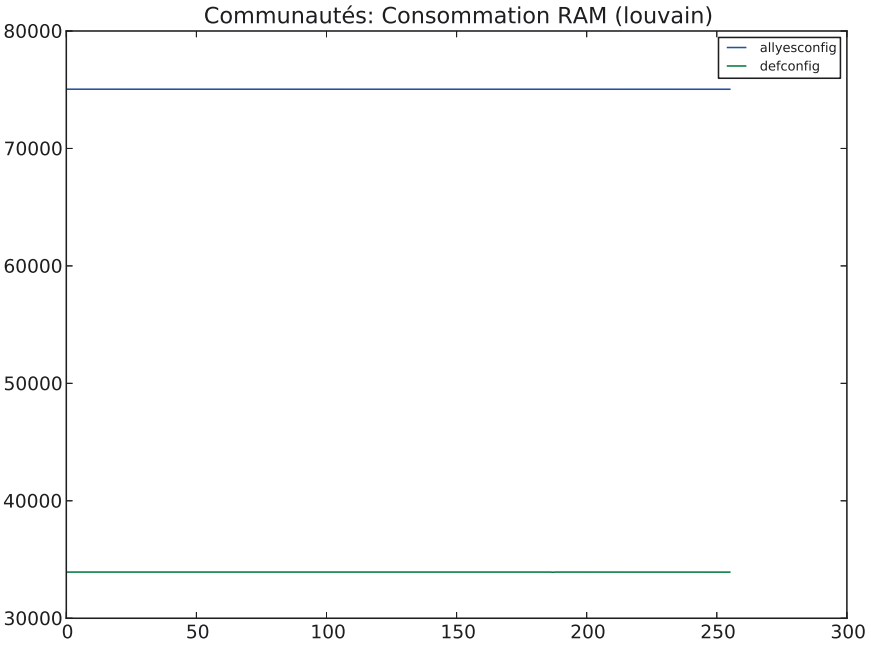


FIGURE B.3 – Mesure de la taille des interconnexions des communautés avec BLONDEL et al. (2008)



(a) Temps de calcul



(b) Consommation mémoire

FIGURE B.4 – Consommation mémoire et processeur pour l'exécution de BLONDEL et al. (2008)

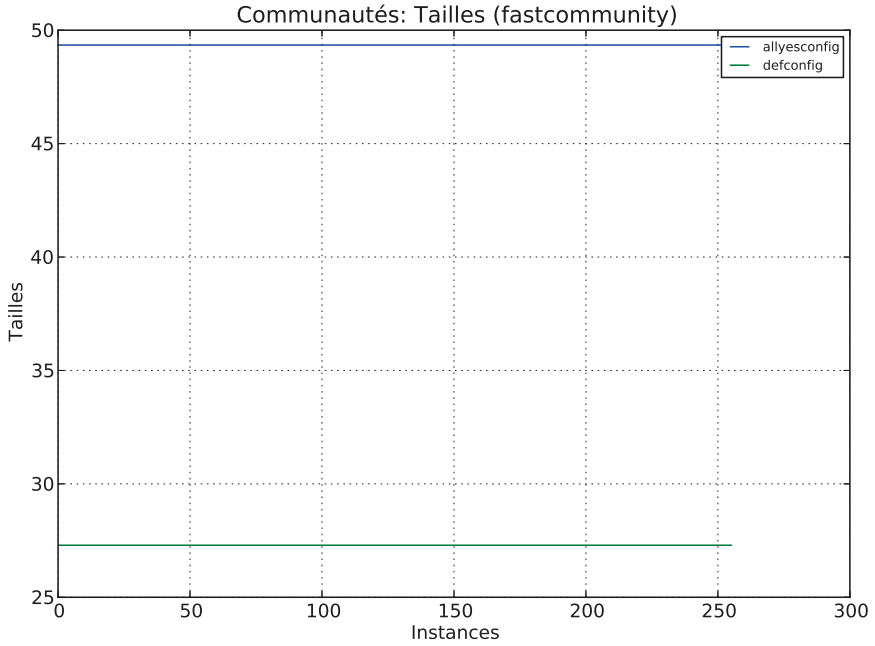
orienté, ni pondéré. Comme pour les autres cas, un jeu de 256 instances a été étudié pour les cas **defconfig** et **allyesconfig** afin de pouvoir observer les éventuelles variations liées à l'utilisation d'aléa.

Les graphiques documentant la taille des communautés (B.5a) ainsi que le taux de concentration en leur sein (B.5b) exposent un comportement différent de celui que nous avons pu observer dans la méthode précédente : ici, sur toutes les instances, le résultat est parfaitement stable ; nous avons ainsi respectivement 27 et 49 nœuds en moyenne dans les communautés détectées. Derrière cette droite, cependant, les données indiquent des écarts très importants. Un si faible nombre de nœuds ne reflète pas correctement la hiérarchie du système de fichier. Si nous nous intéressons au taux de concentration, documenté dans le second graphique, nous voyons d'une part qu'il est très proche entre les deux cas (**defconfig** et **allyesconfig**), la différence entre les deux étant d'un ordre de grandeur inférieur à celle observée pour la méthode précédente. Ensuite nous constatons que ce taux se situe entre 0.80 et 0.825, ce qui parait de nouveau autrement plus élevé que dans la méthode précédente. Ces premiers éléments suggèrent que cette approche est inadaptée à notre problème : elle ne recouvre pas de communautés correspondant au système de fichier (contrairement à la précédente) ; et le taux de concentration des communautés détectées est élevé, ce qui indique que les membres de chacune n'ont pas tant de composants en commun. La faible taille des communautés masque une importante variance, ce qui les rends peu intéressantes pour notre second cas d'application. La forte hétérogénéité des communautés peut cacher des comportements qu'il conviendrait d'étudier.

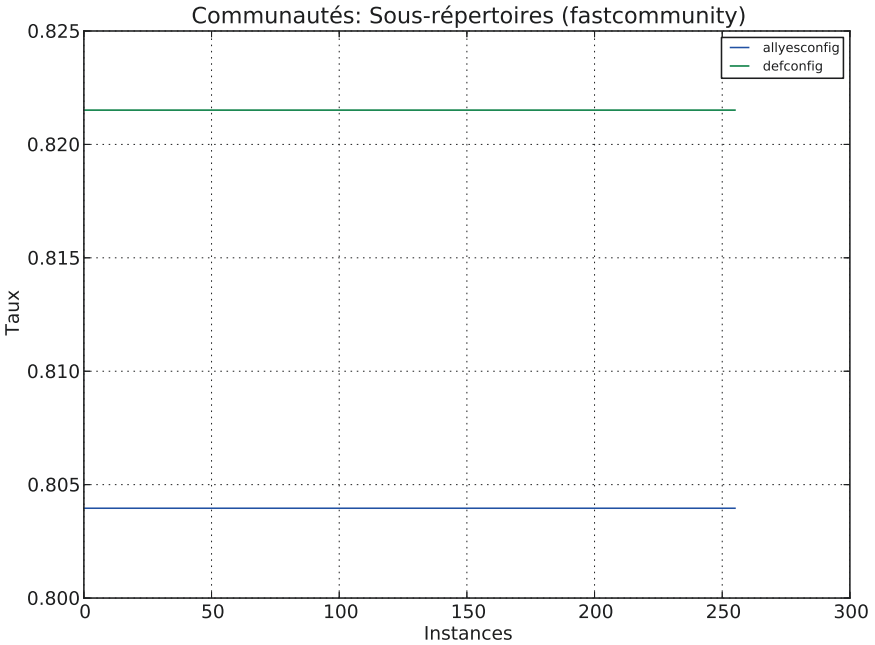
La quantité de communautés détectées est présentée dans le graphique B.6 et est, aussi, complètement stable, dans les deux cas **defconfig** et **allyesconfig**, avec des valeurs respectives de 68 et de 199. Cette parfaite stabilité est, contrairement à la taille, corroborée par toutes les données : les variations de tailles de communautés se font donc par migration de nœuds entre elles.

La taille des interconnexions pour cette méthode est proposée dans le graphique B.7 et est également parfaitement stable avec des valeurs respectives de 0.396 et 0.392 pour les cas **defconfig** et **allyesconfig** : cette fois-ci, l'écart est très faible ; et les valeurs sont également basses et notamment plus basses que celles de la méthode précédente. Cela indique que le nombre d'arcs inter-communautés est plus faible, et donc que les connexions sont plus facilement contrôlables. Le second cas d'application de la détection de communauté deviendrait ainsi intéressant avec cette méthode, en mettant de côté l'importante variance des communautés formées documenté précédemment.

Si le temps de calcul est impacté par la complexité plus importante, comme cela est visible dans le graphique B.8a, la consommation mémoire (B.8b) est par contre très similaire à celle de la méthode précédente, et oscille entre 35MiO pour le cas **defconfig** et 75MiO dans le cas **allyesconfig**. Le temps d'exécution, lui, varie de moins d'une seconde en général pour **defconfig** alors qu'il atteint autour de 100 secondes pour l'autre cas. Malgré quelques variations locales, nous ne pouvons pas constater de tendance particulièrement sur le temps de calcul et celui-ci est plutôt stable pour toutes les instances considérées.



(a) Taille des communautés



(b) Taux de concentration

FIGURE B.5 – Variations des différentes instances de CLAUSET, NEWMAN et MOORE (2004) sur la taille et la concentration des communautés

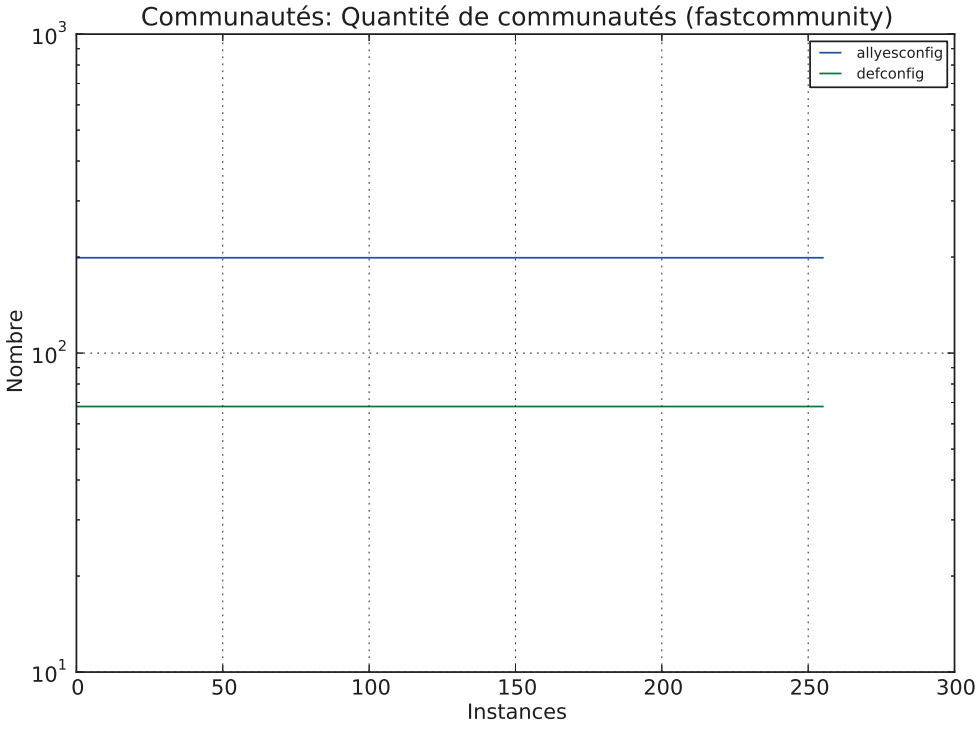


FIGURE B.6 – Variations des différentes instances de CLAUSET, NEWMAN et MOORE (2004) sur le nombre de communautés

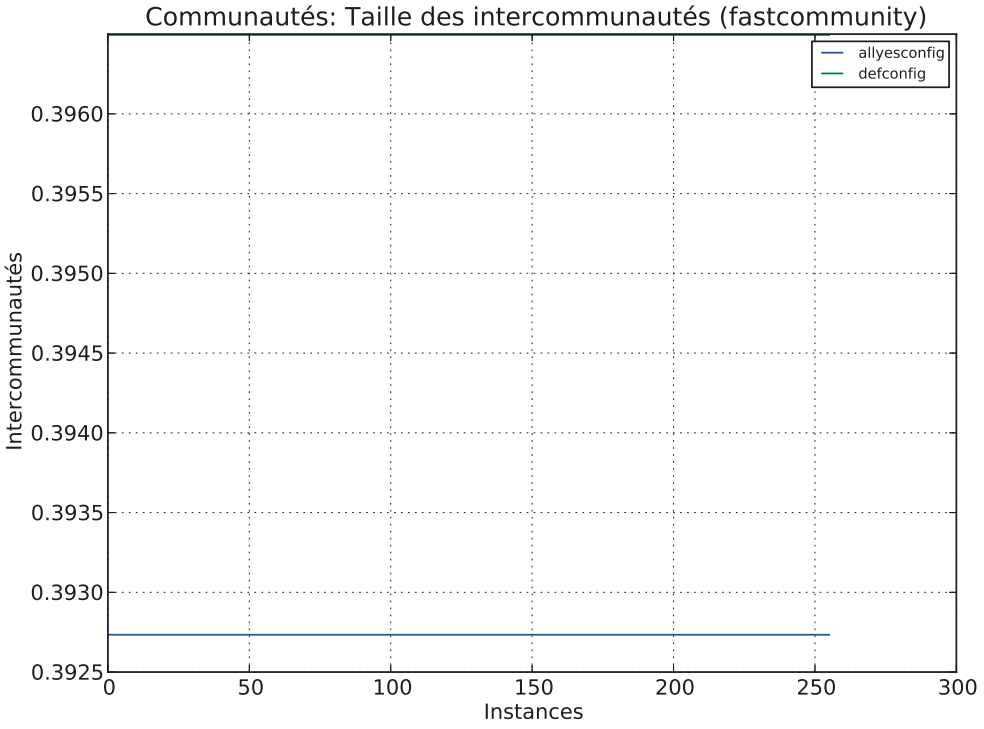
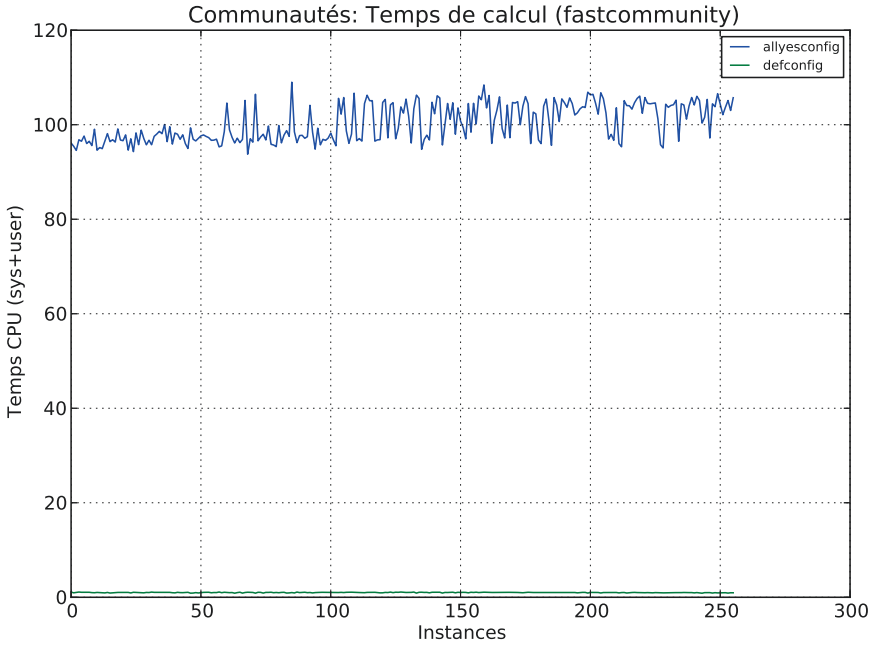
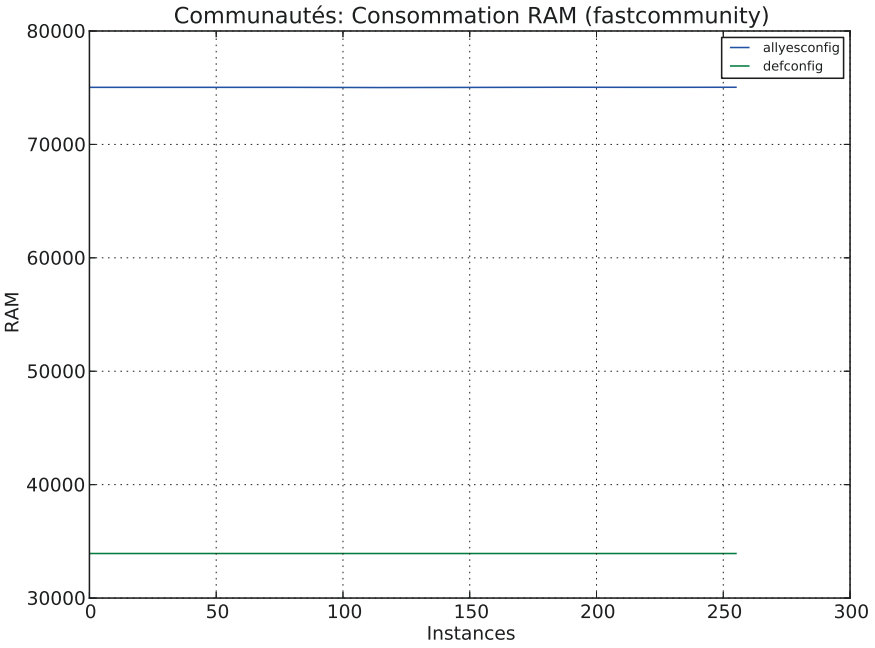


FIGURE B.7 – Mesure de la taille des interconnexions des communautés avec CLAUSET, NEWMAN et MOORE (2004)



(a) Temps de calcul



(b) Consommation mémoire

FIGURE B.8 – Consommation mémoire et processeur pour l'exécution de CLAUSET, NEWMAN et MOORE (2004)

B.3 Résultats pour Hofman et Wiggins (2008)

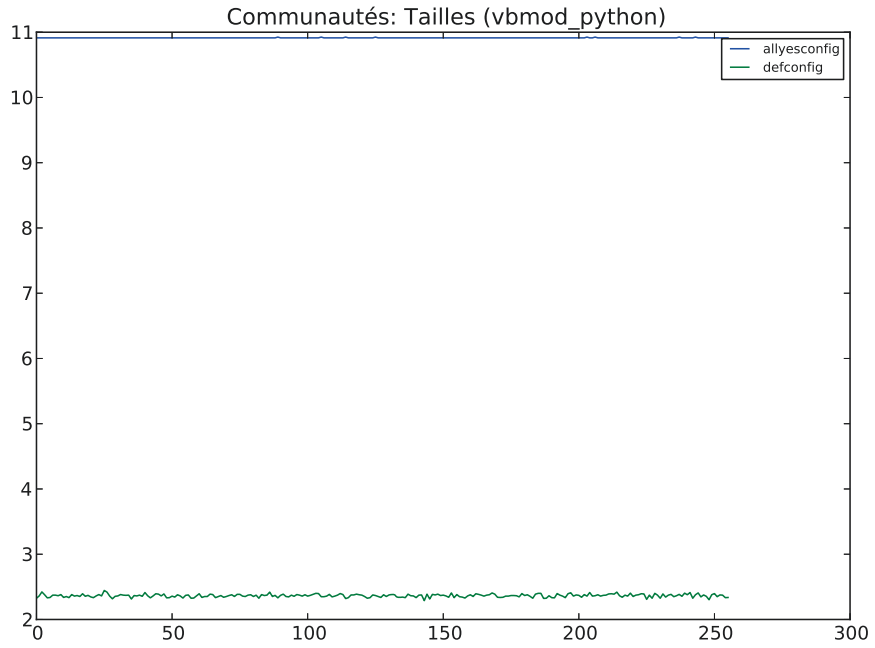
L'implémentation de cette méthode exploitant BAYES repose sur plusieurs paramètres de configuration : `K` indique le nombre de communautés ; `NUM_RESTARTS` défini le nombre de redémarrages de l'algorithme pour les comparaisons des valeurs suivant `K` ; `TOL_DF` indique la tolérance à la variation sur la valeur de retour `F` donnant la qualité de la solution pour un `K` donné. Des valeurs pour ces paramètres ont été sélectionnées après évaluation sur le cas de nos graphes. Tout d'abord, pour les valeurs de `K` à tester, nous pouvons définir une borne minimale comme étant le nombre de répertoires à la racine du noyau, et une borne maximale comme cette quantité à un niveau d'arborescence plus élevé ; ceci nous donne des valeurs de 14 pour la racine, et 248 pour le premier niveau. Si nous allons au second niveau, le nombre de répertoires atteint 872. La méthode, telle qu'elle est implémentée par ses auteurs, va calculer les communautés pour chaque `K` donné et l'évaluer sur la quantité d'énergie `F` correspondante. La valeur de `K` présentant la plus faible quantité d'énergie sera retenue.

L'implémentation est réalisée en PYTHON, mais pour des raisons de performance, les auteurs utilisent une petite section de code `C` pour effectuer une opération qui serait beaucoup trop lente en PYTHON. La complexité évaluée à $O(n^{\alpha})$ avec $\alpha \approx 1.44$ se vérifie sur le temps d'exécution avec la méthode accélérée : l'approche purement PYTHON est plusieurs ordres de grandeur plus lente à l'exécution.

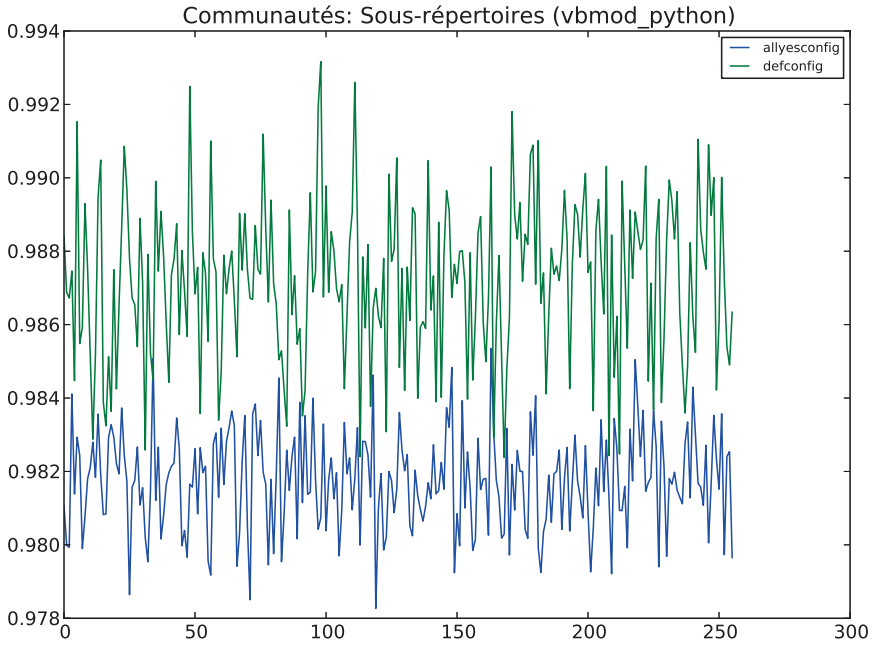
Des essais nous montrent que sur nos graphes, une valeur de tolérance à la variation de l'énergie de l'ordre 0.01 comme celle utilisée par défaut est suffisante pour arrêter l'algorithme : l'échelle des valeurs d'énergie pour nos graphes est largement supérieure, et converge rapidement vers des valeurs stables. De plus, la méthode exploitant de l'aléa, nous allons comme pour les autres effectuer 256 itérations. Le nombre de redémarrages de l'algorithme peut donc être limité sans impacter la variabilité des résultats, celle-ci pouvant s'apprécier sur l'ensemble de nos instances. Nous retenons donc une valeur de redémarrage de 2 au lieu des 25 utilisés par défaut. Quant au nombre de communautés, nous proposons d'évaluer toutes les valeurs comprises entre 10 et 900, par pas de 10.

Les premiers résultats que nous pouvons analyser concernent la taille des communautés détectées ainsi que leur taux de concentration, dans les graphiques respectifs B.9a et B.9b. Pour le cas **defconfig**, la taille détectée est stable et légèrement au-dessus de 2 ; pour **allyesconfig**, la stabilité est toujours là et la valeur atteint environ 11. Si le taux de concentration semble varier de manière importante, c'est que l'échelle est faible ; en réalité tant pour l'instance **defconfig** que pour **allyesconfig**, les valeurs oscillent respectivement autour de 0.981 et 0.987. Celles-ci traduisent une très mauvaise concentration des communautés, c'est-à-dire qu'elles ne contiennent que peu de répertoires communs. Pour le moment, cette méthode de détection ne recouvre pas du tout l'arborescence originale du système de fichier ; par contre, elle est peut-être en mesure de détecter des communautés de taille intéressante (petite) pour le problème d'analyse et de découpage.

Le second graphique que nous proposons est visible en figure B.10, et documente le nombre de communautés qui ont été détectées. Comme attendu à la vue des résultats précédents, le nombre de communautés est important ; il varie autour de 790 pour l'instance **defconfig** et est très stable à environ 900 pour **allyesconfig**. Ce phénomène suggère



(a) Taille des communautés



(b) Taux de concentration

FIGURE B.9 – Variations des différentes instances de HOFMAN et WIGGINS (2008) sur la taille et la concentration des communautés

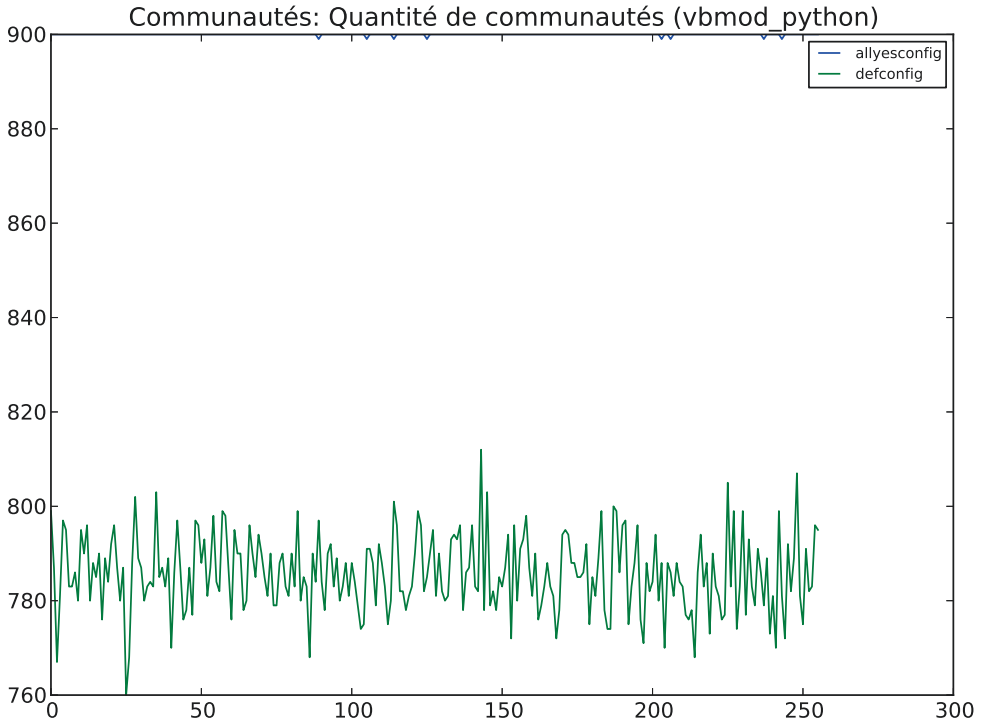


FIGURE B.10 – Variations des différentes instances de HOFMAN et WIGGINS (2008) sur le nombre de communautés

l'impact des différents pilotes de périphériques qui sont regroupés dans des communautés.

La taille des interconnexions est proposée dans le graphique B.11. Nous pouvons ainsi constater, au-delà des variations locales qui semblent importantes, que les deux cas étudiés (**defconfig** et **allyesconfig**) proposent un comportement très proche ; cette taille est quasi identique dans ces cas, et varie entre 0.9988 et 0.9990. Ces valeurs sont très fortes et indiquent une forte population inter-communautés.

Les temps de calculs, présentés dans le graphique B.12a, sont assez important ; ils passent de l'ordre de 2 000 secondes pour traiter une instance de **defconfig** à plus de 12 000 secondes pour les instances de **allyesconfig**. La consommation mémoire, également, est très importante. Celle-ci est proposée dans le graphique B.12b et oscille entre environ 1GiO pour le cas **defconfig** et environ 4GiO pour le cas **allyesconfig**. Cette méthode est donc, conformément à ce qui était prévisible par rapport à la complexité, particulièrement lourde en calcul et en mémoire. Nous devons également rappeler que cette implémentation exploite la version « optimisée » du code, dont une partie est écrite en `C` inclus dans le PYTHON ; sans cette optimisation, les temps de calculs auraient été plusieurs ordres de grandeurs plus importants, et la méthode totalement inexploitable sur nos instances.

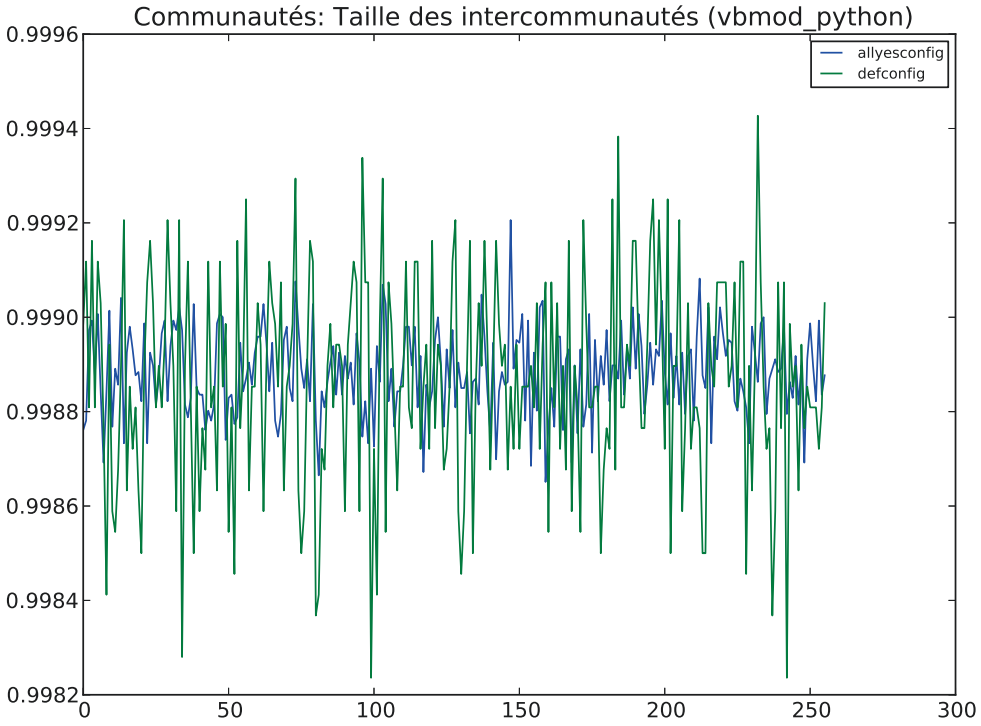
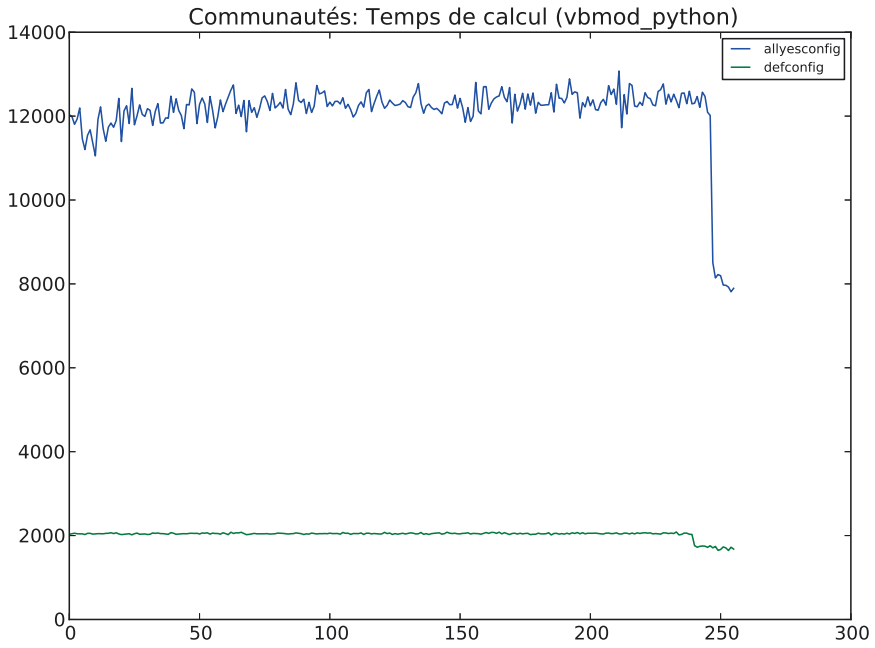
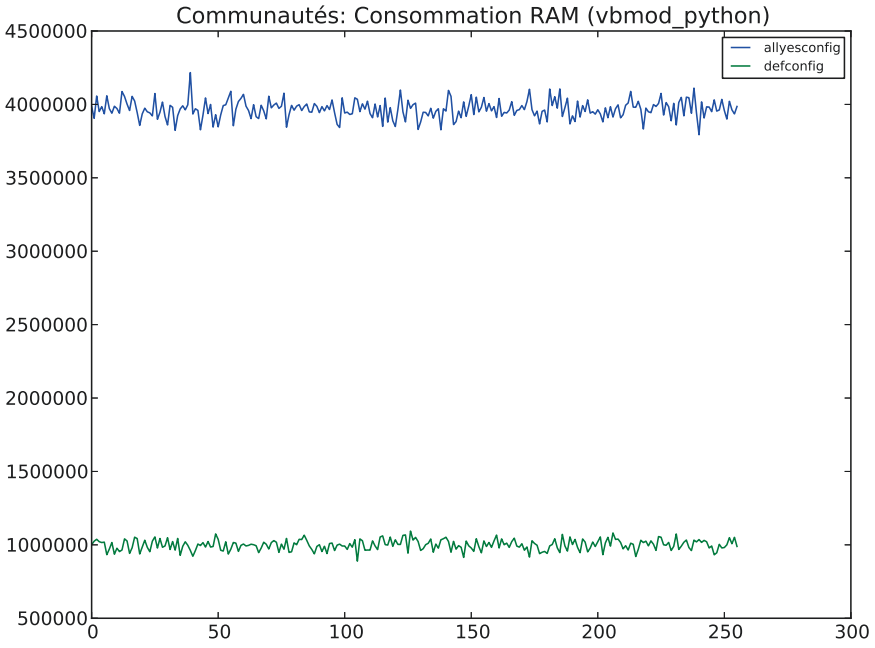


FIGURE B.11 – Mesure de la taille des interconnexions des communautés avec HOFMAN et WIGGINS (2008)



(a) Temps de calcul



(b) Consommation mémoire

FIGURE B.12 – Consommation mémoire et processeur pour l'exécution de HOFMAN et WIGGINS (2008)

B.4 Résultats pour Pons et Latapy (2005)

L'implémentation proposée par les auteurs admet un paramètre qui a une influence sur le résultat de l'analyse : la taille du chemin aléatoire suivi. Celle-ci est mise par défaut à 4, mais l'utilisateur peut explorer des valeurs différentes. L'espace mémoire autorisée pour l'exécution de la méthode n'influe pas sur le résultat mais uniquement sur le temps de traitement, selon les informations fournies par les auteurs.

Dans les graphiques présentés en figure B.13 nous pouvons observer le comportement de l'algorithme pour la détection de communautés sur les cas **defconfig** et **allyesconfig**. Les différentes instances correspondent aux valeurs prises par le paramètre de taille du chemin aléatoire : nous l'avons fait varier entre 1 et 256 par pas de 1, ces bornes étant choisies arbitrairement. Les valeurs présentées dans ces deux graphiques sont les moyennes calculées sur la population des communautés découvertes.

Un premier constat que nous pouvons faire concerne l'influence des paramètres sur les cas **defconfig** et **allyesconfig** : elle n'est pas équivalente. Notamment, nous observons que lorsque le paramètre de taille de la marche aléatoire dépasse 150, à la fois le taux d'homogénéité des sous-répertoires et la taille des communautés découvertes chutent à 0 dans le cas de **defconfig**. Cela s'explique assez bien par la faible population de cette instance, qui ne regroupe que quelques milliers de nœuds.

Au-delà des variations locales marquées, nous pouvons dégager une tendance sur les deux premières courbes visibles dans le graphique B.13a : elles se composent de trois phases. Pour le cas **defconfig**, nous observons d'abord une période de croissance sur l'intervalle de valeur de la taille du chemin aléatoire compris entre 1 et environ 30 ; puis entre 30 et 130, un plateau se forme autour de la valeur 200 ; enfin au-delà, nous notons une très rapide décroissance. Le même phénomène s'observe également pour le cas **allyesconfig**, seules les seuils et le niveau du plateau sont différents ; ainsi la croissance est visible entre 0 et environ 60, puis le plateau se forme sur l'intervalle délimité entre 60 et environ 220. Là aussi la chute est très rapide. Nous pouvons aussi constater que la variabilité semble s'amplifier avec la taille du chemin ; ainsi les valeurs situées jusqu'à une taille de 100 sont plutôt stables. En s'intéressant uniquement à la taille des communautés qui sont générées, nous pouvons conclure qu'un intervalle pertinent de valeurs pour la taille du chemin aléatoire sur ce problème et pour les deux cas du noyau que nous comparons est situé entre environ 60 et 80, comme étant l'intersection de la zone de plateau stable des deux courbes.

Étudions maintenant le taux de bonne reconnaissance des sous-répertoires au sein des communautés présenté dans le graphique B.13b et tel que défini dans la section 4.4.1.1 ; plus ce taux tends vers 0, et plus la communauté détectée est concentrée au sein d'un sous-répertoire. Le premier constat que nous pouvons établir concerne les bornes dans lesquelles les courbes évoluent : pour **defconfig**, elle évolue entre 0.2 et 0.8 en étant particulièrement concentrée entre 0.2 et 0.4 ; pour **allyesconfig**, l'intervalle est plus resserré et se limite entre 0.2 et 0.6. Sur la première courbe, celle de **defconfig**, le taux baisse de manière quasi constante jusqu'au niveau de l'instance 30 environ, où il arrive à son minimum, et reste à varier à proximité de ce taux minimal jusqu'à l'instance 130 environ. Ces seuils sont les mêmes que pour le graphique précédent. Ces résultats ne s'étendent pas au cas **allyesconfig** ; en effet, si la présence d'un plateau s'observe toujours, celui-ci n'oscille

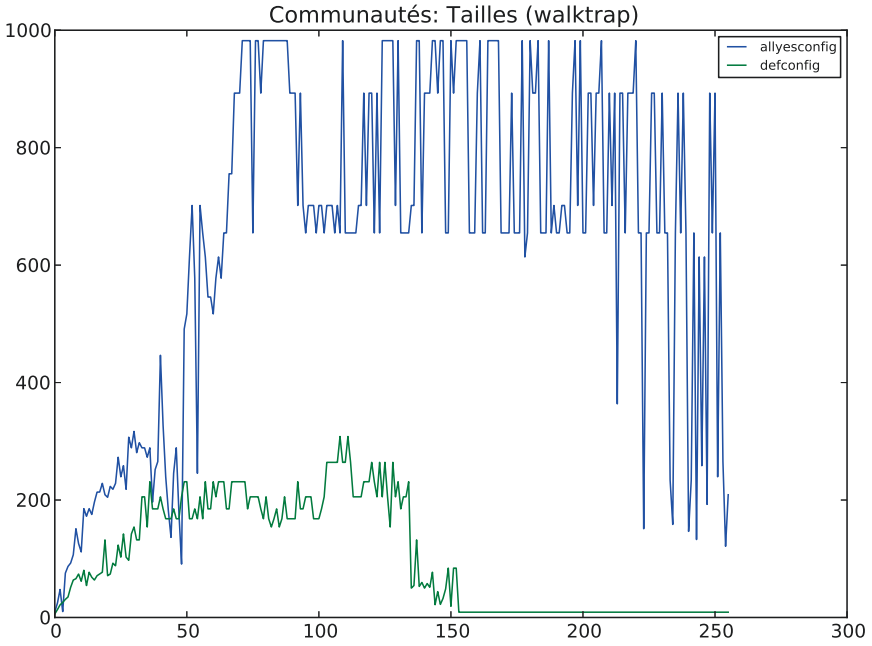
pas autour de la valeur minimale atteinte précédemment. Au lieu d'observer une descente constante et progressive jusqu'au premier pallier situé à environ 60 comme nous aurions pu nous y attendre ; ici, nous constatons une première baisse très rapide, culminant en un minimum situé autour de 10-15 où la valeur atteinte est d'environ 0.2, comme pour **defconfig**. Ensuite, le taux remonte jusqu'à atteindre un plateau oscillant entre 0.4 et 0.5 à partir de l'instance 60 environ.

Nous escomptons que ce taux soit le plus proche de 0, ce qui indiquerait que les communautés détectées soient parfaites et ne contiennent que des fichiers du même sous-répertoire. Contrairement aux données correspondant aux tailles des communautés détectés, il est difficile de trouver ici un intervalle qui satisfasse à un minimum global sur les deux courbes. Si nous reprenons celui proposé lors de l'analyse précédente, à savoir $[60; 80]$, nous englobons les deux plateaux des courbes, mais elle du cas **allyesconfig** n'est pas à son minimum. Nous pouvons, par contre, constater que ce taux est également influencé par la taille des communautés : le minimum est légèrement au-dessus de 0.2, et est atteint avec des communautés d'une taille d'environ 200. La stabilisation du taux entre 0.4 et 0.5 pour **allyesconfig** qui s'opère notamment sur l'intervalle compris entre les instances 60 et 80 peut, dès lors, s'interpréter comme une valeur finalement plutôt intéressante : dans le même temps, les communautés atteignent des tailles importantes, autour de 800. Ces communautés sont moins homogènes.

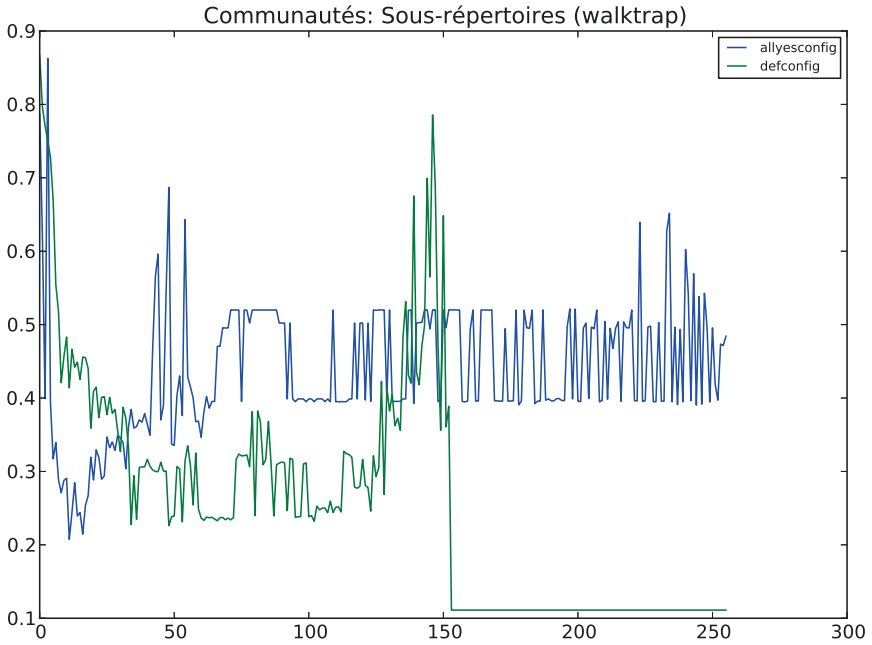
Nous proposons également dans la figure B.14 une visualisation du nombre de communautés détectées dans les cas **defconfig** et **allyesconfig** suivant la valeur du paramètre que nous avons fait varier. La première remarque que nous pouvons faire concerne les différents seuils identifiés précédemment sur les autres graphiques : le comportement est similaire, l'échelle logarithmique choisie permet de masquer une partie des variations fortes que nous observions. L'intervalle estimé entre 60 et 80 correspond également à l'intersection des deux courbes, autour d'un nombre de communautés de l'ordre de la dizaine.

La figure B.15 documente le comportement de la mesure définie dans la section 4.4.1.2, qui rends compte de la forme des connexions entre les communautés. Idéalement, cette mesure doit être minimale : nous pouvons constater que si un comportement similaire à celui observé sur les autres mesures est visible, cependant, l'amplitude est ici bien plus faible. L'exception notable concerne les limites des instances. Si nous nous concentrons sur l'intervalle qui a pour le moment été proposé, c'est-à-dire entre 60 et 80, les valeurs, quelles ques soient les instances et les cas (**defconfig** et **allyesconfig**), varient autour de 0.5.

Le temps de calcul ainsi que la consommation mémoire sont présentés respectivement dans les graphiques B.16a et B.16b. La quantité de mémoire utilisée est uniquement dépendante de la taille de l'instance utilisée, et cela se voit très bien sur le second graphique : les courbes sont plates, quelques variations infimes sont visibles sur le cas **allyesconfig**. Le graphique du temps de calcul montre l'effet de la complexité de l'algorithme, et notamment de la composante quadratique : la taille de l'instance est bien ce qui va influer le plus nettement sur le temps de calcul, la croissance par rapport à la taille du chemin aléatoire étant plutôt linéaire.



(a) Taille des communautés



(b) Taux de concentration

FIGURE B.13 – Influence de la taille du chemin aléatoire dans PONS et LATAPY (2005) sur la détection de communautés

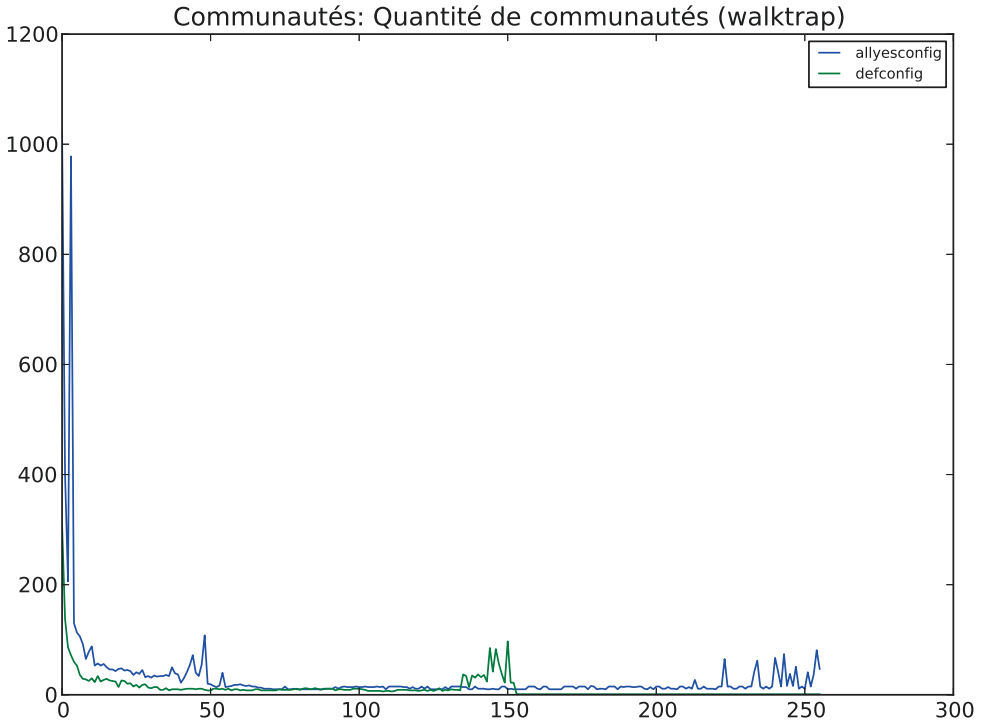


FIGURE B.14 – Quantité de communautés détectées suivant la taille du chemin aléatoire avec PONS et LATAPY (2005)

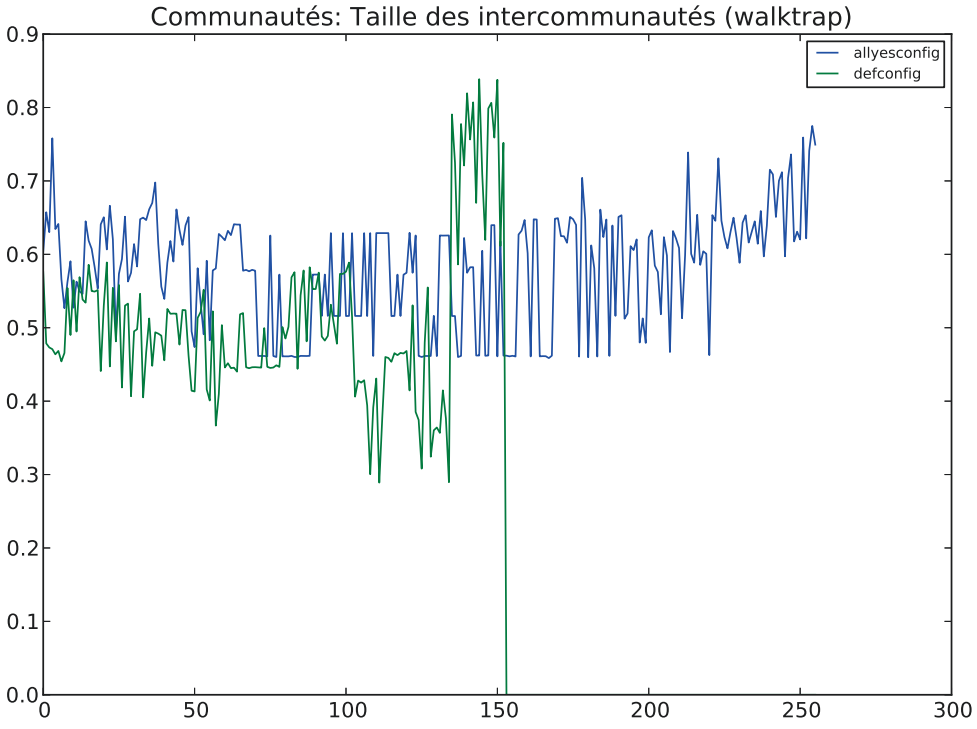
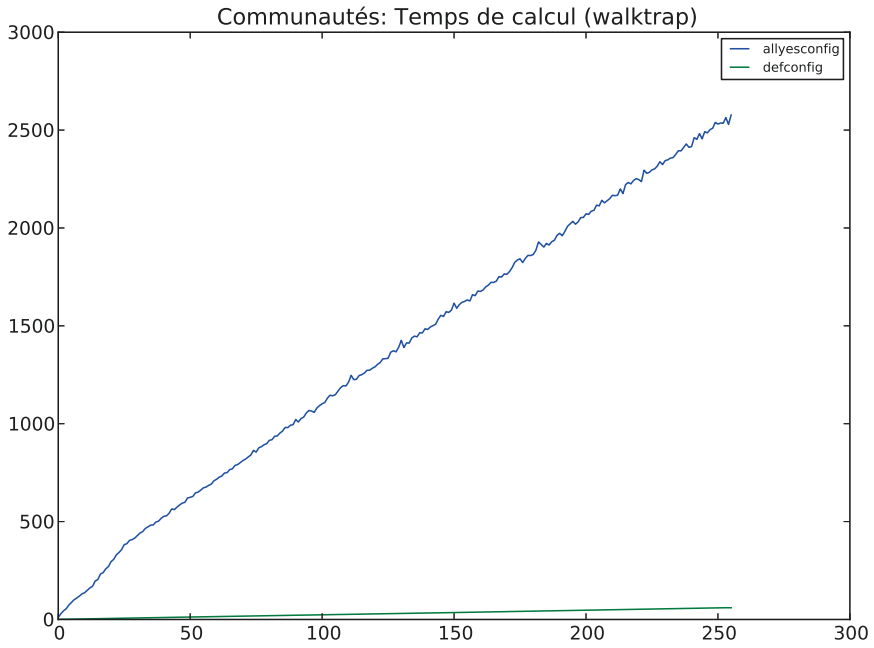
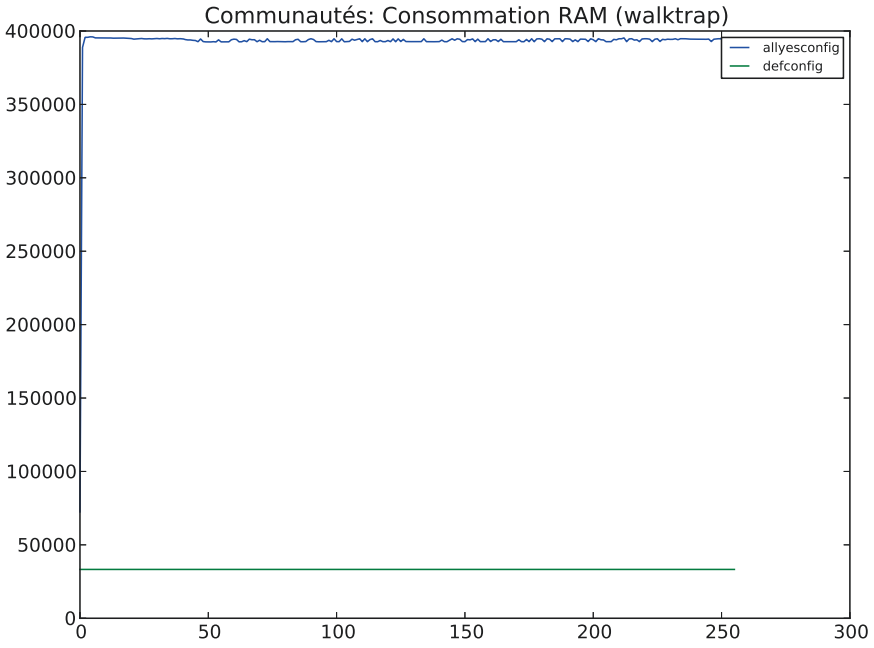


FIGURE B.15 – Mesure de la taille des interconnexions des communautés avec PONS et LATAPY (2005)



(a) Temps de calcul



(b) Consommation mémoire

FIGURE B.16 – Influence de la taille du chemin aléatoire dans PONS et LATAPY (2005) sur l'exécution de l'algorithme

B.5 Résultats pour Radicchi et al. (2004)

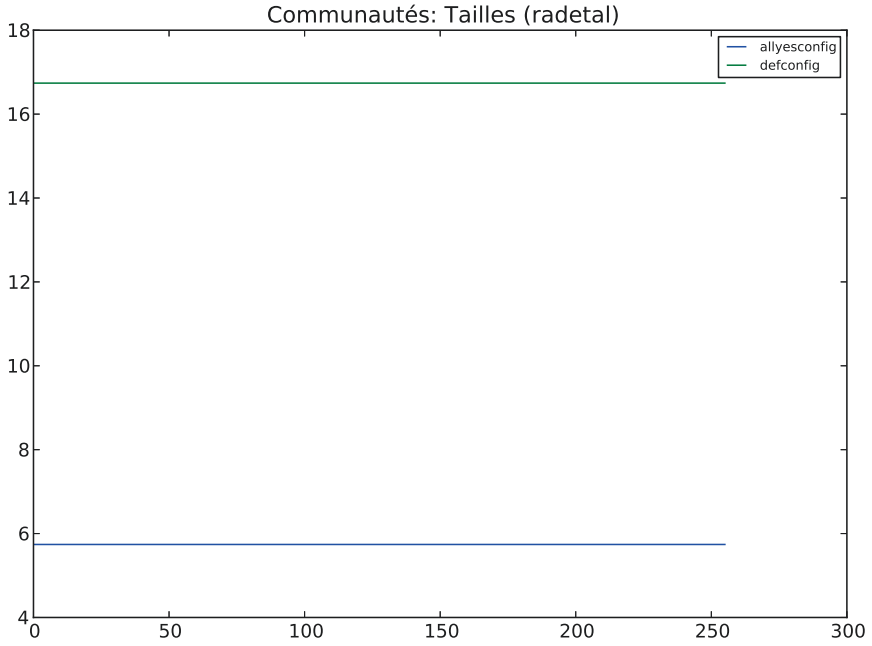
L'implémentation proposée pour cette méthode ne propose qu'un seul paramètre sur lequel nous pouvons agir pour modifier le comportement : détection des communautés avec une méthode du 3^{e} ou du 4^{e} ordre. L'impact quantitatif de cette seconde option n'a pas été mesurée, la complexité devenant beaucoup trop importante ; ainsi, les premiers calculs effectués sur l'instance **defconfig** sont terminés en une cinquantaine de secondes avec une détection de communauté du troisième ordre, lorsqu'il faut de l'ordre de 35 000 secondes dans l'autre cas. Les résultats que nous proposons sont donc ceux obtenus pour les analyses des graphes **defconfig** et **allyesconfig** avec une analyse du troisième ordre. Comme pour les cas précédents, nous avons effectué un total de 256 itérations.

Nous pouvons d'abord voir, dans les graphiques B.17a et B.17b, la taille des communautés qui sont détectées ainsi que leur concentration. Quelle que soit l'instance étudiée, nous constatons dans les deux cas que le nombre ne varie pas : avec **defconfig** on trouve en moyenne un peu moins de six nœuds dans chaque communauté ; avec **allyesconfig**, cette moyenne augmente et passe à environ 17. Le taux de concentration au sein de ces communautés est documenté après, et est très proche dans les deux instances, la différence étant de l'ordre de 0.005 ; la première correspond au cas du noyau compilé en **defconfig**, et est à environ 0.803 alors que la seconde est à un peu moins de 0.809. Ces valeurs restent élevées, même si elles sont plus faibles que celles présentées dans le cas [77] présenté en section B.3 : la comparaison fait d'autant plus sens que les ordres de grandeur des communautés sont assez proches. Si nous nous limitions à ce critère, la méthode présentée serait donc plus pertinente que [77]. L'ordre de grandeur de la concentration indique, cependant, que les communautés qui sont détectées ne sont majoritairement pas constituées de fichiers objets du même sous-répertoire.

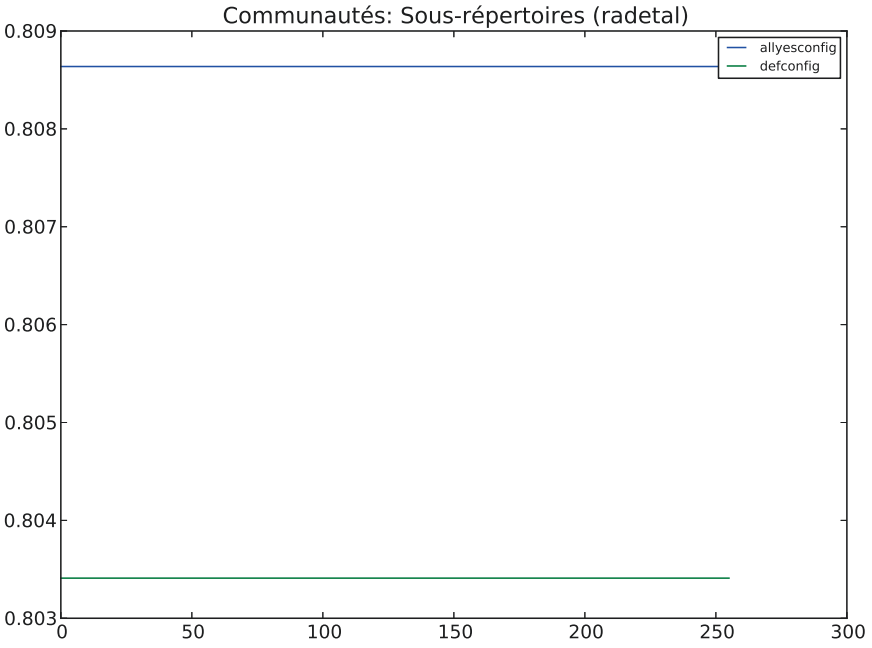
Incidemment, le nombre de communautés qui sont détectées avec cette méthode est entièrement stable. Le graphique qui le documente est proposé en figure B.18 et nous permet, d'abord, de constater que sur l'instance **defconfig**, ce nombre est de l'ordre de la centaine ; alors qu'avec l'instance **allyesconfig**, nous passons la barre du millier.

La taille des intercommunautés qui est proposée dans le graphique B.19 présente, par contre, des propriétés qui commencent à être plus intéressantes. Dans le cas de l'instance **defconfig** nous pouvons voir que la mesure est évaluée en dessous de 0.2, ce qui indique que nous avons assez peu d'arcs intercommunautés, particulièrement comparé à d'autres méthodes. Sur l'instance **allyesconfig**, nous pouvons de plus voir que ce taux augmente de manière non négligeable mais atteint pour valeur 0.6 : cela traduit que dans ce second cas, nous avons toujours un nombre d'arcs entre les communautés qui reste restreint. Cette méthode propose donc des communautés de taille réduite, et un nombre d'arcs intercommunautaires assez réduit, elle semble donc plutôt intéressante dans le cadre d'une analyse contrainte comme nous l'avons décrite précédemment.

Les graphiques B.20a et B.20b documentent respectivement le temps de calcul et la consommation mémoire de l'exécution de cet algorithme. Le temps d'exécution pour l'analyse d'un graphe correspondant à l'instance **defconfig** est très faible, de l'ordre de quelques dizaines de secondes ; ce temps explose et varie autour de 14 000 secondes (un peu moins de 4 h) lorsque nous analysons le graphe **allyesconfig**. De manière similaire, la consommation



(a) Taille des communautés



(b) Taux de concentration

FIGURE B.17 – Variations des différentes instances de RADICCHI et al. (2004) sur la taille et la concentration des communautés

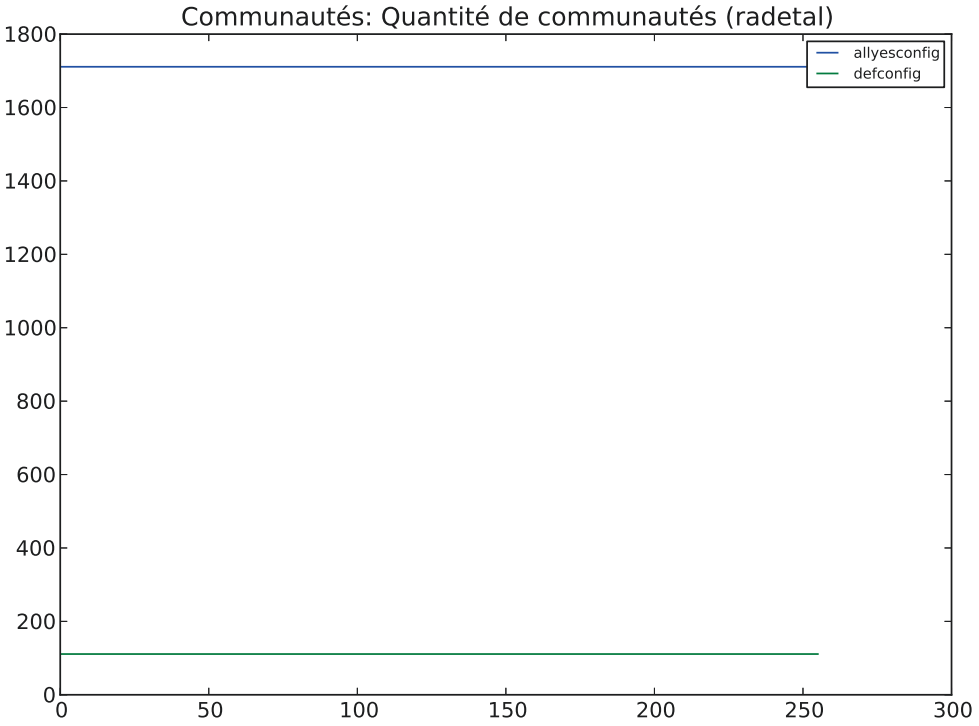


FIGURE B.18 – Variation des différentes instances de RADICCHI et al. (2004) sur le nombre de communautés

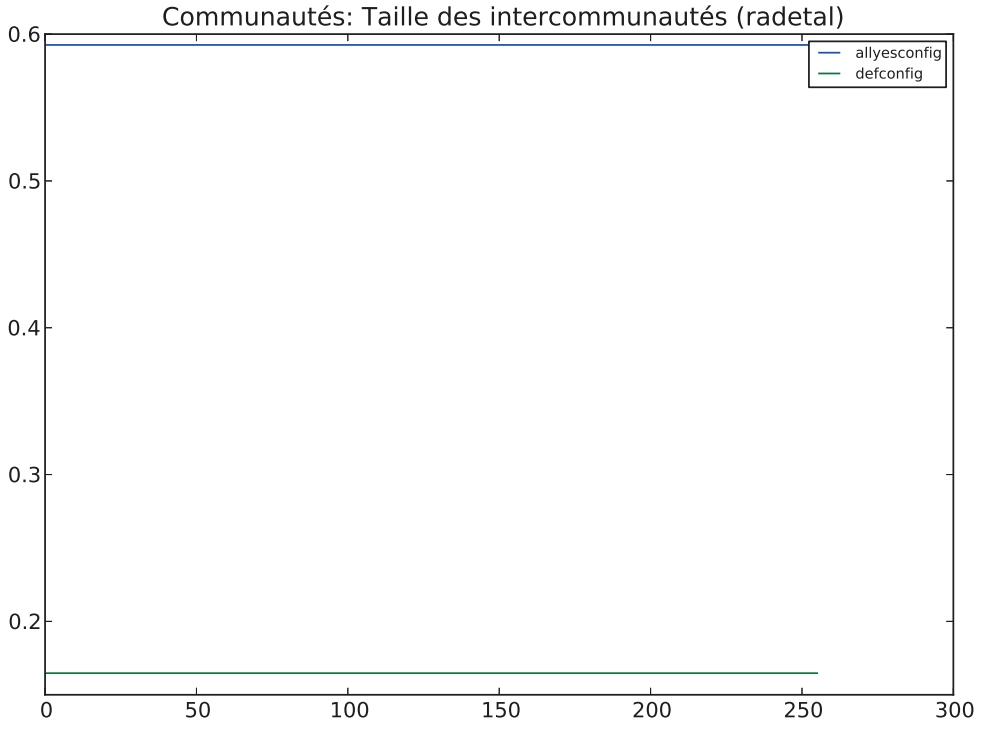


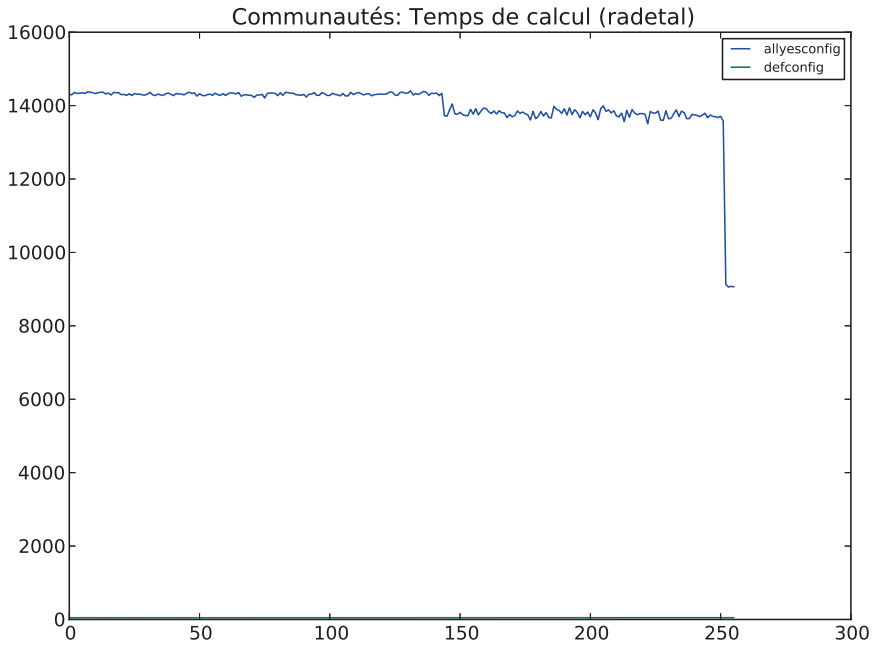
FIGURE B.19 – Mesure de la taille des interconnexions des communautés avec RADICCHI et al. (2004)

mémoire est assez limitée dans le premier cas, de l'ordre d'une centaine de méga-octets ; elle évolue et atteint 4.2GiO avec le passage à la seconde instance. La complexité de cette méthode est donc particulièrement visible avec ces chiffres.

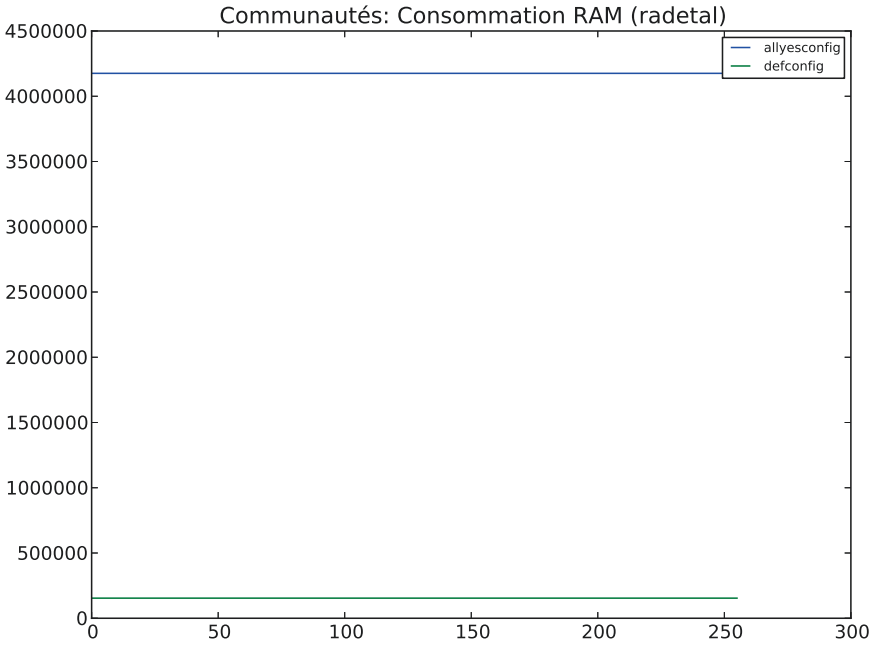
B.6 Résultats pour Zhang, Wang et Zhang (2007)

L'implémentation proposée pour cette méthode accepte plusieurs paramètres à l'exécution. Dans l'optique de construire des communautés d'une taille contrôlée pour aider à l'analyse de code, la possibilité de définir le nombre de communautés nous permet de remplir cet objectif. Un second paramètre contrôle la fréquence d'élagage des communautés, c'est-à-dire la fréquence à laquelle l'algorithme supprimera l'appartenance d'un nœud à une communauté. Cet élagage implique la définition d'une valeur de seuil, le seuil d'élagage, que nous pouvons également contrôler à l'exécution. Par défaut, le comportement est d'avoir une fréquence nulle, ce qui implique de ne jamais faire d'élagage. Un autre paramètre permet de définir le seuil d'appartenance à une communauté, appelé seuil de dominance. Cette implémentation est aussi capable de lire comme entrée un graphe pondéré, ce que nous avons exploité. Des essais préliminaires sur l'impact de certains des paramètres ont été menés afin de déterminer des valeurs pertinentes : le seuil de dominance n'a pas été évalué, puisque nous ne retenons que la valeur maximale pour définir l'appartenance ; l'utilisation d'un seuil d'élagage et d'une fréquence associée n'ont pas amélioré nos résultats. La documentation de l'implémentation indique que l'impact à attendre est une amélioration de la convergence. Cette documentation stipule aussi que l'algorithme peut rester coincé dans des minimums locaux, et suggère donc d'exécuter plusieurs fois la méthode pour éviter ce cas. Cette limitation ne devrait pas nous impacter, étant donné que nous avons effectué pour chaque instance un ensemble de 256 itérations comme pour les méthodes précédentes.

La première série de graphique que nous pouvons lire pour cette méthode est proposée respectivement dans les figures B.21a et B.21b. La taille des communautés qui sont détectées présente un comportement similaire entre les deux cas **defconfig** et **allyesconfig** : une alternance entre environ trois valeurs, et correspondent à des moyennes. Pour **defconfig** les possibilités sont : une communauté avec aucun nœud, une communauté avec quelques (moins d'une dizaine, en général) nœuds et une communauté qui regroupe tous les autres, soit plus de 1800 ; une autre possibilité consiste à avoir seulement deux communautés qui se partagent tous les nœuds. Dans le cas de **allyesconfig**, les possibilités sont les mêmes, la seule différence résidant surtout dans les valeurs : le partage en deux communautés se fait toujours sur la moitié des nœuds, environ, alors que la répartition entre une communauté très peu peuplée, un non peuplée et une très peuplée n'évolue que sur la communauté très peuplée. Nous construisons ainsi des ensembles totalement déséquilibrés. Le second graphique présente le taux de concentration dans les deux cas. Comme dans d'autres méthodes, ce taux est influencé par la quantité de nœuds présents dans une communauté. Pour l'instance **defconfig** comme pour **allyesconfig**, il évolue principalement autour de 0.2 ; cette valeur aurait tendance à indiquer un bon regroupement, avec beaucoup de nœuds du même répertoire. L'analyse des données détaillées montre que ce phénomène s'explique bien par la variabilité préalablement observée : mécaniquement, le



(a) Temps de calcul



(b) Consommation mémoire

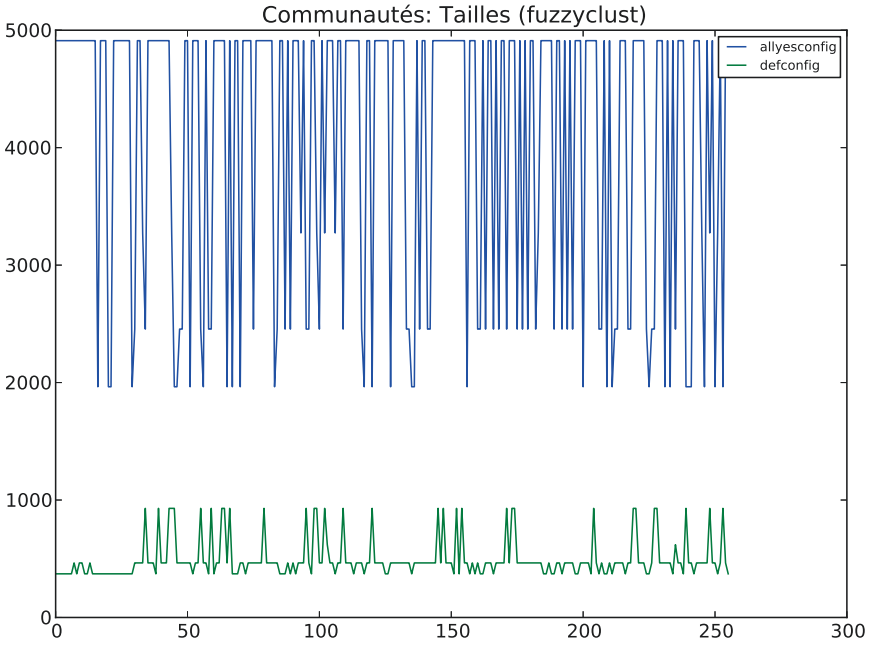
FIGURE B.20 – Consommation mémoire et processeur pour l'exécution de RADICCHI et al. (2004)

taux sera plutôt faible pour les grosses communautés. Sur les petites, la valeur obtenue est plus représentative. Par exemple, sur une instance nous obtenons une communauté constituée de 47 nœuds, et avec un taux de l'ordre de 0.68. Cette valeur indique une assez mauvaise corrélation des différents composants.

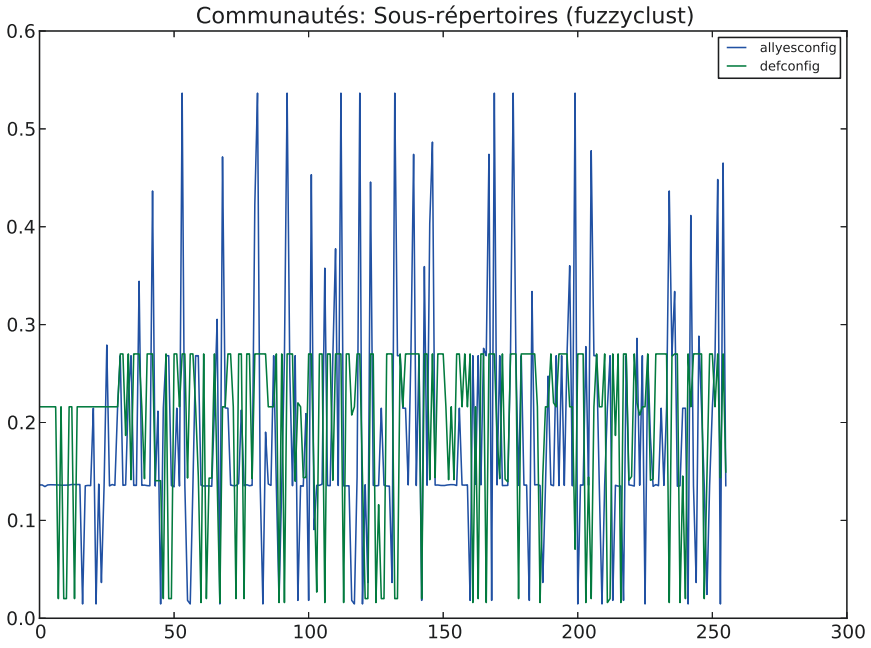
Le nombre de communautés qui ont été détectées est documenté dans le graphique visible en figure B.22. Contrairement aux mesures précédentes, ici, les variations sont faibles et varient entre 2.0 et 5.0. La lecture des données détaillées apporte confirmation que la répartition correspond. De plus, dans ces résultats, nous pouvons voir que les deux instances **defconfig** et **allyesconfig** sont concernées. Ce comportement d'insensibilité à la taille du graphe, en matière de nombre de communautés détectées, aurait tendance à indiquer qu'il s'agit là du résultat que produit cette méthode de détection, et pas d'un artéfact. Nous avons cherché, également, pendant nos essais et après ces résultats, à contraindre le nombre de communautés que l'algorithme devait trouver. Dans tous les cas, finalement, nous n'obtenions que des communautés supplémentaires et vides.

La taille des intercommunautés est documentée dans le graphique B.23. Nous pouvons y lire que dans la très grande majorité des cas sur les instances que nous avons analysées, cette taille est soit nulle ou très proche, soit varie très proche de 0.50. Une valeur nulle indique qu'il n'existe aucun arc entre les communautés, ce qui semble plutôt intriguant. Une plongée dans les données détaillées et les résultats non traités montre qu'une partie non négligeable des sommets, dans ces instances, est classifiée comme n'appartenant à aucune communauté. Cela peut expliquer la présence importante d'instances où il n'y a pas d'arc faisant le lien entre les différentes communautés. De manière similaire, et en nous référant aux résultats précédents, nous avons déjà vu que dans un nombre d'instances important, il existe deux communautés principales qui se partagent la moitié des arcs. Ce comportement explicite le nombre d'instances pour lesquelles la taille des intercommunautés mesurées est proche de 0.50. Ces constats sont valables à la fois pour le cas **defconfig** et **allyesconfig**.

Nous proposons dans les graphiques visibles respectivement en figure B.24a et B.24b. Dans le tableau 4.3 nous avions documenté la complexité des différentes méthodes proposées, cette méthode Zhang, Wang et Zhang (2007) était classée comme l'une des plus complexes à notre disposition. Ces graphiques confirment et éclairent l'impact de cette complexité sur des instances réelles. Commençons par la complexité temporelle qui est illustrée dans le premier graphique donnant le temps de calcul pour les instances. Le graphique montre des variations très fortes dans les temps de calcul, se situant encore une fois sur deux extrèmes : soit assez rapide, soit très lent ; cela se voit très bien sur le cas **allyesconfig**, mais **defconfig** est aussi concerné. Si nous examinons les résultats détaillés, pour ce dernier, nous avons un premier paquet composé d'environ 30 instances dont le temps d'exécution se compte entre 10 et 15 secondes ; puis, quasi sans intermédiaire, celui-ci croît pour atteindre un premier palier d'environ 100 secondes ; enfin, une répartition croissante et constante se réalise sur le reste des instances et le temps d'exécution augmente jusqu'à environ 200 secondes. Les résultats détaillés de **allyesconfig** montre un comportement proche : un premier ensemble d'instances dont le temps d'exécution croît lentement, partant de l'ordre de 500 secondes à environ 2000 secondes, et composé d'environ 125 instances ; puis un phénomène de croissance importante et continue qui atteint des temps de calcul très importants, plus d'une dizaine d'instances dépassant les 30 000 secondes, et plus de la moitié de ce second ensemble dépassant les 15 000 secondes. Nous



(a) Taille des communautés



(b) Taux de concentration

FIGURE B.21 – Variations des différentes instances de ZHANG, WANG et ZHANG (2007) sur la taille et la concentration des communautés

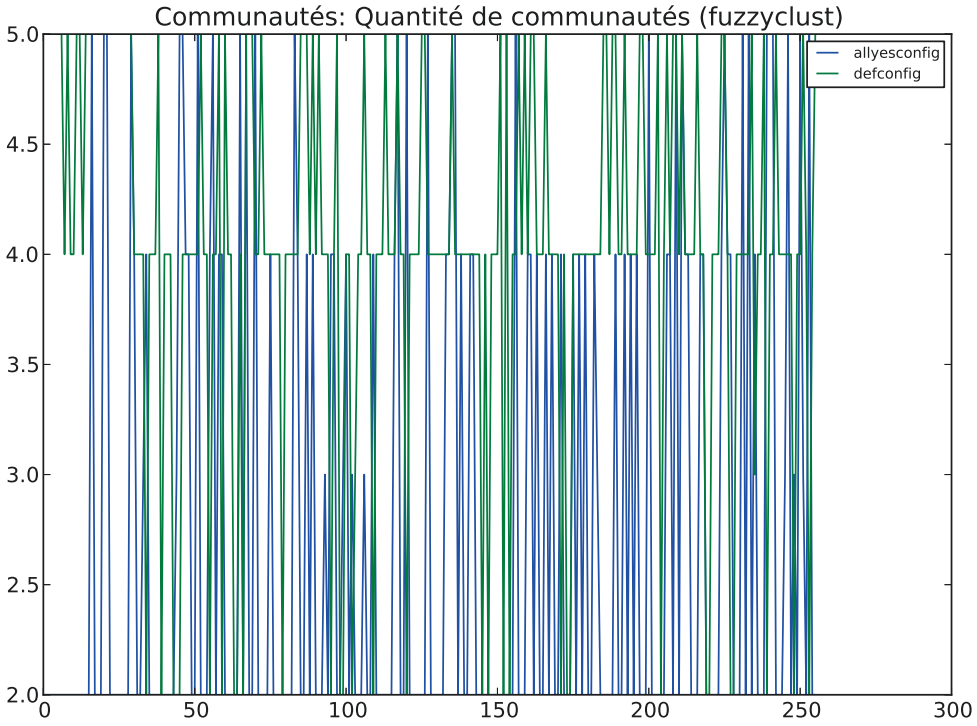


FIGURE B.22 – Variation des différentes instances de ZHANG, WANG et ZHANG (2007) sur le nombre de communautés

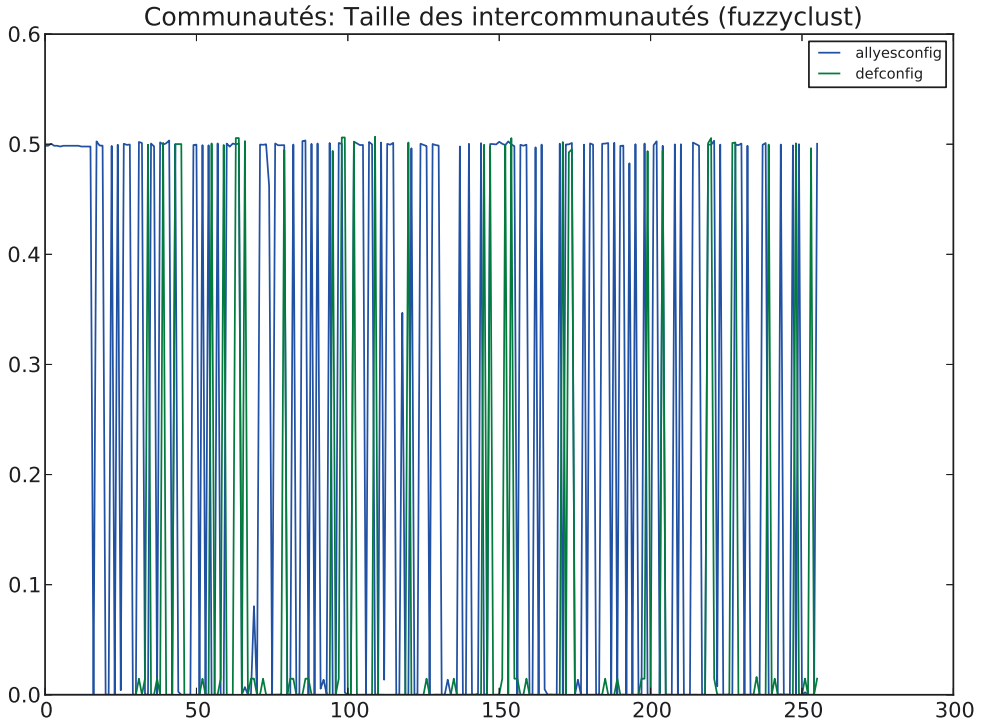
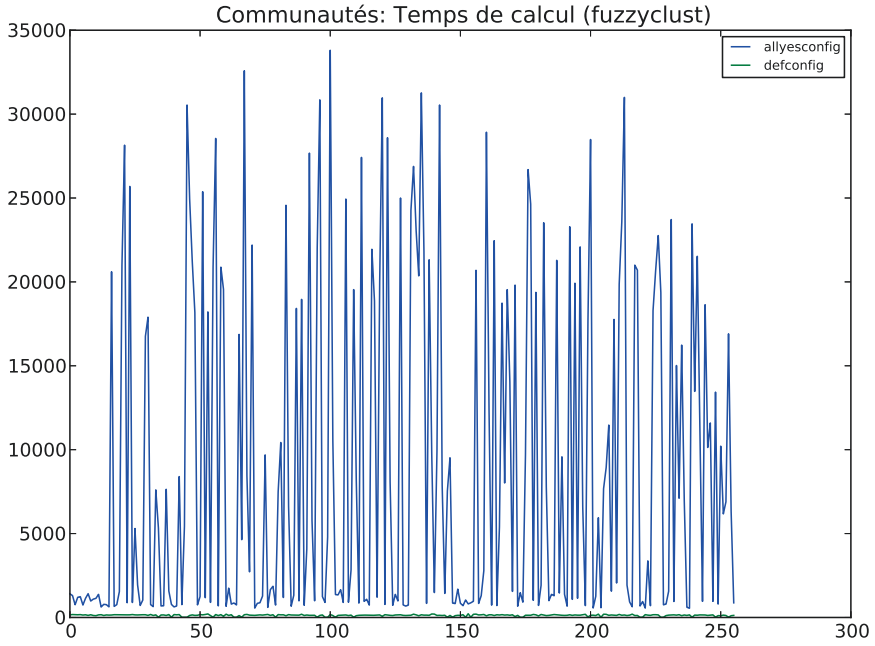
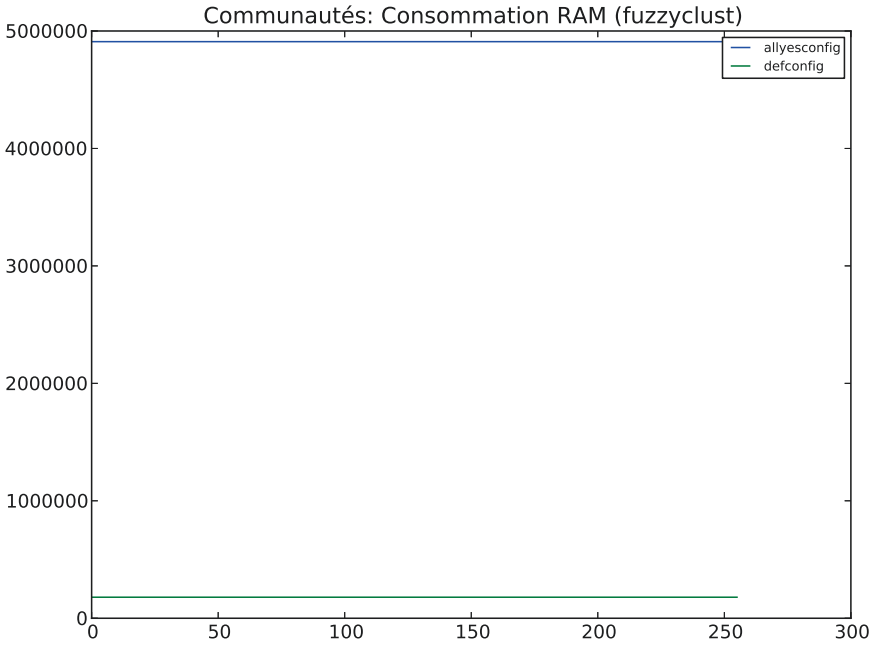


FIGURE B.23 – Mesure de la taille des interconnexions des communautés avec ZHANG, WANG et ZHANG (2007)

n'observons pas, par contre, de rupture brutale entre les deux périodes de croissance dans ce cas. Nous retrouvons là une manifestation de l'avertissement des auteurs de la méthode, qui dans leur implémentation conseillent d'exécuter plusieurs itérations pour s'assurer que la convergence est correcte : nous voyons ici des instances qui convergent très rapidement, alors que d'autres sont beaucoup plus lentes. Le second graphique, proposé en figure B.24b, documente la consommation mémoire des instances. À la différence du temps de calcul, il n'y a ici strictement aucune variation. Nous pouvons par contre constater que la méthode est plutôt gourmande en mémoire, surtout dans le cas **allyesconfig** qui consomme 4.9GiO pour chaque instance, là où celles de **defconfig** ne consomment qu'environ 180MiO.



(a) Temps de calcul



(b) Consommation mémoire

FIGURE B.24 – Consommation mémoire et processeur pour l'exécution de ZHANG, WANG et ZHANG (2007)

Bibliographie

[1] G. Agarwal et D. Kempe. « Modularity-maximizing graph communities via mathematical programming ». English. Dans : *The European Physical Journal B* 66.3 (2008), p. 409–418. issn : 1434-6028. doi : 10.1140/epjb/e2008-00425-1. url : http://dx.doi.org/10.1140/epjb/e2008-00425-1.

[2] Nelson A. Alves. « Unveiling community structures in weighted networks ». Dans : *Phys. Rev. E* 76 (3 sept. 2007), p. 036101. doi : 10.1103/PhysRevE.76.036101. url : http://link.aps.org/doi/10.1103/PhysRevE.76.036101.

[3] Narasimhamurthy Anand et al. *Community Finding in Large Social Networks Through Problem Decomposition*. Rap. tech. 2008.

[4] Jesper Andersen et Julia L. Lawall. « Generic Patch Inference ». Dans : *23rd IEEE/ACM International Conference on Automated Software Engineering*. L'Aquila, Italy, 29 oct. 2008, p. 337–346.

[5] Scott White And. *A Spectral Clustering Approach To Finding Communities in Graphs*. 2005.

[6] A Arenas, A Fernández et S Gómez. « Analysis of the structure of complex networks at different resolution levels ». Dans : *New Journal of Physics* 10.5 (2008), p. 053039. url : http://stacks.iop.org/1367-2630/10/i=5/a=053039.

[7] Alex Arenas, Albert Díaz-Guilera et Conrad J. Pérez-Vicente. « Synchronization Reveals Topological Scales in Complex Networks ». Dans : *Phys. Rev. Lett.* 96 (11 mar. 2006), p. 114102. doi : 10.1103/PhysRevLett.96.114102. url : http://link.aps.org/doi/10.1103/PhysRevLett.96.114102.

[8] A. Arenas et al. « Motif-based communities in complex networks ». Dans : *Journal of Physics A Mathematical General* 41.22, 224001 (juin 2008), p. 224001. doi : 10.1088/1751-8113/41/22/224001. arXiv :0710.0059 [physics.comp-ph].

[9] A Arenas et al. « Size reduction of complex networks preserving modularity ». Dans : *New Journal of Physics* 9.6 (2007), p. 176. url : http://stacks.iop.org/1367-2630/9/i=6/a=176.

[10] J. P. Bagrow et E. M. Bollt. « Local method for detecting communities ». Dans : 72.4, 046108 (oct. 2005), p. 046108. doi : 10.1103/PhysRevE.72.046108. eprint : arXiv:cond-mat/0412482.

[11] Thomas Ball et Sriram K. Rajamani. « Automatically validating temporal safety properties of interfaces ». Dans : *SPIN '01 : Proceedings of the 8th international SPIN workshop on Model checking of software*. Toronto, Ontario, Canada : Springer-Verlag New York, Inc., 19 mai 2001, p. 103–122. ISBN : 3-540-42124-6.

[12] Thomas Ball et Sriram K. Rajamani. *Boolean Programs : A Model and Process for Software Analysis*. Rap. tech. Microsoft Research, 15 fév. 2000.

[13] Thomas Ball et Sriram K. Rajamani. « Checking Temporal Properties of Software with Boolean Programs ». Dans : *In Proceedings of the Workshop on Advances in Verification*. 15 juil. 2000.

[14] Thomas Ball et Sriram K. Rajamani. « The SLAM project : debugging system software via static analysis ». Dans : *POPL '02 : Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Portland, Oregon : ACM, 5 déc. 2002, p. 1–3. ISBN : 1-58113-450-9. DOI : http://doi.acm.org/10.1145/503272.503274.

[15] Thomas Ball et al. « Thorough static analysis of device drivers ». Dans : *EuroSys '06 : Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. Leuven, Belgium : ACM, 16 mai 2007, p. 73–85. ISBN : 1-59593-322-0. DOI : http://doi.acm.org/10.1145/1217935.1217943.

[16] T. Ball, V. Levin et S.K. Rajamani. « A Decade of Software Model Checking with SLAM ». 6 juil. 2011.

[17] T. Ball et al. « SLAM and Static Driver Verifier : Technology transfer of formal methods inside Microsoft ». Dans : *Integrated Formal Methods*. Springer. 15 août 2004, p. 1–20.

[18] T. Ball et al. « The Static Driver Verifier Research Platform ». Dans : *International Conference on Computer Aided Verification*. 12 juil. 2010.

[19] Nikhil Bansal, Avrim Blum et Shuchi Chawla. « Correlation Clustering ». English. Dans : *Machine Learning* 56.1-3 (2004), p. 89–113. ISSN : 0885-6125. DOI : 10.1023/B:MACH.0000033116.57574.95. URL : http://dx.doi.org/10.1023/B:MACH.0000033116.57574.95.

[20] Michael J. Barber. « Modularity and community detection in bipartite networks ». Dans : *Phys. Rev. E* 76 (6 déc. 2007), p. 066102. DOI : 10.1103/PhysRevE.76.066102. URL : http://link.aps.org/doi/10.1103/PhysRevE.76.066102.

[21] Michael J Barber et al. *Searching for Communities in Bipartite Networks*. Rap. tech. arXiv :0803.2854. Comments : 12 pages, 4 figures, to appear in "Proceedings of the 5th Jagna International Workshop : Stochastic and Quantum Dynamics of Biomolecular Systems," C. C. Bernido and M. V. Carpio-Bernido, editors. A version with full-quality figures and larger file size is available at http ://ccm.uma.pt/publications/Barber-Faria-Streit-Strogan-2008.pdf. Mar. 2008.

[22] R. Barnes. *An Algorithm for Partitioning the Nodes of a Graph*. Research Report. IBM Thomas J. Watson Research Center. Center, 1981. URL : http://books.google.fr/books?id=K36VHAAACAAJ.

[23] Jeffrey Baumes et al. « Finding communities by clustering a graph into overlapping subgraphs. » Dans : *IADIS AC* 5 (2005), p. 97–104.

[24] Patrick Bellot, Jean-Philippe Cottin et Jean-François Monin. « Développement et validation de logiciels. Méthodes formelles ». Dans : Techniques de l'Ingénieur, 10 déc. 1995, p. 1–11.

[25] Béatrice Bérard et al. *Vérification de logiciels*. Vuibert Informatique, 15 mai 1999. isbn : 2711786463.

[26] Al Bessey et al. « A few billion lines of code later : using static analysis to find bugs in the real world ». Dans : *Commun. ACM* 53 (2 26 avr. 2010), p. 66–75. issn : 0001-0782. doi : http://doi.acm.org/10.1145/1646353.1646374. url : http://doi.acm.org/10.1145/1646353.1646374.

[27] V. D. Blondel et al. « Fast unfolding of communities in large networks ». Dans : *Journal of Statistical Mechanics : Theory and Experiment* 10 (oct. 2008), p. 8. doi : 10.1088/1742-5468/2008/10/P10008. arXiv :0803.0476.

[28] S. Boccaletti et al. « Detecting complex network modularity by dynamical clustering ». Dans : *Phys. Rev. E* 75 (4 avr. 2007), p. 045102. doi : 10.1103/PhysRevE.75.045102. url : http://link.aps.org/doi/10.1103/PhysRevE.75.045102.

[29] Long Bo et al. « Community Learning by Graph Approximation ». Dans : *ICDM*. 2007, p. 232–241.

[30] Ahmed Bouajjani et al. « Abstract regular (tree) model checking ». Dans : *STTT* 14.2 (2012), p. 167–191.

[31] Ulrik Brandes et al. *On Modularity – NP-Completeness and Beyond*. 2006.

[32] P. Breuer et S. Pickin. « One million (LOC) and counting : Static analysis for errors and vulnerabilities in the Linux kernel source code ». Dans : *Reliable Software Technologies–Ada-Europe 2006* (13 juin 2006), p. 56–70.

[33] P.T. Breuer et S. Pickin. « Approximate verification in an open source world ». Dans : *Innovations in Systems and Software Engineering* 4.1 (27 juil. 2009), p. 87–105.

[34] P.T. Breuer et S. Pickin. « Symbolic approximation : an approach to verification in the large ». Dans : *Innovations in Systems and Software Engineering* 2.3 (8 déc. 2007), p. 147–163.

[35] P.T. Breuer et S. Pickin. « Verification in the large via symbolic approximation ». Dans : *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. 16 nov. 2006, p. 408–415.

[36] P.T. Breuer, S. Pickin et M.L. Petrie. « Detecting deadlock, double-free and other abuses in a million lines of linux kernel source ». Dans : *Software Engineering Workshop, 2006. SEW'06. 30th Annual IEEE/NASA*. 5 fév. 2007, p. 223–233.

[37] P.T. Breuer et M.G. Valls. « Static deadlock detection in the Linux kernel ». Dans : *Reliable Software Technologies-Ada-Europe 2004* (16 juin 2004), p. 52–64.

[38] Julien Brunel et al. « A Foundation for Flow-Based Program Matching Using Temporal Logic and Model Checking ». Dans : *The 36th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*. Savannah, GA, USA, 24 jan. 2009, p. 114–126.

[39] Matthieu Caneill. *Outils de visualisation de résultats d'analyse statique de code des paquets Debian*. Rap. tech. INRIA IRILL ; Polytech Grenoble, 17 août 2013.

[40] A. Capocci et al. « Detecting communities in large networks ». Dans : *Physica A : Statistical Mechanics and its Applications* 352.2–4 (2005), p. 669 –676. ISSN : 0378-4371. DOI : http://dx.doi.org/10.1016/j.physa.2004.12.050. URL : http://www.sciencedirect.com/science/article/pii/S0378437104015729.

[41] Deepayan Chakrabarti. « AutoPart : Parameter-Free Graph Partitioning and Outlier Detection ». Dans : *PKDD*. 2004, p. 112–124.

[42] Zbigniew Chamski et Basile Sarynkévitch. « Les greffons du compilateur GCC : Votre compilateur GCC sur mesure ! » Dans : *Solutions Linux 2010*. 17 mar. 2010.

[43] Hao Chen et David A. Wagner. *MOPS : an Infrastructure for Examining Security Properties of Software*. Rap. tech. UCB/CSD-02-1197. EECS Department, University of California, Berkeley, 28 juil. 2003. URL : http://www.eecs.berkeley.edu/Pubs/TechRpts/2002/5824.html.

[44] H. Chen et D. Wagner. « MOPS : an infrastructure for examining security properties of software ». Dans : *Proceedings of the 9th ACM Conference on Computer and Communications Security*. ACM. 28 juil. 2003, p. 235–244.

[45] Jingchun Chen et Bo Yuan. « Detecting functional modules in the yeast protein–protein interaction network ». Dans : *Bioinformatics* 22.18 (2006), p. 2283–2290.

[46] Andy Chou et al. « An Empirical Study of Operating System Errors ». Dans : 24 oct. 2001, p. 73–88.

[47] Andy Chou et al. *Weird things that surprise academics tryind to commercialize a static checking tool*. 24 août 2005. URL : http://www.stanford.edu/~engler/spin05-coverity.pdf.

[48] E. M. Clarke, E. A. Emerson et A. P. Sistla. « Automatic verification of finite-state concurrent systems using temporal logic specifications ». Dans : *ACM Trans. Program. Lang. Syst.* 8.2 (15 avr. 1986), p. 244–263. ISSN : 0164-0925. DOI : http://doi.acm.org/10.1145/5397.5399.

[49] Aaron Clauset. « Finding local community structure in networks ». Dans : *Phys. Rev. E* 72 (2 août 2005), p. 026132. DOI : 10.1103/PhysRevE.72.026132. URL : http://link.aps.org/doi/10.1103/PhysRevE.72.026132.

[50] Aaron Clauset, Cristopher Moore et Mark EJ Newman. « Hierarchical structure and the prediction of missing links in networks ». Dans : *Nature* 453.7191 (2008), p. 98–101.

[51] Aaron Clauset, Cristopher Moore et MarkE.J. Newman. « Structural Inference of Hierarchies in Networks ». Dans : *Statistical Network Analysis : Models, Issues, and New Directions*. Sous la dir. d'Edoardo Airoldi et al. T. 4503. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, p. 1–13. isbn : 978-3-540-73132-0. doi : 10.1007/978-3-540-73133-7_1. url : http://dx.doi.org/10.1007/978-3-540-73133-7_1.

[52] Aaron Clauset, M. E. J. Newman et Cristopher Moore. « Finding community structure in very large networks ». Dans : *Phys. Rev. E* 70 (6 déc. 2004), p. 066111. doi : 10.1103/PhysRevE.70.066111. url : http://link.aps.org/doi/10.1103/PhysRevE.70.066111.

[53] Thomas Reps Computer et Thomas Reps. « Program Analysis via Graph Reachability ». Dans : *Information and Software Technology* 40 (30 nov. 1998), p. 5–19.

[54] Jernej Copic, Matthew O. Jackson et Alan Kirman. « Identifying Community Structures from Network Data via Maximum Likelihood Methods ». Dans : *The B.E. Journal of Theoretical Economics* 9.1 (sept. 2009), p. 1–40. url : http://ideas.repec.org/a/bpj/bejtec/v9y2009i1n30.html.

[55] Robert DeLine et Manuel Fähndrich. *The Fugue protocol checker : Is your software Baroque ?* Rap. tech. 15 juil. 2004.

[56] J.-C. Delvenne, S. N. Yaliraki et M. Barahona. « Stability of graph communities across time scales ». Dans : *ArXiv e-prints* (déc. 2008). arXiv :0812.1811 [physics.soc-ph].

[57] L. Donetti et M. A. Muñoz. « Detecting network communities : a new systematic and efficient algorithm ». Dans : *Journal of Statistical Mechanics : Theory and Experiment* 10 (oct. 2004), p. 12. doi : 10.1088/1742-5468/2004/10/P10012. eprint : arXiv:cond-mat/0404652.

[58] Stijn M. v. van Dongen. « Graph Clustering by Flow Simulation ». Thèse de doct. University of Utrecht, 29 mai 2000. url : http://www.library.uu.nl/digiarchief/dip/diss/1895620/full.pdf.

[59] Jean-Pierre Eckmann et Elisha Moses. « Curvature of co-links uncovers hidden thematic layers in the World Wide Web ». Dans : *Proceedings of the National Academy of Sciences* 99.9 (2002), p. 5825–5829. doi : 10.1073/pnas.032093399. eprint : http://www.pnas.org/content/99/9/5825.full.pdf+html. url : http://www.pnas.org/content/99/9/5825.abstract.

[60] Dawson Engler et Madanlal Musuvathi. « Static Analysis versus Software Model Checking for Bug Finding ». Dans : *Verification, Model Checking, and Abstract Interpretation*. Sous la dir. de Bernhard Steffen et Giorgio Levi. T. 2937. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 12 jan. 2004, p. 405–427. url : http://dx.doi.org/10.1007/978-3-540-24622-0_17.

[61] Dawson Engler et al. « Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions ». Dans : 16 déc. 2002, p. 1–16.

[62] Weinan E, Tiejun Li et Eric Vanden-Eijnden. « Optimal partition and effective dynamics of complex networks ». Dans : *Proceedings of the National Academy of Sciences* 105.23 (2008), p. 7907–7912. DOI : 10.1073/pnas.0707563105. eprint : http://www.pnas.org/content/105/23/7907.full.pdf+html. URL : http://www.pnas.org/content/105/23/7907.abstract.

[63] I. Farkas et al. « Weighted network modules ». Dans : *New Journal of Physics* 9 (juin 2007), p. 180. DOI : 10.1088/1367-2630/9/6/180. eprint : arXiv:cond-mat/0703706.

[64] Gary William Flake, Steve Lawrence et C. Lee Giles. « Efficient Identification of Web Communities ». Dans : *In Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM Press, 2000, p. 150–160.

[65] Gary William Flake et al. « Self-Organization and Identification of Web Communities ». Dans : *IEEE Computer* 35 (2002), p. 66–71.

[66] Santo Fortunato. « Community detection in graphs ». Dans : *Physics Reports* 486.3-5 (25 jan. 2010), p. 75–174. ISSN : 0370-1573. DOI : http://dx.doi.org/10.1016/j.physrep.2009.11.002. URL : http://www.sciencedirect.com/science/article/pii/S0370157309002841.

[67] Marco Gaertler, Robert Görke et Dorothea Wagner. « Significance-Driven Graph Clustering ». Dans : *Algorithmic Aspects in Information and Management*. Sous la dir. de Ming-Yang Kao et Xiang-Yang Li. T. 4508. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, p. 11–26. ISBN : 978-3-540-72868-9. DOI : 10.1007/978-3-540-72870-2_2. URL : http://dx.doi.org/10.1007/978-3-540-72870-2_2.

[68] Pablo M Gleiser et Leon Danon. « Community structure in jazz ». Dans : *Advances in complex systems* 6.04 (2003), p. 565–573.

[69] Benjamin H Good, Yves-Alexandre de Montjoye et Aaron Clauset. « Performance of modularity maximization in practical contexts ». Dans : *Physical Review E* 81.4 (2010), p. 046106.

[70] Gösta Grahne et Jianfei Zhu. « Efficiently Using Prefix-trees in Mining Frequent Itemsets ». Dans : *FIMI*. 27 avr. 2004.

[71] Steve Gregory. « Finding Overlapping Communities Using Disjoint Community Detection Algorithms ». Dans : *Complex Networks*. Sous la dir. de Santo Fortunato et al. T. 207. Studies in Computational Intelligence. Springer Berlin Heidelberg, 2009, p. 47–61. ISBN : 978-3-642-01205-1. DOI : 10.1007/978-3-642-01206-8_5. URL : http://dx.doi.org/10.1007/978-3-642-01206-8_5.

[72] V. Gudkov et al. « Community detection in complex networks by dynamical simplex evolution ». Dans : 78.1, 016113 (juil. 2008), p. 016113. DOI : 10.1103/PhysRevE.78.016113. arXiv :0710.0550 [cond-mat.dis-nn].

[73] Roger Guimerà et Luís A. Nunes Amaral. « Nunes Amaral. Functional cartography of complex metabolic networks ». Dans : *Nature* (2005).

[74] Roger GUIMERÀ, Marta SALES-PARDO et Luís A. Nunes AMARAL. « Modularity from fluctuations in random graphs and complex networks ». Dans : *Phys. Rev. E* 70 (2 août 2004), p. 025101. DOI : 10.1103/PhysRevE.70.025101. URL : http://link.aps.org/doi/10.1103/PhysRevE.70.025101.

[75] Peter HABERMEHL et al. « Forest automata for verification of heap manipulation ». Dans : *Formal Methods in System Design* 41.1 (2012), p. 83–106.

[76] M. B. HASTINGS. « Community detection as an inference problem ». Dans : *Phys. Rev. E* 74 (3 sept. 2006), p. 035102. DOI : 10.1103/PhysRevE.74.035102. URL : http://link.aps.org/doi/10.1103/PhysRevE.74.035102.

[77] Jake M. HOFMAN et Chris H. WIGGINS. « Bayesian Approach to Network Modularity ». Dans : *Phys. Rev. Lett.* 100 (25 juin 2008), p. 258701. DOI : 10.1103/PhysRevLett.100.258701. URL : http://link.aps.org/doi/10.1103/PhysRevLett.100.258701.

[78] Petter HOLME, Mikael HUSS et Hawoong JEONG. « Subnetwork hierarchies of biochemical pathways ». Dans : *Bioinformatics* 19.4 (2003), p. 532–538.

[79] Yanqing HU et al. « Community detection by signaling on complex networks ». Dans : *Phys. Rev. E* 78 (1 juil. 2008), p. 016115. DOI : 10.1103/PhysRevE.78.016115. URL : http://link.aps.org/doi/10.1103/PhysRevE.78.016115.

[80] R Kang-Xing JIN, David C PARKES et Patrick J WOLFE. « Analysis of bidding networks in eBay : aggregate preference identification through community detection ». Dans : *Proceedings of AAAI workshop on plan, activity and intent recognition (PAIR)*. 2007.

[81] B. W. KERNIGHAN et S. LIN. « An Efficient Heuristic Procedure for Partitioning Graphs ». Dans : *The Bell system technical journal* 49.1 (1970), p. 291–307.

[82] Alexey V. KHOROSHILOV et al. « Establishing Linux Driver Verification Process ». Dans : *Ershov Memorial Conference*. 16 mar. 2010, p. 165–176.

[83] Gerwin KLEIN et al. « seL4 : formal verification of an OS kernel ». Dans : *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. SOSP '09. Big Sky, Montana, USA : ACM, 25 oct. 2009, p. 207–220. ISBN : 978-1-60558-752-3. DOI : http://doi.acm.org/10.1145/1629575.1629596. URL : http://doi.acm.org/10.1145/1629575.1629596.

[84] Daniel KROENING, Edmund CLARKE et Karen YORAV. « Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking ». Dans : *Proceedings of DAC 2003*. ACM Press, 24 jan. 2003, p. 368–371. ISBN : 1-58113-688-9.

[85] Daniel KROENING, Edmund CLARKE et Karen YORAV. *Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking*. Rap. tech. School of Computer Science, Carnegie Mellon University, 1er mar. 2003.

[86] J. M. KUMPULA et al. « Sequential algorithm for fast clique percolation ». Dans : 78.2, 026109 (août 2008), p. 026109. DOI : 10.1103/PhysRevE.78.026109. arXiv :0805.1449 [physics.soc-ph].

[87] M. Kwiatkowska, G. Norman et D. Parker. « A Framework for Verification of Software with Time and Probabilities ». Dans : *Proc. 8th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS'10)*. Sous la dir. de K. Chatterjee et T. Henzinger. T. 6246. LNCS. Springer, 13 déc. 2010, p. 25–45.

[88] David Lacey. « Program Transformation using Temporal Logic Specifications ». Thèse de doct. Oxford University Computing Laboratory, 15 août 2003.

[89] A. Lancichinetti, F. Radicchi et J. J. Ramasco. « Statistical significance of communities in networks ». Dans : 81.4, 046110 (avr. 2010), p. 046110. DOI : 10.1103/PhysRevE.81.046110. arXiv :0907.3708 [physics.soc-ph].

[90] Julia L. Lawall, Gilles Muller et Richard Urunuela. « Tarantula : Killing Driver Bugs Before They Hatch ». Dans : *The 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*. Chicago, IL, 16 mar. 2005, p. 13–18.

[91] Julia L. Lawall et al. *WYSIWIB : A Declarative Approach to Finding Protocols and Bugs in Linux Code*. Rap. tech. 08/1/INFO. Nantes, France : Ecole des Mines de Nantes, 8 juil. 2008.

[92] Julia L. Lawall et al. « WYSIWIB : A Declarative Approach to Finding Protocols and Bugs in Linux Code ». Dans : *The 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Estoril, Portugal, 2 juil. 2009, p. 43–52.

[93] Julia Lawall, Gilles Muller et Nicolas Palix. « Enforcing the Use of API Functions in Linux Code ». Dans : *8th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS '09)*. Charlottesville, VA, USA, 4 mar. 2009.

[94] S. Lehmann et L. K. Hansen. « Deterministic modularity optimization ». English. Dans : *The European Physical Journal B* 60.1 (2007), p. 83–88. ISSN : 1434-6028. DOI : 10.1140/epjb/e2007-00313-2. URL : http://dx.doi.org/10.1140/epjb/e2007-00313-2.

[95] Sune Lehmann, Martin Schwartz et Lars Kai Hansen. « Biclique communities ». Dans : *Phys. Rev. E* 78 (1 juil. 2008), p. 016108. DOI : 10.1103/PhysRevE.78.016108. URL : http://link.aps.org/doi/10.1103/PhysRevE.78.016108.

[96] E. A. Leicht et M. E. J. Newman. « Community Structure in Directed Networks ». Dans : *Phys. Rev. Lett.* 100 (11 mar. 2008), p. 118703. DOI : 10.1103/PhysRevLett.100.118703. URL : http://link.aps.org/doi/10.1103/PhysRevLett.100.118703.

[97] Jure Leskovec et al. « Microscopic evolution of social networks ». Dans : *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. KDD '08. Las Vegas, Nevada, USA : ACM, 2008, p. 462–470. ISBN : 978-1-60558-193-4. DOI : 10.1145/1401890.1401948. URL : http://doi.acm.org/10.1145/1401890.1401948.

[98] D. Li et al. « Synchronization Interfaces and Overlapping Communities in Complex Networks ». Dans : *Phys. Rev. Lett.* 101 (16 oct. 2008), p. 168701. DOI : 10.1103/PhysRevLett.101.168701. URL : http://link.aps.org/doi/10.1103/PhysRevLett.101.168701.

[99] Alexandre Lissy. « On our way to apply model-checking to the kernel ». Dans : *Linux Driver Verification Workshop (ISoLA 2012)*. Heraklion, Crete, oct. 2012.

[100] Alexandre Lissy, Stéphane Laurière et Patrick Martineau. « Faults in patched kernel ». Dans : *Linux Symposium*. Ottawa, Canada, juin 2011.

[101] Alexandre Lissy, Stéphane Laurière et Patrick Martineau. « Model Checking the Linux Kernel ? » Dans : Brussels, Belgium, fév. 2011.

[102] Alexandre Lissy et Jean Parpaillon. « Code Quality Platform ». Dans : *Linux Symposium*. Ottawa, Canada, juil. 2012.

[103] Alexandre Lissy, Jean Parpaillon et Patrick Martineau. « Clustering the Kernel ». Dans : *Linux Symposium*. Ottawa, Canada, juil. 2012.

[104] Zhenmin Li et Yuanyuan Zhou. « PR-Miner : Automatically extracting implicit programming rules and detecting violations in large software code ». Dans : *In ESEC/FSE*. ACM Press, 15 fév. 2006, p. 306–315.

[105] Zhenmin Li et al. « CP-Miner : a tool for finding copy-paste and related bugs in operating system code ». Dans : *OSDI'04 : Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. San Francisco, CA : USENIX Association, 21 jan. 2005, p. 20–20.

[106] Zhenmin Li et al. « CP-Miner : Finding Copy-Paste and Related Bugs in Large-Scale Software Code ». Dans : *IEEE Transactions on Software Engineering* 32 (22 août 2006), p. 176–192. ISSN : 0098-5589. DOI : http://doi.ieeecomputersociety.org/10.1109/TSE.2006.28.

[107] Françcois Lorrain et Harrison C. White. « Structural equivalence of individuals in social networks ». Dans : *The Journal of Mathematical Sociology* 1.1 (1971), p. 49–80. DOI : 10.1080/0022250X.1971.9989788.

[108] F. Luccio et M. Sami. « On the Decomposition of Networks in Minimally Interconnected Subnetworks ». Dans : *Circuit Theory, IEEE Transactions on* 16.2 (1969), p. 184–188. ISSN : 0018-9324. DOI : 10.1109/TCT.1969.1082924.

[109] David Lusseau et al. « The bottlenose dolphin community of Doubtful Sound features a large proportion of long-lasting associations ». English. Dans : *Behavioral Ecology and Sociobiology* 54.4 (2003), p. 396–405. ISSN : 0340-5443. DOI : 10.1007/s00265-003-0651-y. URL : http://dx.doi.org/10.1007/s00265-003-0651-y.

[110] J. MacQueen. « Some methods for classification and analysis of multivariate observations ». Dans : *Proc. Fifth Berkeley Symp. on Math. Statist. and Prob.* T. 1. Univ. of Calif. Press, 1967, p. 281–297.

[111] Claire P. Massen et Jonathan P. K. Doye. « Identifying communities within energy landscapes ». Dans : *Phys. Rev. E* 71 (4 avr. 2005), p. 046101. DOI : 10.1103/PhysRevE.71.046101. URL : http://link.aps.org/doi/10.1103/PhysRevE.71.046101.

[112] Robert Mokken. « Cliques, clubs and clans ». Dans : *Quality & Quantity : International Journal of Methodology* 13.2 (1979), p. 161–173. URL : http://EconPapers.repec.org/RePEc:spr:qualqt:v:13:y:1979:i:2:p:161-173.

[113] Matthieu Moy. « Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level ». Thèse de doct. INPG, Grenoble, France, 9 déc. 2005. URL : http://www-verimag.imag.fr/~moy/phd/.

[114] M. Mungan et J. J. Ramasco. « Stability of Maximum likelihood based clustering methods : exploring the backbone of classifications (Who is keeping you in that community ?) » Dans : *ArXiv e-prints* (sept. 2008). arXiv :0809.1398 [physics.soc-ph].

[115] Michael Swift Muthukaruppan et al. « Recovering Device Drivers ». Dans : *In OSDI*. 15 déc. 2004, p. 1–16.

[116] M. E. J. Newman. « Analysis of weighted networks ». Dans : *Phys. Rev. E* 70 (5 nov. 2004), p. 056131. DOI : 10.1103/PhysRevE.70.056131. URL : http://link.aps.org/doi/10.1103/PhysRevE.70.056131.

[117] M. E. J. Newman. « Fast algorithm for detecting community structure in networks ». Dans : *Phys. Rev. E* 69 (6 juin 2004), p. 066133. DOI : 10.1103/PhysRevE.69.066133. URL : http://link.aps.org/doi/10.1103/PhysRevE.69.066133.

[118] M.E. J. Newman. « A measure of betweenness centrality based on random walks ». Dans : *Social Networks* 27.1 (2005), p. 39 –54. ISSN : 0378-8733. DOI : http://dx.doi.org/10.1016/j.socnet.2004.11.009. URL : http://www.sciencedirect.com/science/article/pii/S0378873304000681.

[119] M. E. J. Newman et M. Girvan. « Finding and evaluating community structure in networks ». Dans : *Phys. Rev. E* 69 (2 fév. 2004), p. 026113. DOI : 10.1103/PhysRevE.69.026113. URL : http://link.aps.org/doi/10.1103/PhysRevE.69.026113.

[120] M. E. J. Newman et E. A. Leicht. « Mixture models and exploratory analysis in networks ». Dans : *Proceedings of the National Academy of Sciences* 104.23 (2007), p. 9564–9569. DOI : 10.1073/pnas.0610537104. eprint : http://www.pnas.org/content/104/23/9564.full.pdf+html. URL : http://www.pnas.org/content/104/23/9564.abstract.

[121] V. Nicosia et al. « Extending the definition of modularity to directed graphs with overlapping communities ». Dans : *Journal of Statistical Mechanics : Theory and Experiment* 3 (mar. 2009), p. 24. DOI : 10.1088/1742-5468/2009/03/P03024. arXiv :0801.1647 [physics.data-an].

[122] A. Jefferson Offutt, Mary Jean Harrold et Priyadarshan Kolte. « A software metric system for module coupling ». Dans : *J. Syst. Softw.* 20 (3 15 mar. 1993), p. 295–308. ISSN : 0164-1212. DOI : http://dx.doi.org/10.1016/0164-1212(93)90072-6. URL : http://dx.doi.org/10.1016/0164-1212(93)90072-6.

[123] Jun Ohkubo et Kazuyuki Tanaka. « Nonadditive Volume and Community Detection Problem in Complex Networks ». Dans : *Journal of the Physical Society of Japan* 75.11 (2006), p. 115001. DOI : 10.1143/JPSJ.75.115001. URL : http://jpsj.ipap.jp/link?JPSJ/75/115001/.

[124] Yoann Padioleau, Julia L. Lawall et Gilles Muller. « Semantic Patches, Documenting and Automating Collateral Evolutions in Linux Device Drivers ». Dans : *Ottawa Linux Symposium (OLS 2007)*. Ottawa, Canada, 15 mai 2007.

[125] Yoann Padioleau, Julia L. Lawall et Gilles Muller. « SmPL : A Domain-Specific Language for Specifying Collateral Evolutions in Linux Device Drivers ». Dans : *International ERCIM Workshop on Software Evolution (2006)*. Lille, France, 12 fév. 2007.

[126] Yoann Padioleau, Julia L. Lawall et Gilles Muller. « Understanding Collateral Evolution in Linux Device Drivers ». Dans : *The first ACM SIGOPS EuroSys conference (EuroSys 2006)*. Leuven, Belgium, 16 mai 2007, p. 59–71.

[127] Yoann Padioleau et al. « Documenting and Automating Collateral Evolutions in Linux Device Drivers ». Dans : *EuroSys 2008*. Glasgow, Scotland, 2 avr. 2008, p. 247–260.

[128] Yoann Padioleau et al. « Semantic Patches for Documenting and Automating Collateral Evolutions in Linux Device Drivers ». Dans : *PLOS 2006 : Linguistic Support for Modern Operating Systems*. San Jose, CA, 22 oct. 2006.

[129] Yoann Padioleau et al. *Towards Documenting and Automating Collateral Evolutions in Linux Device Drivers*. Research Report 6090. INRIA, 15 jan. 2007. URL : https://hal.inria.fr/inria-00123142.

[130] Nicolas Palix, Julia Lawall et Gilles Muller. *Herodotos : A Tool to Expose Bugs' Lives*. Anglais. Research Report RR-6984. INRIA, 21 juil. 2009, p. 16. URL : http://hal.inria.fr/inria-00406306/PDF/RR-6984.pdf.

[131] Nicolas Palix, Julia Lawall et Gilles Muller. « Tracking code patterns over multiple software versions with Herodotos ». Dans : *AOSD '10 : Proceedings of the 9th International Conference on Aspect-Oriented Software Development*. Rennes et Saint-Malo, France : ACM, 19 mar. 2010, p. 169–180. ISBN : 978-1-60558-958-9. DOI : http://doi.acm.org/10.1145/1739230.1739250.

[132] Nicolas Palix et al. *Faults in Linux : Ten Years Later*. Anglais. Research Report RR-7357. INRIA, 11 août 2010, p. 21. URL : http://hal.inria.fr/inria-00509256/PDF/RR-7357.pdf.

[133] Gergely Palla et al. « Uncovering the overlapping community structure of complex networks in nature and society ». Dans : *Nature* 435 (juin 2005), p. 814 –818. ISSN : 0028-0836. DOI : 10.1038/nature03607.

[134] S. Papadopoulos et al. « Bridge Bounding : A Local Approach for Efficient Community Discovery in Complex Networks ». Dans : *ArXiv e-prints* (fév. 2009). arXiv :0902.0871 [physics.data-an].

[135] C. Peterson et J. Anderson. « A mean field theory learning algorithm for neural networks ». Dans : *Complex systems* 1 (1987), p. 995–1019.

[136] Pascal Pons. « Post-Processing Hierarchical Community Structures : Quality Improvements and Multi-scale View ». Dans : *CoRR* abs/cs/0608050 (2006).

[137] Pascal Pons et Matthieu Latapy. « Computing Communities in Large Networks Using Random Walks ». Dans : *Computer and Information Sciences - ISCIS 2005*. Sous la dir. de pInar Yolum et al. T. 3733. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, p. 284–293. isbn : 978-3-540-29414-6. doi : 10.1007/11569596_31. url : http://dx.doi.org/10.1007/11569596_31.

[138] H. Post et W. Küchlin. « Integrated static analysis for Linux device driver verification ». Dans : *Integrated Formal Methods*. Springer. 20 sept. 2007, p. 518–537.

[139] Filippo Radicchi et al. « Defining and identifying communities in networks ». Dans : *Proceedings of the National Academy of Sciences of the United States of America* 101.9 (2004), p. 2658–2663. doi : 10.1073/pnas.0400054101. eprint : http://www.pnas.org/content/101/9/2658.full.pdf+html. url : http://www.pnas.org/content/101/9/2658.abstract.

[140] U. N. Raghavan, R. Albert et S. Kumara. « Near linear time algorithm to detect community structures in large-scale networks ». Dans : 76.3, 036106 (sept. 2007), p. 036106. doi : 10.1103/PhysRevE.76.036106. arXiv :0709.2938 [physics.soc-ph].

[141] José J. Ramasco et Muhittin Mungan. « Inversion method for content-based networks ». Dans : *Phys. Rev. E* 77 (3 mar. 2008), p. 036122. doi : 10.1103/PhysRevE.77.036122. url : http://link.aps.org/doi/10.1103/PhysRevE.77.036122.

[142] Matthew J Rattigan, Marc Maier et David Jensen. « Using structure indices for efficient approximation of network properties ». Dans : *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2006, p. 357–366.

[143] Jörg Reichardt et Stefan Bornholdt. « Clustering of sparse data via network communities—a prototype study of a large online market ». Dans : *Journal of Statistical Mechanics : Theory and Experiment* 2007.06 (2007), P06016. url : http://stacks.iop.org/1742-5468/2007/i=06/a=P06016.

[144] Jörg Reichardt et Stefan Bornholdt. « Detecting Fuzzy Community Structures in Complex Networks with a Potts Model ». Dans : *Phys. Rev. Lett.* 93 (21 nov. 2004), p. 218701. doi : 10.1103/PhysRevLett.93.218701. url : http://link.aps.org/doi/10.1103/PhysRevLett.93.218701.

[145] Jörg Reichardt et Stefan Bornholdt. « Statistical mechanics of community detection ». Dans : *Phys. Rev. E* 74 (1 juil. 2006), p. 016110. doi : 10.1103/PhysRevE.74.016110. url : http://link.aps.org/doi/10.1103/PhysRevE.74.016110.

[146] Jörg Reichardt et Stefan Bornholdt. « When are networks truly modular ? » Dans : *Physica D : Nonlinear Phenomena* 224.1–2 (2006). Dynamics on Complex Networks and Applications, p. 20 –26. issn : 0167-2789. doi : http://dx.doi.org/10.1016/j.physd.2006.09.009. url : http://www.sciencedirect.com/science/article/pii/S0167278906003678.

[147] J. Reichardt et D. R. White. « Role models for complex networks ». English. Dans : *The European Physical Journal B* 60.2 (2007), p. 217–224. ISSN : 1434-6028. DOI : 10.1140/epjb/e2007-00340-y. URL : http://dx.doi.org/10.1140/epjb/e2007-00340-y.

[148] Wei Ren et al. « Simple probabilistic algorithm for detecting community structure ». Dans : *Phys. Rev. E* 79 (3 mar. 2009), p. 036111. DOI : 10.1103/PhysRevE.79.036111. URL : http://link.aps.org/doi/10.1103/PhysRevE.79.036111.

[149] Sune Rievers. *Finding Bugs in Open Source Software using Coccinelle*. 13 jan. 2010.

[150] Alexander W. Rives et Timothy Galitski. « Modular organization of cellular networks ». Dans : *Proceedings of the National Academy of Sciences* 100.3 (2003), p. 1128–1133. DOI : 10.1073/pnas.0237338100. eprint : http://www.pnas.org/content/100/3/1128.full.pdf+html. URL : http://www.pnas.org/content/100/3/1128.abstract.

[151] P. Ronhovde et Z. Nussinov. « Local resolution-limit-free Potts model for community detection ». Dans : *ArXiv e-prints* (mar. 2008). arXiv :0803.2548 [physics.soc-ph].

[152] P. Ronhovde et Z. Nussinov. « Multiresolution community detection for megascale networks by information-based replica correlations ». Dans : *ArXiv e-prints* (déc. 2008). arXiv :0812.1072 [physics.soc-ph].

[153] Martin Rosvall et Carl T. Bergstrom. « An information-theoretic framework for resolving community structure in complex networks ». Dans : *Proceedings of the National Academy of Sciences* 104.18 (2007), p. 7327–7331. DOI : 10.1073/pnas.0611034104. eprint : http://www.pnas.org/content/104/18/7327.full.pdf+html. URL : http://www.pnas.org/content/104/18/7327.abstract.

[154] Martin Rosvall et Carl T. Bergstrom. « Maps of random walks on complex networks reveal community structure ». Dans : *Proceedings of the National Academy of Sciences* 105.4 (2008), p. 1118–1123. DOI : 10.1073/pnas.0706851105. eprint : http://www.pnas.org/content/105/4/1118.full.pdf+html. URL : http://www.pnas.org/content/105/4/1118.abstract.

[155] M. Rosvall, D. Axelsson et C. T. Bergstrom. « The map equation ». Dans : *European Physical Journal Special Topics* 178 (nov. 2009), p. 13–23. DOI : 10.1140/epjst/e2010-01179-1. arXiv :0906.1405 [physics.soc-ph].

[156] Jianhua Ruan et Weixiong Zhang. *An Efficient Spectral Algorithm for Network Community Discovery and Its Applications to Biological and Social Networks*. 2007.

[157] Marta Sales-Pardo et al. « Extracting the hierarchical organization of complex systems ». Dans : *Proceedings of the National Academy of Sciences* 104.39 (2007), p. 15224–15229. DOI : 10.1073/pnas.0703740104. eprint : http://www.pnas.org/content/104/39/15224.full.pdf+html. URL : http://www.pnas.org/content/104/39/15224.abstract.

[158] Basile Starynkévitch. « Extending the GCC compiler with MELT to suit your needs ». Dans : *RMLL 2010*. 9 juil. 2010.

[159] Philipp Schuetz et Amedeo Caflisch. « Efficient modularity optimization by multistep greedy algorithm and vertex mover refinement ». Dans : *Phys. Rev. E* 77 (4 avr. 2008), p. 046112. doi : 10.1103/PhysRevE.77.046112. url : http://link.aps.org/doi/10.1103/PhysRevE.77.046112.

[160] Stephen B. Seidman. « Network structure and minimum degree ». Dans : *Social Networks* 5.3 (1983), p. 269 –287. issn : 0378-8733. doi : http://dx.doi.org/10.1016/0378-8733(83)90028-X. url : http://www.sciencedirect.com/science/article/pii/037887338390028X.

[161] Stephen B. Seidman et Brian L. Foster. « A graph-theoretic generalization of the clique concept ». Dans : *Journal of Mathematical Sociology* 6 (1978), p. 139–154.

[162] Huawei Shen et al. « Detect overlapping and hierarchical community structure in networks ». Dans : *Physica A : Statistical Mechanics and its Applications* 388.8 (2009), p. 1706 –1712. issn : 0378-4371. doi : http://dx.doi.org/10.1016/j.physa.2008.12.021. url : http://www.sciencedirect.com/science/article/pii/S0378437108010376.

[163] Jirí Síma et Satu Elisa Schaeffer. « On the NP-Completeness of Some Graph Cluster Measures ». Dans : *CoRR* abs/cs/0506100 (2005).

[164] Herbert A. Simon. « The architecture of complexity ». Dans : *Proceedings of the American Philosophical Society*. 1962, p. 467–482.

[165] Julio Sincero et al. « Facing the Linux 8000 Feature Nightmare ». Dans : *Proceedings of ACM European Conference on Computer Systems (EuroSys 2010), Best Posters and Demos Session*. Sous la dir. d'ACM SIGOPS. Paris, France, 14 avr. 2010. url : http://www4.informatik.uni-erlangen.de/Publications/2010/sincero_tartler_eurosys-life.pdf.

[166] S.-W. Son, H. Jeong et J. D. Noh. « Random field Ising model and community structure in complex networks ». English. Dans : *The European Physical Journal B - Condensed Matter and Complex Systems* 50.3 (2006), p. 431–437. issn : 1434-6028. doi : 10.1140/epjb/e2006-00155-4. url : http://dx.doi.org/10.1140/epjb/e2006-00155-4.

[167] Victor Spirin et Leonid A. Mirny. « Protein complexes and functional modules in molecular networks ». Dans : *Proceedings of the National Academy of Sciences* 100.21 (2003), p. 12123–12128. doi : 10.1073/pnas.2032324100. eprint : http://www.pnas.org/content/100/21/12123.full.pdf+html. url : http://www.pnas.org/content/100/21/12123.abstract.

[168] Basile Starynkevitch et Pierre Vittet. « Extending GCC with MELT for Free Software ». Dans : *GNU Hackers Meeting in Paris, 2011*. IRILL, Paris, France, août 2011. url : http://www.gnu.org/ghm/2011/paris/slides/basile-starynkevitch+pierre-vittet-gcc-melt.pdf.

[169] Henrik Stuart. « Hunting bugs with Coccinelle ». Thèse de doct. Department of Computer Science, University of Copenhagen, 8 août 2008.

[170] Henrik STUART et al. « Towards Easing the Diagnosis of Bugs in OS Code ». Dans : *4th Workshop on Programming Languages and Operating Systems*. Stevenson, Washington, 18 oct. 2007, p. 1–5.

[171] Michael M. SWIFT, Brian N. BERSHAD et Henry M. LEVY. « Improving the reliability of commodity operating systems ». Dans : 15 nov. 2004, p. 207–222.

[172] Michael M. SWIFT et al. « Nooks : an architecture for reliable device drivers ». Dans : *EW 10 : Proceedings of the 10th workshop on ACM SIGOPS European workshop*. Saint-Emilion, France : ACM, 1er juil. 2002, p. 102–107. DOI : http://doi.acm.org/10.1145/1133373.1133393.

[173] Reinhard TARTLER et al. « Feature Consistency in Compile-Time Configurable System Software ». Dans : *Proceedings of the EuroSys 2011 Conference (EuroSys '11)*. Salzburg, 18 avr. 2011. ISBN : 978-1-4503-0634-8. URL : http://www4.informatik.uni-erlangen.de/Publications/2011/tartler_11_eurosys.pdf.

[174] Gergely TIBÉLY et János KERTÉSZ. « On the equivalence of the label propagation method of community detection and a Potts model approach ». Dans : *Physica A : Statistical Mechanics and its Applications* 387.19–20 (2008), p. 4982 –4984. ISSN : 0378-4371. DOI : http://dx.doi.org/10.1016/j.physa.2008.04.024. URL : http://www.sciencedirect.com/science/article/pii/S0378437108003932.

[175] A. L. TRAUD et al. « Comparing Community Structure to Characteristics in Online Collegiate Social Networks ». Dans : *ArXiv e-prints* (sept. 2008). arXiv :0809.0690 [physics.soc-ph].

[176] Joshua R. TYLER, Dennis M. WILKINSON et Bernardo A. HUBERMAN. « Communities and technologies ». Dans : sous la dir. de Marleen HUYSMAN, Etienne WENGER et Volker WULF. Deventer, The Netherlands, The Netherlands : Kluwer, B.V., 2003. Chap. Email as spectroscopy : automated discovery of community structure within organizations, p. 81–96. ISBN : 1-4020-1611-5. URL : http://dl.acm.org/citation.cfm?id=966263.966268.

[177] A. VAZQUEZ. « Bayesian approach to clustering real value, categorical and network data : solution via variational methods ». Dans : *ArXiv e-prints* (mai 2008). arXiv :0805.2689 [physics.data-an].

[178] Alexei VAZQUEZ. « Population stratification using a statistical model on hypergraphs ». Dans : *Phys. Rev. E* 77 (6 juin 2008), p. 066106. DOI : 10.1103/PhysRevE.77.066106. URL : http://link.aps.org/doi/10.1103/PhysRevE.77.066106.

[179] Clark VERBRUGGE, Phong CO et Laurie HENDREN. « Generalized Constant Propagation A Study in C ». Dans : *In 6th Int. Conf. on Compiler Construction, volume 1060 of Lec. Notes in Comp. Sci.* Springer, 25 avr. 1996, p. 74–90.

[180] Cédric VERNOU. *Forester usage example*. Avr. 2013. URL : https://github.com/Orwel/forester_example.

[181] Pierre VITTET. « How to write a GCC plugin with MELT ? » Dans : *Rencontres Mondiales du Logiciel Libre*. Strasbourg, France, juil. 2011. URL : http://2011.rmll.info/IMG/pdf/MELT-RMLL-2011.pdf.

[182] I. Vragović et al. « Diversity-induced synchronized oscillations in close-to-threshold excitable elements arranged on regular networks : Effects of network topology ». Dans : *Physica D : Nonlinear Phenomena* 219.2 (2006), p. 111 –119. ISSN : 0167-2789. DOI : http://dx.doi.org/10.1016/j.physd.2006.05.017. URL : http://www.sciencedirect.com/science/article/pii/S0167278906001904.

[183] Ken Wakita et Toshiyuki Tsurumi. « Finding Community Structure in Mega-scale Social Networks ». Dans : *CoRR* abs/cs/0702048 (2007).

[184] Gaoxia Wang, Yi Shen et Ming Ouyang. « A vector partitioning approach to detecting community structure in complex networks ». Dans : *Computers & Mathematics with Applications* 55.12 (2008), p. 2746 –2752. ISSN : 0898-1221. DOI : http://dx.doi.org/10.1016/j.camwa.2007.10.028. URL : http://www.sciencedirect.com/science/article/pii/S0898122107007742.

[185] S. Wasserman et K. Faust. *Social Network Analysis : Methods and Applications*. Structural Analysis in the Social Sciences. Cambridge University Press, 1994. ISBN : 9780521387071. URL : http://books.google.fr/books?id=CAm2DpIqRUIC.

[186] D. J. Watts et S. H. Strogatz. « Collective dynamics of'small-world'networks. » Dans : *Nature* 393.6684 (1998), p. 409–10.

[187] Dennis M. Wilkinson et Bernardo A. Huberman. « A method for finding communities of related genes ». Dans : *Proceedings of the National Academy of Sciences* 101.suppl 1 (2004), p. 5241–5248. DOI : 10.1073/pnas.0307740100. eprint : http://www.pnas.org/content/101/suppl_1/5241.full.pdf+html. URL : http://www.pnas.org/content/101/suppl_1/5241.abstract.

[188] T. Witkowski et al. « Model checking concurrent linux device drivers ». Dans : *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM. 7 fév. 2008, p. 501–504.

[189] F. Wu et B.A. Huberman. « Finding communities in linear time : a physics approach ». English. Dans : *The European Physical Journal B - Condensed Matter and Complex Systems* 38.2 (2004), p. 331–338. ISSN : 1434-6028. DOI : 10.1140/epjb/e2004-00125-x. URL : http://dx.doi.org/10.1140/epjb/e2004-00125-x.

[190] Yichen Xie et Alex Aiken. « Saturn : A scalable framework for error detection using Boolean satisfiability ». Dans : *ACM Trans. Program. Lang. Syst.* 29.3 (6 nov. 2007), p. 16. ISSN : 0164-0925. DOI : http://doi.acm.org/10.1145/1232420.1232423.

[191] Bo Yang et Jiming Liu. « Discovering global network communities based on local centralities ». Dans : *ACM Trans. Web* 2.1 (mar. 2008), 9 :1–9 :32. ISSN : 1559-1131. DOI : 10.1145/1326561.1326570. URL : http://doi.acm.org/10.1145/1326561.1326570.

[192] Xifeng Yan, Jiawei Han et Ramin Afshar. « CloSpan : Mining Closed Sequential Patterns in Large Datasets ». Dans : *In SDM*. 2 mai 2003, p. 166–177.

[193] Hao YUAN et Patrick EUGSTER. « An Efficient Algorithm for Solving the Dyck-CFL Reachability Problem on Trees ». Dans : *ESOP '09 : Proceedings of the 18th European Symposium on Programming Languages and Systems*. York, UK : Springer-Verlag, 1er avr. 2009, p. 175–189. ISBN : 978-3-642-00589-3. DOI : http://dx.doi.org/10.1007/978-3-642-00590-9_13.

[194] L. YU et al. « Categorization of common coupling and its application to the maintainability of the Linux kernel ». Dans : *IEEE Transactions on Software Engineering* (3 août 2005), p. 694–706.

[195] K. YUTA, N. ONO et Y. FUJIWARA. « A Gap in the Community-Size Distribution of a Large-Scale Social Networking Site ». Dans : *ArXiv Physics e-prints* (jan. 2007). eprint : arXiv:physics/0701168.

[196] W. W. ZACHARY. « An information flow model for conflict and fission in small groups ». Dans : *Journal of Anthropological Research* 33 (1977), p. 452–473.

[197] Hugo ZANGHI, Christophe AMBROISE et Vincent MIELE. « Fast online graph clustering via Erdős–Rényi mixture ». Dans : *Pattern Recognition* 41.12 (2008), p. 3592 – 3599. ISSN : 0031-3203. DOI : http://dx.doi.org/10.1016/j.patcog.2008.06.019. URL : http://www.sciencedirect.com/science/article/pii/S0031320308002483.

[198] Mina ZAREI et Keivan Aghababaei SAMANI. « Eigenvectors of network complement reveal community structure more accurately ». Dans : *Physica A : Statistical Mechanics and its Applications* 388.8 (2009), p. 1721 –1730. ISSN : 0378-4371. DOI : http://dx.doi.org/10.1016/j.physa.2009.01.007. URL : http://www.sciencedirect.com/science/article/pii/S0378437109000338.

[199] Shihua ZHANG, Rui-Sheng WANG et Xiang-Sun ZHANG. « Identification of overlapping community structure in complex networks using fuzzy -means clustering ». Dans : *Physica A : Statistical Mechanics and its Applications* 374.1 (2007), p. 483 –490. ISSN : 0378-4371. DOI : http://dx.doi.org/10.1016/j.physa.2006.07.023. URL : http://www.sciencedirect.com/science/article/pii/S0378437106008119.

[200] Haijun ZHOU. « Distance, dissimilarity index, and network community structure ». Dans : *Phys. Rev. E* 67 (6 juin 2003), p. 061901. DOI : 10.1103/PhysRevE.67.061901. URL : http://link.aps.org/doi/10.1103/PhysRevE.67.061901.

[201] Haijun ZHOU. « Network landscape from a Brownian particle's perspective ». Dans : *Phys. Rev. E* 67 (4 avr. 2003), p. 041908. DOI : 10.1103/PhysRevE.67.041908. URL : http://link.aps.org/doi/10.1103/PhysRevE.67.041908.

[202] Haijun ZHOU et Reinhard LIPOWSKY. « Network Brownian Motion : A New Method to Measure Vertex-Vertex Proximity and to Identify Communities and Subcommunities ». Dans : *Computational Science - ICCS 2004*. Sous la dir. de Marian BUBAK et al. T. 3038. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, p. 1062–1069. ISBN : 978-3-540-22116-6. DOI : 10.1007/978-3-540-24688-6_137. URL : http://dx.doi.org/10.1007/978-3-540-24688-6_137.

[203] Etay Ziv, Manuel Middendorf et Chris H. Wiggins. « Information-theoretic approach to network modularity ». Dans : *Physical Review E* 71.4 (avr. 2005), p. 046117+. DOI : 10.1103/physreve.71.046117. URL : http://dx.doi.org/10.1103/physreve.71.046117.

Résumé :

Dans cette thèse nous nous intéressons à intégrer dans la distribution Linux produite par Mandriva une assurance qualité permettant de proposer des garanties de propriétés sur le code exécuté. Le processus de création d'une distribution implique l'utilisation de logiciels de provenances diverses pour proposer un assemblage cohérent et présentant une valeur ajoutée pour l'utilisateur. Ceci engendre une moindre maîtrise potentielle sur le code. Un audit manuel permet de s'assurer que celui-ci présente de bonnes propriétés, par exemple, en matière de sécurité. Le nombre croissant de composants à intégrer, et la croissance de la quantité de code de chacun amènent à avoir besoin d'outils pour permettre une assurance qualité. Après une étude de la distribution nous choisissons de nous concentrer sur un paquet critique, le noyau Linux : nous proposons un état de l'art des méthodes de vérifications appliquées à ce contexte particulier, et identifions le besoin d'améliorer la compréhension de la structure du code source, la question de l'explosion combinatoire et le manque d'intégration des outils d'analyse de l'état de l'art. Pour répondre à ces besoins nous proposons une représentation du code source sous la forme d'un graphe, et l'utilisons pour aider à la documentation et à la compréhension de l'architecture du code. Des méthodes de détection de communautés sont évaluées sur ce cas pour répondre au besoin de l'explosion combinatoire. Enfin nous proposons une architecture intégrée dans le système de construction de la distribution permettant d'intégrer des outils d'analyse et de vérification de code.

Mots clés :

Linux, distribution, vérification, qualité logicielle, communautés

Abstract :

In this thesis we are interested in integrating to the Linux distribution produced by Mandriva quality assurance level that allows ensuring user-defined properties on the source code used. The core work of a distribution and its producer is to create a meaningful aggregate from software available. Those softwares are free and open source, hence it is possible to adapt it to improve end user's experience. Hence, there is less control over the source code. Manual audit can of course be used to make sure it has good properties. Examples of such properties are often referring to security, but one could think of others. However, more and more software are getting integrated into distributions and each is showing an increase in source code volume: tools are needed to make quality assurance achievable. We start by providing a study of the distribution itself to document the current status. We use it to select some packages that we consider critical, and for which we can improve things with the condition that packages which are similar enough to the rest of the distribution will be considered first. This leads us to concentrating on the Linux kernel: we provide a state of the art overview of code verification applied to this piece of the distribution. We identify a need for a better understanding of the structure of the source code. To address those needs we propose to use a graph as a representation of the source code and use it to help document and understand its structure. Specifically we study applying some state of the art community detection algorithm to help handle the combinatory explosion. We also propose a distribution's build system-integrated architecture for executing, collecting and handling the analysis of data produced by verifications tools.

Keywords :

Linux, distribution, verification, software quality, communities