UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN MATHÉMATIQUES ET INFORMATIQUE APPLIQUÉES

PAR
DANIEL ST-YVES

DÉPENDANCES ET GESTION DES MODIFICATIONS DANS LES SYSTÈMES ORIENTÉS OBJET : UTILISATION DES GRAPHES DE CONTRÔLE

DÉCEMBRE 2007

DÉPENDANCES ET GESTION DES MODIFICATIONS DANS LES SYSTÈMES ORIENTÉS OBJET : UTILISATION DES GRAPHES DE CONTRÔLE

Daniel St-Yves

SOMMAIRE

L'analyse de l'impact de changement joue un rôle primordial dans la maintenance des systèmes logiciels. Elle est d'autant plus importante dans le cas des systèmes complexes et de grande envergure. Elle permet aux développeurs d'évaluer les effets possibles d'un changement dans le code d'un système. Ce mémoire présente, dans un premier temps, une étude expérimentale de trois techniques statiques de l'analyse de l'impact de changement : une technique basée sur les graphes d'appels traditionnels, une technique basée sur les graphes de contrôle réduits aux appels et une technique basée sur le slicing statique. La technique basée sur les graphes de contrôle réduits aux appels est une nouvelle technique que nous avons proposée au début de ce travail de recherche [Badri 05, St-Yves 06] pour supporter l'analyse de l'impact des changements dans les systèmes orientés objet. Nous avons développé un outil, structuré en plusieurs composantes, et intégrant les trois techniques. L'outil développé a été intégré à l'environnement de développement Eclipse. Suite aux résultats obtenus [Badri 07a, Badri 07b], nous présentons une seconde technique, que nous avons définie combinant les propriétés des graphes de contrôle et du slicing traditionnel ainsi que les résultats de l'étude empirique qui a été conduite pour l'évaluer. La dernière étude empirique a été conduite sur plusieurs techniques, incluant celle définie, sur plusieurs versions d'un large projet *open-source* Java (JMol). Les changements observés sur les différentes générations du projet, ont été collectés à partir des différentes versions du logiciel. Les ensembles retournés par les techniques des méthodes potentiellement affectées suite à un changement, ont été comparés aux changements réels observés. Les résultats ont démontré que la nouvelle technique, unifiant les concepts des graphes de contrôle et de données, améliore le temps machine de recherche d'impact, précise les ensembles de résultats et réduit l'effort de recherche de l'impact.

DEPENDANCIES AND IMPACT ANALYSIS IN OBJECT-ORIENTED SYSTEMS: A CONTROL GRAPH APPROACH

Daniel St-Yves

ABSTRACT

Impact analysis plays an important role in software maintenance, in particular in the case of complex and large-scale systems. It allows developers assessing the possible effects of a change in a software system. This thesis presents, in a first step, an experimental study of three static impact analysis techniques: a technique based on traditional call graphs, a technique based on control call graphs and a technique based on program slicing. The technique based on control call graphs is a new technique that we proposed at the beginning of this research work [Badri 05, St-Yves 06] to support change impact analysis in Object-Oriented Systems. We have developed a tool that supports the three techniques, structured as multiple modules integrated into the Eclipse environment. Following the results obtained [Badri 07a, Badri 07b], we introduce a second technique that combine the properties of the control call graph and the traditional slicing and also, the results of the performed empirical study. We conducted a comparative experimental study between the considered techniques on several versions of a Java (open-source) large project (Jmol). The observed changes were collected from different successive versions of the considered software. The sets of potentially affected methods returned by the three techniques, after a given change, were compared to the observed changes. The results showed that the new technique, unifying the control and data graph concepts, improve the computing time for impact research, precise the results sets and reduce the effort of impact research.

REMERCIEMENTS

Cette large étude n'aurait pu se réaliser sans la participation de plusieurs éléments clés dans ma vie.

Tout d'abord, je remercie de tout cœur l'énergie, la passion et la compétence dont mes directeurs de maîtrise, Linda Badri et Mourad Badri, professeurs au Département de mathématiques et d'informatique de l'Université du Québec à Trois-Rivières, ont fait preuve tout au long de ce projet. Vous m'avez montré qu'il est possible de trouver une solution à tout problème. Je suis bien heureux d'avoir été à votre charge.

Je tiens aussi à remercier mon père Claude St-Yves, qui ma encouragé moralement et financièrement, sans qui je n'aurais pas pu réaliser ce grand projet. Je remercie également ma proche famille, ma sœur Geneviève, ma mère Claudette et ma tendre moitié de m'avoir encouragé dans mon cheminement.

J'aimerais remercier mes collègues et amis sans qui cette étude n'aurait pas été aussi plaisante.

Cette étude a été rendue possible grâce à la contribution financière du CRSNG (Conseil de Recherches en Sciences Naturelles et en Génie du Canada) et à la fondation de l'UQTR.

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS ET DES SIGLES

CIA	Analyse d'Impact de Changements (change impact analysis)
REA	Ripple Effect Analysis (analyse de propagation de changement)
OO	Object-Oriented (orienté-objet)
CP	Change Proposition (proposition de changement)
CG	Call Graph (graphe d'appels)
CGd	Direct Call Graph (graphe d'appels directs)
CGi	Indirect Call Graph (graphe d'appels indirects)
CCG	Control Call Graph (graphe de contrôle réduit aux appels)
DCPG	Data Control Path Graph
DCP	Data Control Path
PS	Program Slicing (analyse de slice de programme)
FS	Forward Slicing
FS*	Direct Forward Slicing
BS	Backward Slicing

CHAPITRE 1

ANALYSE DE L'IMPACT : PROBLEMATIQUE, OBJECTIFS ET DÉMARCHE

L'évolution de la majorité des systèmes logiciels actuels présente des enjeux cruciaux. Plusieurs de ces systèmes ont nécessité d'importants efforts en temps de développement et de mise à jour. Il est donc important de faciliter leur évolution. Cet aspect représente une tâche essentielle, mais néanmoins complexe [Barros 95]. La maintenance a été pendant longtemps, et reste encore, reconnue comme étant une étape très coûteuse du processus de développement des logiciels [Sommerville 07, Pressman 07]. Deux des activités de base de la maintenance des systèmes logiciels sont la compréhension du système et l'évaluation des effets d'un changement [Barros 95]. La seconde activité est, en fait, étroitement liée à la première. En effet, selon Lee [Lee 00], pour comprendre les effets potentiels d'un changement donné, un développeur doit d'abord comprendre le système. La compréhension d'un système facilite l'analyse des effets d'un changement. Pour réduire les coûts importants de la maintenance, il est donc nécessaire de faciliter leur compréhension en utilisant des approches permettant, entre autres, d'évaluer les effets potentiels d'un changement dans différentes circonstances [Black 01, Basil 01, Han 97, Bohner 96-2, Rajlich 00]. Par ailleurs, la diversité ainsi que la complexité des interdépendances pouvant exister entre les composants d'un système, orienté-objet en particulier, rendent cette tâche encore plus difficile [Kung 94, Li 96, Turver 94]. Un changement au niveau d'un système, même mineur, peut conduire à des effets inattendus (ripple-effect) qui ne sont ni faciles ni évidents à détecter [Law 03].

L'analyse de l'impact d'un changement logiciel, appelée souvent *analyse de l'impact*, joue un rôle primordial dans ce contexte. L'analyse de l'impact permet aux développeurs d'évaluer les effets possibles d'un changement donné apporté au code source d'un programme. Les informations fournies par une analyse d'impact peuvent être utilisées pour la planification des changements, pour exécuter les changements ainsi que pour

suivre les effets d'un changement [Orso 03]. L'analyse de l'impact (CIA – Change Impact Analysis) consiste en une collection de techniques permettant d'évaluer les effets d'un ensemble de changements opérés sur le code source d'un programme [Ryder 01, Lustman 02]. Le CIA implique plusieurs techniques telles que l'analyse de la propagation (REA – Ripple Effect Analysis). Le REA est un processus itératif utilisé pour assurer la consistance et l'intégrité d'un système avant et après que les changements aient été implémentés [Yau 78, Wang 96, Black 01]. Il permet aux développeurs d'estimer les effets possibles d'un changement et de déterminer les parties éventuelles d'un système pouvant être affectées après ce changement. L'utilisation de telles techniques est essentielle pour l'estimation de l'effort requis après un changement. Le processus REA permet aux mainteneurs d'évaluer l'étendue et la propagation d'un changement donné sur un système [Black 01, Wang 96, Haider 05]. Même si le souci de supporter, d'une manière générale, l'analyse d'impact n'est pas récent, nous observons un manque crucial d'outils efficaces, en particulier pour les systèmes orientés objet. Les techniques récentes, dans ce domaine, bien que donnant pour certaines d'entre elles des résultats intéressants, doivent être encore expérimentées pour démontrer leur efficacité. Il existe encore de nombreux défis à relever pour arriver à offrir des solutions viables dans ce domaine.

L'analyse de l'impact est souvent utilisée pour évaluer les effets possibles d'un changement après son implémentation [Law 03]. Cependant, des approches plus proactives pourraient utiliser les techniques d'analyse de l'impact pour prédire les effets possibles d'un changement avant qu'il ne soit implémenté [Bohner 96, Law 03]. Les avantages dans ce cas seraient nombreux. En particulier, elles permettront aux développeurs de choisir, parmi plusieurs façons d'implémenter un changement, la solution présentant le moindre impact estimé. Les approches prédictives permettent, en effet, de donner un aperçu global sur les efforts requis en termes de coût et de planification [Law 03, Lee 00]. Barros, entre autres auteurs, discute dans [Barros 95] de l'importance de déterminer les effets des changements planifiés avant qu'ils ne soient

implémentés. L'analyse prédictive de l'impact requiert, cependant, que les mainteneurs spécifient l'endroit approximatif des changements.

Les techniques d'analyse de l'impact peuvent être divisées en deux classes principales [Bohner 96] : l'analyse de la traçabilité et l'analyse des dépendances. Nous nous intéressons aux techniques basées sur l'analyse des dépendances. Plusieurs techniques ont été proposées dans ce contexte [Briand 99, Bohner 96, Law 03, Lee 00, Li 95, Li 96, Ren 04, Ryder 01]. Les techniques d'analyse de l'impact peuvent être dynamiques [Law 03, Orso 03, Korel 90], statiques [Bohner 96, Kung 94, Badri 05], ou des techniques combinant à la fois l'analyse statique et l'analyse dynamique [Ren 04]. Les travaux effectués et présentés dans ce document mettent l'emphase sur les techniques statiques. Les techniques statiques d'analyse de l'impact sont basées sur une analyse statique du code source d'un programme et ne nécessitent pas son exécution.

Nous présentons, une étude expérimentale portant sur quatre approches d'analyse statique de l'impact des changements: technique basée sur les graphes d'appels traditionnels, technique basée sur les graphes de contrôle réduits aux appels (CCG – Control Call Graph), technique basée sur le slicing traditionnel et un modèle hybride unifiant les propriétés du slicing et des graphes de contrôle. La technique basée sur les graphes de contrôle réduits aux appels est une technique que nous avons proposée dans nos premiers travaux [Badri 05]. Elle utilise un nouveau modèle basé sur les chemins de contrôle réduits aux appels obtenus par analyse statique du code. Elle permet, en particulier, de supporter une analyse prédictive de l'impact d'un changement. Elle est basée sur les appels entre méthodes (granularité niveau méthode) et intègre le contrôle conduisant aux appels. Les détails des instructions atomiques ne contenant pas d'appels sont ignorés. Les chemins générés sous forme compactée, à partir du modèle considéré, sont utilisés pour identifier les composants potentiellement affectés suite à un changement et qui doivent être testés à nouveau.

De plus, nous présentons un second modèle basé sur le regroupement des instructions entre chemins de contrôles. Le nouveau modèle permet d'abstraire l'information résultante d'un « slice » pour faciliter sa compréhension. Il se base sur le cheminement des données et le cheminement des contrôles pour prédire un ensemble de méthodes affectées par un changement. Pour les deux approches proposées, le processus est itératif (réductions successives des composants à tester à nouveau). Par ailleurs, travaillant au niveau granularité des méthodes rend, tel que mentionné dans [Law 03], l'analyse plus appropriée dans la pratique. Les deux autres techniques, basées respectivement sur les graphes d'appels traditionnels et le « slicing » traditionnel, ont été prises de la littérature.

Nous nous sommes intéressés aux programmes orientés objet, en particulier les programmes Java. Les techniques proposées sont, cependant, assez générales et peuvent être adaptées facilement au paradigme de programmation impératif. Contrairement aux techniques basées sur les graphes d'appels traditionnels, notre technique (CCG) est plus précise et retourne des ensembles de méthodes potentiellement affectées moins importants (en taille) [Badri 05]. L'expérimentation conduite et présentée dans [Badri 05], portant sur une comparaison partielle entre le CCG et les graphes d'appels, a confirmé les conclusions de plusieurs autres recherches. Elle s'est limitée, néanmoins, à une analyse de la taille des ensembles retournés et n'a pas approfondie l'analyse de la qualité, en termes de précision, des ensembles retournés. Les techniques basées sur les graphes d'appels traditionnels peuvent être grandement imprécises tel que souligné dans [Law 03, Ren 04]. Elles peuvent, en fait, identifier un impact lorsqu'il n'existe pas et échouer lorsqu'il existe. Le modèle utilisé dans le cadre de notre technique étend les graphes d'appels traditionnels en intégrant le contrôle conduisant aux appels. Le nouveau modèle, plus précis que les graphes d'appels traditionnels, est utilisé pour générer les différents chemins de contrôle (chemin de contrôle réduit aux appels) en éliminant les chemins infaisables. Ceci permet d'éviter les inconvénients des approches dynamiques qui extraient ces mêmes chemins par analyse des traces d'exécution. Ces dernières tentent de déterminer le profil opérationnel du programme par l'analyse de ses

traces d'exécution. Elles semblent donner de bons résultats, mais sont néanmoins contraintes à la qualité des données utilisées comme mentionné dans [Orso 03]. Elles se basent aussi sur des ensembles de test pour exécuter le programme et déterminer son profil opérationnel. Le problème avec ces approches est, à notre avis, qu'elles partent de l'hypothèse que de tels ensembles existent et qu'ils sont complets. Dans le cas où ces hypothèses sont vérifiées, leur utilisation est pertinente. Par contre, dans le cas de nombreux systèmes industriels actuels, ces ensembles de test n'existent pas ou sont incomplets. Les développer, surtout dans le cas d'applications industrielles complexes, peut être très coûteux en temps, ressources et efforts. Ces hypothèses peuvent se transformer dans plusieurs cas en des contraintes sérieuses. En effet, de nombreux systèmes industriels actuels souffrent à la base d'un manque cruel d'information sur leur historique d'évolution. Nous devons donc développer des approches alternatives moins coûteuses.

Par ailleurs, les techniques d'analyse de l'impact basées sur le « program slicing » évaluent l'impact d'un changement une fois implémenté. Nous introduisons, à la fin de ce mémoire, un nouveau modèle d'analyse qui s'inspire du « slicing » et du CCG pour déterminer un ensemble de résultats; une liste de méthodes et des regroupements d'instructions pouvant être affectées lors d'un changement. Cette nouvelle technique, résultat d'une unification des deux premières, réduit l'information retournée par une analyse de « slice » et permet de prédire l'impact d'un changement, tout comme le CCG sans implémenter préalablement le changement.

Nous avons développé un outil qui supporte nos approches selon un processus itératif. Les autres techniques considérées ont également été implémentées. Il permet d'identifier, en fonction des choix des programmeurs, les ensembles de méthodes qui sont susceptibles d'être affectées directement suite à un changement. Ces méthodes doivent être re-testées. Les ensembles indirects (effet de propagation) ne sont considérés que si les méthodes correspondantes, déterminées directement, ont été modifiées. Nous

avons, dans un second temps, conduit une étude expérimentale comparative entre les quatre approches d'analyse statique de l'impact des changements. Cette étude a porté sur plusieurs générations d'un grand projet (open-source) Java. Les changements observés ont été collectés sur les différentes versions du projet considéré. Les ensembles de méthodes potentiellement affectées, suite à un changement, retournés par les techniques ont été comparés aux changements réels observés. Ces techniques ont également été comparées sur la base de critères de performance. Les résultats de l'étude sont reportés et discutés dans ce mémoire.

CHAPITRE 2

EFFET D'IMPACT PAR DÉPENDANCE MODULAIRE

Le concept orienté-objet (OO) est aujourd'hui bien connu des développeurs. Il est largement utilisé par de nombreuses entreprises de développement logiciel. C'est par l'interaction des objets qu'il est possible pour une application OO d'effectuer un certain traitement. Le paradigme objet se base sur une multitude de concepts tels que l'héritage, la modularité, le polymorphisme et l'encapsulation. La complexité de ces techniques rend les analyses d'impact très difficiles à effectuer pour de larges systèmes [Black 05]. Par ailleurs, le changement dans un logiciel est une opération essentielle et continue. Il peut correspondre à l'implémentation de nouveaux besoins. Il peut, aussi, correspondre à la modification des besoins déjà implémentés ou bien à l'adaptation du système existant à un nouvel environnement. Ces modifications peuvent avoir un impact qui se propage de module en module à travers la totalité d'un système. Black [Black 01] ressort deux types de propagations d'information dans les différents éléments OO.

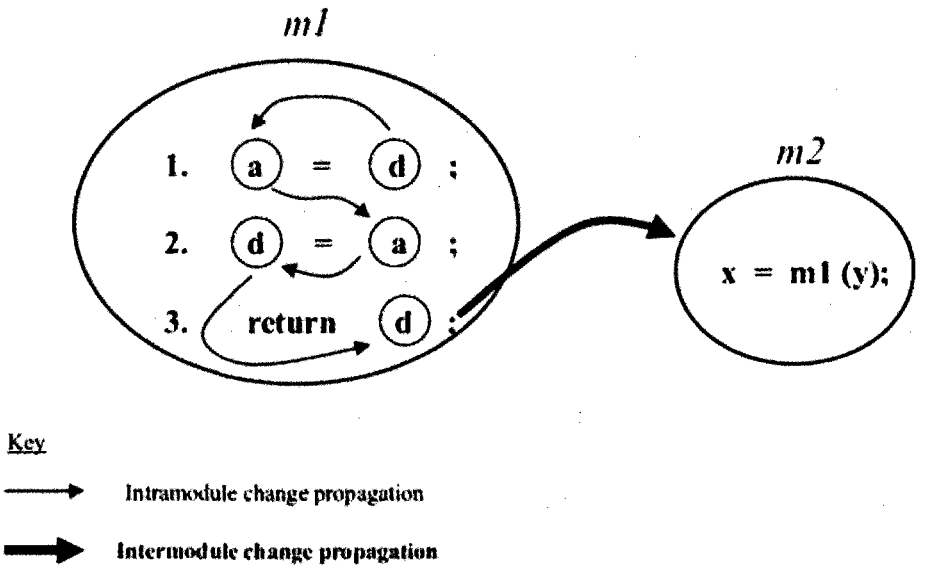


Figure 1. Propagation de changement intra et inter modulaire [Black 05].

Propagation Intra-modulaire : L'information lors de l'exécution d'une application est véhiculée par l'utilisation de variables. Ces variables sont utilisées lors de calculs pouvant affecter la valeur de plus amples données. Dans la *Figure 1*, la variable *D* affecte la variable *A,* car la valeur de *A* est dépendante de la valeur de *D.* La propagation des données, entre variables d'un même module, crée des interdépendances entre les éléments internes du module et peut affecter son comportement.

Propagation Inter-modulaires : Le résultat d'un module, appelé fonction en Java, est transmis par un mécanisme de message appelé invocation. Le concept d'invocation en OO est essentiel, car il permet de lier les différents composants d'un système. Il propage les données de variables entre modules. Cette propagation complexe entre les modules favorise leurs interdépendances. Dans la *Figure 1,* le module *M2* utilise par invocation le module *M1*. *M2* transmet une valeur à *M1*, pouvant l'affecter d'une donnée corrompue en entrée de module. Le module *M1* fait certains traitements pouvant affecter *M2* par un retour de données et en sortie de paramètres (largement utilisé dans les langages C et C++).

Les relations intra et inter modulaires des éléments orientés objet créent des dépendances entre les modules d'un système. Le changement d'un ensemble de ces éléments affecte les liens d'utilisation des données et propage directement et indirectement le nouveau calcul à travers les modules du système. C'est en analysant les liens de dépendances, des variables et invocations d'un système, qu'il est possible de connaître les effets d'une modification.

CHAPITRE 3

ÉTAT DE L'ART : UN BREF APERÇU

Plusieurs travaux reliés à l'analyse de l'impact ont été publiés dans la littérature [Briand 99, Bohner 96, Law 03, Lee 00, Li 95, Li 96, Ren 04, Ryder 01, Yau 80]. L'une des premières techniques d'analyse d'impact, présentée par Bohner et Arnold dans [Bohner 96], était basée sur les graphes d'appels. Dans cette approche, lorsqu'une procédure P est modifiée, toutes les procédures et fonctions appelées par P (directement et indirectement) étaient considérées comme potentiellement affectées et devaient être re-testées. L'analyse d'impact, dans le cadre de telles approches, pouvait souvent conduire à des résultats imprécis comme mentionné dans [Law 03, Ren 04]. Briand et al. [Briand 99] ont proposé un modèle basé sur les principales métriques de couplage. Les résultats obtenus ont montré que les conséquences d'un changement n'étaient pas totalement capturées par les métriques considérées. La principale raison avancée était que les métriques considérées ne capturaient pas toutes les dépendances.

Lee et al. [Lee 00] ont proposé une technique statique d'analyse d'impact pour les systèmes orientés-objet. La technique consiste à construire des graphes de flot de contrôle (niveau instruction) et des graphes de flot de données. Ces graphes sont ensuite transformés en un graphe de dépendances données orienté-objet. Ce graphe est utilisé pour calculer la fermeture transitive de l'impact d'un changement. Horwitz et al. [Horwitz 04] ont utilisé un graphe de dépendances système. Dans [Law 03], Law et al. ont proposé une technique dynamique d'analyse de l'impact (PathImpact) basée sur les chemins d'exécution (whole path profiling). Cette technique requiert un code instrumenté pour capturer les traces d'exécution et un algorithme de compactage pour réduire (compacter) ces traces (obtention de chemins d'exécution réduits). Comparée aux autres techniques, telles que la fermeture transitive des graphes d'appels et le slicing

statique, cette technique donne des résultats plus précis. Une étude expérimentale comparative [Orso 04] entre les méthodes dynamiques d'analyse de l'impact, CoverageImpact [Orso 03] et PathImpact [Law 03], a démontré que la technique proposée par Law et al. [Law 03] retourne dans plusieurs cas des ensembles d'impacts plus précis que ceux retournés par la technique proposée par Orso et al. [Orso 03]. L'étude a, cependant, révélé qu'elle utilise plus d'espace pour sauvegarder les données (7 à 30 fois plus). Cette étude a aussi révélé que la différence relative dans la précision entre les deux techniques varie considérablement d'une version à une autre du programme considéré et dépend étroitement des endroits où les changements ont été effectués.

Weiser introduit dans [Weiser 79] une technique de slicing basée sur les graphes d'appels. Deux ensembles de données sont définis : variables définies et variables référencées, pour chaque nœud (instruction) du graphe [Danicic 99]. L'algorithme de Weiser a été adapté par plusieurs auteurs [Horwitz 04, Jiang 91, Ottenstein 84]. Bishop [Bishop 04] a focalisé sur l'amélioration des performances (temps) de l'analyse d'impact utilisant des techniques traditionnelles de slicing. Le modèle de dépendance utilisé (Program Dependency Model) est mis à jour de façon incrémentale. Le temps d'analyse est réduit en mettant à jour uniquement les modèles de dépendances des méthodes qui ont été affectées. Wang et al. [Wang 96] ont discuté le rôle du « program slicing » (PS) dans le processus REA. Ils mentionnent que le « backward slicing » est utile pour connaître la validité syntaxique de certaines variables et le « forward slicing » (FS) est utilisé pour connaître l'impact sémantique d'un changement.

Les approches statiques basées sur les graphes d'appels traditionnels sont imprécises tel que mentionné précédemment. Celles qui sont basées sur le slicing statique évaluent les effets d'un changement une fois qu'il est implémenté. Les approches dynamiques sont relativement plus précises, mais néanmoins plus coûteuses. Elles requièrent une multitude de tests complets et des données fiables sur le profil opérationnel du

programme analysé. Cependant, l'existence de telles batteries de tests ou de telles données est problématique. Plusieurs projets d'envergure souffrent du manque cruel de données les concernant. La documentation de base de ces projets est souvent désuète. Pour que ce type d'approches puisse être utilisé efficacement, elles exigent des pré-requis importants qui sont malheureusement souvent absents ou largement contraignants dans la plupart des grosses organisations. Les développer dans ce but engendrerait de plus amples coûts. Elles sont donc, à notre avis, de part leurs exigences, réduites à des cas limités.

Les approches statiques basées sur les graphes de contrôles réduits aux appels sont apparues comme pouvant offrir un bon compromis entre les approches statiques traditionnelles (graphes d'appels et slicing) et les approches dynamiques. Elles permettent de repousser les limites des approches basées sur les graphes d'appels car elles sont plus précises [Badri 05]. Elles indiquent dans quelles conditions les appels sont effectués, comment les appels dans une exécution donnée peuvent s'enchaîner, s'ils sont séquentiels, exclusifs, etc. L'intégration du contrôle aux appels permet de se renseigner sur la dynamique du programme. Par ailleurs, comparées aux techniques basées sur le slicing traditionnel, notre approche (telle que implémentée – outil de support) permet de prédire les effets d'un changement avant qu'il ne soit implémenté. Cette possibilité permet aux développeurs d'évaluer, en termes d'efforts, plusieurs solutions possibles pour un changement donné et de retenir celle qui est la moins coûteuse.

CHAPITRE 4

LES GRAPHES DE CONTROLE REDUITS AUX APPELS ET L'ANALYSE DE L'IMPACT

4.1 Graphes d'appels

Citées comme des approches fondamentales pour la prédiction de l'impact d'un changement, les techniques basées sur les graphes d'appels ont été largement utilisées [Bohner 96]. Plusieurs travaux ont démontré qu'elles peuvent être fortement imprécises. Nous donnons, dans ce qui suit, deux définitions de base relatives respectivement à un graphe orienté et à un graphe d'appels.

Définition 1 : *Un graphe orienté G = (S, A) est composé d'un ensemble fini d'éléments appelés nœuds et d'un ensemble de paires $A \subseteq S \times S$ d'éléments appelés arcs. Un arc (x, y) représente un lien orienté entre l'origine x et la destination y.*

Définition 2 : *Un graphe d'appels est un graphe orienté Ga = (M, R). Il est composé d'un ensemble fini M d'éléments appelés nœuds correspondants à des méthodes M = {M1, M2 ...} et un ensemble $R \subseteq M \times M$ d'éléments appelés arcs. Un arc (Mi, Mj) représente un appel de la méthode Mi vers la méthode Mj.*

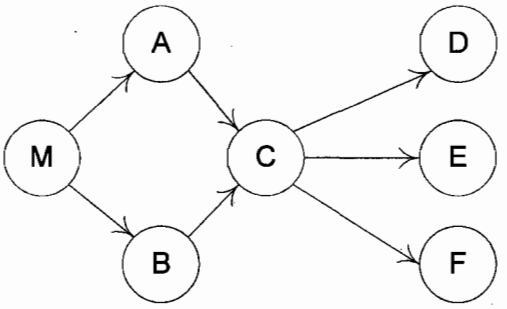


Figure 2. Exemple d'un graphe d'appels.

Considérons le simple exemple donné par la *Figure 2* (pris de [Law 03]). Nous ne pouvons déterminer, à partir du graphe, quelles sont les conditions à la base de la propagation de l'impact d'un changement à la méthode M sur les autres méthodes. Une autre limite des graphes d'appels est qu'ils ne peuvent pas capturer la propagation d'un impact résultant d'un retour à une méthode, comme mentionné dans [Law 03]. Supposons que nous effectuons une modification sur la méthode E de l'exemple (*Figure 2)*. Les effets de ce changement peuvent se propager par retour à C, B, A et M. De telles informations ne peuvent être déduites des graphes d'appels. Nous ne pouvons déduire, en effet, si après le retour à C, l'impact est propagé à A, B, dans les deux cas ou aucun. Cette faiblesse a également été introduite et discutée dans [Ren 04]. Les graphes d'appels capturent, en fait, la structure locale des appels potentiels. Ils ignorent cependant les autres aspects reliés à leur contrôle. Le comportement des appels est beaucoup plus complexe dans la réalité que ne le présente les graphes d'appels.

4.1.1 Algorithme transitif implémenté

Les graphes d'appels directs (CGd) et indirects (CGi) sont représentés à partir de matrices. C'est à partir de ces matrices qu'il est possible de calculer la transitivité des appels.

Tableau 1. Matrice adjacences (1.1) et matrice de chemins (1.2)

	A	B	C	D	E	F	M
A			1				
B			1				
C				1	1	1	
D							
E							
F							
M	1	1					

(1.1)

	A	B	C	D	E	F	M
A			*1*	**1**	**1**	**1**	
B			*1*	**1**	**1**	**1**	
C				*1*	*1*	*1*	
D							
E							
F							
M	*1*	*1*	**1**	**1**	**1**	**1**	

(1.2)

Le *tableau 1.1* est une matrice où chaque rangée et colonne représentent une méthode du système. Les chiffres binaires (1 et 0) représentent, entre les éléments, l'existence de liens d'appels. En reprenant la *Figure 2*, nous avons construit la matrice adjacences (1.1). Cette matrice décrit les liens d'appels directs entre méthodes. Par exemple, (A,C) = 1 car la méthode A appelle la méthode C. La rangée d'une méthode X renseigne sur les méthodes que X invoque. Au contraire, la colonne de la méthode X renseigne sur les méthodes qui invoquent X.

La transitivité des appels (*tableau 1.2*) se calcule à partir de la matrice adjacences. La matrice résultante, nommée la matrice de chemin (*path matrix)*, permet de connaître le cheminement direct et indirect des appels. Par exemple, en regardant la rangée de la méthode A, on ressort les méthodes appelées directement et indirectement par A soit : C, D, E, F. Le code source Java pour calculer la matrice de chemins est donnée en Annexe 1.

4.2 Graphes de contrôle réduits aux appels

Les graphes de contrôle réduits aux appels (CCG) permettent de synthétiser le comportement du programme [Badri 96, Badri 99]. Ils précisent les enchaînements lors de l'exécution des appels. Nous donnons dans ce qui suit, les définitions relatives respectivement à un graphe de contrôle et à un graphe de contrôle réduit aux appels.

Définition 3 **:** *Un graphe de flot de contrôle est un graphe orienté. Les nœuds de ce graphe représentent soit des points de décision (« if-then-else, while, case, etc. »), une instruction ou un bloc séquentiel d'instructions. Un bloc séquentiel d'instructions est une séquence d'instructions telles que si nous exécutons la première instruction, nous sommes sûrs d'exécuter les autres, et toujours dans le même sens. Un arc orienté lie un nœud N_i à un nœud N_j s'il est possible d'exécuter l'instruction correspondante à N_j*

immédiatement après celle associée au nœud N_i. Les arcs du graphe indiquent le transfert de contrôle d'un nœud à l'autre.

Définition 4 **:** *Un graphe de contrôle réduit aux appels est un graphe de flot de contrôle dans lequel les nœuds, représentant les instructions ne conduisant pas à des appels, sont éliminés.*

4.2.1 Concept

Considérons la portion de programme donnée par la *Figure 3.1*. Les Si représentent des séquences d'instructions ne contenant pas d'appels de méthodes. Le graphe d'appels correspondant est donné par la *Figure 3.2*. Le graphe de contrôle réduit aux appels correspondant est donné par la *Figure 3.3*.

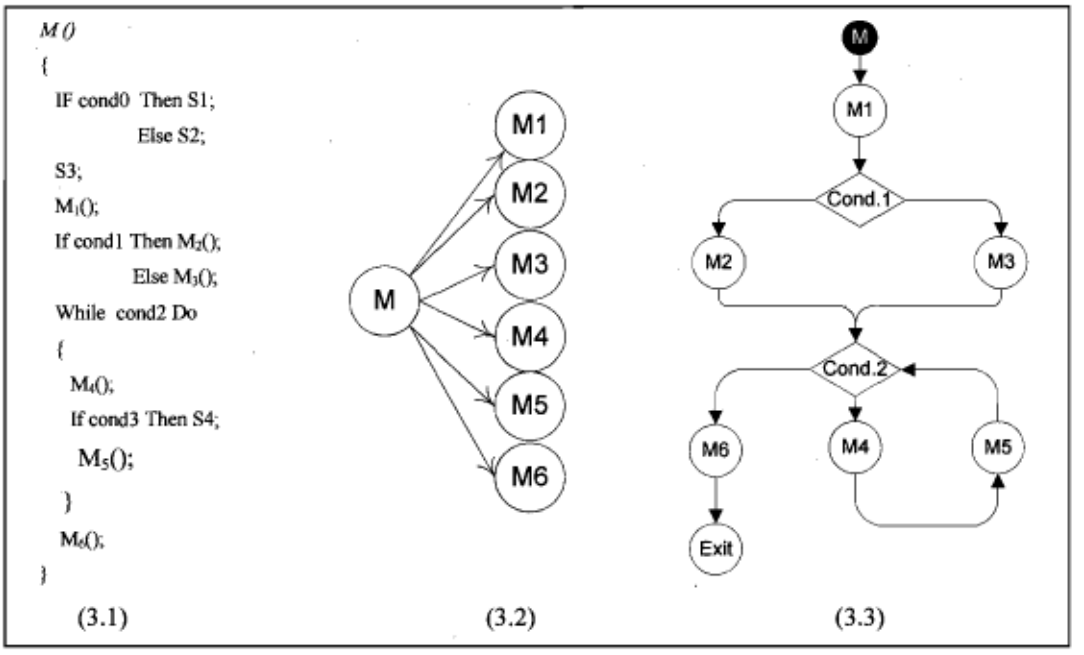


Figure 3. Code source d'une méthode, graphe d'appels et graphes de contrôle réduit aux appels correspondants [Badri 05].

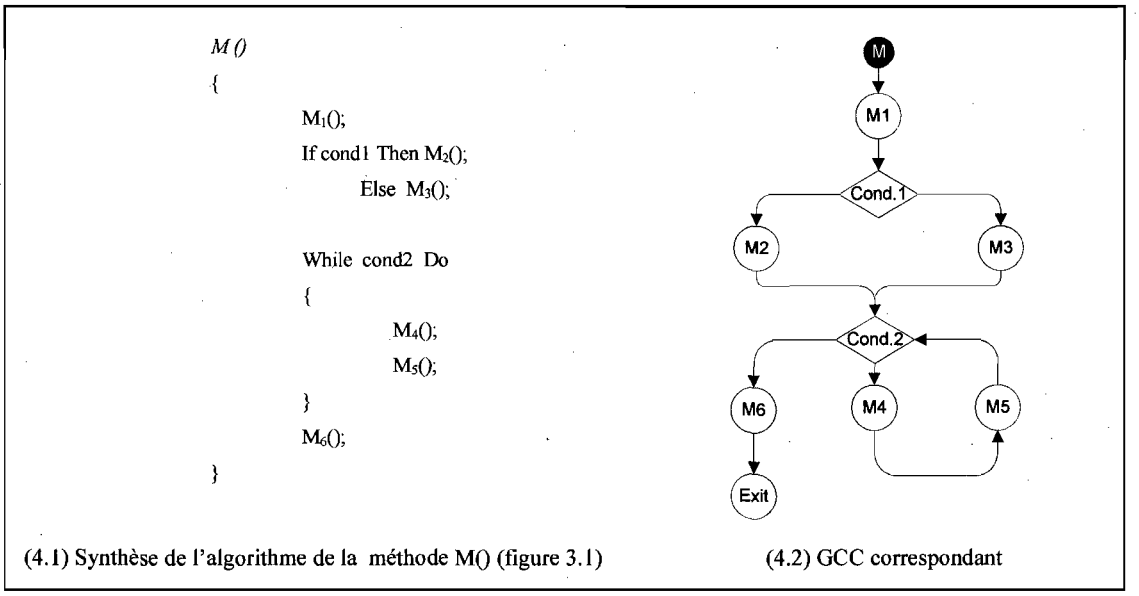
Dans les graphes d'appels, la notion d'enchaînement dans l'exécution des appels est complètement absente. En effet, le graphe d'appels indique uniquement la liste des méthodes appelées par une méthode donnée (dans le cas de l'exemple, M appelle M1, M2, M3, etc). Contrairement aux graphes de contrôle réduits aux appels, dans un graphe d'appels, nous ne pouvons savoir, pour deux méthodes données appelées par une méthode M, quelle est la méthode qui est appelée en premier. Nous ne pouvons savoir, non plus, si les deux méthodes sont appelées exclusivement ou conditionnellement. La notion d'enchaînement dans l'exécution et celle du contrôle lié aux appels sont très importantes pour l'analyse de l'impact des changements. Elles renseignent le comportement du programme. Ceci explique, en partie, le manque de précision dans les résultats de l'analyse de l'impact basée sur les graphes d'appels. Les graphes de contrôle réduits aux appels permettent de spécifier de façon précise le contexte d'un appel donné (conditionnel, inconditionnel, itératif ou autre) et son lien avec les autres appels (exclusif, avant ou après). Si nous considérons l'exemple, la modification de la méthode M2 n'a aucun effet sur la méthode M3 (elles sont exclusives). Par ailleurs, grâce au contrôle, nous pouvons déterminer l'ordre des appels des méthodes, celles qui précèdent la méthode M2 par exemple et celles qui la suivent dans l'exécution. Les méthodes M1 et M6 s'exécutent systématiquement lors de l'exécution de la méthode M(). Par contre, l'exécution des méthodes M2, M3, M4 et M5 est conditionnelle aux conditions 1 et 2.

4.2.2 Analyse prédictive de l'impact

La technique proposée est organisée en plusieurs étapes [Badri 05]: (1) Analyse du code source du programme et génération des graphes de contrôle réduits aux appels des méthodes, (2) Construction des chemins de contrôle réduits aux appels et compactés, (3) Analyse de l'impact basée sur les chemins réduits et compactés. Nous illustrons sommairement, dans ce qui suit, les principales étapes de l'approche par un exemple simple.

Étape 1. Analyse du code source et génération des graphes de contrôle réduits aux appels :

L'objectif de cette étape consiste à effectuer une analyse statique du code source en vue de construire une synthèse des algorithmes des différentes méthodes. Ces algorithmes permettent la construction des graphes de contrôle réduits aux appels. Ces graphes fournissent un aperçu global du contrôle. La *Figure 4.1* donne la synthèse de l'algorithme de la méthode M() définie par la *Figure 3.1*. Les instructions ne conduisant pas à des appels ont été éliminées. La *Figure 4.2* donne le graphe de contrôle correspondant. L'outil développé permet de visualiser graphiquement les graphes de contrôle générés.



(4.1) Synthèse de l'algorithme de la méthode M() (figure 3.1) (4.2) GCC correspondant

Figure 4. Construction des graphes de contrôle réduit aux appels.

Étape 2. Génération des chemins de contrôle compactés :

Le but de cette étape est d'analyser les graphes de contrôle obtenus lors de l'étape précédente en vue de générer les chemins de contrôle compactés. A partir de ces chemins, nous pouvons générer l'ensemble des chemins de contrôle réduits aux appels, en éliminant les chemins théoriques infaisables. Ces chemins compactés permettent d'être renseigné sur le comportement dynamique du programme.

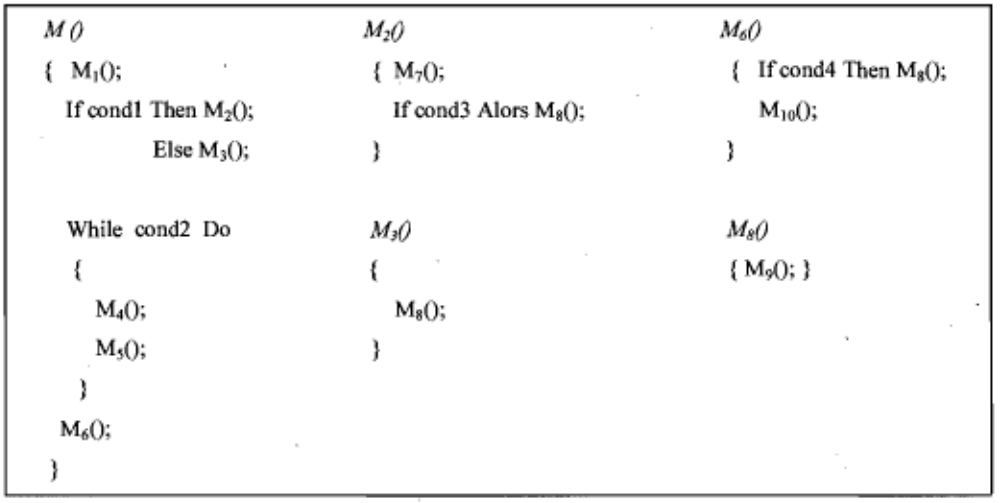


Figure 5. Construction des graphes de contrôle réduits aux appels.

La *Figure 5* donne la synthèse (contrôle réduit aux appels) de plusieurs méthodes que nous considérons pour illustrer notre approche.

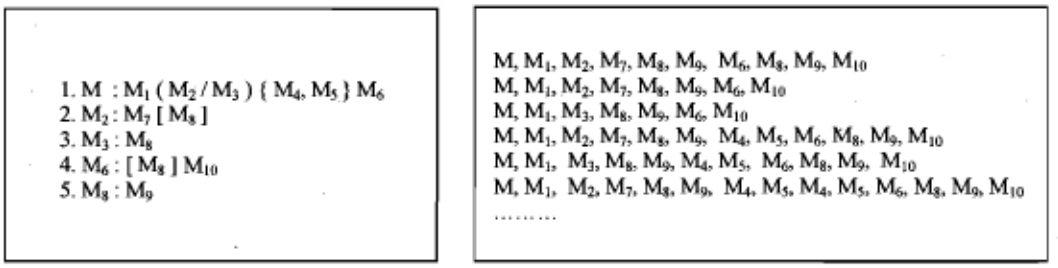


Figure 6. *Chemins d'appels compactés (gauche 6.1) et possibles (droite 6.2)*

La *Figure 6.1* présente les chemins de contrôle compactés correspondants aux méthodes de la *Figure 5*. Nous utilisons plusieurs notations pour exprimer le contrôle dans les séquences d'appels. La notation {séquence} exprime l'itération dans l'exécution des séquences (ou parties de séquences). La séquence entre { } peut être exécutée 0 ou plusieurs fois. La séquence (séquence 1 / séquence 2) exprime l'alternative dans l'exécution des deux séquences. La séquence [séquence] exprime le fait que la séquence en question peut être exécutée comme elle peut ne pas l'être. La *Figure 6.2* illustre une partie des chemins possibles qui peuvent être déduits de la *Figure 6.1*. Les chemins de contrôle compactés sont automatiquement générés par analyse du code. Ceci représente un avantage important relativement aux approches dynamiques qui tentent de les obtenir par l'instrumentation du code et compactage des traces d'exécution.

Étape 3. Identification de l'impact d'un changement:

Notre technique considère, si nous envisageons d'effectuer un changement sur une méthode M, uniquement l'impact qui peut se propager à travers tout chemin de contrôle incluant la méthode M. Ce principe a déjà été appliqué dans [Law 03]. Toute méthode appelée après M, appelée par M, et toute méthode appelant M est incluse dans l'ensemble des méthodes potentiellement impactées.

Par ailleurs, il a été démontré que l'ensemble des méthodes potentiellement affectées par un changement dépend fortement de l'endroit du changement. D'où l'importance de la prise en compte de cet aspect dans la prédiction des effets d'un changement. Il permet de réduire la taille des ensembles retournés suite à un changement. L'outil développé offre à l'utilisateur la possibilité de choisir une section du code qu'il prévoit changer et retourne l'ensemble des méthodes qui risquent d'être affectées. L'utilisateur peut donc, par simulations successives, évaluer l'impact de plusieurs solutions et choisir celle qui présente le moindre coût. Cette possibilité peut apporter une aide importante à un développeur.

Soit MP_{Mi} l'ensemble des méthodes M_j qui appellent directement la méthode M_i. Une méthode M_j appelle directement une méthode M_i lorsque l'appel de la méthode M_i figure directement dans le corps de la méthode M_j. L'ensemble MP_{Mi} est construit par analyse des séquences compactées. Soit M_{Mi} l'ensemble des méthodes M_k appelées directement par la méthode M_i et se trouvant après l'endroit où la modification a lieu (dans la méthode M_i). Une méthode M_k est appelée directement par la méthode M_i si son appel figure directement dans le corps de la méthode M_i. Prenons pour exemple la séquence suivante de la méthode Mi : M_k [M_l] M_m : si la modification a lieu après l'appel de la méthode M_k, seules les méthodes M_l et M_m seront considérées (suite de la séquence). Soit MS_{Mi} l'ensemble des méthodes qui sont appelées après exécution de M_i. Pour construire cet ensemble, nous considérons l'ensemble des séquences compactées dans lesquelles apparaît la méthode M_i. L'ensemble I_{Mi} des méthodes impactées suite à une éventuelle modification d'une méthode M_i correspond à: $I_{Mi} = MP_{Mi} \cup M_{Mi} \cup MS_{Mi}$

Considérons maintenant l'ensemble des séquences données à la *Figure 6.1*. Supposons que l'on prévoie modifier la méthode M_6. La séquence de la méthode M_6 est : [M_8] M_{10}. L'outil développé permet de visualiser la séquence de la méthode à modifier et offre à l'utilisateur la possibilité de marquer l'endroit du changement. Dans le cas de l'exemple de la méthode M_6, plusieurs cas sont possibles.

Cas 1: ***Le changement a lieu avant l'appel de la méthode M_8.***

MP_{M6} = { M }, si on se réfère aux séquences de la *Figure 6.1*, la méthode M_6 est appelée uniquement par la méthode M.

M_{M6} = { M_8, M_{10} }, en se référant à la séquence de la méthode M_6, les méthodes appelées par la méthode M_6 après l'endroit où a lieu le changement sont M_8, M_{10}.

$MS_{Mi} = \Phi$, cet ensemble correspond à l'ensemble des méthodes qui s'exécutent après M_6. Si on se réfère à la *Figure 6.1*, la méthode M_6 apparaît uniquement au niveau de la séquence 1 et aucune méthode ne s'exécute après elle.

L'impact dans le cas de ce changement sera alors: $\{M, M_8, M_{10}\}$.

Cas 2: ***Le changement a lieu après l'appel de la méthode M_8.***

$MP_{Mi} = \{ M \}$, $M_{Mi} = \{ M_{10} \}$, $MS_{Mi} = \Phi$, l'impact dans le cas de ce changement sera alors: $I_{Mi} = \{ M, M_{10} \}$.

Cas 3: ***Le changement a lieu après l'appel de la méthode M_{10}.***

$MP_{Mi} = \{ M \}$, $M_{Mi} = \Phi$, $MS_{Mi} = \Phi$, l'impact dans le cas de ce changement sera alors : $I_{Mi} = \{ M \}$.

L'ensemble I_{Mi} est différent dans les trois cas. Dans le premier cas, il comporte trois méthodes, dans le second deux méthodes et dans le dernier une seule méthode. Ceci reflète l'importance de préciser l'endroit exact du changement. Ceci permet aussi de réduire la taille des ensembles des méthodes susceptibles d'être affectées suite à un changement donné. Grâce aux séquences compactées, l'outil développé offre à l'utilisateur la possibilité de préciser, dans une méthode pour laquelle il prévoit un changement, l'endroit exact de ce changement. Il peut simuler plusieurs changements possibles pour une modification donnée et évaluer comparativement leurs impacts respectifs. Cet aspect peut être d'une grande aide pour les mainteneurs lorsqu'ils sont en face de plusieurs solutions possibles.

4.2.3 Analyse d'impact incrémentale

Dans le cadre de notre approche, nous nous sommes intéressés uniquement à l'ensemble des méthodes susceptibles d'être affectées directement. Nous construisons, en fait, l'ensemble des méthodes susceptibles d'être affectées par un changement donné de façon incrémentale. Cette façon de faire permet d'éviter l'obtention éventuellement d'ensembles trop larges comme c'est le cas auprès de certaines approches. Lors du processus de changement, seules les méthodes appartenant à l'ensemble I_{Mi} seront prises en considération. Si, suite à la modification de la méthode M_i, une méthode $M_j \subseteq I_{Mi}$ subit également des modifications après sa vérification, nous répétons le processus qui correspond à construire l'ensemble des méthodes impactées I_{Mj} par la méthode M_j. Nous répétons, de façon itérative, le processus jusqu'à ce qu'aucune méthode parmi les méthodes affectées par une méthode donnée ne soit changée. Par ailleurs, lors du processus de test de régression, seules les méthodes appartenant à l'ensemble I_{Mi} seront prises en considération (en premier). Si, suite à la modification de la méthode M_i une méthode $M_k \subseteq I_{Mi}$ présente quelques problèmes, nous répétons le processus qui correspond à construire l'ensemble des méthodes impactées I_{Mk} par la méthode M_k. Cette démarche nous permet d'orienter de façon itérative le processus des tests de régression.

4.3 Étude expérimentale

4.3.1 Introduction

Dans cette section, une évaluation expérimentale comparative entre l'approche CCG que nous proposons et celle basée sur les CG. Elle constitue la première investigation empirique de notre approche. Cette expérimentation a été effectuée grâce à un environnement, composé de plusieurs outils, que nous avons développés.

L'environnement supporte les différentes phases de notre approche. Nous avons également implémenté l'approche basée sur les graphes d'appels. L'approche basée sur les graphes d'appels a été implémentée de deux façons: la première (CGi), correspondant aux approches traditionnelles basées sur les graphes d'appels, en considérant les appels directs et indirects entre méthodes et la deuxième (CGd), en considérant uniquement les appels directs entre méthodes (comme c'est le cas pour notre approche). Dans le premier cas, l'ensemble des méthodes appelées par une méthode donnée représente l'ensemble des méthodes qu'elle appelle directement et indirectement. Dans le deuxième cas, l'ensemble des méthodes appelées par une méthode donnée représente l'ensemble des méthodes qu'elle appelle directement.

Nous avons choisi deux projets Java 1.4 «open-source» disponibles sur sourceforge.net : JFTP et OpenWfe. JFTP est une application graphique supportant les protocoles SMB, SFTP, NFS, HTTP et permettant des transferts de fichiers. Elle comporte au total 57 classes et 443 méthodes. OpenWfe est un engin permettant d'effectuer des «workflot». Il comporte une suite complète d'outils pour la gestion des processus industriels. Il est composé au total de 77 classes et de 465 méthodes. Ces deux applications ont subi plusieurs changements dans le temps. Par analyse de leurs versions successives, nous avons identifié les modifications effectuées dans leur code par comparaison binaire de deux versions différentes pour chacune des deux applications. Nous avons sélectionné 21 méthodes (dans les deux projets) ayant subi des modifications (ajouts et/ou suppressions). Pour chacune des 21 méthodes, nous avons analysé, en considérant la dernière version de leur code, l'impact du changement selon les approches GAd (Graphe d'Appels - direct), CGi (Graphes d'Appels – indirect) et CCG. Les résultats obtenus pour les projets JFTP et OpenWfe sont donnés respectivement dans *les tableaux 2 et 3*.

Tableau 2. Résultats pour le projet JFTP.

JFTP Version: 1.10; #Class: 57; #Methods: 443									
No.	**CGi**	**CGd**	**CCG**	**CCG - CGi**	**CGi - CCG**	**CCG - CGd**	**CGd - CCG**	**CCG / CGi**	**CCG / CGd**
1	70	8	8	0	62	0	0	0,11	1,00
2	18	8	14	6	10	6	0	0,78	1,75
3	54	3	4	1	51	1	0	0,07	1,33
4	3	3	3	0	0	0	0	1,00	1,00
5	44	10	18	8	34	8	0	0,41	1,80
6	2	2	2	0	0	0	0	1,00	1,00
7	70	5	4	1	67	1	2	0,06	0,80
8	88	10	10	0	78	0	0	0,11	1,00
9	11	8	8	0	3	0	0	0,73	1,00
10	9	4	4	0	5	0	0	0,44	1,00
11	17	10	10	0	7	0	0	0,59	1,00
12	108	21	5	0	103	0	16	0,05	0,24
13	70	5	5	0	65	0	0	0,07	1,00
14	19	7	5	0	14	0	2	0,26	0,71
15	7	4	4	0	3	0	0	0,57	1,00
16	6	3	7	4	3	4	0	1,17	2,33
17	30	18	15	6	21	6	9	0,50	0,83
18	56	9	6	1	51	1	4	0,11	0,67
19	52	12	14	2	40	3	1	0,27	1,17
20	43	10	12	2	33	2	0	0,28	1,20
21	71	14	11	0	60	0	3	0,15	0,79
Avr.	**40,3**	**8,29**	**8,05**	**1,48**	**33,8**	**1,52**	**1,76**	**0,42**	**1,08**
Sum	**848**	**174**	**169**	**31**	**710**	**32**	**37**		

Tableau 3. Résultats pour le projet OpenWfe.

OpenWfe Version: 1.4.7; #Class: 77; #Methods: 465									
No.	**CGi**	**CGd**	**CCG**	**CCG - CGi**	**CGi - CCG**	**CCG - CGd**	**CGd - CCG**	**CCG / CGi**	**CCG / CGd**
1	12	2	2	0	10	0	0	0,17	1,00
2	8	4	5	1	4	1	0	0,63	1,25
3	4	4	4	0	0	0	0	1,00	1,00
4	5	5	5	0	0	0	0	1,00	1,00
5	5	2	2	0	3	0	0	0,40	1,00
6	5	3	3	0	2	0	0	0,60	1,00
7	14	5	9	4	9	4	0	0,64	1,80
8	2	2	2	0	0	0	0	1,00	1,00
9	14	6	6	0	8	0	0	0,43	1,00
10	3	2	2	0	1	0	0	0,67	1,00
11	8	5	6	1	3	1	0	0,75	1,20
12	4	4	7	3	0	3	0	1,75	1,75
13	2	2	2	0	0	0	0	1,00	1,00
14	19	7	3	0	16	0	4	0,16	0,43
15	9	9	7	1	3	1	3	0,78	0,78
16	2	2	2	0	0	0	0	1,00	1,00
17	7	4	14	9	2	10	0	2,00	3,50
18	2	2	2	0	0	0	0	1,00	1,00
19	3	3	2	0	1	0	1	0,67	0,67
20	10	2	2	0	8	0	0	0,20	1,00
21	13	3	2	0	11	0	1	0,15	0,67
Avr.	**7,19**	**3,71**	**4,24**	**0,90**	**3,86**	**0,95**	**0,43**	**0,76**	**1,14**
Sum.	**151**	**78**	**89**	**19**	**81**	**20**	**9**		

4.3.2 Discussion des résultats

Les *tableaux 2 et 3* présentent, en particulier, le nombre de méthodes impactées lors d'une modification d'une méthode donnée. Dans le cas de l'application JFTP, si nous considérons par exemple la méthode 12, l'approche basée sur les CGd suggère 21

méthodes, alors que la technique que nous proposons suggère uniquement 5 méthodes. Les CGi suggèrent quant à eux 108 méthodes. Les ensembles identifiés, constitués de méthodes susceptibles d'être affectées suite à un changement, sont uniquement constitués de méthodes internes au projet. Les méthodes appartenant à des librairies externes n'ont pas été considérées. En moyenne, l'ensemble que suggèrent les CGi est 5 fois plus grand que celui suggéré par les CCG et 4.5 fois plus grand que celui suggéré par les CGd. Pour les 21 modifications considérées, les CGi indiquent au total 848 méthodes (deux fois le nombre total de méthodes) à vérifier, ce qui est énorme en termes d'efforts et de coût. Pour chaque méthode susceptible d'être modifiée, les CGi précisent en moyenne 34 nouvelles méthodes à vérifier. La taille des ensembles suggérés par les CCG est en moyenne inférieure à la moitié (42%) de celle proposée par les CGi. Les CGd donnent des ensembles de méthodes impactées relativement comparables à ceux donnés par les CCG. L'examen détaillé des résultats et l'analyse du code des applications révèlent cependant des différences importantes. Certains changements dans le cas des CGd proposent des ensembles assez larges. La colonne (CGd – CCG) met en évidence cette différence. Si on considère par exemple le cas de la méthode 12 (*tableau 2*), les CGd retournent un ensemble dont la taille est 21 alors que les CCG retournent un ensemble dont la taille est 5. Après analyse du code, nous avons pu remarquer que notre technique a éliminé plusieurs méthodes qui ne risquent pas d'être impactées en cas de modification de la méthode 12. Cette élimination est due à l'intégration du contrôle dans le modèle que nous proposons. Ce modèle permet de déterminer les différents chemins de contrôle et élimine l'imprécision des CG. Par ailleurs, le fait que l'outil développé offre la possibilité de préciser exactement à quel niveau la modification peut avoir lieu dans une méthode, donne plus de précision et permet de réduire la taille des ensembles obtenus contrairement aux approches basées sur les CG. Nous pouvons remarquer dans *les tableaux 2 et 3*, pour les 21 modifications considérées, que pour l'application JFTP les CGd indiquent 37 méthodes que les CCG ignorent et que pour l'application OpenWfe les CGd indiquent 9 méthodes que les CCG ignorent également.

Par ailleurs, nous avons remarqué que, dans certains cas, la taille des ensembles proposés par les CCG est plus importante que celle des ensembles proposés par les CGd. La colonne CCG-CGd des tableaux reflète bien cette différence. Pour la méthode 17 (*tableau 3*) par exemple, les CCG donnent un ensemble dont la taille est de 14, alors que les CGd donnent un ensemble dont la taille est de 4. Cette différence est liée au fait que les CGi et les CGd ne capturent pas la propagation de l'impact due au retour de la méthode. Autrement dit, toutes les méthodes s'exécutant après la méthode modifiée ne sont pas considérées dans l'ensemble proposé aussi bien par les CGi que par les CGd contrairement au CCG. Pour l'application JFTP nous pouvons remarquer dans les tableaux, pour les 21 modifications considérées, que les CCG donnent 32 méthodes que les CGd ne capturent pas et que pour l'application OpenWfe les CCG donnent 20 méthodes que les CGd ne capturent également pas.

Les résultats obtenus démontrent que la technique basée sur les CCG est plus précise que les techniques basées sur les CG traditionnels. Cette étude expérimentale constitue notre première investigation dans ce domaine. Par ailleurs, nous pensons que notre approche peut constituer un compromis intéressant entre les approches statiques classiques et les approches dynamiques telles que celle proposée dans [Law 03]. Notre approche est en effet comparable à celle proposée dans [Law 03]. Elle présente cependant des différences importantes : (1) notre approche est statique, (2) les chemins d'exécution sont générés par analyse des traces d'exécution alors que dans notre cas ils sont générés par analyse des séquences compactées et (3) l'outil développé offre la possibilité de préciser l'endroit de la modification. Il est clair que ceci nécessite une investigation expérimentale plus poussée avant de tirer des conclusions finales.

CHAPITRE 5

COMPARAISON EXPÉRIMENTALE DES APPROCHES STATIQUES

Ce chapitre présente une étude expérimentale complémentaire à la première sur trois techniques statiques d'analyse de l'impact : technique basée sur les graphes d'appels traditionnels, technique basée sur les graphes de contrôle réduits aux appels et technique basée sur le slicing. La technique basée sur les graphes de contrôle réduits aux appels est une nouvelle technique dont nous avons discuté dans le chapitre antérieur. Elle a fait l'objet d'une première expérimentation et a été publiée dans [Badri 05]. Elle utilise un nouveau modèle basé sur les chemins de contrôle réduits aux appels obtenus par analyse statique du code. Les chemins de contrôle, sous forme compactée, sont utilisés pour supporter l'analyse de l'impact. La première expérimentation s'est limitée, comme mentionné précédemment, à sa comparaison avec les graphes d'appels relativement à la taille des ensembles retournés. Dans le cadre de cette nouvelle expérimentation, nous allons examiner la « qualité » des ensembles retournés et les évaluer à l'aide de métriques. Ce sont les résultats obtenus lors de la première étude et les observations que nous avions faites qui nous ont conduits à affiner notre démarche.

Les deux techniques basées respectivement sur les graphes d'appels traditionnels et le slicing ont été prises de la littérature. Nous avons conduit une étude expérimentale comparative, entre les trois techniques considérées, sur plusieurs générations d'un grand projet réel (open-source) Java. Les changements observés ont été collectés à partir des différentes versions successives du projet (JMOL). Suite à un changement, les ensembles retournés par les trois techniques, des méthodes potentiellement affectées, ont été comparés aux changements réels observés. Les trois techniques ont également été comparées sur la base de certains critères de performance. Les résultats de l'étude sont rapportés et discutés dans ce chapitre. Nous donnons également un aperçu de

l'environnement que nous avons développé pour supporter l'approche proposée ainsi que l'expérimentation que nous avons conduite.

5.1 Slicing statique

Le *Program Slicing* (PS) permet de réduire un programme complexe en un programme plus simple. Effectuer un *slicing* sur un programme permet de supprimer les instructions non nécessaires à un certain point d'exécution, appelé critère de slice. Ceci a pour but de faciliter la compréhension de son fonctionnement et de diminuer la complexité et la grosseur de ses composants [Tip 94]. Weiser introduit dans [Weiser 79] le PS comme une nécessité au débogage et à la compréhension de programmes. Son algorithme, appelé *Static Slicing,* retourne un ensemble d'instructions affectant directement ou indirectement la valeur d'un critère de slice. Il est basé sur une analyse du flot de données dans un graphe de contrôle. L'analyse résultante de son algorithme se nomme *backward slicing* (BS). L'analyse inverse du *backward slicing* se nomme *forward slicing* (FS). Elle se définie comme étant l'ensemble des éléments d'un programme potentiellement affectés par les valeurs des variables du critère de slice [Tip 94, Wang 96, Lucia 01]. Selon [Wang 96], de part sa nature, le *forward slicing* est souvent utilisé pour identifier la propagation d'un changement dans un programme. Quelques auteurs ont utilisé le FS statique dans leur recherche pour comparer les performances, en particulier la grosseur des ensembles retournés des parties potentiellement affectées, dans un contexte d'analyse d'impact de changements [Wang 96, Law 03, Orso 03].

Wang et al [Wang 96] situent le rôle du PS dans un processus REA. Ils ont implémenté un prototype pour l'analyse de programmes COBOL utilisant le *backward* et le *forward slicing.* Le prototype utilise plusieurs notions de PS : *backward, forward, constraint slicing (impact direct)* et *control change.* Suite à plusieurs expérimentations, ils concluent que la propagation indirecte retourne un trop grand ensemble d'éléments (supposés) affectés comparativement à l'approche directe. Par ailleurs, selon [Wang 96],

il serait préférable d'analyser itérativement le résultat direct lors du processus de changement. Malheureusement, les travaux effectués par Wang et al. [Wang 96] n'ont pas exploré la validité (justesse – qualité de la précision) des résultats retournés. Ils n'ont pas précisé si les résultats « directs » sont meilleurs, en termes de qualité de précision, que ceux obtenus d'une manière « indirecte ». La façon d'évaluer la qualité de la précision sera abordée ultérieurement dans la section 5.3.3. Law et al. [Law 03] concluent que le *Static Slicing* peut être coûteux en ressources et peut retourner des résultats imprécis lorsque des comportements dynamiques sont analysés.

```
public class A
{
  private int v = 0;
  public A(int a)
  {
    v = a ;
  }
}
public class B
{
   static A buildA(int a)
   {
     return new A(C.getInt(a));
   }
}
public class C
{
   static int getInt(int a){return a*a;}
}
```

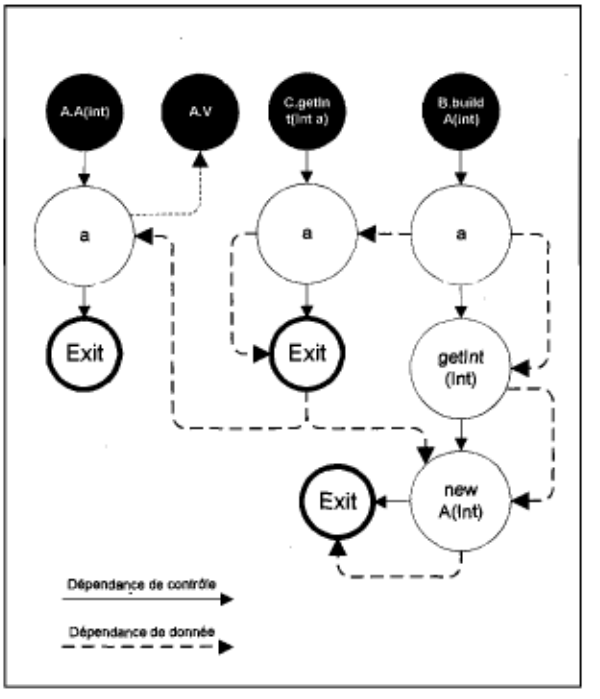


Figure 7. Exemple de modèle de slice utilisé

Malgré la simplicité théorique du PS, l'approche reste complexe et difficile à implémenter [Gallagher 04]. Peu d'outils stables et disponibles supportent l'implémentation du *forward slicing* pour des applications orientées objets. On peu citer *Unravel* pour le langage C [Unravel], *CodeSurfer* pour le langage C++ [Codesurfer] et

Kaveri, du projet Indus, pour le langage Java [Kaveri]. Ce dernier n'a pas pu être utilisé dans nos expérimentations en raison de son instabilité. Une implémentation du *forward slicing* basée sur l'approche formelle de Weiser a été réalisée [Weiser 79]. Nous avons implémenté, en fait, une version plus simple du *static forward slicing,* en omettant le *control change* similaire à la définition du *Data Slicing* de [Zhang 07]. La version du FS que nous avons implémentée se concentre sur l'identification des méthodes comportant un ou plusieurs cheminements de données impactées. À partir d'un graphe de contrôle, nous ajoutons des liens de dépendance de données intra et inter modulaires entre les diverses instructions atomiques (instructions irréductibles) d'une méthode. En parcourant ces liens de cheminement de données, à partir d'un ensemble de critères de slice, il est possible de déterminer les instructions subséquentes pouvant être affectées dans la méthode modifiée et les autres méthodes d'un système (voir *Figure 7*).

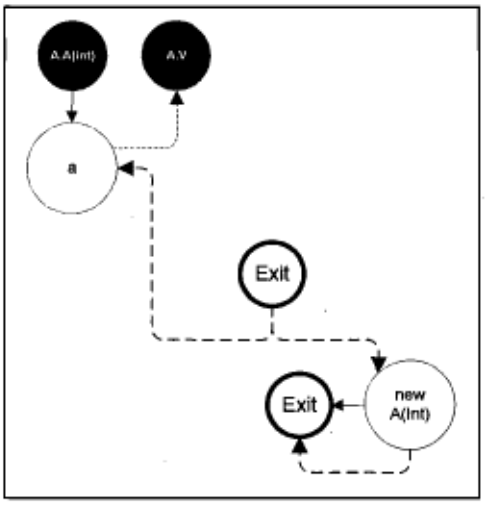


Figure 8. Résultat d'une passe de slice

L'implémentation considère aussi les variables de classes affectées pouvant affecter de plus amples méthodes par le cheminement des données. Cette implémentation permet, ainsi, de ressortir les diverses méthodes et instructions affectées d'une manière directe et indirecte dans la totalité du système analysé.

Ainsi, à partir d'un nœud dans le graphe, il est possible de suivre le cheminement de dépendance entre les éléments. Par exemple, selon l'exemple de la *Figure 7*, si le critère de slice est l'instruction de retour (*return a*a)* de la méthode *static int getInt(int a)* de la classe C, le résultat de la slice sera l'ensemble des nœuds décrit par la *Figure 8,* soit, les méthodes *B.buildA(int a)* et *A.A(int)* ainsi que la variable de classe *A.V*.

5.2 Analyse itérative de l'impact : un outil de support

L'approche proposée est supportée par un outil « *plug-in* », que nous avons développé pour l'environnement de développement Éclipse [Eclipse]. Éclipse est une communauté open source dont les projets sont orientés vers la construction d'une plateforme de développement ouverte et extensible, permettant le développement, le déploiement et la gestion d'un logiciel durant son cycle de vie. Récemment, quelques auteurs ont implémenté leurs outils d'analyse d'impact à partir de l'API d'Éclipse [Ryder 04, Rajlich 05]. Son API permet, en fait, la création d'un *Arbre Syntaxique Abstrait* (AST). Ce dernier est une représentation en structure d'arbre du code d'un programme. L'utilisation des AST d'Éclipse permet de faciliter et de simplifier le développement et l'analyse du code source d'un projet. L'API d'Éclipse peut lier les invocations aux méthodes appelées lors de la création des AST facilitant l'analyse des références entre méthodes. De plus, leur utilisation rend l'outil indépendant de la version du langage Java analysé. L'outil peut aussi bien analyser du code écrit en Java1.2 ou Java1.6 et très probablement les prochaines versions selon les mises à jour de l'API d'Éclipse, ce qui présente de nombreux avantages. La structure de l'outil est construite au dessus des AST d'Éclipse permettant ainsi, de lier l'éditeur d'Éclipse à notre outil et de circonscrire la grande quantité d'informations que possèdent les AST aux structures des CCG (*Figure 9*).

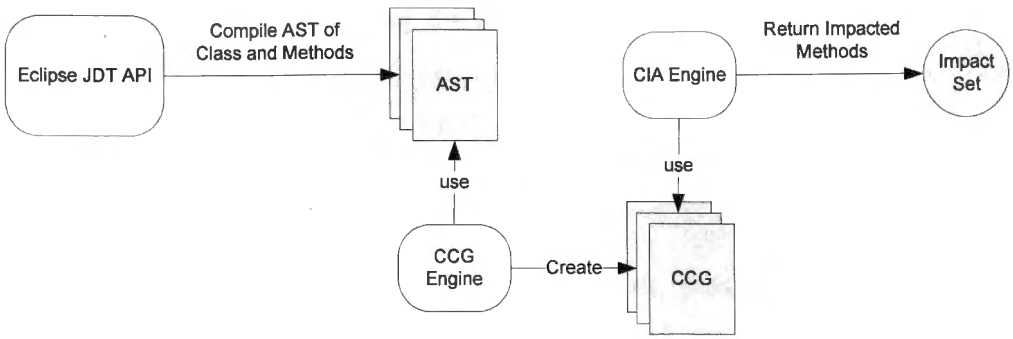


Figure 9. Flot opérationnel de l'outil d'analyse d'impact de changement

Dans un premier temps, l'outil demande à l'environnement d'Éclipse de compiler un ensemble d'AST pour un projet analysé. Par la suite, l'engin CCG se charge de convertir l'information que l'on retrouve dans les AST en structure standard CCG pour chaque méthode du projet. Finalement, l'engin d'analyse utilise l'ensemble des CCG pour capturer les méthodes affectées. L'interaction entre l'outil et l'utilisateur, pour effectuer une analyse, est réduite au minium. Le processus d'analyse s'effectue en 3 étapes :

1. **Localisation du futur changement :** L'outil permet à l'utilisateur d'indiquer l'emplacement des futures modifications. Cette opération se fait en positionnant le curseur dans une méthode à partir de l'éditeur de fichier Java d'Éclipse (*Figure 10*). Cette façon de procéder permet d'évaluer (par simulations successives) plusieurs possibilités éventuelles pour un même changement et garder celle présentant le moindre coût.

```
public static BaseAtomType get(int atomicNumber) {

  Enumeration iter = typePool.elements();
  while (iter.hasMoreElements()) {
    BaseAtomType at = (BaseAtomType) iter.nextElement();
    if (atomicNumber == at.getAtomicNumber()) {
      BaseAtomType atr = get(at.getRoot());
      return atr;
    }
  }
  return null;
}
```

Figure 10. Localisation de l'emplacement d'un changement.

2. **Analyse de la propagation du changement :** L'utilisateur demande à l'outil d'effectuer une analyse d'impact de changement à partir de la position du curseur. L'outil compile les CCG et retourne, selon l'approche décrite dans la section 4.2.2, différentes vues des résultats.

3. **Vue des résultats :** Les résultats sont, en fait, présentés sous deux formes : textuelle et graphique. La vue textuelle se traduit par une liste des méthodes et classes affectées. Elle est illustrée par la *Figure 11*. Elle comporte deux fenêtres : la première précise la classe ou la méthode qui sera modifiée et la seconde, l'ensemble des classes (méthodes) qui risquent d'être impactées. La vue graphique, illustrée par la *Figure 12*, permet à l'utilisateur d'avoir une vue globale de l'étendu de la propagation du changement. La visualisation de la propagation peut se faire, selon le choix de l'utilisateur, sous deux formes. La première montre les éléments du code directement affectés. Tandis que la seconde montre l'ensemble des éléments qui risquent d'être affectés (directement et indirectement).

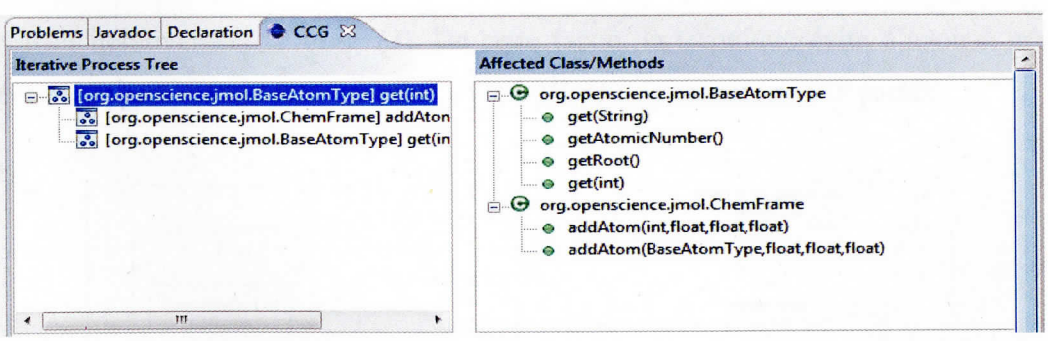


Figure 11. Présentation des résultats sous forme textuelle.

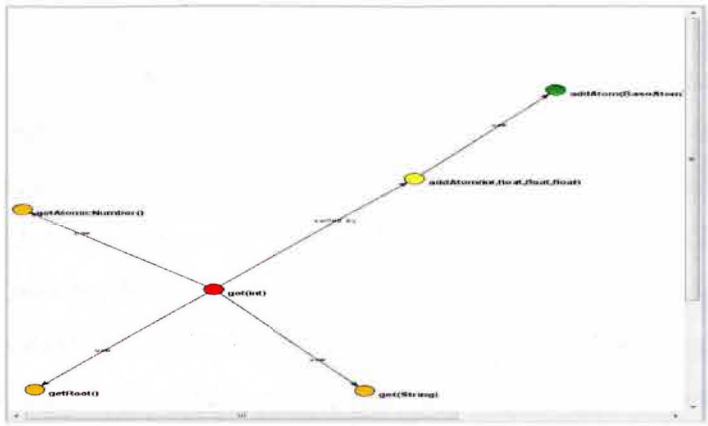


Figure 12. Présentation des résultats sous forme graphique.

L'outil permet d'effectuer des analyses prédictives d'impact itératives. En effet, l'utilisateur peut sélectionner une classe (méthode) dans la liste des classes/méthodes affectées (*Figure 11, fenêtre de droite*) et relancer le processus d'analyse. Il peut, par ailleurs, revenir à l'état précédent en sélectionnant la classe/méthode au niveau de la *fenêtre 1*. Dans le cadre de l'approche proposée, nous considérons, pour des raisons de simplification, uniquement l'ensemble des éléments (classes, méthodes) qui risquent d'être affectés directement suite à un changement. L'ensemble des éléments impactés est construit itérativement (*Figure 13*). De cette façon, la technique évite d'avoir à gérer d'éventuels gros ensembles pouvant contenir plusieurs éléments non impactés.

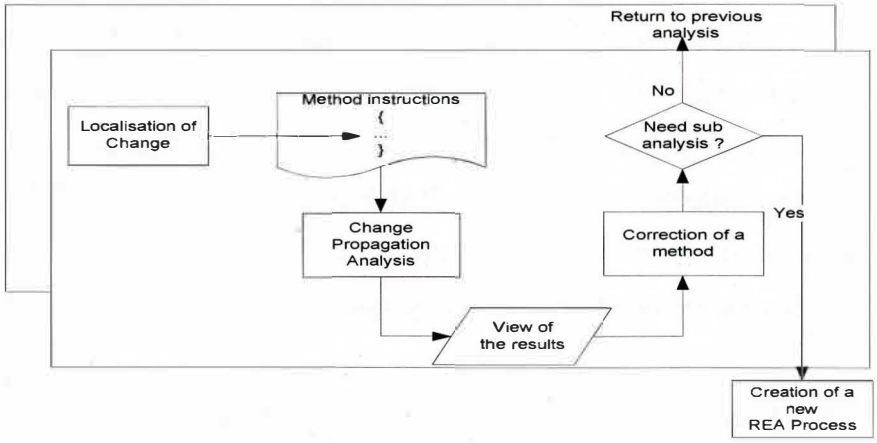


Figure 13. Processus itératif de construction des ensembles impactés.

Dans le processus, seuls les éléments qui risquent d'être affectés directement suite à une modification d'une méthode M_i sont considérés. Supposons que cet ensemble est noté I_{Mi}. Si après la modification de la méthode M_i une autre méthode M_j appartenant à I_{Mi} nécessite une adaptation, nous répétons le processus qui revient à construire l'ensemble I_{Mj} correspondant. Ce processus est répété itérativement jusqu'à ce qu'il n'y ait plus de méthodes impactées par le changement. On évite ainsi, de considérer pour les tests de régression, un ensemble pouvant être relativement grand de méthodes ne nécessitant pas d'être re-testées. Cette démarche permet d'orienter le processus des tests de régression sur les éléments devant être réellement testés à nouveau.

5.3 Étude expérimentale

5.3.1 Environnement

Une expérimentation a été effectuée sur plusieurs générations d'un projet réel *« open source »* Java, appelé *Jmol* [Jmol], disponible sur *SourceForge* [SourceForge]. *Jmol* est une application gratuite de visualisation de molécules pour les étudiants, éducateurs et chercheurs en chimie et biochimie. Le projet représente une bonne étude de cas concrète et a déjà été utilisé dans d'autres travaux, entre autres [Tansatalis 05]. Nous nous sommes intéressés dans le cadre de cette étude en particulier, à déterminer l'exactitude des résultats des approches considérées dans l'identification des bonnes classes impactées après un changement initial dans *Jmol*. Dans l'expérimentation, il s'agissait d'identifier, entres deux révisions majeures de *Jmol,* les changements que les classes ont subis. Après avoir ciblé un point initial de changement identifié après analyse du code et des différents changements opérés d'une version à la suivante, nous avons appliqué les approches choisies et compilé les classes ayant bien été identifiées lors de l'analyse.

Nous avons appliqué l'expérimentation sur les trois approches sélectionnées que nous avons implémentées par des outils pour Éclipse : CG, CCG et FS. L'objectif principal de

l'analyse était de cibler les méthodes affectées d'une version du système, suite à un ensemble de modifications, pour connaître l'impact réel des changements et de les comparer aux résultats des approches. Par ailleurs, les changements observés d'une version à une autre ont été collectés sur les différentes versions du projet. Les ensembles retournés par les trois techniques des méthodes potentiellement affectées ont été comparés aux changements réels observés. Les trois techniques ont été comparées sur la base de plusieurs critères de performance. Les analyses ont été effectuées sur une machine avec processeur Turion 64x2 Mobile 1.80 GHz possédant 1918 MB de mémoire. Les résultats de l'étude sont présentés et discutés dans les sections suivantes.

5.3.2 Méthodologie de l'expérimentation

Notre expérimentation utilise une méthodologie similaire à celle utilisée dans [Hassan 04]. Nous l'avons adaptée à notre projet pour affiner l'analyse à un niveau de granularité méthode, filtrer les méthodes et enfin évaluer les performances des trois approches. Le même travail a été, aussi, réalisé pour le niveau classe.

5.3.2.1 Identification des méthodes et des classes impactées

L'utilisation de l'outil de comparaison binaire d'Éclipse nous a permis de comparer, de façon répétée pour toutes les versions analysées, deux versions majeures successives de l'application, l'une antérieure à l'autre, pour identifier les classes et les méthodes qui ont changé lors du passage d'une version à l'autre. Ceci nous a permis de connaître, par exemple, pour chaque version de l'application, l'ensemble des méthodes qui ont changées lors des processus de changement qu'elles ont connus. L'ensemble des informations collectées nous a permis, selon l'approche adoptée, d'évaluer l'exactitude des résultats retournés par les trois approches considérées. Par ailleurs, pour simplifier l'expérimentation, nous avons considéré seulement les éléments qui existent entre deux

versions successives. Les éléments de code ayant été par exemple supprimés, ont été ignorés.

5.3.2.2 Filtrage des éléments n'induisant aucun REA

La nature multiple des modifications, dans les différents éléments d'une application, ajoute à la complexité de l'expérimentation. Il est inutile, lors de l'expérimentation, de considérer toutes les modifications qu'une classe ou une méthode peut subir. Pour calculer adéquatement l'ensemble des éléments qui ont réellement été affectés, il est important de filtrer les éléments qui n'engendrent aucune propagation de changement tel que mentionné dans [Hassan 04]. Les méthodes qui ont été identifiées comme modifiées sont filtrées manuellement selon ces critères :

1. **Reformatage de code :**
    a. Réécriture d'une instruction sans modifier sa sémantique
    b. Changement d'espaces et de lignes vides
2. **Changement d'éléments informatifs :**
    a. Changement de balises informatives (description de paramètre d'une méthode)
    b. Changement de commentaires
3. **Modification de l'identification des éléments orientés objet :**
    a. Renommer globalement une variable, méthode, classe, package, etc.
    b. Modification globale de la définition d'une méthode (renommée un paramètre).

Le reformatage de code (1) pour améliorer la visibilité et la compréhension du code source d'une application n'implique pas un REA puisque l'ajout d'espaces et de lignes vides ne modifie en rien l'exécution et le calcul de l'impact. De plus, l'information par balise (commentaires) (2) qui renseigne le programmeur sur l'utilisation de certains composants et leur raison d'être n'affecte pas l'exécution d'une application, seulement sa compréhension. Finalement, propager le changement de nom d'un élément localement ou globalement (3) dans la totalité d'un système ne change pas la sémantique de cet élément.

5.3.2.3 Identification d'un changement initial

Lorsqu'un programmeur applique des modifications sur un système, il débute par une certaine méthode (ou classe). Il est possible que d'autres méthodes subissent des modifications à cause de ce premier changement. Cette seconde vague de changements fait partie de la propagation du changement qu'il faut capturer à différents niveaux : granularité classe et granularité méthode.

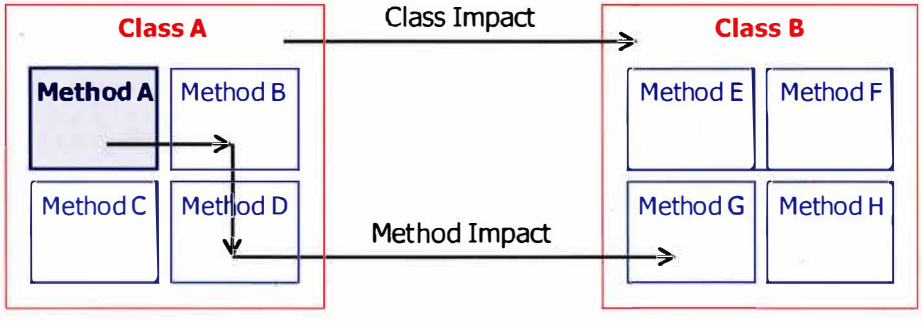


Figure 14. Chemin d'impact inter modulaire d'un changement de la méthode A.

Il est difficile de bien cibler une méthode initialement modifiée. Pour ce faire, nous avons donné plus de poids aux classes comportant un grand nombre de méthodes modifiées. Par la suite, la méthode comportant un grand nombre d'instructions modifiées

est choisie comme étant la méthode ayant subit le changement initial (ou le plus de changements initiaux) dans la version antérieur. En comparant les classes A, B et C de la *Figure 15,* il est possible de déterminer approximativement l'importance d'une méthode pour une version donnée d'un programme. Dans ce cas-ci, la méthode ayant subit 10 changement sera choisit comme point de départ pour l'analyse.

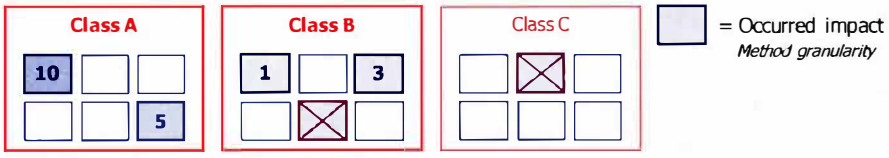


Figure 15. *Poids d'importance donné aux méthodes selon leur nombre de changements.*

Cette tâche n'a pas été facile, car pour déterminer ces éléments de base nous avons du combiner l'utilisation de l'outil d'Éclipse à une analyse (manuelle) du code. À notre connaissance, il n'existe aucun outil (stable et fiable) permettant par analyse de deux versions successives d'une application, d'identifier de façon précise les changements « initiaux » et ceux qui sont plutôt des conséquences « changements » des changements « initiaux ».

Pour chaque instruction ayant subit un changement dans la méthode initiale, nous avons appliqué les approches. Les résultats sont donnés sous forme d'un ensemble de classes et méthodes. La pertinence (« justesse ») des résultats des approches est calculée en comparant les ensembles retournés aux méthodes (classes) identifiées comme ayant réellement été modifiées par le programmeur.

5.3.2.4 Utilisation de l'outil comparatif

Comparer les différentes versions d'une application est une tâche essentielle à l'expérimentation. Il est nécessaire de connaître les méthodes qui ont changé entre une

première version plus ancienne, et une seconde version plus récente. Pour connaître les fichiers qui ont été modifiés entre deux versions d'une application, nous avons utilisé en premier lieu l'outil de rapport de *commit* d'un serveur CVS. Cet outil permet de retourner les fichiers qui ont été modifiés pour atteindre la nouvelle version. Ceci nous permet de connaître sur une centaine de fichiers, lesquels ont été modifiés par le programmeur.

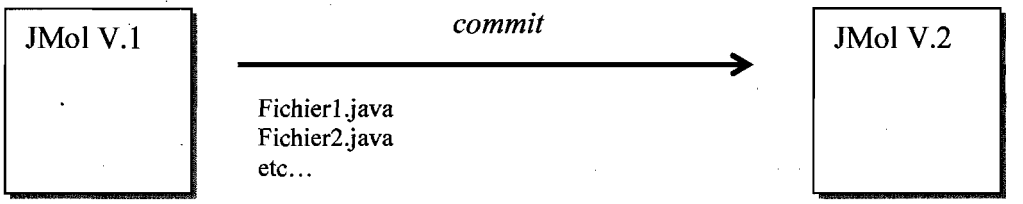


Figure 16. Enregistrement (commit) entre deux versions de JMol

Par la suite, l'utilisation de l'outil de comparaison binaire, disponible dans l'environnement de développement Eclipe, a permis de comparer les fichiers modifiés d'une version à une autre, pour connaître les sections de code qui ont subit des modifications. Le filtrage des modifications, expliqué dans la section précédente, se fait à cette étape de la préparation de l'expérimentation.

Bien que l'outil permet de connaître les différences entre les modules d'un système, il ne peu pas filtrer automatiquement l'information inutile. Par ailleurs, il ne peu pas, non plus, indiquer l'ordre chronologique des changements effectués dans un fichier. Il est donc impossible de connaître exactement le cheminement réel des modifications.

5.3.3 Métriques

Hassan et al. [Hassan 04] ont défini deux métriques pour comparer les approches REA entre elles. Ils proposent la métrique *recall* qui est le pourcentage du nombre d'éléments

bien identifiés (*PO*) de l'approche par rapport au nombre réel d'éléments qui ont été changés lors du processus de maintenance. Cette mesure indique la sensibilité de l'approche et permet de connaître la quantité d'éléments, bien ciblés du système, ainsi que la charge de travail nécessaire pour connaître les méthodes à changer dans ce système. Par exemple, un *recall* de 0.3 signifie que l'approche a bien identifié 30% des méthodes modifiées dans un système, mais que 70% des méthodes réellement modifiées n'ont pas été identifiées. Un *recall* de 1 signifie que la totalité des méthodes impactées ont bien été ciblées par l'approche.

La pertinence d'une approche se calcule aussi en quantité d'éléments qu'elle retourne. La mesure *precision* est le pourcentage d'éléments bien identifiés (*PO*) de l'approche par rapport au nombre total d'éléments identifiés (*P*). La mesure permet de connaître l'effort de recherche des méthodes réellement affectées à l'intérieur de l'ensemble des résultats retournés. Par exemple, une *precision* de 0.5 implique que sur 2 méthodes retournées par l'approche, l'une d'elle n'est pas réellement impactée par un changement. Une précision de 1 signifie que tous les éléments qu'elle retourne sont impactés par un changement.

5.3.3.1 Analyse de performance

De plus, nous avons évalué en termes de performance (temps d'analyse et quantité mémoire requise) les trois approches. Le temps de l'analyse se divise en deux parties. La première calcule le temps qu'Éclipse prend pour construire la structure AST. Le deuxième calcule le temps de création du modèle de dépendance de l'approche à partir des AST et du calcul de l'impact.

5.3.4 Résultats et discussions

Collecter les données réelles observées des changements et les analyser pour pouvoir démarrer notre évaluation des approches n'a pas été facile. Le travail a été, dans

certaines étapes, très laborieux et très minutieux. Notre expérimentation s'est limitée aux sept premières versions successives de Jmol qui en compte actuellement une dizaine. Les données collectées étaient assez significatives pour permettre notre expérimentation. Le *tableau 4* donne quelques statistiques descriptives sur les différentes versions du projet Jmol. Nous pouvons observer que la taille en ligne de code est, passée de 18357 (première version) à 30288 (dernière version). Cela représente une augmentation de presque 65%. Le nombre de classes est passé de 237 à 315 (33 %). Le nombre de méthodes est passé de 1073 à 1732 (61 %). Le passage de la version 1 à 2 n'a pas été retenu (peu de données). Les résultats obtenus sont présentés dans les trois tableaux suivants: granularité méthode (*Tableau 5)*, granularité classe (*Tableau 6)* et performance (*Tableau 7*).

Tableau 4. Statistiques descriptives sur les différentes versions de Jmol.

Versions	# att. pub.	# att. prot.	# att. priv.	# classes	# méthodes	LOC
1	100	47	569	237	1073	18357
1.1	155	48	585	256	1153	20228
1.2	155	48	588	259	1164	22548
2	65	35	575	206	950	18166
3	64	52	590	214	996	18666
4	59	54	535	225	956	19324
5	59	54	589	249	1079	21908
6	181	44	529	293	1656	28252
7	183	49	541	313	1718	29650
8	182	49	553	315	1732	30288

Tableau 5. « Recall » et « Precision » des trois approches au niveau méthode.

REV	OCCURRED	CGi				CGd				CCG				FS			
	M	P	PO	Recall	Prec.	P	PO	Recall	Prec.	P	PO	Recall	Prec.	P	PO	Recall	Prec.
2-3	33	300	18	0.55	0.06	28	9	0.27	0.32	43	10	0.30	0.23	9	3	0.09	0.33
3-4	38	226	6	0.16	0.03	29	2	0.05	0.07	11	3	0.08	0.27	19	3	0.08	0.16
4-5	22	88	7	0.32	0.08	12	6	0.27	0.50	14	6	0.27	0.43	47	7	0.32	0.15
5-6	93	291	23	0.25	0.08	35	3	0.03	0.09	34	3	0.03	0.09	88	15	0.16	0.17
6-7	45	30	6	0.13	0.20	17	2	0.04	0.12	37	2	0.04	0.05	2	1	0.02	0.50
	231	935	60	0.28	0.09	121	22	0.13	0.22	139	24	0.15	0.22	165	29	0.13	0.26
				average				average				average				average	

Le *tableau 5* résume les résultats obtenus pour une granularité « méthode ». Les trois approches évaluées figurent dans le tableau : CGi (graphes d'appels indirects), CGd (Graphes d'appels directs), CCG (Graphes d'appels réduits au contrôle) et FS (Forward slicing). La colonne *M* indique le nombre de méthodes réellement modifiées entre les deux révisions. Le nombre total de changements observés est de 231, ce qui donne une moyenne d'environ 46 changements entre deux versions. La colonne *P* (pour toutes les approches) note le nombre de méthodes totales que l'approche retourne et la colonne *PO* (même chose) note les méthodes qui ont bien été ciblées par l'approche. En moyenne, le CGi identifie correctement 28% des méthodes, ce qui est plus que les autres approches. Par contre, il obtient un grand *recall* en omettant la précision de résultats retournés qui est aussi la plus petite de toutes les approches avec 9%. Il retourne un ensemble de résultats excessifs et ne serait pas praticable pour un programme en situation réelle de développement, en particulier de grande taille. Sa version directe, CGd, identifie correctement 13% des méthodes avec une précision accrue de 22%. Ces premières conclusions complètent donc et confirment les remarques au sujet des approches basées sur les graphes d'appels formulées dans les premières parties du mémoire. Le CCG identifie correctement 15% des méthodes tandis que le FS identifie, tout comme le CGd, 13% des méthodes. La faible valeur du *recall* des approches s'explique du fait que l'analyse s'effectue sur une seule méthode initiale et qu'une révision peut impliquer le changement initial de plusieurs méthodes calculées dans *M*. Un approfondissement de l'analyse du code permettrait certainement d'affiner ces résultats. Actuellement, au meilleur de notre connaissance, aucun outil ne permet, par analyse de deux versions d'une même application, d'identifier la totalité des méthodes initiales (changements initiaux) ainsi que l'ensemble de méthodes changées qu'elles impliquent (conséquences des changements).

Le CCG semble identifier correctement plus de méthodes que le CGd en ajoutant la capture des méthodes affectées en retour d'appel. Les révisions 2-3 et 3-4 sont des exemples où le CCG capture une méthode de plus que le CGd en analysant les retours

d'appels. Pour les révisions 2-3, 4-5 et 6-7 la précision est la même pour le CGd et CCG malgré la valeur élevée de P de ce dernier. C'est dû, en grande partie, à la localisation du changement permettant au CCG de réduire l'ensemble des méthodes directement affectées. Par contre, dans certaines situations, un grand ensemble de résultats peut être retourné à cause de sa capture des retours d'appels. En effet, pour la révision 6-7, le CCG pousse plus loin sa recherche que le CGd sans augmenter son *recall*.

Le CCG cible les méthodes appelées à partir d'une localisation de changements sans discriminer leur sémantique. Il ne filtre pas les méthodes appelées par l'utilisation de données, mais par leur emplacement dans les contrôles lui permettant ainsi d'obtenir un meilleur *recall* que le FS. Par contre, il perd en *precision* lorsque la méthode analysée comporte beaucoup d'appels qui ne peuvent être discriminés par l'exclusion mutuelle du cheminement de contrôle; révisions 5-6 et 6-7 où les modifications sont effectuées en début de méthodes. En discriminant les cheminements de données, le FS augmente la confiance de ses résultats à bien cibler les méthodes réellement impactées.

Le FS traditionnel analyse directement et indirectement les cheminements de données. L'indirecte du FS, sans être limité dans sa profondeur de recherche transitive, semble être la cause de sa perte de *precision* et cela se remarque avec les révisions 4-5 et 5-6 où le FS retourne un ensemble de résultats excessifs comparés au CCG. Le CCG limite l'indirecte au retour d'appels et réduit considérablement l'ensemble de ses résultats tout en conservant un bon *recall*. Même en limitant la profondeur (nombre d'appels subséquent) de sa recherche indirecte, le CCG obtient un meilleur *recall* que le FS. De plus, le CGd obtient le même *recall* que le FS. Cela pourrait indiquer que les liens syntaxiques directs entre méthodes expriment mieux une propagation de changements que les liens subséquents indirects. [Zhang 07] rapporte qu'il observe une distance très petite, granularité instruction, de la propagation d'impacts en effectuant un FS dynamique. Nous observons aussi une petite distance pour le FS statique.

Tableau 6. « Recall » et « Precision » des trois approches au niveau classe.

REV	OCCURRED	CGi				CGd				CCG				FS			
	C	P	PO	Recall	Prec.	P	PO	Recall	Prec.	P	PO	Recall	Prec.	P	PO	Recall	Prec.
2-3	15	64	11	0.73	0.17	15	7	0.47	0.47	23	7	0.47	0.30	6	5	0.33	0.83
3-4	20	42	10	0.50	0.24	15	4	0.20	0.27	5	3	0.15	0.60	4	2	0.10	0.50
4-5	14	25	7	0.50	0.28	10	6	0.43	0.60	12	7	0.50	0.58	16	6	0.43	0.38
5-6	43	50	18	0.42	0.36	19	10	0.23	0.53	19	10	0.23	0.53	43	8	0.19	0.19
6-7	29	17	8	0.28	0.47	17	5	0.17	0.29	18	7	0.24	0.39	1	1	0.03	1.00
	121	198	54	0.49	0.30	76	32	0.30	0.43	77	34	0.32	0.48	70	22	0.22	0.58
				average				average				average				average	

Le *tableau 6* compile les résultats pour les mêmes révisions que le *tableau 5*, mais à un niveau de granularité supérieure : les classes. Les résultats sont présentés de façon similaire que ceux donnés par le *tableau 5*. Le CCG possède un *recall* de 32%, plus fort que le FS qui possède un *recall* de 22% et le CGd avec 30%. En contrepartie, le FS possède une *precision* de 58% tandis que le CCG possède une *precision* de 48%. Le *tableau 6* démontre que le CGi peu être utile à un niveau classe pour indiquer la propagation des changements entre classes. Elle est par contre l'approche qui obtient une fois de plus la plus petite précision. Le CCG se démarque du CGd en ayant cette fois-ci une meilleure *precision* et *recall*. Comparé au FS, le CCG obtient un plus grand *recall,* signe qu'il est moins restrictif dans l'analyse que le FS. Par contre, il obtient encore une fois une *precision* plus petite que le FS. En regardant les *Figures 17 et 18*, il est possible de visualiser les caractéristiques des approches. Plus une approche suit la ligne diagonale, plus elle est balancée entre son *recall* et *precision*. Plus une approche se rapproche du coin supérieur droit, plus elle est efficace. Le FS obtient une très grande *precision,* dans les deux diagrammes, comparé aux autres approches, mais obtient un très faible *recall* plus marqué dans la granularité classe. Le CGi quand à lui, obtient le meilleur *recall* des approches dans les deux granularités. Il obtient aussi la plus faible précision dans les deux granularités. Le CGd et CCG quand à eux, se situent dans le milieu des diagrammes. Ils semblent être plus balancés que le CGi et FS. Dans la *Figure 18*, on remarque que le CCG tend à obtenir une meilleure performance que le CGd en se rapprochant du coin supérieur droit.

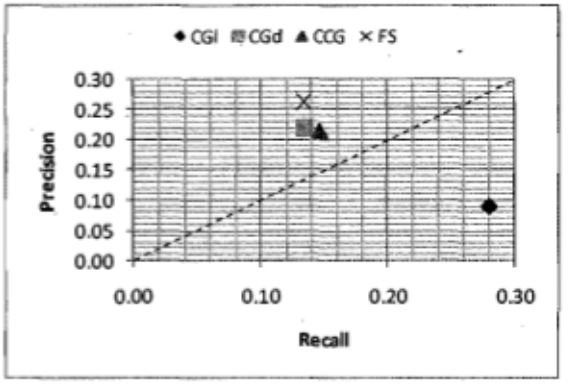


Figure 17. Diagramme « Recall » vs « Precision » : niveau méthode.

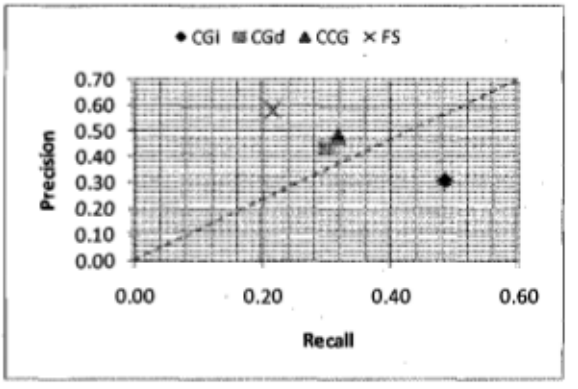


Figure 18. Diagramme « Recall » vs « Precision » : niveau classe.

Tableau 7. Performance des quatre approches

REV	CGi			CGd			CCG			FS		
	AST	Analysis	Mem.	AST	Analysis	Mem.	AST	Analysis	Mem.	AST	Analysis	Mem.
2-3	1638	10795	49.8	1695	156	47.1	1671	140	41.2	1695	10584	48.5
3-4	2096	12630	62.5	1971	174	50.1	1732	171	44.2	1826	10888	45.3
4-5	1834	11369	58.9	1957	130	52.6	1763	234	39.3	1544	11325	44.2
5-6	1988	16850	72.6	2011	195	59.1	2044	187	45.0	2054	16021	51.2
6-7	2808	46473	94.4	2305	265	75.0	2627	152	75.3	2614	27308	63.6
avg.	2073	19623	67.6	1988	184	56.8	1967	177	49.0	1947	15225	50.6

Le *tableau 7* rapporte le temps, en milliseconde, de la création de la structure AST d'Éclipse pour chaque approche ainsi que le temps d'analyse, ce qui inclut la création de

leur structure de dépendance respective. L'utilisation de la mémoire des outils est aussi notée à chaque expérimentation. Le calcul de la mémoire consommée se fait en calculant la différence de consommation de mémoire de l'outil avant et après l'exécution d'une analyse. À partir des résultats rapportés, nous pouvons observer que le temps de création de l'AST d'Éclipse est très stable pour les trois approches et n'affecte en rien le temps de calcul des résultats. En moyenne, la création d'un AST prend 2 secondes. Les deux approches directes obtiennent le meilleur temps d'analyse qui est d'environ 1/10 de seconde, ce qui est relativement très rapide. Pour les approches CGi et FS, le temps d'analyse est très imposant. En moyenne, ils prennent 19 et 15 secondes respectivement pour retourner un résultat. Pour faire 5 analyses, le CGi et FS prennent plus d'une minute tandis que le CGd et CCG prennent moins d'une seconde. La quantité de mémoire utilisée est similaire dans toutes les approches. Cela est dû à l'utilisation intensive des AST d'Éclipse pour l'implémentation des approches. Le CGi semble tout de même utiliser plus de mémoire. La création de matrices pour déterminer les relations indirectes pourrait expliquer une utilisation accrue de la mémoire.

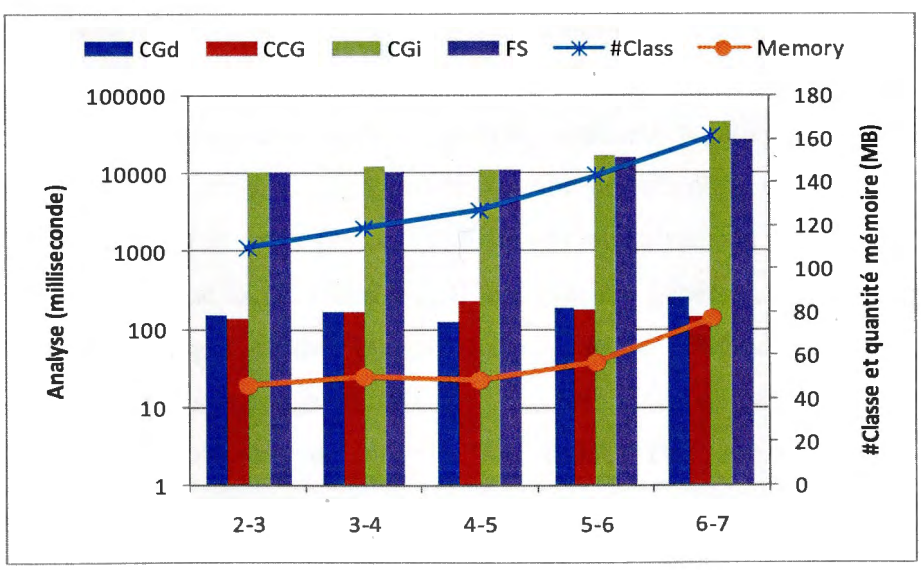


Figure 19. Diagramme du temps d'analyse comparativement au nombre de classes et quantité de mémoire par révision.

Dans la *Figure 19*, on ressort le temps requis (axe de gauche) pour effectuer l'analyse de chaque révision et la quantité de mémoire moyenne requise selon le nombre de classes pour l'ensemble des approches (axe de droite). Visiblement, il existe une connexion entre le nombre de classes qui augmente d'une révision à une autre, le temps requis pour effectuer l'analyse et la quantité moyenne de mémoire requise pour l'ensemble des approches. Plus il y a d'informations à analyser, plus la quantité de mémoire requise par les approches augmente. Cette remarque est aussi valable pour le temps d'analyse. Par ailleurs, les approches directes sont largement plus rapides que les approches indirectes et semblent être plus ou moins affectées par l'augmentation des classes entre les révisions.

5.4 Conclusion de l'expérimentation

Dans ce chapitre, nous avons présenté une étude expérimentale portant sur trois approches d'analyse statiques de l'impact des changements : technique basée sur les graphes d'appels traditionnels, technique basée sur les graphes de contrôle réduits aux appels et technique basée sur le « slicing » traditionnel. Au regard des deux métriques utilisées *recall* et *precision*, notre approche fournie des résultats mieux balancés que ceux du FS. Sur le plan de la performance (temps d'analyse), la technique CCG est nettement meilleure que la technique FS. D'une manière générale, les deux approches possèdent des avantages et des inconvénients. L'étude effectuée a révélé que la différence relative dans la précision entre les deux techniques ne permet pas d'observer dans les résultats obtenus de grandes variations. Des études expérimentales complémentaires nous permettrons certainement d'affiner cette comparaison avant de tirer des conclusions finales à ce sujet.

La présente étude a, néanmoins, confirmé l'intérêt de prendre en compte le flot de contrôles et le flot de données dans l'analyse de l'impact. L'unification des deux approches est une piste qui semblait intéressante. Nous l'avons considérée dans nos travaux. Elle fera l'objet du prochain chapitre.

Finalement, la difficulté d'identifier les changements initiaux est évidente. Il est très laborieux d'analyser manuellement un système, dans le but de connaître les propagations de changements multiples, à partir d'une simple liste de fichiers modifiés simultanément que les CVS retournent.

CHAPITRE 6

VERS UNE UNIFICATION DES FLOTS DE CONTRÔLES ET DE DONNÉES : LE DATA CONTROL PATH GRAPH

Nous présentons, dans ce chapitre, une nouvelle approche statique : Le *Data Control Path Graph* (DCPG). Le DCPG détermine l'impact de changements sur des applications orientées objet en se basant sur l'utilisation conjointe, de façon unifiée, des graphes de contrôles (chemins de contrôle) et d'un modèle de cheminement de données (chemins de données) inspiré du slicing. Nous étendons la granularité de l'analyse à un ensemble d'instructions délimitées par les contrôles. Cette réunion d'instructions (sous forme de bloc) permet de réduire l'information qu'un programmeur doit considérer lors d'une analyse.

Le nouveau modèle, créé par unification des deux concepts, est un graphe permettant de rassembler les fonctionnalités communes d'un système et de les lier syntaxiquement (différents types de dépendances). Les nœuds de chaque instruction, dans un graphe de contrôles, sont unifiés pour former un cheminement appelé *Data Control Path* (DCP). L'approche ajoute au graphe de contrôle classique, discuté dans les chapitres précédents, les liens de dépendance de données que l'on retrouve entre les vertex d'instructions d'un Program Dependencies Graph (PDG) [Tip 94]. Le modèle résultant est un graphe permettant de cibler les zones, à l'intérieur des corps de méthodes, qui seront affectées lors d'un changement donné. Les bénéfices attendus d'une telle technique portent, dans l'ensemble, sur une amélioration de la précision des méthodes réellement affectées et l'abstraction dé la grande quantité d'information relative aux dépendances. Ceci facilitera, à notre point de vue, le travail de l'utilisateur du modèle. L'objectif principal du nouveau modèle, utilisant à la fois le cheminement de contrôle et le cheminement de données, est d'améliorer les résultats du *forward slicing* traditionnel ainsi que ceux du

modèle basé sur les graphes de contrôles réduits aux appels. Ce second développement de modèle s'inspire des résultats et conclusions du précédent chapitre. L'idée de base porte sur une unification des deux approches étudiées et expérimentées dans le précédent chapitre (graphe de contrôle réduit aux appels et forward slicing).

Arnold [Arnold 93] indique que le désavantage des graphes de contrôles est le manque de cheminement de données. Ceci est aussi valable, dans une certaine mesure, pour les graphes de contrôle réduits aux appels (modèle implémenté et expérimenté en premier lieu). De plus, il indique que la grosseur des tranches de programmes retournées par une analyse de FS traditionnel peut être trop grande pour être utile à un programmeur. Notre approche complète le manque d'information du graphe de contrôles. Elle ajoute le cheminement des données et limite le résultat d'une analyse de flot de données à une granularité plus précise qu'une méthode et plus abstraite que la variable de la simple instruction. Ceci nous semble un compromis intéressant.

6.1 Support aux graphes traditionnels

Bishop [Bishop 04] insiste sur la nécessité d'améliorer le temps machine d'analyse du REA. Notre approche fait l'intégration du flot de données au graphe de contrôles et permet d'effectuer une analyse se rapprochant des techniques de « slicing » tout en supportant les approches qui utilisent les graphes de contrôles et leurs avantages par conséquent. Ceci permet de tirer profit des avantages des deux approches discutées dans le chapitre précédent. L'utilisation d'un graphe hybride peut, par ailleurs, réduire conséquemment le temps et la mémoire requis pour permettre l'utilisation simultanée de plusieurs modèles d'analyse. De plus, l'abstraction de l'information dans le DCPG permet de réduire la quantité d'information à analyser et le temps d'analyse nécessaire.

Par ailleurs, Lindvall conclut dans [Lindvall 99], après une expérimentation sur la quantité de changements d'un système, que le corps des méthodes est l'élément le plus modifié pour accommoder les appels aux nouvelles méthodes. Il est donc important de supporter la prédiction de modification du corps des méthodes. Selon Kung [Kung 94], il existe 3 classes de modifications possibles à l'intérieur d'une méthode. Des lignes d'instructions peuvent être supprimées, modifiées et ajoutées. Pour notre approche, une ligne d'instruction supprimée se définie comme une ligne d'instruction que l'on ne peut plus retrouver d'une version à une autre à sa position relative (en omettant les espaces et lignes blanches). Une position est relative à deux instructions précédentes et suivantes. Il est difficile de bien définir ces modifications et de les discriminer entre deux versions d'une application. Une modification peut, par exemple, être interprétée comme l'ajout d'une nouvelle instruction ou bien la suppression de l'instruction et l'ajout d'une nouvelle.

L'analyse prédictive de l'impact de changements par flot de données ne peut pas être effectuée dans toutes circonstances de modifications. En effet, Hassan [Hassan 04] indique qu'il n'est pas possible de prévoir la propagation de changements lors d'une addition de nouvelles entités à partir du flot de données. Le chemin, qu'empruntent les données, reste incomplet tant que le processus d'ajout n'a pas été complètement propagé dans le système. Pour connaître l'importance de supporter l'analyse d'ajout d'instructions dans l'implémentation des méthodes, une recherche sur la quantité d'instructions ajoutées par rapport à la totalité des changements à été effectuée à partir du projet jEdit écrit en JAVA. Avec l'outil de comparaison de fichiers de l'environnement de développement Éclipse, nous avons noté le nombre de modifications, ajouts et suppressions effectués entre deux révisions majeures et récentes (9484 et 9500). L'outil d'Éclipse permet d'indiquer les éléments ajoutés et supprimés d'un projet. Nous avons filtré les modifications de commentaires et formatages usuels du code. Par la comparaison des modifications unifiées, nous avons créé le tableau suivant (*Tableau 8)*:

Tableau 8. Changements effectués sur les instructions des méthodes entre deux versions majeures de jEdit.

jEdit Rev. 9484 - 9500			
# Methode	**Modified**	**Suppressed**	**Added**
1	0	1	0
2	2	0	0
3	0	2	0
4	2	0	2
5	4	2	7
6	1	0	1
7	1	0	0
8	0	1	0
9	2	0	0
10	1	0	3
11	4	0	0
12	2	0	2
13	5	0	0
14	4	0	0
15	2	0	0
Total	**30**	**60**	**15**
%	**58.82**	**11.76**	**29.41**

Pour chaque méthode, le nombre de changements effectués entre les deux révisions a été calculé manuellement. Les résultats montrent que 29% des changements effectués entre les deux révisions sont des ajouts d'instructions. La grande majorité des changements sont des modifications d'instructions. Bien que l'application jEdit soit assez mature, l'ajout de code reste encore très présent dans les modifications des méthodes du système. Il est donc nécessaire de permettre à un programmeur d'effectuer des analyses prédictives pour presque 30% des changements internes modulaires d'un système.

Pour prévoir l'effet d'un ajout à un emplacement, il faut utiliser des techniques qui utilisent la localisation du changement à partir du flot de contrôles. La technique CCG permet cette sorte d'analyse. Cependant, elle capture la majorité des appels à partir d'un point dans une méthode, ce qui donne des résultats plus précis que les CG mais moins précis que le slicing statique traditionnel. La combinaison du flot de données et du flot de contrôles permet d'effectuer une analyse prédictive à l'interne d'une méthode. La

technique statique DCPG a été conçue pour prédire l'effet potentiel d'un changement. En ciblant un ensemble d'instructions comme emplacement de modifications possibles, le DCPG a une chance plus accrue de découvrir des relations de dépendance de données qui peuvent être affectées lors d'un ajout d'instructions.

6.2 Simplification des méthodes d'analyse

Dépendamment de l'approche implémentée dans un outil, la difficulté d'analyse d'impact résulte de la configuration d'une analyse, de la compréhension des résultats et de la quantité d'informations retournées. La configuration des outils pour effectuer une analyse peut être exhaustive, c'est-à-dire requérant beaucoup de connaissances et plusieurs essais afin de trouver le bon ensemble d'options fonctionnelles pour l'application. Le *program slicing* permet une grande quantité d'options pour plusieurs types de slices. Le temps pour configurer ces options et la formation nécessaire à leur utilisation augmentent le temps d'analyse humain. Il y a un réel besoin dans le domaine de faciliter les analyses et la compréhension des résultats. Il faut développer des outils qui permettent d'effectuer des analyses simplement et facilement, avec un minimum de configuration et de manipulation et qui supportent la majorité des processus de maintenance.

6.3 Le Data Control Path Graph

Le *Data Control Path Graph* est un graphe modélisant le flot de contrôles et les dépendances de données. Le graphe représente une méthode contenue dans les classes d'un système orienté-objet. Il se base sur les graphes de contrôles pour permettre l'utilisation de graphes traditionnels et l'abstraction des instructions. On ajoute une notion de lien de dépendance, que l'on retrouve dans les PDG [Tip 94], entre deux regroupements d'instructions.

Définition 5: *Un Path Graph (PG) est un graphe de contrôles (définition 3) où les blocs séquentiels d'instructions d'une méthode représentent des chemins appelés path (voir Figure 20). Un path est un ensemble d'instructions séparées par l'entrée ou un contrôle d'une méthode et la fin de ce contrôle ou de la méthode. Le graphe est étendu avec un nœud d'entrée lié aux nœuds de paramètres d'entrée de la méthode et, finalement, d'un nœud de sortie de méthode.*

Pour effectuer une analyse de flot de données, nous implémentons des liens de dépendance de données entre les instructions selon la définition de Danicic [Danicic 99].

Définition 6: *Un Data Control Path Graph (DCPG) est un Path Graph (définition 5) où des arcs sont ajoutés entre les nœuds du graphe pour démontrer la dépendance de données (voir Figure 21). Un nœud N2 est dépendant des données du nœud N1 s'il y a une variable V référencée dans N2 et définie dans N1 et qu'il y a un arc qui part du nœud N1 au nœud N2 sans redéfinition de la variable V. Un arc qui part du nœud N1 à N2 signifie que N2 est dépendant de la donnée de N1 et que le résultat de l'exécution de N2 est affecté par l'exécution de N1.*

La *Figure 20* est une implémentation naïve de la simulation d'un client qui achète des produits d'une manufacture tant qu'il possède de l'argent. Le *Path Graph* (PG), graphe à droite, est la représentation des *paths* contenus dans l'implémentation.

À partir de ce graphe, il est possible de déterminer les cheminements s'excluant mutuellement. Dans l'exemple, les instructions du path3 ne peuvent affecter directement les instructions du path4. De plus, la création de *path* permet de regrouper les instructions servant à atteindre collectivement un but. Par exemple, le bloc d'instructions de la méthode M est séparé en 2 *paths*. L'utilité du *path1* est l'instanciation des variables de la méthode tandis que le *path5* permet de terminer la méthode en libérant les

variables utilisées. Malgré que les deux *paths* se retrouvent dans un même bloc d'instructions, les deux ont une fonction différente (cheminement en termes de contrôle et de données). Il faut donc les considérer différemment. Cette idée de regroupement d'instructions s'appuie sur l'hypothèse qu'un programmeur va généralement regrouper les instructions essentielles à une certaine fonctionnalité (tâche à réaliser) dans une méthode. Par exemple, les instructions du *path3* (*Figure 20)* servent à vendre un produit. La vente d'un produit est liée à la satisfaction immédiate du client. Les *paths* peuvent s'assimiler à la notion de scénario (incluant le contrôle et les données) d'exécution.

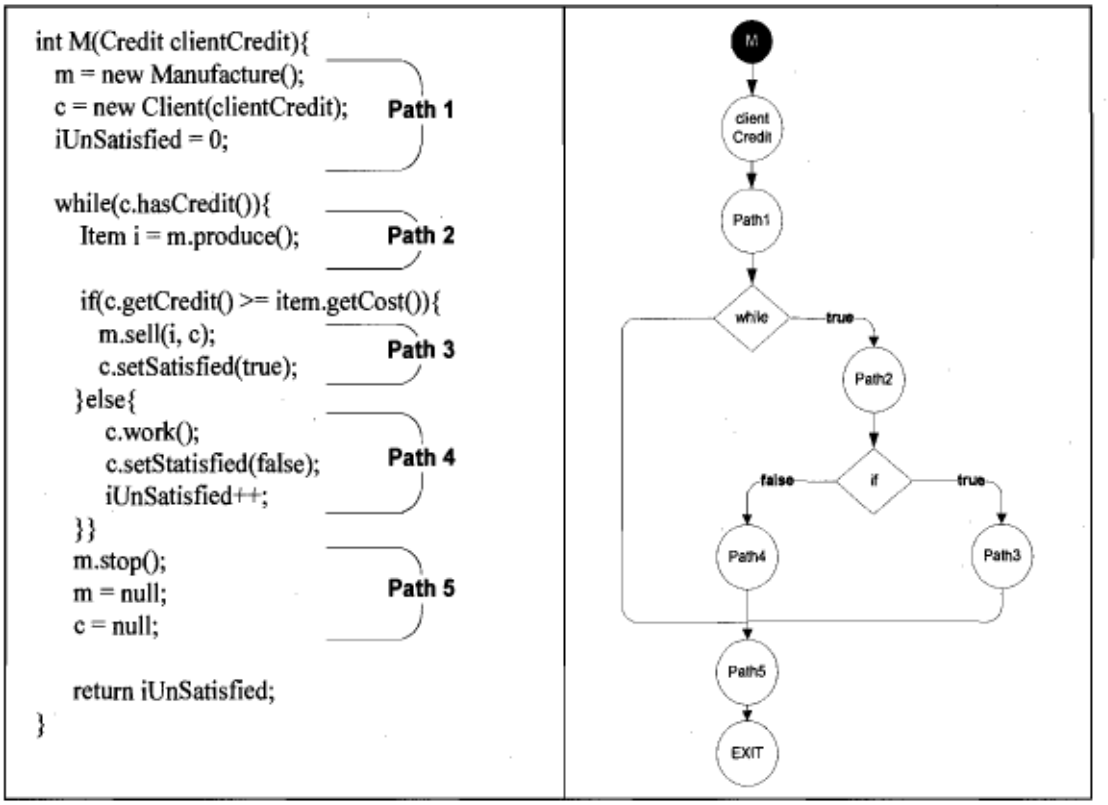


Figure 20. Exemple simple d'un PG

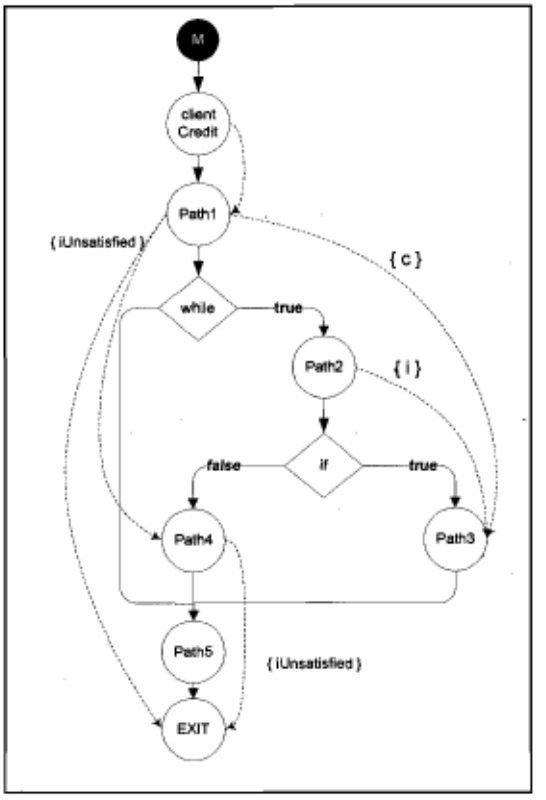


Figure 21. Data Control Path Graph de la Figure 20.

Les liens de dépendances de données sont utilisés pour connaître les relations d'utilisation de variables entre les fonctionnalités (différentes tâches élémentaires – scénarios implémentés) d'une méthode. Le *Data Control Path Graph* permet de résumer les liens multiples de dépendances de données entre *paths* d'une méthode. Dans cette dernière représentation, on voit très bien que l'instanciation des variables utiles à la méthode est utilisée dans le 3^{e} et 4^{e} cheminement. Ils représentent le processus d'achat et de travail d'un client. Il est aussi notable de voir que le 3^{e} et 5^{e} cheminement n'affectent sémantiquement aucun autre cheminement dans la méthode. D'ailleurs, le 5^{e} cheminement est indépendant du reste des fonctionnalités de la méthode.

6.4 Analyse de l'impact avec le Data Control Path Graph

L'analyse du cheminement de données entre *path* permet de connaître les dépendances possibles entre les diverses fonctionnalités d'une méthode ciblée pour des changements (CP) et les fonctionnalités de méthodes utilisées ou l'utilisant. Nous continuons l'exemple de la *Figure 20* en ajoutant le corps d'une méthode nommée *sell(Item i, Client c)* et son graphe DCPG dans la *Figure 22.*

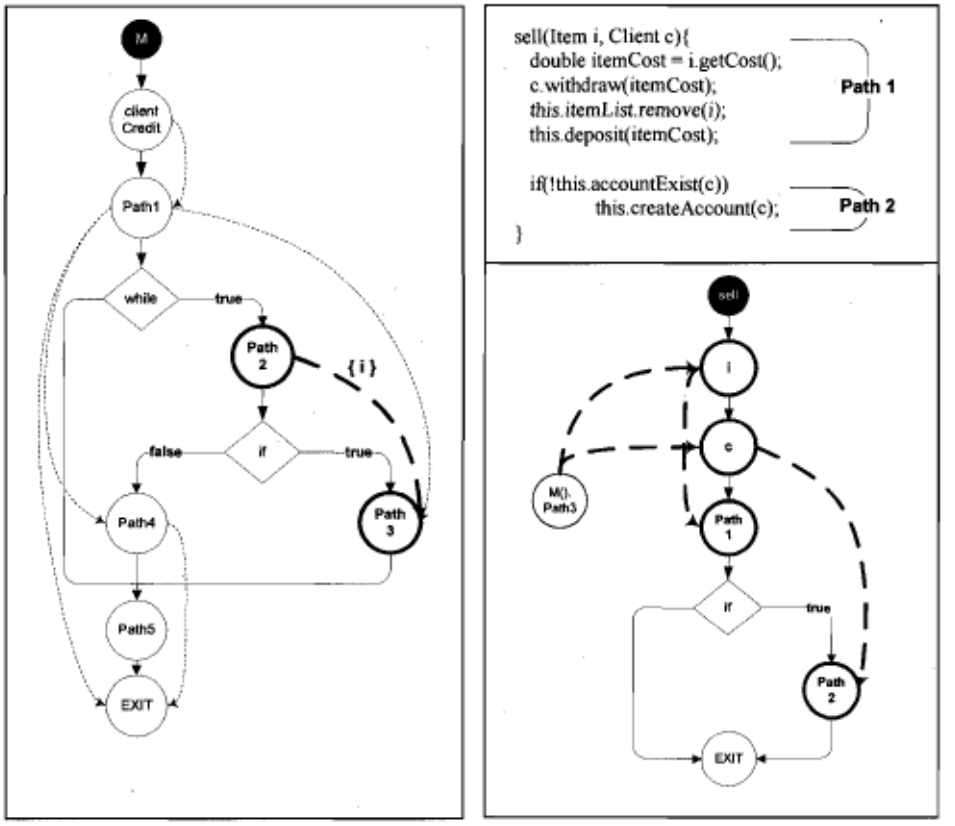


Figure 22. Analyse Inter modulaire de DCPG

Le calcul de l'analyse par cheminement de données se base sur les liens de dépendances de données entre les nœuds de *path* du graphe DCPG. Une méthode M*i* affecte une

méthode M*j* si M*j* utilise des données qui sont affectées par argument d'appels ou par retour de méthode. Soit M*i* la méthode devant subir des changements, *P* le *path* subissant la localisation des changements dans M*i*. Nous définissons les ensembles calculés à partir des liens de dépendances du graphe DCPG.

MCDmi : L'ensemble des *path* directement affectés à partir de *P*. L'ensemble inclus les liens de dépendances de données intra modulaires de M*i* et inter modulaires entre *Mi* et *Mj*.

MCLmi : L'ensemble des *path* qui utilisent directement M*i* et où M*i* retourne une valeur affectée.

MCAmi : L'ensemble des *path*, provenant de *MCLmi*, qui sont affectées par le retour de M*i*.

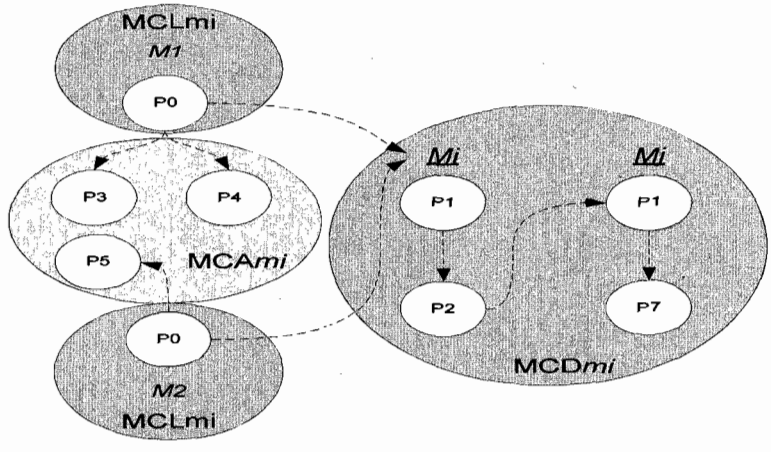


Figure 23. Exemple d'interactions des ensembles lors de l'analyse de path

La *Figure 23* montre l'interaction des ensembles de résultats. Chaque nœud noté *P#* représente un *path*. Les liens entre les nœuds et les ensembles représentent l'affectation entre les *path* et les ensembles. Par exemple, *P0* de *M1* utilise une valeur retournée de

Mi et est donc affecté par ce retour formant un ensemble de résultats *MCLmi*. Ce retour affecte par la suite de plus amples *path* dans *M1* par le cheminement des données donnant un second ensemble de résultat *MCAmi*.

L'ensemble des *path* potentiellement affectés est donné par :

$$I_{mi} = MCD_{mi} \cup MCL_{mi} \cup MCA_{mi} \cup P$$

Remarquez que lorsque M*i* ne retourne aucune valeur affectée, l'ensemble

$$MCL_{mi} \cup MCA_{mi} = \{\phi\}.$$

À partir de la *Figure 21*, si la proposition de changement se situe dans le *Path2,* il est possible de prévoir l'impact de changements entre les fonctionnalités de la méthode. En parcourant tous les cheminements de données en partant du nœud *Path2*, il est possible d'identifier les *Paths* dépendant de ses données, dans ce cas-ci, le *Path3*. Une analyse inter-modulaire se poursuit en suivant ce cheminement entre les méthodes d'un système. Dans ce cas-ci, M.*Path2* affecte, par l'entremise de l'argument *i,* M.*Path3*. Puisque les *paths* font abstraction de tous les liens de dépendances sortant, la totalité des arguments d'une méthode subséquente affectée doivent être considérés comme affectés. Dans ce cas-ci, la méthode *sell()* (méthode ajoutée) est affectée et, en considérant ses arguments comme affectés, on ressort les *Path1 et Path2*.

En ne considérant que les méthodes définies dans l'exemple (*Figures 20-22)*, l'ensemble résultant des *paths* affectés est {M().*path2*, M().*path3*, Sell().*path1*, Sell().*path2*}. De cet ensemble on peut ressortir l'ensemble des méthodes affectées: {M(), Sell()}.

6.5 Étude expérimentale

6.5.1 Introduction

L'étude expérimentale du DCPG s'est déroulée dans le même cadre que l'expérimentation précédente (Chapitre 5). Avec le même environnement et la même méthodologie, nous avons recueilli les résultats et noté les performances du DCPG. Dans ce contexte, nous comparons le *Forward Slicing* traditionnel (FS, Chapitre 5) avec le *Forward Slicing* directe (noté FS*), une adaptation du FS qui considère seulement les méthodes directement affectées, à la manière du CCG, par l'analyse FS. Nous ajoutons à l'expérimentation nos deux approches proposées, le *Control Call Graph* (CCG, Chapitre 4) ainsi que le *Data Control Path Graph* (DCPG), approche proposée dans les précédentes sections. Nous avons suivi, dans l'ensemble, la même méthodologie (et environnement) d'expérimentation que celle présentée dans le chapitre précédent.

6.5.2 Résultats

Le *tableau 9* rapporte les résultats obtenus du DCPG et du FS*. Les résultats du CCG et du FS sont exactement les mêmes que l'expérimentation du Chapitre 5.

Tableau 9. « Precision » et « Recall » des approches niveau méthode.

REV	OCCURRED	DCPG				CCG				FS				FS*			
	M	P	PO	Recall	Prec.	P	PO	Recall	Prec.	P	PO	Recall	Prec.	P	PO	Recall	Prec.
2-3	33	17	5	0.15	0.29	43	10	0.30	0.23	9	3	0.09	0.33	8	3	0.09	0.38
3-4	38	9	3	0.08	0.33	11	3	0.08	0.27	19	3	0.08	0.16	8	3	0.08	0.38
4-5	22	11	5	0.23	0.45	14	6	0.27	0.43	47	7	0.32	0.15	10	4	0.18	0.40
5-6	93	32	3	0.03	0.09	34	3	0.03	0.09	88	15	0.16	0.17	20	1	0.01	0.05
6-7	45	4	1	0.02	0.25	37	2	0.04	0.05	2	1	0.02	0.50	2	1	0.02	0.50
	231	73	17	0.10	0.29	139	24	0.15	0.22	165	29	0.13	0.26	48	12	0.08	0.34
				average				average				average				average	

À une granularité méthode, nous pouvons classer la performance des techniques en ordre croissant à partir de leur *recall* (FS*, DCPG, FS, CCG) et à partir de leur *precision* (CCG, FS, DCPG, FS*). Le FS semble être l'approche la plus balancée entre le *recall* et la *precision* suivit de près par le CCG (*Figure 24).* Le FS* et le CCG sont deux approches qui se distinguent avec de très grands résultats pour chacune des métriques utilisées. Le CCG possède le plus grand *recall* mais aussi la plus petite *precision* avec 0.15 et 0.22 respectivement. Le FS* possède quant à lui, au contraire du CCG, le plus petit *recall* et la plus grande *precision* avec 0.08 et 0.34 respectivement.

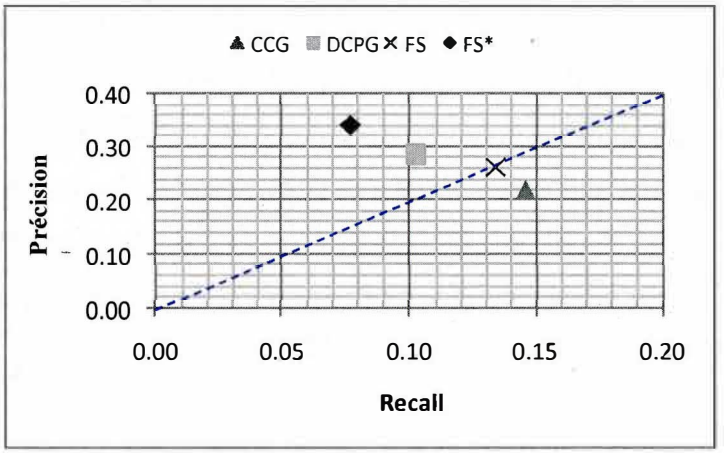


Figure 24. Diagramme « Recall » vs « Precision » : niveau méthode

Le DCPG possède une plus grande *precision* que le CCG et le FS avec un résultat de 0.29 contrairement à 0.22 et 0.26. Il améliore la précision de son ensemble de résultats par rapport au CCG en propageant son analyse par les liens de dépendances de données. Il précise aussi son résultat par rapport au FS en réduisant son analyse aux méthodes directes. Le DCPG possède un *recall* plus grand que le FS*. Il améliore légèrement son *recall* par l'abstraction des liens de dépendances et en considérant plus de possibilités par l'utilisation des *Paths.* Ce n'est pourtant pas assez pour posséder un *recall* plus important que le FS et CCG. Malheureusement, en utilisant des *paths*, le DCPG perd en *precision* lorsqu'on le compare au FS*.

Tableau 10. « Precision » et « Recall » des approches niveau classe.

REV	OCCURRED	DCPG				CCG				FS				FS*			
	C	P	PO	Recall	Prec.	P	PO	Recall	Prec.	P	PO	Recall	Prec.	P	PO	Recall	Prec.
2-3	15	13	6	0.40	0.46	23	7	0.47	0.30	6	5	0.33	0.83	6	5	0.33	0.83
3-4	20	5	2	0.10	0.40	5	3	0.15	0.60	4	2	0.10	0.50	4	2	0.10	0.50
4-5	14	10	6	0.43	0.60	12	7	0.50	0.58	16	6	0.43	0.38	8	6	0.43	0.75
5-6	43	19	10	0.23	0.53	19	10	0.23	0.53	43	8	0.19	0.19	15	8	0.19	0.53
6-7	29	2	2	0.07	1.00	18	7	0.24	0.39	1	1	0.03	1.00	1	1	0.03	1.00
	121	49	26	0.25	0.60	77	34	0.32	0.48	70	22	0.22	0.58	34	22	0.22	0.72
				average				average				average				average	

À une granularité classe, nous pouvons classer la performance des méthodes en ordre croissant à partir de leur *recall* (FS/FS*, DCPG, CCG) et à partir de leur *precision* (CCG, FS, DCPG, FS*). Au niveau des classes, le FS et FS* ont tout deux détecté le même nombre de classes correctement avec un *recall* de 0.22. Ils diffèrent par contre par leur *precision* où le FS* (0.72) performe mieux que le FS (0.58). Le DCPG possède un meilleur *recall* que les deux techniques de FS, soit de 0.25. Le CCG possède quant à lui le plus grand *recall* qui est de 0.32 et la plus petite *precision* soit 0.48. Le FS* possède la plus grande *precision* de 0.72, mais aussi le plus petit *recall* avec 0.22. Le DCPG améliore le FS légèrement avec une *precision* accrue de 0.60 contrairement à 0.58 et grandement comparé au CCG qui est de 0.48.

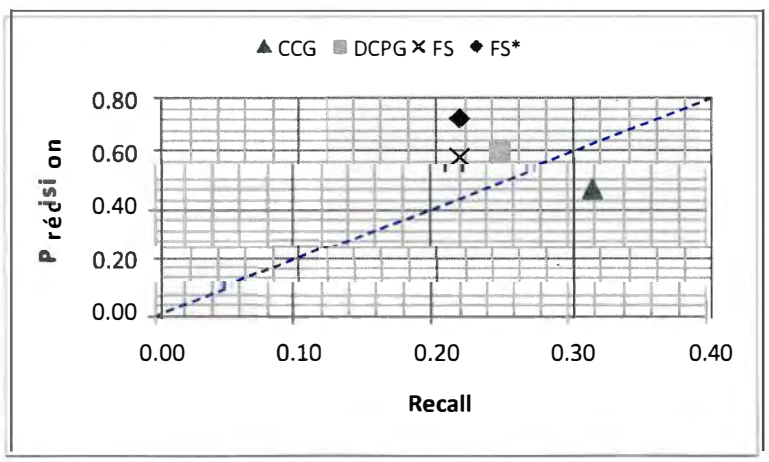


Figure 25. Diagramme « Recall » vs « Precision » : niveau classe.

Tableau 11. Performances des trois approches

REV	DCPG			CCG			FS		
	AST	Analysis	Mem.	AST	Analysis	Mem.	AST	Analysis	Mem.
2-3	1611	1117	29.3	1671	140	41.2	1695	10584	48.5
3-4	1872	1076	33.0	1732	171	44.2	1826	10888	45.3
4-5	1528	1155	39.0	1763	234	39.3	1544	11325	44.2
5-6	2013	1341	35.8	2044	187	45.0	2054	16021	51.2
6-7	2616	1409	56.5	2627	152	75.3	2614	27308	63.6
avg.	1928.00	1219.60	38.72	1967	177	49.0	1947	15225	50.6

Le DCPG réduit considérablement le temps d'analyse requis par rapport au FS. On parle ici d'environs 14 secondes en moyenne ce qui est une amélioration d'environs de 90%. L'abstraction du cheminement de données réduit considérablement le nombre de cheminements répétitifs à analyser. En conséquence, le DCPG performe plus rapidement que le FS.

6.5.3 Discussions

Le forward slicing gagne en précision en considérant minutieusement le cheminement des données entre chaque instruction d'une méthode. Par contre, il perd en *recall,* lorsqu'on le compare au CCG, par son manque de recherche et une grande restriction des liens de dépendances. L'objectif du nouveau modèle DCPG est d'améliorer les performances du slicing traditionnel par l'abstraction des chemins de données et par l'intégration du cheminement de contrôle. Abstraire le cheminement de données permet d'augmenter la vision de recherche du FS. L'utilisation du cheminement du contrôle est essentielle pour créer cette abstraction. Par ailleurs, l'utilisation du DCPG simplifie l'ensemble de retour en réunissant les instructions sous une même « fonctionnalité logique ».

En somme, le DCPG semble garder une bonne balance entre les résultats au niveau méthode et classe. Il améliore le *recall* FS* et la *precision* du CCG et FS au niveau des méthodes. Il possède aussi un meilleur *recall* que le FS* et FS au niveau des classes. Il est une bonne alternative au FS et FS* lorsqu'il est nécessaire d'abstraire l'information de dépendance de données entre les fonctionnalités d'un système. Il améliore son *recall* en sacrifiant un peu de *precision.* Le DCPG reprend le modèle d'analyse du CCG pour capturer le plus d'informations possibles en étant moins restrictif que les techniques de FS et essaie, en même temps, de bien contrôler la propagation de l'analyse avec l'abstraction des liens de dépendances de données. Le DCPG semble donc être une alternative viable et est un essai positif pour améliorer les métriques (*recall* et *precision*) qui en pratique, semblent s'exclure.

CONCLUSIONS

Le processus de maintenance des systèmes logiciels est complexe et coûteux. Les systèmes orientés objet n'échappent pas à cette règle. La nature même de ces systèmes, due aux dépendances multiples qui peuvent exister entre leurs constituants, peut rendre cette tâche très complexe. En utilisant des méthodologies et des techniques comme celles qui ont été abordées dans ce mémoire, les industries pourraient bénéficier d'actions plus sûres et plus rapides pour effectuer la mise à jour de leurs systèmes. Un développeur non renseigné sur les composants d'un système peut diriger sa recherche de modifications modulaires par des outils d'analyse d'impact.

Dans ce mémoire, nous avons présenté, dans un premier temps et comme partie principale du présent travail de recherche, une étude expérimentale comparative considérant trois approches statiques de l'analyse de l'impact: une technique basée sur les graphes d'appels traditionnels, une technique basée sur les graphes de contrôle réduits aux appels et une technique basée sur le slicing statique. La technique basée sur les graphes de contrôle réduits aux appels est une nouvelle technique que nous avons proposée. Elle est une nouvelle technique statique qui supporte l'analyse d'impact prédictive. Nous avons focalisé sur les programmes orientés objet, les programmes Java en particulier. Nous croyons par contre, que la technique peut s'adapter facilement au paradigme impératif. Elle utilise un nouveau modèle basé sur la réduction du flot de contrôle aux appels. Basé sur une analyse statique du code source d'un programme, elle permet de générer les cheminements de flot de contrôle réduit aux appels. Les chemins générés, dans une forme compactée, sont utilisés pour déterminer l'impact d'un changement. La technique permet de déterminer, d'une manière incrémentale, les composants pouvant être potentiellement affectés par un certain changement et devant être analysés (à des fins de test) de nouveau.

Dans ce contexte, une première expérimentation sur diverses versions de deux applications Java à été conduite. Ces deux applications ont subi plusieurs changements dans le temps. Par analyse de leurs versions successives, nous avons identifié les modifications effectuées dans leur code par comparaison binaire de deux versions différentes pour chacune des deux applications. L'étude a révélé la quantité d'informations renvoyées des approches utilisant les graphes d'appels traditionnels et ceux basées sur les graphes de contrôle réduits aux appels. L'expérimentation démontre l'inefficacité du graphe d'appel indirecte (CGi) à circonscrire un ensemble de méthodes affectées. De plus, elle semble indiquer que le graphe de contrôle réduit aux appels renvoie des méthodes plus précisément que les autres approches basées sur les graphes d'appels. L'expérimentation conduite (en premier lieu) n'a cependant pas évalué la qualité, en termes de précision entre autres, des résultats retournés par les approches considérées.

Une seconde expérimentation, plus approfondie, pour comparer les approches statiques, incluant le *forward slicing* traditionnel et le *forward slicing* directe, aux graphes de contrôle réduits aux appels a été effectuée. Cette seconde expérimentation étend la première en ajoutant une notion de granularité classe en plus de la granularité méthode. De plus, son objectif était de comparer les résultats obtenus des techniques avec les résultats réels observés. Ces derniers ont été manuellement identifiés. Cette seconde expérimentation démontre que le CCG retourne des ensembles moins précis que le FS, mais plus susceptibles de contenir des méthodes réellement affectées. Sa perte de précision est due à la capture plus large de liens directs. De plus, l'expérimentation semble indiquer que les liens directs entre méthodes seraient de meilleurs indicateurs d'impacts.

Une première tentative d'unification du modèle basé sur les chemins de contrôle et du modèle basé sur les chemins de données (slicing) à été effectuée. Ce modèle, appelé, *Data Control Path Graph*, se base sur les chemins (*paths*) de contrôles et sur la

réduction des instructions en les regroupant en un ensemble entre chaque *path*. Les *paths* sont augmentés de liens de cheminement des données. La technique DCPG reprend la méthodologie d'analyse itérative et les ensembles définis pour le CCG. En réunissant les instructions et en abstrayant les cheminements de données, le DCPG améliore substantiellement les performances du FS. Lors de la seconde expérimentation, le DCPG a été comparé aux trois autres approches selon les mêmes critères d'évaluation. Les résultats de l'expérimentation démontrent que l'abstraction des instructions et de leurs cheminements de données améliore le *recall* du DCPG mais réduit la *precision*. Le DCPG est une bonne alternative aux techniques de *slice* traditionnel lorsqu'il n'est pas nécessaire de connaître en détail les éléments objets impactés. Il performe mieux que le FS et possède un meilleur *recall* que le FS direct au niveau des classes. L'analyse ressort aussi, au niveau des classes, que le FS direct performe semblablement sinon mieux que le FS traditionnel, ce qui alimente l'importance du direct pour déterminer les classes.

Finalement, les résultats concernant la vitesse de calcul des analyses ont démontré que les techniques se limitant aux liens directs, calculent plus rapidement les résultats, en millième de seconde, comparé à une dizaine de seconde pour l'indirect. De plus, leur consommation de mémoire en est diminuée.

RECOMMENDATIONS

Cette étude s'est basée en grande partie sur les dépendances entre éléments des systèmes orientés-objet, à partir du cheminement de contrôles et de données. Notre étude nous a permis, par ailleurs, de nous rendre compte du manque crucial d'outils pour bien conserver (et gérer) l'information lors des modifications successives d'un système. À ce jour, il est impossible de déterminer adéquatement les étapes de modifications, qui ont eu lieu, dans un système avec une granularité plus précise que le fichier. Les relations existantes entres ces étapes est essentielle pour connaître statistiquement l'importance des dépendances inter modulaires.

ANNEXE 1

```
/**
     * This function will create a path matrix from an adjacency matrix. The path matrix
     * is the result of the of Warshall-Floyd algorithm.
     *
     * @param adjacencyMatrix :
     *              A matrix fill with 0 and 1 values.
     * @return int[][] : A matrix having the path of the adjMatrix. Contain 0 and 1
values.
     */
    public static int[][] get_WarshallFloyd_PathMatrix(int[][] adjacencyMatrix) {

        //Grosseur max du tableau carré
        int max = adjacencyMatrix.length;

        //Copie de la matrice initiale
        int[][] pathMatrix = new int[max][max];
        for(int i=0; i<max; i++)
            for(int j=0; j<max; j++)
                pathMatrix[i][j] = adjacencyMatrix[i][j];


        //Algorithme de Warshall-Floyd prit dans la litérature.
        int n = 0;
        while(n < max){

            for(int i=0; i<max; i++)
                for(int j=0; j<max; j++)
                        if((pathMatrix[i][n] == 1) && (pathMatrix[n][j] == 1))
                            pathMatrix[i][j] = 1;
        }

        return pathMatrix;
    }
```

BIBLIOGRAPHIE

[Badri 05] Badri, L., Badri, M., and St-Yves, D. 2005. Supporting Predictive Change Impact Analysis: A Control Call Graph Based Technique. In Proceedings of the 12th Asia-Pacific Software Engineering Conference (Apsec'05) - Volume 00 (December 15 - 17, 2005). APSEC. IEEE Computer Society, Washington, DC, 167-175.

[Badri 07-a] L. Badri, M. Badri & Daniel St-Yves: Analyse de l'impact des changements: Une étude expérimentale sur deux approches statiques, in Revue Génie Logiciel, Paris, Numéro 82, Septembre 2007.

[Badri 07-b] L. Badri, M. Badri & Daniel St-Yves: Static Impact Analysis for Object-Oriented Programs: An experimental comparison on reduced-call-path and slicing based approaches, in Proceedings of the 20th International Conference on Software and Systems Engineering and their Applications, Paris, Décembre 2007.

[Barros 95] S. Barros, Th. Bodhun, A. Escudie, J.P. Voidrot, Supporting Impact Analysis : A semi automated technique and associated tool. Proc. of the 1995 IEEE Conf. on Software Maintenance, pp. 42-51, Piscataway, NJ, 1995.

[Basil 01] Sarita Basil and Rudolf K. Keller, Software Visualization Tools: Survey and Analysis 2001.

[Bishop 04] Luke Bishop, Incremental impact analysis for object-oriented software, Master Thesis, Iowa State University 2004.

[Black 01] Black, S. 2001. Computing ripple effect for software maintenance. Journal of Software Maintenance 13, 4 (Sep. 2001), 263.

[Black 05] Black S.E. and Rosner P.E. Measuring Ripple Effect for the ObjectOriented Paradigm [Conference] // IASTED International Conference on Software Engineering, 15th-17th. - Austria : [s.n.], February 2005.

[Briand 99] Briand, L.C., Wust, J., Lounis, H., Using coupling measurement for impact analysis in object-oriented systems. Proc. of the IEEE International Conf. on Software Maintenace (ICSM '99), 30 Aug.-3 Sept. 1999, pp.475 – 482.

[Bohner 96] S.A. Bohner and R. Arnold, Software Change Impact Analysis. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[Bohner 96-2] S.A. Bohner, Impact Analysis in the Software Change Process : A Year 2002 Perspective. Proc. of the International Conf. on Software Maintenance, 4-8 Nov. 1996, pp.42 – 51.

[Codesurfer] CodeSurfer. *CodeSurfer*, [En ligne]. http://www.grammatech.com/products/codesurfer/overview.html (Consulté le 19 octobre 2007).

[Danicic 99] Sebastian Danicic, Dataflow Minimal Slicing, PhD. Thesis. 1999, University of North London.

[Eclipse] Eclipse. *Eclipse - an open development platform*, [En ligne]. http://www.eclipse.org (Consulté le 19 octobre 2007).

[Gallagher 04] K. B. Gallagher, "Some Notes on Interprocedural Program Slicing," scam, pp. 36-42, Source Code Analysis and Manipulation, Fourth IEEE International Workshop on (SCAM'04), 2004

[Haider 05] H Bilal, S Black, Using the Ripple Effect to Measure Software Quality, Software Quality Management-International Conference, 2005

[Han 97] J. Han. Supporting Impact Analysis and Change Propagation in Software Engineering Environments, Proceedings. In Proc. of the 8th Intl. Workshop on Software Technology and Engineering Practice (STEP'97), London, England, pp. 172-182, July 1997.

[Hassan 04] Hassan, A.E.; Holt, R.C., "Predicting change propagation in software systems," Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on , vol., no.pp. 284- 293, 11-14 Sept. 2004

[Horwitz 04] S. Horwitz, T. Reps, and W. Binkley D, Interprocedural slicing using dependence graphs. ACM SIGPLAN Notices, Vol. 39, Issue 4, April 2004.

[Jiang 91] J. Jiang, X. Zhou, and D. J. Robson, Program slicing for C, The problems in implementation. IEEE Inter. Conference on Software Maintenance, 1991.

[Jmol] JMol. *Jmol: an open-source Java viewer for chemical structures in 3D*, [En ligne]. http://jmol.sourceforge.net/ (Consulté le 19 octobre 2007).

[Kaveri] Kaveri. *Indus - Kaveri*, [En ligne]. http://projects.cis.ksu.edu/docman/view.php/12/90/kaveri-ug.pdf (Consulté le 19 octobre 2007).

[Korel 90] B. Korel and J. Laski, Dynamic slicing in computer programs. Journal of Systems Software, 13(3): 187-195, 1990.

[Kung 94] Kung, D. C., Gao, J., Hsia, P., Wen, F., Toyoshima, Y., and Chen, C., Change Impact identification in object-oriented software maintenance. Proc. of the International Conf. on Software Maintenance, pp. 202-211, 1994.

[Law 03] J. Law, G. Rothermel, Whole Program Path-Based Dynamic Impact Analysis. Proc. of the International Conf. on Software Engineering, pp. 308-318, 2003.

[Lee 00] Michelle Lee, A. Jefferson Offutt and Roger T. Alexander, Algorithmic Analysis of the Impacts of Changes to Object-Oriented Software. IEEE, pp. 61-70, 2000.

[Li 95] Wei Li and Sallie Henry, Maintenance support for object-oriented programs. The Journal of Software Maintenance, Research and Practice, 7(2):131-147, March-April 1995.

[Li 96] L. Li and A. J. Offutt, Algorithmic analysis of the impact of changes to object-oriented software. Proc. of the IEEE International Conf. on Software Maintenance, CA, USA, pp 171-184, 1996.

[Lucia 01] De Lucia, A. Program slicing: Methods and applications. In 1 st IEEE International Workshop on Source Code Analysis and Manipulation (Florence, Italy, 2001), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 142-149.

[Lustman 02] M. Ajmal Chaumun, Hind Kabaili, Rudolf K. Keller and F. Lustman, A change impact model for changeability assessment in object-oriented software systems, Science of Computer Programming, Volume 45, Issues 2-3, , November-December 2002, Pages 155-174.

[Orso 03] A. Orso, T. Apiwattanapong, and M.J. Harrold, Leveraging field data for impact analysis and regression testing. Proc. of European Software Engineering Conf. And ACM SIGSOFT Symp. On the foundations of software Engineering (ESEC/FSE'03), Helsinki, Finland, Sept. 2003.

[Orso 04] A. Orso, T. Apiwattanapong, J.Law, G. Rothermel, and M.J. Harrold, An Empirical Comparison of Dynamic Impact Analysis Algorithms. Proc. of the International Conf. on Software Engineering (ICSE'04), , pp. 491-500, Edinburg, Scotland, 2004.

[Ottenstein 84] L. M. Ottenstein and K. J. Ottenstein, The program dependence graph in software development environments. SIGPLAN Notices, Vol.19, pp. 177-184, 1984.

[Rajlich 00] Rajlich, V. 2000. Modeling software evolution by evolving interoperation graphs. Ann. Softw. Eng. 9, 1-4 (Jan. 2000), 235-248.

[Rajlich 05] Jonathan Buckner, Joseph Buchta, Maksym Petrenko, Vaclav Rajlich, "JRipples: A Tool for Program Comprehension during Incremental Change," iwpc, pp. 149-152, 13th International Workshop on Program Comprehension (IWPC'05), 2005.

[Ren 04] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley, Chianti: A Tool for Change Impact Analysis of Java Programs. OOPSLA'04, Vancouver, British Columbia, Canada, Oct. 24-28, 2004.

[Ryder 01] Barbara G. Ryder and Frank Tip, Change Impact Analysis for object-Oriented Programs. In ACM SIGPLAN-SIGSOFT Worshop on Program Analysis for Software Tools and Engineering, pages 46-53. ACM Press, 2001.

[St-Yves 06] D. St-Yves, L. Badri & M. Badri : Analyse prédictive de l'impact des changements, Concours des affiches scientifiques, Université du Québec à Trois-Rivières, Trois-Rivières, Avril 2006. Ce poster a reçu le prix de la deuxième meilleure affiche du département de mathématiques et d'informatique (UQTR).

[Sommerville 04] Ian Sommerville, Software Engineering, Seventh edition, Pearson, Addison Wesley, 2004.

[SourceForge] SourceForge. *SourceForge.net*, [En ligne]. http://sourceforge.net/ (Consulté le 19 octobre 2007).

[Tip 94] F. Tip, A survey of program slicing techniques. Journal of Programming Language, vol. 3, pp.121-189, 1995.

[Tsantalis 05] Tsantalis, N.; Chatzigeorgiou, A.; Stephanides, G., "Predicting the probability of change in object-oriented systems," Software Engineering, IEEE Transactions on , vol.31, no.7pp. 601- 614, July 2005

[Turver 94] Richard J.Turver and Munro Malcom, An early impact analysis technique for software maintenance. The Journal of Software Maintenance, Research and Practice, 18(12):35-52, January-February 1994.

[Unravel] Unravel. *The Unravel Project*, [En ligne]. http://www.itl.nist.gov/div897/sqg/unravel/unravel.html (Consulté le 19 octobre 2007).

[Wang 96] Yamin Wang and Wei-Tek Tsai and Xiaoping Chen and Sanjai Rayadurgam 1996. The Role of Program Slicing in Ripple Effect Analysis. SEKE, p. 369-376

[Warren 75] Warren, H. S. (1975). A modification of Warshall's algorithm for the transitive closure of binary relations. G. Manacher, IBM Tomas J. Watson Research Center.

[Weiser 79] M. Weiser, Program slices: formal, psychological, and pratical investigations of program abstraction method. PhD thesis, University of Michigan, Ann Arbor, 1979.

[Yau 78] Yau, S.S.; Collofello, J.S.; MacGregor, T., "Ripple effect analysis of software maintenance," Computer Software and Applications Conference, 1978. COMPSAC '78. The IEEE Computer Society's Second International , vol., no.pp. 60- 65, 1978

[Yau 80] S.S. Yau , J. S. Collofello, Some Stability Measures for software maintenance. IEEE Transactions on Software Engineering, 6(6): pp. 545-552, November 1980.

[Zhang 07] Zhang, X., Gupta, N., and Gupta, R. 2007. A study of effectiveness of dynamic slicing in locating real faults. Empirical Softw. Engg. 12, 2 (Apr. 2007), 143-160.