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
NICOLAS FRÉCHETTE

TESTS DE RÉGRESSION DANS LES SYSTÈMES ORIENTÉS OBJET : UNE APPROCHE STATIQUE BASÉE SUR LE CODE

LE 20 AVRIL 2010

TESTS DE RÉGRESSION DANS LES SYSTÈMES ORIENTÉS OBJET : UNE APPROCHE STATIQUE BASÉE SUR LE CODE

Nicolas Fréchette

SOMMAIRE

Le test de régression est un processus utilisé pour déterminer si un programme modifié rencontre toujours ses spécifications ou si de nouvelles erreurs ont été introduites au niveau des fonctionnalités qui existaient déjà avant les modifications. Pour réduire le coût, nous procèdons en général, de façon conservatrice, à une sélection parmi les tests existants. Une approche conservatrice sélectionne tous les tests susceptibles de démontrer une faute dans la version modifiée du programme P. En d'autres mots, l'approche n'éliminera jamais de tests qui pourraient révéler des erreurs découlant des modifications. L'exécution des tests sélectionnés par une approche conservatrice assure, en fait, le même niveau de confiance que si l'intégralité des tests avait été exécutée. Par ailleurs, puisque les tests existants ne sont pas en mesure de couvrir la totalité des chemins d'exécution, en particulier ceux engendrés par les ajouts/modifications/retraits de code, nous devons aussi générer de nouveaux cas de tests.

Ce mémoire présente une approche intégrée permettant dans un premier temps, la sélection de tests de régression de façon conservatrice et, dans un second temps, la génération de nouveaux cas de tests pour les chemins non couverts par les tests existants. Cette principale caractéristique distingue notre approche des approches similaires présentées dans la littérature. Elle détermine la couverture à effectuer à partir de l'ensemble des chemins d'exécution possibles du programme modifié. Les approches existantes considèrent uniquement les chemins d'exécution couverts par la suite de tests originale. Cette caractéristique permet de gérer certains éléments qui ne sont pas supportés par les autres approches: amélioration d'un ensemble de tests de départ qui seraient déficients et couverture des nouveaux chemins d'exécution possibles suite aux

modifications. L'approche définie présente aussi l'avantage important de ne requérir aucune instrumentation du code.

L'implémentation de l'approche, des exemples théoriques ainsi qu'une étude de cas concrète sont présentés dans ce mémoire. Les résultats obtenus sont aussi évalués grâce à certaines métriques. L'étude réalisée permet de constater que l'approche proposée permet d'obtenir des résultats égaux ou supérieurs à ceux obtenus avec d'autres approches.

REGRESSION TEST SELECTION IN OBJECT-ORIENTED SYSTEMS: A STATIC CODE BASED APPROACH

Nicolas Fréchette

ABSTRACT

Regression testing is a testing process that is used to determine if a modified program still meets its specifications or if new errors have been introduced on functionalities that already existed before the changes. To reduce the cost of regression testing, we do safe tests selection among existing tests suit. A safe technique selects all tests that can reveal faults in the modified version of the program and ensures the same level of confidence as the execution of all tests of the original tests suit. Moreover, because existing tests can't cover all paths due to additions/changes/withdrawals of code, our approach also generates new test cases.

The difference between our approach and other similar techniques in the literature is the way used to determine coverage that should be done. Required coverage is determined from all possible execution paths of the modified version of the program. Existing techniques only consider execution paths that are covered by the original tests suit. This characteristic allows our approach to handle certain situations that are not supported by others similar approaches, improves and completes original tests suit or cover new execution paths generated by code changes. The defined approach presents also an important advantage knowing that it does not require code instrumentation.

The implementation of the proposed technique, some simple examples and a real case study are presented in this work. Obtained results are also evaluated using some metrics. The study demonstrates that in certain situations, our approach allows to obtain equal or better results than those obtained with other similar approaches.

REMERCIEMENTS

Le présent projet n'aurait pas été possible sans la contribution et le support de plusieurs personnes. À leur façon, ils m'ont permis de progresser à travers les différentes étapes de ce travail.

Je tiens d'abord à remercier M. Mourad Badri et Mme Linda Badri, professeurs au Département de Mathématiques et d'Informatique de l'UQTR, qui m'ont dirigé et orienté tout au long de ce travail. Leur expertise et leur disponibilité ont été d'une grande aide.

Je remercie M. Daniel St-Yves, M. Fadel Toure et M. Pierre-Luc Vincent pour leurs contributions respectives. Leurs apports ont été d'une grande utilité.

Je remercie finalement mon employeur, Cogeco Cable Inc., pour son support et sa flexibilité ainsi que ma famille, ma conjointe et mes amis pour leur soutien inconditionnel. Leur support a été apprécié tout au long de ce projet.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

INTRODUCTION

Tests de régression

Avec le temps, le coût associé à la maintenance d'un logiciel représente une part prédominante du coût global de développement [Chen 94, Harrold 97]. Dans les années 90, les dépenses associées à la maintenance ont été estimées à plus de 70% du coût total des logiciels [Harrold 97]. Un grand pourcentage de cette dépense en maintenance est consacré aux tests. Les tests de régression sont utilisés pour déterminer si un programme modifié rencontre encore ces spécifications ou si de nouvelles erreurs ont été introduites. Nous nous assurons que les changements apportés n'ont pas affecté les fonctionnalités existantes.

L'amélioration du processus de tests de régression a pour but de réduire les coûts liés à la maintenance. Cependant, les considérations économiques ne sont pas les seules raisons pour améliorer ce processus. Les tests de régression apportent un niveau de confiance au bon fonctionnement du logiciel et améliorent sa fiabilité [Li 99]. Ils impliquent un compromis entre le coût de réexécution des tests et le risque d'éviter des erreurs introduites par les effets secondaires des modifications apportées au logiciel [Skoglund 09]. Les tests de régression sont très importants. Selon Onoma & al. [Onoma 98], le secret dans la livraison d'un logiciel de qualité est de faire de bons tests de régression. Plusieurs types d'approches de tests de régression ont été présentés dans la littérature : sélection, réduction, priorisation, génération, exécution, etc. [Skoglund 09].

Problématique

Plusieurs études ont démontré l'avantage de réutiliser les suites de test existantes dans le but de diminuer les efforts de tests requis. Comme il est très long et coûteux de ré-exécuter de façon systématique tous les tests de la suite originale, la question est de savoir quels tests exactement devons-nous ré-exécuter en cas de modifications. Dans cette optique, plusieurs approches sélectives de tests de régression ont été proposées dans

la littérature pour sélectionner, parmi les suites de test existantes, les tests qui peuvent démontrer des erreurs dans la version modifiée du programme.

Puisque les modifications effectuées peuvent introduire de nouveaux chemins d'exécution (modifier éventuellement la structure de ceux qui existent), l'ensemble original de tests ne suffit habituellement pas à couvrir l'intégralité des chemins d'exécution possibles de la version modifiée du programme. Par exemple, dans le cas d'ajout de nouvelles fonctionnalités, les cas de tests de la version originale du programme ne contiennent aucun test relatif à celles-ci. Dans ce contexte, de nouveaux cas de tests doivent être construits et ajoutés à la suite originale. Les approches dites sélectives qui existent dans la littérature ne couvrent pas à cet aspect important de la problématique.

Objectifs

L'approche que nous préconisons pour les tests de régression s'intéresse non seulement à la sélection de tests de régression à partir d'un ensemble de tests existants, mais également à la génération de nouveaux cas de tests afin de couvrir les nouveaux chemins d'exécution de la version modifiée du programme, qui ne sont pas couverts par les tests existants.

Organisation

Ce mémoire est composé de cinq chapitres. Le chapitre 1 résume l'état de l'art des travaux portant sur les tests de régression basés sur le code. Le chapitre 2 présente certains concepts de base qui sont à l'origine des présents travaux. Le chapitre 3 présente la méthodologie de l'approche proposée. Le chapitre 4 présente des exemples théoriques d'application de notre approche ainsi qu'une comparaison avec des approches similaires de la littérature. Le chapitre 5 décrit l'outil implémenté supportant notre approche. Le chapitre 6 présente les métriques utilisées pour évaluer l'approche ainsi qu'une étude de cas réelle. En conclusion, un résumé du travail réalisé est présenté et des suggestions de travaux futurs sont proposées.

CHAPITRE 1

ÉTAT DE L'ART

1.1 Introduction

Aujourd'hui, la plupart des systèmes sont développés de façon évolutive [Rajlich 00, Runeson 03]. L'automatisation des tests est un domaine qui bénéficie spécifiquement des approches évolutives puisque celles-ci permettent la réutilisation des investissements réalisés dans les programmes de test et les environnements. De cette façon, des coûts plus élevés de développement lié au soutien à l'automatisation des tests peuvent être justifiés et une augmentation de la qualité des logiciels est atteinte [Skoglund 04].

Les tests de régression constituent une activité de maintenance importante du cycle de vie du logiciel qui a reçu une attention considérable ces dernier temps [Li 99]. Bien que les tests de régression soient souvent considérés comme étant une activité de maintenance, ils devraient être effectués en fait chaque fois qu'il y a un changement de code ou de spécification (durant le processus de développement). Les tests de régression ne doivent pas être limités uniquement à la phase de maintenance du cycle de vie du logiciel [Li 99].

Trois types de tests peuvent être effectués durant le cycle de vie d'un logiciel : les tests unitaires, les tests d'intégration et les tests de système. Ils révèlent tous différents types de fautes. Les tests de régression peuvent être appliqués à chacun de ses niveaux [Li 99].

1.2 Approche unitaires

Plusieurs techniques s'intéressent aux tests unitaires. Gupta et al. [Harrold 96] ont proposé une approche de tests de régression unitaire, basée sur les flux de données dans un programme, utilisant le "program slicing". Comme les approches traditionnelles

basées sur les flux de données, celle-ci détermine les associations "definition-use" (définition-utilisation/affectation) et utilise certains critères de pertinence afin de sélectionner les associations à tester [Harrold 96]. Dans un graphe de flux de contrôle (CFG), chaque nœud correspond à une instruction et chaque arc représente le flux de contrôle entre les instructions du programme [Harrold 96]. Les définitions et les utilisations de variables sont attachées aux nœuds du graphe de flux de contrôle [Harrold 96]. Les associations "definition-use" sont représentées par le trio (s,u,v) dans lequel v représente la valeur de la variable définie dans l'instruction s qui est utilisée dans l'instruction ou l'arc u [Harrold 96]. La technique sélectionne deux types d'association "definition-use" : les associations affectées directement en raison de suppression/insertion de définitions et les associations affectées indirectement en raison d'un changement d'une valeur calculée ou d'un chemin conditionnel.

L'approche applique les algorithmes "ForwardWalk" et "BackwardWalk" sur les CFGs pour sélectionner les associations à tester [Harrold 96]. L'algorithme "BackwardWalk" (figure 1) parcours le CFG du programme original par derrière, à partir du point d'édition (endroit de la modification), à la recherche de définitions reliées a l'instruction modifiée [Harrold 96].

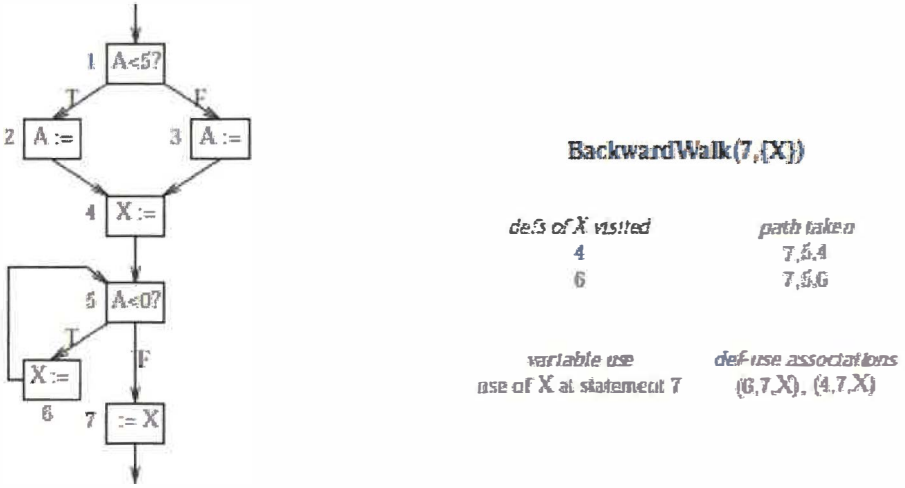
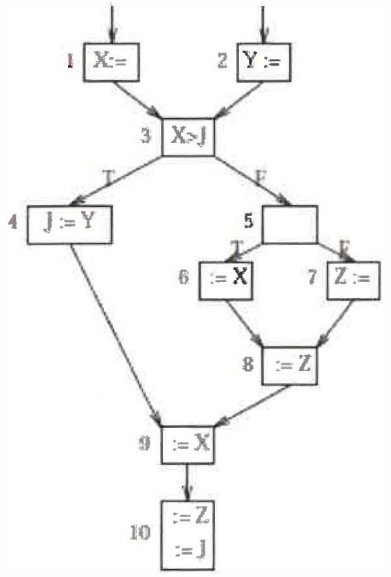


Figure 1. Exemple de "BackwardWalk" [Harrold 96].

L'algorithme "ForwardWalk" (figure 2) parcours le CFG du programme original par devant, à partir du point d'édition, à la recherche de définitions et d'utilisations affectées par le résultat de l'édition (le changement) [Harrold 96].



ForwardWalk({(2,Y)}, false)

variable	*def-use association*	*path taken*
Y	(2,4,Y)	2,3,4
J	(4,10,J)	4,9,10

ForwardWalk({(3,X),(3,Y)}, true)

variable	*def-use association*	*path taken*
X	(1,6,X)	3,5,6
Y	(2,4,Y)	2,3,4
J	(4,10,J)	4,9,10
Z	(7,8,Z)	7,8
	(7,10,Z)	7,8,9,10

Figure 2. Exemple de "ForwardWalk" [Harrold 96].

Puisque l'algorithme basé sur le slicing identifie explicitement, de façon directe et indirecte, les associations affectées, les tests de régression requis pour tester l'association sont déterminés assez facilement [Harrold 96]. Rappelons que le test de régression est un processus utilisé pour déterminer si un programme modifié rencontre toujours ses spécifications ou si de nouvelles erreurs ont été introduites au niveau des fonctionnalités qui existaient déjà avant les modifications. En appliquant cette technique, il n'est pas requis de maintenir un historique de tests, ni de recalculer les flux de données pour permettre les tests de régression sélectifs [Harrold 96].

Korel et AI-Yami ont développé une approche pour automatiser la génération de tests de régression unitaires dans laquelle chaque cas de test généré détecte au moins une erreur [Korel 98]. L'approche est applicable lorsque les spécifications des fonctionnalités n'ont pas changées suite au changement. Ce résultat est atteint en utilisant la version originale du programme dans le processus de génération des données de test.

Spécifiquement, l'approche tente de générer automatiquement des données d'entrée pour lesquelles le programme original et sa version modifiée produisent des résultats différents. Si des données en entrée sont trouvées, une erreur est découverte car chaque version devrait produire le même résultat [Korel 98]. L'erreur peut être dans le programme original, dans le programme modifié ou dans les deux programmes.

La méthode présentée est intégrée à TESTGEN, un système de génération de données de test pour programmes Pascal. Le testeur fournit le code source du module original Mold et sa version modifiée Mnew [Korel 98]. Il doit aussi identifier une paire de paramètres de sortie équivalente dans Mold et Mnew. Supposons que (Yi, Zp) ... (Ys, Zt) sont les paires de paramètres identifiés par le testeur [Korel 98]. Basé sur cette information, un préprocesseur génère automatiquement un pilote (figure 3) qui, similairement à celui présenté ci-dessous (figure 3), exécute les modules Mold et Mnew avec le même input et compare les résultats obtenus avec les paramètres de sortie précédemment sélectionnés :

Generate input $x_1, x_2, .., x_k$
call M_{old} (**in** $x_1, x_2, .., x_k$, **out** $y_1, y_2, .., y_m$);
call M_{new} (**in** $x_1, x_2, .., x_k$, **out** $z_1, z_2, .., z_n$);
if $y_i \neq z_0$ **then Report_Error;**
....
if $y_s \neq z_t$ **then Report_Error;**

Figure 3. Pilote d'exécution [Korel 98].

Le but de l'approche est de trouver les éditions ayant pour conséquence que le module modifié se comporte différemment du module original [Korel 98]. Par conséquent, les méthodes existantes de génération automatique de données de test par "boite-blanche" peuvent être utilisées pour générer les tests. L'expérimentation des auteurs démontre que leur approche peut améliorer les chances de trouver des erreurs logicielles comparées aux autres approches existantes de tests de régression. L'avantage est que la méthode est entièrement automatisée et que chaque cas de test généré révèle une erreur. La figure 4 présente un exemple de cas de test généré par l'approche qui

consiste essentiellement en des paramètres d'entrée qui produiront des données en sortie différente sur Mold et Mnew.

```
Test #1: n=5, a=(6,2,-7,9,3)                          /* a typical valid case
Test #2: n=-4, a=0                                    /* a typical invalid case
Test #3: n=14, a=(2,-1,3,5,88,12,23,-3,16,22,14,-22,33,11)   /* a typical invalid case
Test #4: n-0, a=0                                     /* a boundary case
Test #5: n=1, a=(6)                                   /* a boundary case
Test #6: n=-1, a=0                                    /* a boundary case
Test #7: n=10, a=(6,2,-7,3,9,11,23,1,6,22)            /* a boundary case
Test #8: n=9, a=(7,5,-12,-5,3,11,25,2,5)              /* a boundary case
Test #9: n=11, a=(2,-1,3,5,88,-12,23,3,16,22,14)      /* a boundary case
Test #10: n=6, a=(9,8,7,6,2,-2)                       /* elements in descending order
Test #11: n=6, a=(-1,2,3,4,9,12)                      /* elements in ascending order
Test #12: n=5, a=(4,4,4,4,4)                          /* all elements equal
```

Figure 4. Exemple de tests générés par l'approche de Korel et al. [Korel 98].

Ball [Ball 98] propose une approche utilisant la forte relation entre la sélection de tests de régression basée sur les flux de contrôle (CRTS) et les automates finis déterministes (DFA) [Ball 98]. Nous observons qu'un graphe de contrôle (CFG) G peut être vu comme un DFA, ayant l'état de départ s et l'état final x, acceptant le langage L(G) qui constitue l'ensemble des chemins complets de G. Un CFG du programme original et celui de sa version modifiée sont générés et leur intersection est ensuite définie [Ball 98]. Puisque celle-ci capture précisément le but des CRTS, elle constitue la base d'une famille d'algorithmes paramétrés par l'information dynamique collectée à partir de l'exécution de la suite de tests originale sur la version originale du programme. La figure 5 présente le graphe d'intersection G'' généré à partir des CFG G et G' qui sont respectivement représentatifs de P et P' :

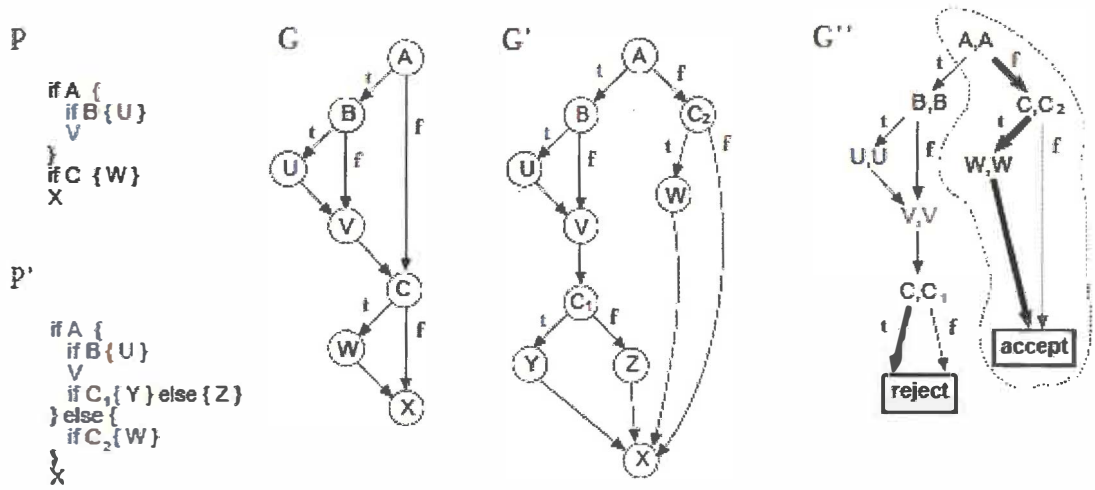


Figure 5. Graphe d'intersection [Ball 98].

L'auteur définit une condition d'optimalité forte (arc-optimalité) pour les algorithmes de CRTS et reformule l'algorithme de Rothermel & Harrold [Harrold 97] afin d'utiliser un graphe d'intersection plutôt que deux CFGs. Supposons Pp représentant l'ensemble des chemins, G le CFG du programme original, G' le CFG du programme modifié, I l'intersection de G et G'. Un algorithme de CRTS est arc-optimal si pour chaque chemin p, tel que Pp $\subseteq$ I(G, G'), l'algorithme signale que p est dans I(G, G'). Les 3 algorithmes découlant de cette théorie apportent des améliorations intéressantes. Le premier élimine tous les tests que l'algorithme de Rothermel & Harrold. Il élimine en général plus de tests, de façon conservatrice, pour le même coût. Les deux autres algorithmes sont plus précis que l'algorithme de Rothermel & Harrold mais avec un coût plus élevé [Ball 98].

1.3 Approches intégrées

Deux principales familles de techniques de tests de régression au niveau des tests d'intégration ont été présentées jusqu'à maintenant dans la littérature : les techniques basées "firewall" et les techniques basées "arcs dangereux" [Skoglund 09].

Le premier groupe de techniques est le groupe de techniques "firewall" dans lesquelles les dépendances des parties modifiées sont isolées dans un firewall [Harrold

96]. Les éléments inclus dans le firewall peuvent être des éléments qui interagissent avec des éléments modifiés, des éléments qui sont des ancêtres directs d'éléments modifiés ou des éléments qui sont des descendants directes d'éléments modifiés. Les cas de tests couvrant les parties incluses dans le firewall sont sélectionnés dans cette approche pour être ré-exécutés. Les techniques de firewall sont simples et faciles à utiliser particulièrement lors de petits changements [Harrold 96]. Différents niveaux de granularité ont été utilisés tel que les dépendances entre les modules, les fonctions et les classes.

Leung & White introduisent le concept de firewall afin de supporter les tests de régression au niveau intégration [White 90]. Pour un ou des modules dans lesquels des changements ont été effectués, le firewall inclue l'ensemble des modules qui doivent être re-testés [White 90]. Le concept appliqué est que lorsqu'une erreur est détectée, le changement doit être fait, si possible, de façon à éviter que le Firewall soit étendu par l'inclusion d'autres modules [White 90]. L'analyse des dépendances entre les modules dans les graphes d'appels s'effectue en considérant 3 cas de base [White 90]. Les possibilités pour un module sont les suivantes : pas de changement dans a, changement de code dans a et changement dans les spécifications de a [White 90].

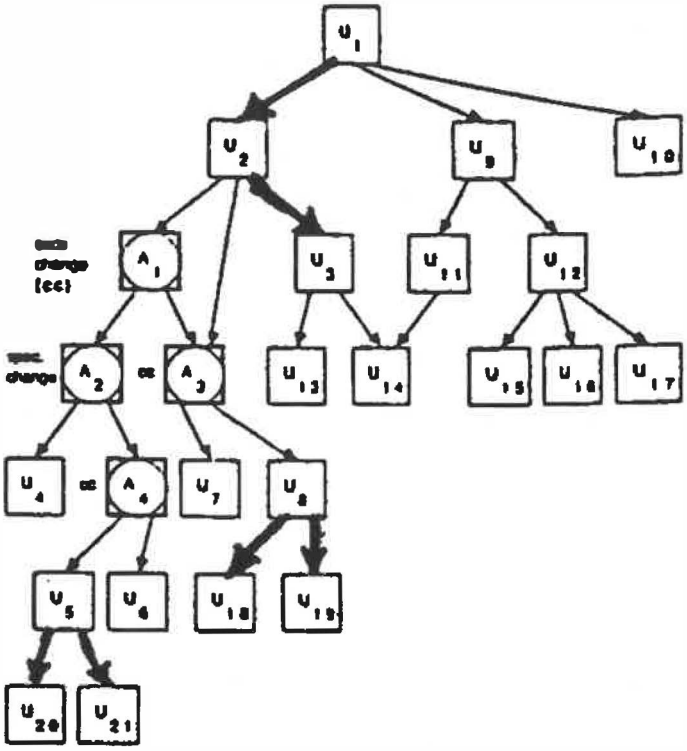


Figure 6. Exemple d'utilisation des cas de bases et du calcul du firewall [White 90].

Dans l'exemple présenté ci-haut (figure 6), les modules A1, A2, A3 et A4 ont été modifiés [White 90]. Les tests (U2-A1), (U2-A3), (U2-A4), (A3-U7), (A3-U8), (A4-U5), (A4-U6) doivent être ré-exécutés [White 90]. Le firewall est représenté par les flèches foncées qui séparent les modules modifiés du reste du graphe d'appels [White 90].

Leung & White présentent une méthodologie de tests de régression qui peut être utilisée aussi bien pour les tests d'intégration que pour les tests de système [White 92]. La technique implique les tests de régression d'un module dans laquelle sont considérées les dépendances dues au flux de contrôle et au flux de données [White 92]. Les dépendances de flux de contrôle sont modélisées à travers des graphes d'appels et un firewall est défini pour inclure tous les modules affectés qui doivent être re-testés [White 90, White 92]. L'approche définit aussi un firewall des flux de données affectées par les modules modifiés [White 92].

Dans cette méthodologie, un module est considéré comme étant une unité de programme tel que des procédures, routines, packages dont les paramètres d'entrée et de sortie sont clairement identifiés [White 92]. Ils utilisent en entrée les éléments suivants : les modules modifiés, la correspondance entre les spécifications fonctionnelles et les tests fonctionnels, les graphes d'appels (statiques) des modules et la matrice module/test [White 92]. La matrice module/test est obtenue dynamiquement et représente les modules rencontrés par chaque cas de test disponibles pour tester une fonctionnalité. Cette matrice enregistre aussi quels modules appellent un module donné, c'est à dire les liens du graphe d'appel impliqués.

Lorsqu'un changement est effectué au niveau du logiciel, un sous ensemble des données de test est automatiquement sélectionné et exécuté en tant que tests de régression. Ce sous-ensemble est déterminé en examinant la matrice module/test pour chaque test impliquant des modules modifiés [White 92]. En utilisant la correspondance entre les spécifications fonctionnelles et les tests, ainsi que les tests de régression identifiés précédemment, les fonctionnalités modifiées qui sont associés à ces tests peuvent être identifiées [White 92]. L'utilisation des modules inclus dans le firewall des modules modifiés, pour chaque lien entre un module et un module modifié, permet de déterminer combien de tests sont traversés par ces liens [White 92]. S'il y a des cas où un lien dans le firewall n'est pas exécuté, il s'agit d'une indication que d'autres tests sont requis afin d'effectuer une couverture complète [White 92].

Pour ce qui est du firewall des flux de données, le firewall consiste, en gros, à l'ensemble des variables globales et de leurs données de tests associées qui sont affectées par les modules modifiés [White 92]. Il est requis de vérifier les éléments inclus dans le firewall afin de s'assurer que l'effet des changements effectués aux modules ne s'est pas étendu aux variables globales [White 92]. Cette approche systématique pour tester les variables globales constitue un complément au firewall qui permet de supporter les dépendances de flux de données [White 92]. Deux aspects sont considérés dans ce cas, c'est-à-dire qu'on s'intéresse aux modules qui définissent la valeur d'une variable globale et aux modules qui utilisent une variable globale (figure 7).

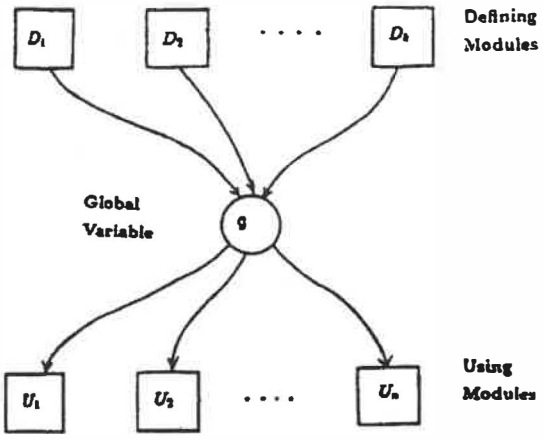


Figure 7. Définition et utilisation d'une variable globale g [White 92].

Supposons une variable globale g, la façon de procéder est de demander à un développeur expérimenté de générer des valeurs de g pour tester de façon fiable chaque module. Une première matrice sera ensuite générée. Il s'agit de la matrice d'utilisation de la variable globale (figure 8). Cette matrice garde la trace des valeurs de tests de g qui ont été utilisées pour tester chaque module.

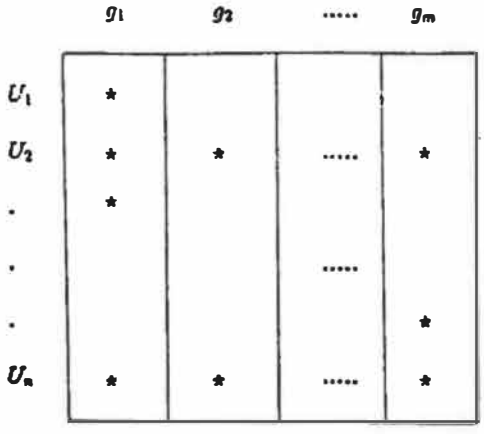


Figure 8. Exemple de matrice d'utilisation de la variable globale [White 92].

On effectue ensuite la construction d'une deuxième matrice à partir des données obtenues; la matrice de définition de la variable globale (figure 9) [White 92]. Cette

matrice enregistre les données en entrée envoyées aux différents modules pour obtenir une valeur de test spécifique de g [White 92].

D_1	Y_{11}	X	.....	Y_{m1}
D_2	X	Y_{22}	.....	Y_{m2}
.	.	.		.
.	.	.	.....	.
.	.	.		.
D_k	Y_{1k}	Y_{2k}	.....	Y_{mk}
	g_1	g_2	.....	g_m

Figure 9. Exemple de matrice de définition de la variable globale [White 92].

Ayant en main toutes ces données, il est maintenant possible d'évaluer l'impact des modules définissant et utilisant les variables globales. D'abord, si la matrice d'utilisation indique que la variable globale g pourrait être affectée par le changement effectué au module U, le testeur doit alors ré-exécuter les valeurs de test des g indiquées par la matrice. Ensuite, si la matrice de définition indique que la variable globale pourrait être affectée par un changement dans la définition D, le développeur doit alors exécuter la valeur de g indiquée par la matrice de définition.

Une autre approche présentée par White et al. [White 97] effectue la sélection de tests de régression pour les systèmes orientés objet. Cette technique, qui effectue la construction du firewall par analyse du modèle ORD (diagramme de relations entre les objets), constitue une extension du concept de firewall présenté précédemment. Contrairement au firewall présenté dans le cadre des programmes procéduraux, le firewall proposé ici est construit en termes de classes et d'objets. Lorsqu'un changement est effectué sur une classe ou un objet, les autres classes et objets qui interagissent avec elle doivent être contenus dans le firewall. Le firewall est construit de manière à accommoder les différences que présentent les relations classe-objet (héritage). La

précision de cette approche se limite aux dépendances entre classes. Ils sélectionnent les cas de tests correspondants aux classes impactées [White 97]. Les classes incluses dans le firewall, mais qui restent non-testées, indiquent que des tests supplémentaires doivent être générés [White 97].

Chen et al. [Chen 97] proposent aussi une méthode s'intéressant à la sélection des tests de régression pour les logiciels orientés objet. Puisque les relations impliquées dans les programmes orientés objet sont beaucoup plus complexes et difficiles à identifier que dans les paradigmes traditionnels, ce problème constitue un obstacle majeur au niveau des tests de régression [Chen 97]. La méthode est aussi basée sur le concept de firewall et sur le marquage de toutes les classes touchées par un cas de test [Chen 97]. À partir du firewall contenant les classes impactées, ils peuvent identifier toutes les classes affectées lorsqu'un programme est modifié [Chen 97]. Avec le marquage, ils peuvent aussi identifier tous les cas de test de la suite originale qui doivent être ré-exécutés suite au changement. Un processus, étape par étape, est proposé pour identifier les relations entre les classes et les cas de test, déterminer le firewall et sélectionner les cas de test à re-tester.

La technique de sélection de tests de régression à partir du firewall a été étendue par White et al. afin de répondre à la difficulté de tester les interfaces graphiques utilisateur (GUI) due au grand nombre d'états, d'entrées et d'évènements [White 03]. Un autre sérieux problème avec les GUI est que ce ne sont pas tous les effets générés par les tests qui sont observables [White 03]. En utilisant la fondation des séquences complètes d'interactions (CIS), définis lors de travaux précédents, ils proposent une nouvelle méthode de tests de régression des applications GUI [White 03]. Le CIS est une séquence d'objets GUI et de relations, qui collaborent pour produire une réponse à l'usager qui appelle la fonctionnalité [White 03]. Cette approche de tests de régression sélective consiste à tenir un niveau d'écart entre les objets GUI dépendants des objets GUI modifiés et le reste du code [White 03]. Les objets qui sont inclus dans le firewall doivent être re-testés [White 03]. L'étude de cas présentée démontre l'efficacité lors de son application sur des programmes complexes.

Skoglund & Runeson présentent une évaluation empirique de la combinaison de la technique de sélection de tests de régression par firewall et d'une technique de test de scénario (use-case), qui s'appuie sur l'analyse du « byte code » Java, dans le cadre de tests de régression des systèmes industriels distribués à grande échelle [Skoglund 05]. L'étude a été effectuée sur un logiciel distribué complexe appartenant à une des plus grandes banques de Suède. Les effets de l'utilisation de scénarios de test avec la sélection de tests de régression sont présentés dans [Skoglund 05]. La figure 10 présente un diagramme de dépendances de classes avec les cas de tests et le firewall déterminé après la modification de la classe D [Skoglund 05]. Dans cet exemple, les cas de tests TC1 et TC2 doivent être ré-exécutés.

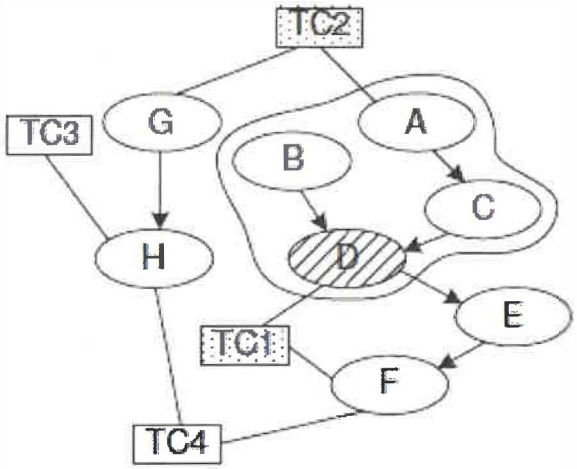


Figure 10. Diagramme de dépendances de classes, cas de tests et firewall [Skoglund 05].

Étant donnée l'ampleur du système, des outils (composants) ont été conçus ou récupérés afin d'effectuer les tâches suivantes : extraire les dépendances de toutes les classes du système, identifier les différences entre les versions du système afin de déterminer les changements et déterminer quels cas de test utilisent quelle classe [Skoglund 05]. Les cas de test sont définis en tant que scénarios pour chaque application. Chaque scénario contient une série de tâches qui doivent être effectuées par le testeur pour valider les fonctionnalités de l'application [Skoglund 05]. Chaque tâche se compose d'un certain nombre de cas de tests exécutés l'un après l'autre. Les scénarios contiennent

des cas de test qui peuvent être dépendants ou indépendants. Afin d'organiser les cas de test et les tâches, un graphe dirigé est construit [Skoglund 05]. La procédure consiste à collecter les dépendances du programme p, exécuter les tests existants sur p afin de collecter les classes utilisées pas les cas de test, générer les signatures MD5 des classes de p et p', comparer les signatures MD5 pour produire la liste des changements et finalement extraire les tâches (cas de test) qui utilisent des classes modifiées ou celles qui en dépendent [Skoglund 05]. L'algorithme MD5, pour Message Digest 5, est une fonction de hachage cryptographique qui permet d'obtenir l'empreinte numérique d'un fichier. La signature MD5 d'une classe est générée à partir du contenu du fichier de la classe et contient une emprunte numérique de 128 bit du fichier [Skoglund 05]. Puisqu'une tâche contient plusieurs cas de test, si au moins un cas de test sélectionné est inclus dans une tâche, la tâche doit être ré-exécutée [Skoglund 05]. Le résultat de l'utilisation de la technique de sélection par firewall est que seulement une portion des tests est sélectionnée [Skoglund 05]. Ils démontrent que l'utilisation des scénarios de tests, dans lesquels les cas de test sont dépendants, affecte le nombre de cas de test sélectionnés, autant que la localisation et le nombre de changements effectués dans le système [Skoglund 05].

Le deuxième groupe de techniques est basé sur une approche proposée par Harrold & al. pour les langages procéduraux connue sous le nom de "DejaVu1" (niveau unitaire) et "DejaVu2" (niveau intégré) [Harrold 97]. Le premier algorithme constitue une approche intra-procédurale tandis que le deuxième constitue une approche inter-procédurale. Les deux algorithmes utilisent les graphes de flux de contrôle (CFG) pour sélectionner des tests parmi la suite originale. Le CFG d'une méthode P contient un nœud pour chaque instruction simple ou conditionnelle de la méthode et les arcs (chemins) entre les nœuds représentent le flux de contrôle entre les instructions (figure 11).

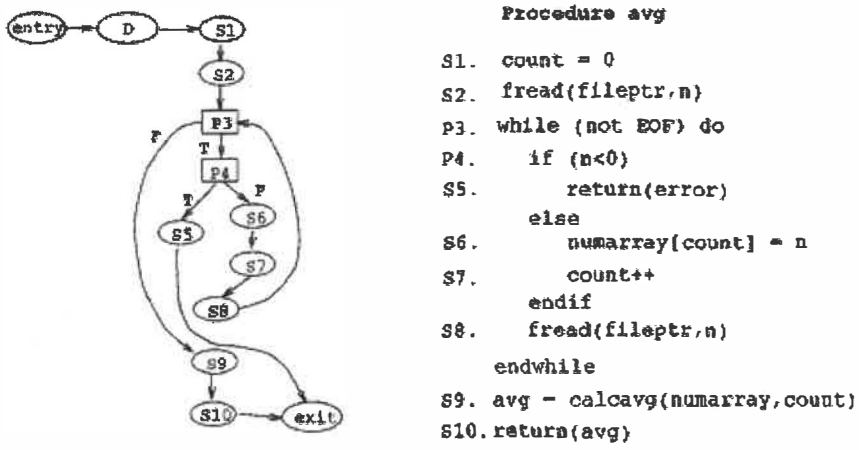


Figure 11. CFG de la procédure avg [Harrold 97].

"DejaVu1" et "DejaVu2" ne requièrent pas de connaître les endroits où le code a été modifié ce qui diminue le coût des tests de régression [Harrold 97]. "DejaVu2" génère de façon automatique un CFG du programme et de sa version modifiée. L'algorithme maintient aussi une table afin de conserver l'historique à savoir quels arcs (liens entre chaque instruction du code) du programme original sont impliqués par chacun des cas de test [Harrold 97]. Le fonctionnement consiste à comparer les deux CFGs, nœud par nœud et arc par arc. Si une différence est constatée lors de la comparaison, les cas de test correspondants sont sélectionnés à partir de la table d'historique et seront ré-exécutés [Harrold 97]. Deux traces d'exécutions sont équivalentes si elles ont la même longueur et si le texte représentant leurs éléments est identique. Un cas de test t est dit "traversé de modification" si sa trace d'exécution est non équivalente pour p et p'. Les auteurs ont d'ailleurs prouvé que "DejaVu2" est un algorithme conservateur dans un contexte de tests de régression contrôlé, à savoir le programme modifié est testé dans les mêmes conditions que le programme original [Harrold 97]. Bien que cette approche conservatrice puisse sélectionner certains cas de tests qui peuvent être éliminés, elle a la particularité d'être plus précise que les approches similaires de la littérature [Harrold 97]. La figure 12 illustre l'idée générale derrière l'approche proposée.

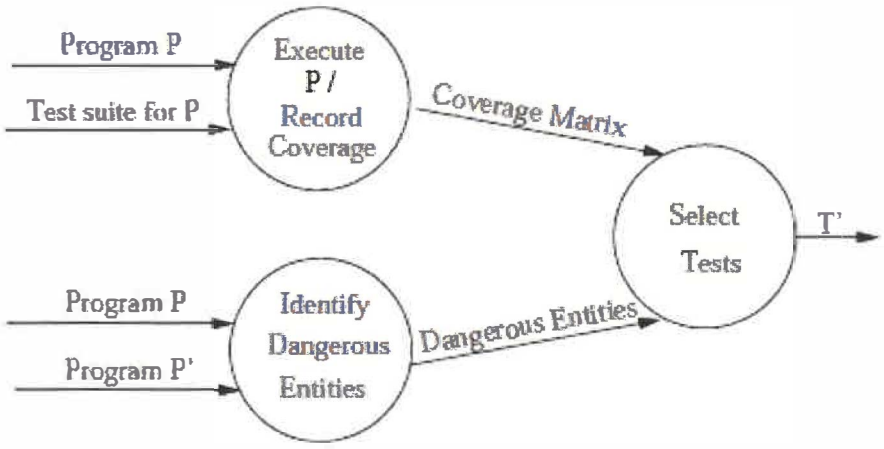


Figure 12. Système générale de sélection de tests de régression [Harrold 97].

Les concepts orientés objet tels que l'héritage et le polymorphisme présentent des problèmes uniques de maintenance et de tests de régression. Harrold et al. présentent une technique pour les tests de régression sélectifs qui supporte les logiciels orientés objet et qui tente de répondre à cette problématique [Harrold 94]. En se basant sur les graphes de dépendances de programme (PDG), l'algorithme construit des graphes de dépendances de classe (CIDG) et utilise ces graphes pour déterminer les tests, parmi la suite de tests originale, qui pourraient produire des résultats différents lorsqu'exécutés sur le programme original et sur sa version modifiée [Harrold 94]. Les PDG (Program Dependance Graph) représentent à la fois les dépendances de contrôle et les dépendances de données dans un seul graphe [Harrold 94]. Les dépendances de contrôle entre les instructions du programme (nœud) sont représentées par des arcs droits (Control Dependance Edge). Les dépendances de données, quant à elles, sont représentées par des arcs pointillés (Data Dependance Edge) [Harrold 94]. La figure 13 représente le PDG de la procédure "search" et son ensemble de tests original T.

```
E31     int List::search(int& loc, int item)
        {
S32         loc = 0;
P33         while (loc < numentries)
            {
P34             if (list[loc] == item)
S35                 return 0;
                else
S36                 ++loc;
            }
S37         return -1;
        }
```

TEST HISTORY INFORMATION

test	list	item	region trace
T1	empty	2	E31,R7,P33,R9
T2	1,2,3	2	E31,R7,P33,R8,P34,R11,R10
T3	1,2,3	4	E31,R7,P33,R8,P34,R10,R9
T4	1,2,3	1	E31,R7,P33,R8,P34,R10

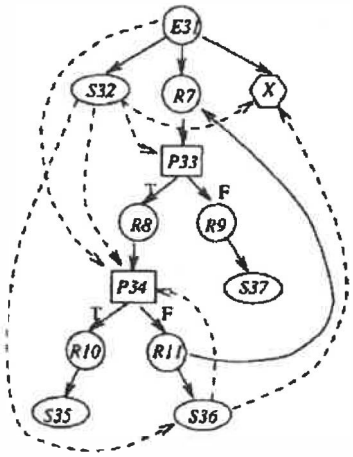


Figure 13. PDG de la fonction "search" et son ensemble de tests T [Harrold 94].

Les IPDGs (graphes de dépendances inter-procédurales), utilisés dans le contexte des programmes procéduraux, consistent en un regroupement de PDGs [Harrold 94]. Supposons un programme p, l'IPDG de p constitue l'ensemble PDGs individuels des procédures dans p avec des arcs ajoutés pour représenter les dépendances de contrôle et de données inter procédurales [Harrold 94]. Par contre, les IPDGs requièrent un programme ayant un seul point d'entrée. Pour adapter ce concept aux classes qui possèdent plusieurs points d'entrée, ils utilisent les CIDGs (Class Dependance Graph - figure 14).

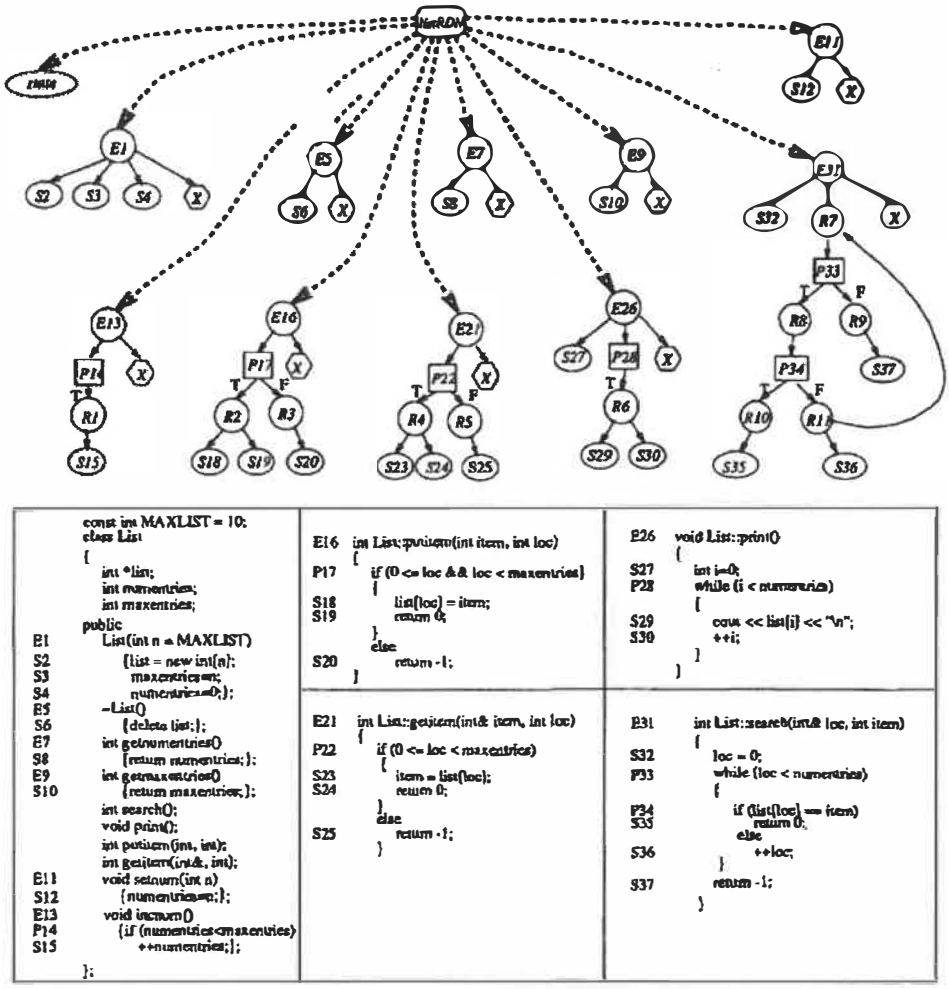


Figure 14. CIDG de la classe List [Harrold 94].

L'algorithme SelectClassTests est utilisé pour illustrer les dépendances de contrôle et de données d'une classe à travers les CIDGs pour ensuite sélectionner les tests de régression requis [Harrold 94]. L'algorithme prend en entrée une classe C, sa version modifiée C', la liste des méthodes publiques PubM et PubM' de C et C' ainsi que l'ensemble de tests T utilisé pour tester originalement C [Harrold 94]. Le même principe est utilisé pour les classes modifiées et dérivées. Cette approche, strictement basée sur le code, supporte l'héritage, le polymorphisme et les liaisons dynamiques [Harrold 94].

Harrold et al. proposent une technique de sélection de tests de régression, strictement basée sur l'analyse du code, dans le cadre des programmes orientés objet C++ [Harrold 00]. L'idée principale consiste à construire une représentation orientée objet

(graphe de contrôle) du programme et de sa version modifiée pour sélectionner, à partir de la suite de tests originale, les tests de régression qui exécutent du code modifié [Harrold 00]. L'approche utilise les modèles ICFG (Interprocedural Control Flow Graph) et les modèles CCFG (Class Control Flow Graph) qui sont dérivés des CFGs (Control Flow Graph) [Harrold 00].

```
1  #include <iostream.h>
2  #include <stdlib.h>
3  #define UP 1
4  #define DOWN 2
5  typedef int Direction;
6
7  class Elevator {
8  public:
9     Elevator (int l_top_floor) {
10      current_floor = 1;
11      current_direction = UP;
12      top_floor = l_top_floor;
13      bottom_floor = 1;
14    }
15
16    virtual ~Elevator() {}
17
18    void up() {
19       current_direction = UP;
20    }
21
22    void down() {
23       current_direction = DOWN;
24    }
25
26    Direction direction() {
27       return current_direction;
28    }
29
55
56 private:
57    add(int &a, const int &b) {
58       a = a+b;
59    }
60
61    int valid_floor(int floor) {
62       if ((floor > top_floor) ||
                (floor < bottom_floor)
63          return 0;
64       return 1;
65    };
66
67 protected:
68    int current_floor;
69    Direction current_direction;
70    int top_floor;
71    int bottom_floor;
72 };
73
74 void main (int argc, char **argv) {
75    Elevator *e_ptr;
76
77    e_ptr = new Elevator(10);
78    e_ptr->go(2);
79 }
```

Figure 15. Code de la classe Elevator [Harrold 00].

La figure 15 présente le code de la classe Elevator tandis que la figure 16 illustre l'ICFG généré à partir de la classe.

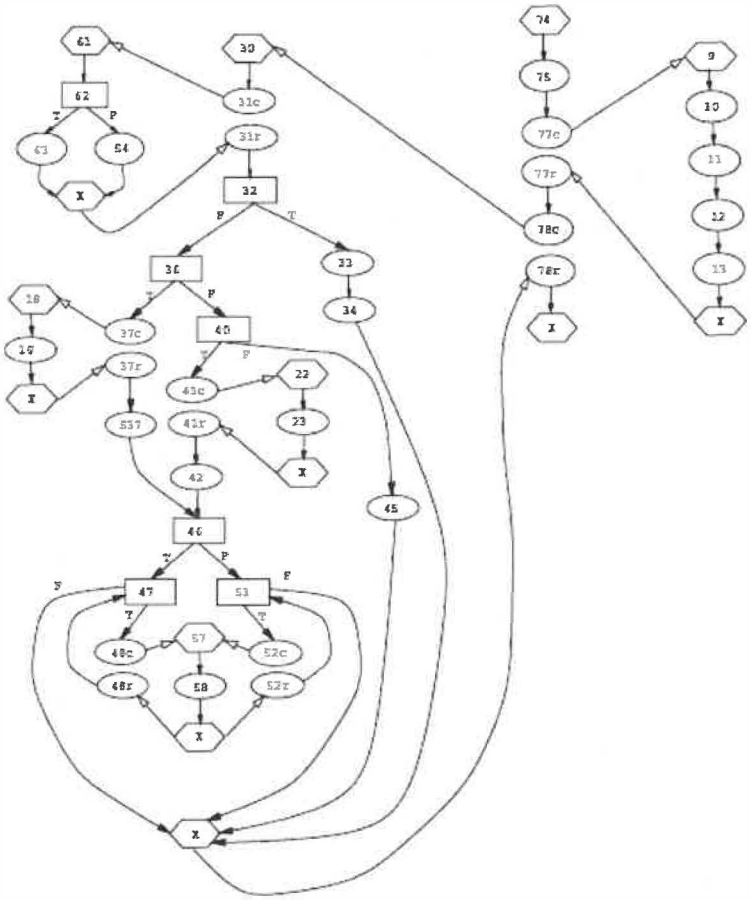


Figure 16. ICFG de la classe Elevator [Harrold 00].

L'ICFG contient un CFG pour chaque méthode d'un programme [Harrold 00]. Comme le ICFG, le CCFG est une collection de CFGs des méthodes d'une classe auquelles une structure est ajoutée (entrée, sortie, constructeur, appels, retours, etc.) [Harrold 00]. L'utilisation du CCFG, permet à l'approche de supporter l'héritage, le polymorphisme, la liaison dynamique et le passage d'objet en paramètre [Harrold 00]. Pour sélectionner les tests de régression des classes modifiées et dérivées, l'approche utilise une version adaptée de l'algorithme "SelectTests" présenté dans l'article de Harrold & al. [Harrold 97]. Les CCFGs sont aussi utilisés pour représenter les classes dérivées [Harrold 00]. Dans ce cas, les CCFGs sont composés de CFGs hérités de la classe de base et de CFGs représentant les nouvelles méthodes de la classe dérivée [Harrold 00]. L'approche a aussi la particularité d'accommoder d'importantes

caractéristiques du paradigme orienté objet telles que les tests intra classe et interclasses, les objets passés en paramètres, etc. [Harrold 00].

Harrold et al. présentent la première technique de sélection de tests de régression supportant les caractéristiques propres au langage Java [Harrold 01]. Elle constitue une adaptation de l'approche de Harrold & al., présentée précédemment [Harrold 97], qui utilise une représentation basée sur les flux de contrôle (CFG) de la version originale et de la version modifiée du programme, pour sélectionner les tests de régression à ré-exécuter [Harrold 01]. Une représentation fidèle du programme Java et de toutes ses caractéristiques est construite sous forme de modèle de contrôle nommé "Java Interclass Graph" (JIG) [Harrold 01]. Un modèle JIG contient un CFG pour chaque méthode interne de l'ensemble des classes [Harrold 01]. La figure 17 illustre le JIG des classes E1, E2 et E3.

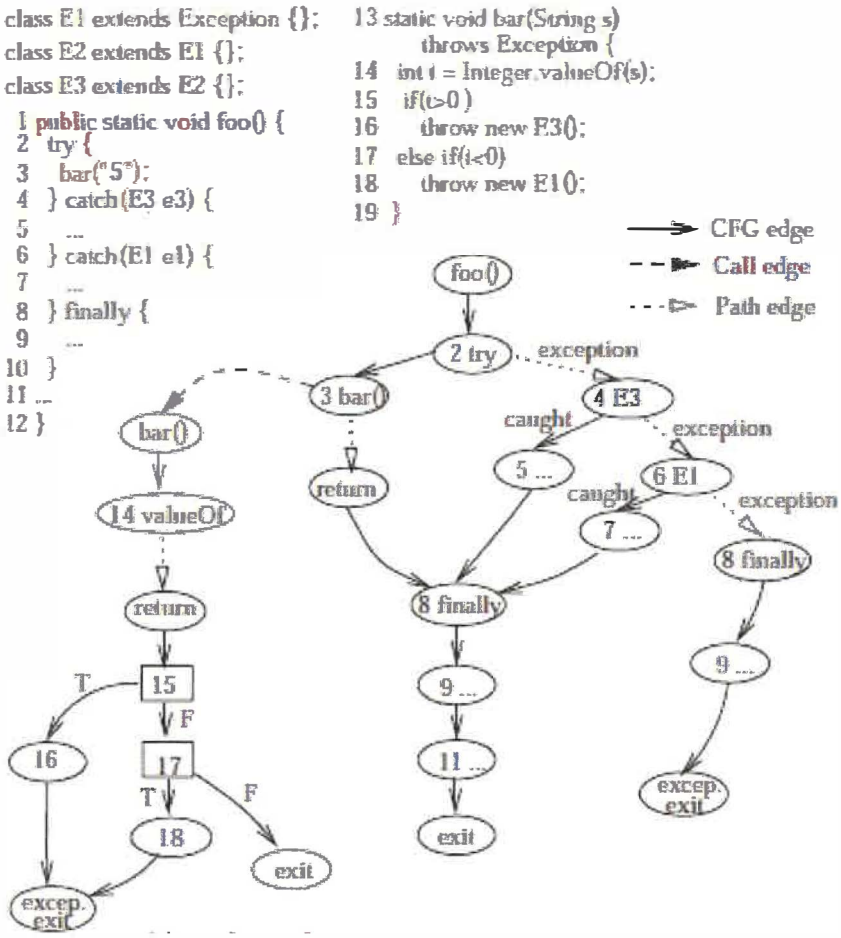


Figure 17. JIG des classes E1, E2 et E3 [Harrold 01].

Dans un JIG, chaque appel et retour de méthode sont représentés [Harrold 01]. Le JIG est conçu de manière à accommoder les caractéristiques du langage Java (héritage, polymorphisme, liaison dynamique et gestion des exceptions) [Harrold 01]. Le JIG supporte 5 caractéristiques propres à Java : l'information sur le type des variables et des objets, les méthodes internes et externes, les interactions inter procédurales à travers les appels à des méthodes internes ou externes par des méthodes internes, les interactions inter procédurales à travers les appels de méthodes internes par des méthodes externes et la gestion des exceptions [Harrold 01]. La technique comporte 3 étapes principales. La construction des modèles représentant les flux de contrôle (JIG), l'analyse des modèles pour déterminer les entités dangereuses et la sélection des tests de régression à partir de la matrice de couverture [Harrold 01]. Supposons P(i) et P'(i), l'exécution des programmes P et P' avec les données i en entrée. Une entité dangereuse constitue une entité de programme telle que, pour des données en entrée i appliquées à P pour couvrir e, P(i) et P'(i) se comportent différemment à cause des différences entre les programmes P et P' [Harrold 01]. Le fonctionnement de l'approche est illustré dans la figure 18.

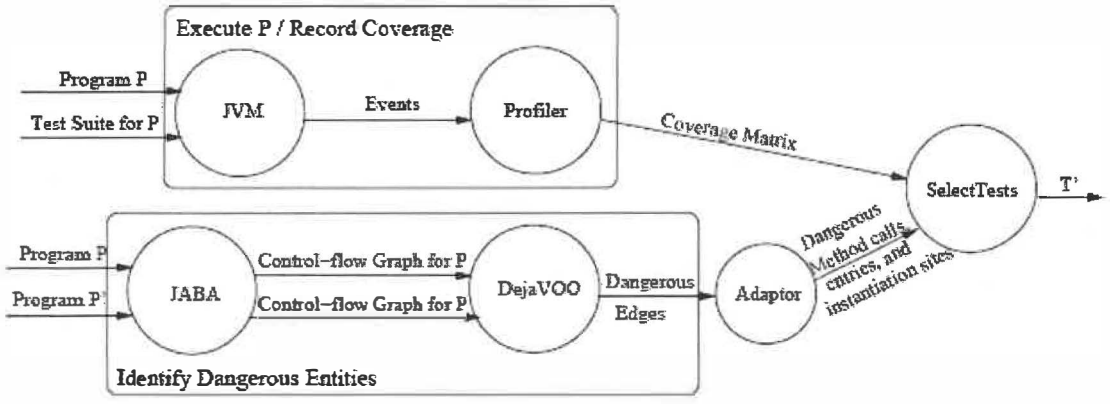


Figure 18. JIG des classes E1, E2 et E3 [Harrold 01].

La matrice de couverture est obtenue par l'exécution de la suite de tests originale sur le programme P qui a préalablement été instrumenté. La technique assure que n'importe quel cas de test, qui ne couvre pas d'entité dangereuse, se comportera de la même manière pour *P* et *P'*, et ne pourra exposer de nouvelles erreurs dans *P'* [Harrold

01]. Elle peut être appliquée sur des programmes incomplets. Sa précision s'étend jusqu'aux structures de contrôle des méthodes membres des classes.

Koju & al. [Koju 03] présentent une technique conservatrice (safe) de sélection de tests de régression, pour les tests d'intégration, dans le cadre d'environnements de développement basés sur les machines virtuelles [Koju 03]. Des exemples d'application de l'approche utilisant le Microsoft Intermediate Language (MSIL) du framework Microsoft .Net Framework sont présentés dans leurs travaux. La technique, basée sur les travaux de Harrold et al. [Harrold 01], consiste à créer des Control Flow Graphs (CFG) à partir du code intermédiaire MSIL de manière à résoudre certains éléments non supportés par l'approche originale de Harrold et al. [Harrold 01]. On parle ici de difficultés causées par les délégués, l'analyse du code à grande échelle ainsi que la gestion des exceptions. Un algorithme détecte ensuite les chemins (arcs) dangereux en comparant les CFGs avant et après modification [Koju 03]. La détection des arcs dangereux est basée sur les algorithmes de Harrold & al. [Harrold 97, Harrold 01]. L'évaluation de l'approche sur 10 exemples différents démontre une réduction significative du coût des tests de régression.

Zhao, Xie & Li [Li 06] présentent une technique de sélection de tests de régression, basée sur l'analyse du code, dans le cadre des programmes orientés-aspect (AspectJ). La technique, inspirée de l'approche pour Java proposée par Harrold et al [Harrold 01], peut être appliquée au niveau de la modification individuelle d'un aspect ou d'une classe aussi bien que pour le programme en entier. La technique utilise plusieurs types de modèles de contrôle pour sélectionner, parmi la suite de tests originale, les tests de régression à exécuter lors de la modification d'un programme. Comme pour la technique d'Harrold et al. [Harrold 01], la notion de base consiste à détecter les arcs dangereux basés sur les modèles de contrôle du programme original et de sa version modifiée. La détection des arcs dangereux est basée sur celle proposée par Harrold & al. [Harrold 97, Harrold 01]. Le modèle de contrôle proposé pour AspectJ est composé de différents types de graphes de contrôle pour capturer différents niveaux d'information de flux de contrôle. Les graphes CFG sont utilisés comme base pour modéliser les

interactions. Les figures 20 et 21 illustrent respectivement un exemple d'ACFG et un exemple de SCFG du programme AspectJ présenté à la figure 19.

```
 ce0 public class Point {
  s1   protected int x, y;
 me2   public Point(int _x, int _y) {
  s3     x = _x;
  s4     y = _y;
       }
 me5   public int getX() {
  s6     return x;
       }
 me7   public int getY() {
  s8     return y;
       }
 me9   public void setX(int _x) {
 s10     x = _x;
       }
me11   public void setY(int _y) {
 s12     y = _y;
       }
me13   public void printPosition() {
 s14     System.out.println("Point at("+x+","+y+")");
       }
me15   public static void main(String[] args) {
 s16     Point p = new Point(1,1);
 s17     p.setX(2);
 s18     p.setY(2);
       }
     }
ce19 class Shadow {
 s20   public static final int offset = 10;
 s21   public int x, y;

me22   Shadow(int x, int y) {
 s23     this.x = x;
 s24     this.y = y;
me25   public void printPosition() {
 s26     System.outprintln("Shadow at
                 ("+x+","+y+")");
       }
     }
```

```
ase27 aspect PointShadowProtocol {
 s28   private int shadowCount = 0;
 me29  public static int getShadowCount() {
 s30     return PointShadowProtocol.
                 aspectOf().shadowCount;
       }
 s31   private Shadow Point.shadow;
 me32  public static void associate(Point p, Shadow s){
 s33     p.shadow = s;
       }
 me34  public static Shadow getShadow(Point p) {
 s35     return p.shadow;
       }

 pe36  pointcut setting(int x, int y, Point p):
         args(x,y) && call(Point.new(int,int));
 pe37  pointcut settingX(Point p):
         target(p) && call(void Point.setX(int));
 pe38  pointcut settingY(Point p):
         target(p) && call(void Point.setY(int));

 ae39  after(int x, int y, Point p) returning :
           setting(x, y, p) {
 s40     Shadow s = new Shadow(x,y);
 s41     associate(p,s);
 s42     shadowCount++;
       }
 ae43  after(Point p): settingX(p) {
 s44     Shadow s = new getShadow(p);
 s45     s.x = p.getX() + Shadow.offset;
 s46     p.printPosition();
 s47     s.printPosition();
       }
 ae48  after(Point p): settingY(p) {
 s49     Shadow s = new getShadow(p);
 s50     s.y = p.getY() + Shadow.offset;
 s51     p.printPosition();
 s52     s.printPosition();
     }
   }
```

Figure 19. Exemple de programme AspectJ [Ball 98].

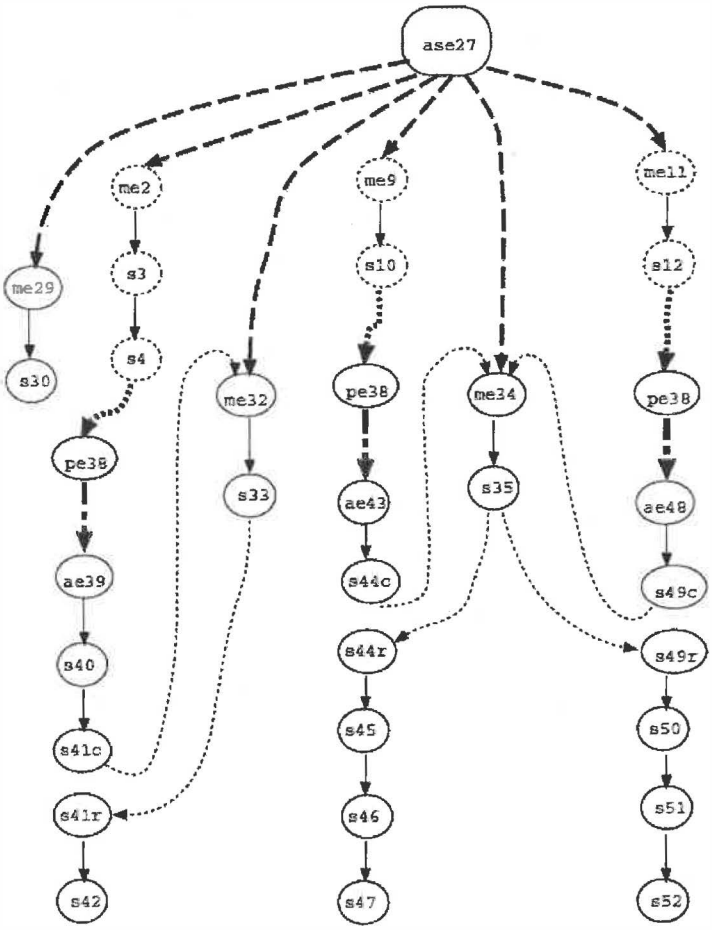


Figure 20. ACFG correspondant à l'aspect PointShadowProtocol [Ball 98].

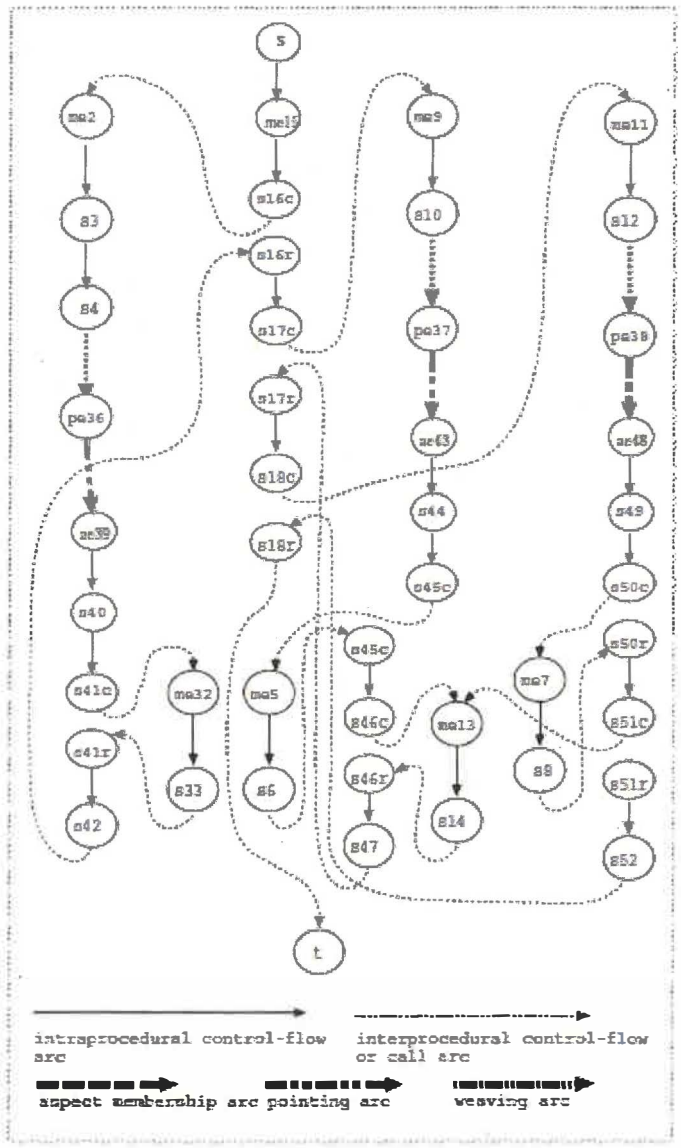


Figure 21. SCFG du programme AspectJ [Ball 98].

Le graphe ACFG (Aspect Control Flow Graph) modélise les relations à l'intérieur d'un aspect, tandis que le SCFG (System Control Flow Graph) modélise les interactions aspect-classe et inter modules. Ces graphes constituent, en fait, des collections de CFGs.

1.4 Approches systèmes

Les tests de système consistent à tester l'intégralité d'un programme pour s'assurer que ce dernier respecte bien ses spécifications [Skoglund 09]. Les tests de système doivent vérifier que tous les éléments du système ont été proprement intégrés et qu'ils effectuent correctement les fonctions désirées. Les tests de systèmes peuvent être effectués sans avoir de connaissance sur l'implémentation de l'application.

Des approches ont été développées afin d'effectuer des tests de régression au niveau système. TestTube, un système issu des travaux de Chen & al. 1994 [Chen 94], en est un exemple. Il a été développé dans les laboratoires d'AT&T Bell et peut être utilisé pour effectuer des tests de régression sélectifs à tous les niveaux (unitaire, intégration et système) [Chen 94]. L'idée de base derrière Testube est illustrée dans la figure 22 [Chen 94]. Les boites représentent les sous-programmes et les cercles représentent les variables [Chen 94]. Les flèches représentent les relations de dépendance statiques et dynamiques (ex : références de variable, appels de fonctions, etc.) [Chen 94]. Dans cet exemple, les entités foncées sont modifiées [Chen 94]. Le test 3 est sélectionné et les tests 1 et 2 sont exclus [Chen 94].

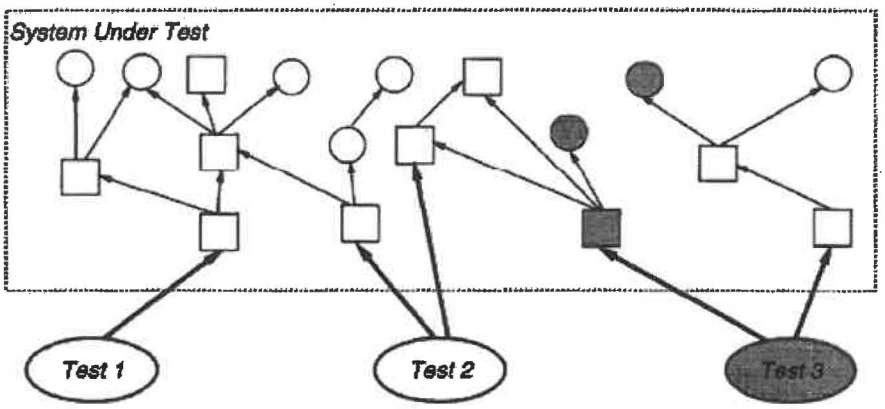


Figure 22. Sélection de tests de régression via Testube [Chen 94].

L'approche, qui combine analyse statique et dynamique, partitionne le système en entités de code, pilote l'exécution des cas de test, analyse les relations avec le système testé et détermine l'ensemble d'entités de code que le test couvre [Chen 94]. Testube identifie, par analyse statique du code, les fonctions, les types, les variables et les macros couverts par chaque cas de test de la suite de tests existantes [Chen 94]. Chaque fois que le programme est modifié, Testube identifie les entités qui ont été changées pour créer la nouvelle version du programme et sélectionne les cas de test qui lui sont associés [Chen 94]. Pour chaque version du système sous test, une base de données contenant les informations à propos des entités qui composent le système est créée [Chen 94]. Cette base de données est utilisée pour analyser et créer la liste des différences entre les entités du programme original et modifié [Chen 94]. Cette liste est ensuite comparée à la liste des traces d'entités (obtenues grâce à l'instrumentation du code) de chaque cas de test [Chen 94].

Testube sélectionne uniquement les tests qui couvrent les entités qui ont changées pour effectuer les tests de régression de la nouvelle version [Chen 94]. Le système a été conçu pour supporter les programmes procéduraux écrits en langage C [Chen 94]. L'expérimentation présentée démontre une réduction significative dans le nombre de cas de test requis pour tester les changements typiques de logiciel [Chen 94].

1.5 Points forts et points faibles

Plusieurs approches ont été proposées afin de supporter la sélection de tests de régression [White 90, White 92, Chen 94, Harrold 94, Harrold 96, Harrold 97, Chen 97, White 97, Korel 98, Ball 98, Onoma 98, Harrold 00, Harrold 01, White 03, Koju 03, Skoglund 05]. Parmi elles, certaines s'intéressent aux tests de régression au niveau unitaire [Harrold 96, Korel 98, Ball 98] et niveau système [Chen 94]. D'autres s'intéressent aux tests d'intégration [White 90, White 92, Harrold 94, Harrold 97, White 97, Chen 97, Onoma 98, Harrold 00, Harrold 01, White 03, Koju 03, Skoglund 05] et plus précisément dans les systèmes orientés objet [Harrold 94, Chen 97, White 97, Harrold 00, Harrold 01] comme le fait l'approche proposée dans ce mémoire. Dans ce

groupe, figurent des approches basées sur les « arcs dangereux » [Harrold 00, Harrold 01, Harrold 94] et d'autres basées sur le concept de « firewall » [Chen 97, White 97].

Les approches basées sur les arcs dangereux [Harrold 94, Harrold 00, Harrold 01] sont très précises, car elles tiennent compte des chemins de contrôle lors de la sélection des tests de régression. Ce type d'approches est d'ailleurs le plus précis de la littérature actuellement, car il tend à sélectionner moins de tests que les types d'approches similaires afin d'atteindre un niveau de confiance équivalent. Ces approches sont dites conservatrices (safe), car elles n'écartent aucun cas de test susceptible de révéler une erreur. Par contre, due à la fine granularité (niveau instructions) des informations à recueillir par instrumentation, elles ont tendance à être très coûteuses. De plus, les approches basées sur les arcs dangereux ne permettent pas d'identifier les chemins d'exécution non couverts par les tests existants et par le fait même ne supportent pas la génération de nouveaux cas de tests.

Les approches basées firewall [Chen 97, White 97] sont un peu moins précises que les approches basées sur les arcs dangereux, car elles ne tiennent pas compte des chemins de contrôle et ont tendance à sélectionner plus de tests que l'approche d'Harrold et al. afin d'atteindre le même niveau de confiance. Ce type d'approches se contente de travailler à un niveau de granularité moindre (niveau classe) pour effectuer la sélection des tests de régressions. Il requiert donc une instrumentation beaucoup moins coûteuse. Par contre, les approches par firewall requièrent les spécifications de P et P'. Cet aspect constitue un désavantage puisque dans un contexte industriel réel, les spécifications des applications sont rarement disponibles. Les approches par firewall ne supportent pas la génération de nouveaux cas de test afin de couvrir les éléments nouveaux.

CHAPITRE 2

GRAPHES DE CONTRÔLE RÉDUITS AUX APPELS

2.1 Graphes de contrôle

Les graphes de flot de contrôle (CFG) ont été initialement développés par Cota et Sargent [Cots 89, Cots 90a, Cots 90b, Cots 90c, Cots 90d] en tant que modèle de représentation du langage pour la simulation d'événements discrets. Le but de ce langage de représentation est de rendre l'information explicite pour permettre le développement de simulation d'exécution d'algorithmes.

L'information des flux de contrôle est nécessaire pour effectuer des techniques d'analyse de programmes, tel que les dépendances entre les flux de données et les flux de contrôle ainsi que des techniques d'ingénierie logicielle tel que le slicing et les tests [Wu 03]. Les CFG sont une représentation de la relation entre les flux de contrôle qui existe dans un programme, dans lequel les nœuds représentent des instructions et les arcs représentent les flux de contrôle entre les instructions [Wu 03]. Pour que ces techniques d'analyse de programme et d'ingénierie logicielle soient sûres et utiles, les CFG doivent incorporer tout les flux de contrôle qui peuvent survenir durant l'exécution d'un programme [Wu 03]. Voici la définition d'un graphe de flot de contrôle :

Définition 1 : 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 Ni à un nœud Nj s'il est possible d'exécuter l'instruction correspondante à Nj immédiatement après celle associée au nœud Ni. Les arcs du graphe indiquent le transfert de contrôle d'un nœud à l'autre.

2.2 Graphes de contrôle réduits aux appels

Les graphes de contrôle réduits aux appels (CCG) permettent de synthétiser le comportement d'un programme. Ils précisent les enchaînements lors de l'exécution des appels. Nous donnons, dans ce qui suit, la définition relative à un graphe de contrôle réduit aux appels [Badri 05, St-Yves 08]. Ils représentent une forme réduite des CFGs.

Définition 2 : 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.

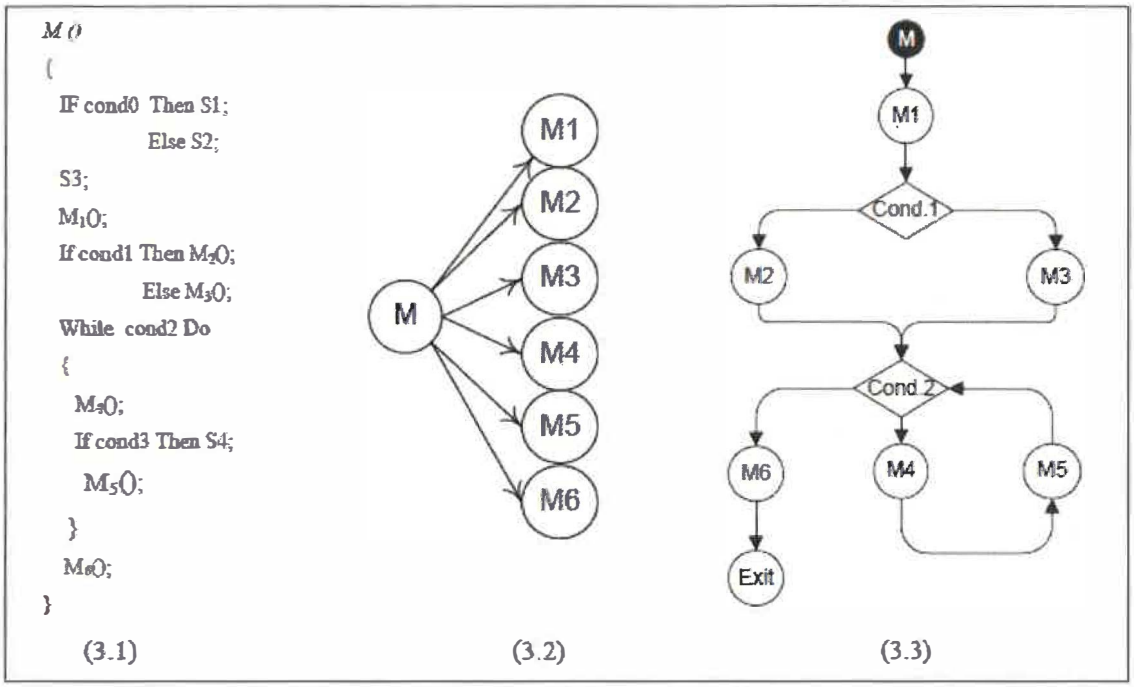


Figure 23. Graphe d'appels et graphe de contrôle réduit aux appels [St-Yves 08].

Considérons la portion de programme donnée par la figure 23(3.1). Les S_i représentent des séquences d'instructions ne contenant pas d'appels de méthodes. Le graphe d'appels correspondant est présenté à la figure 23(3.2). Le graphe de contrôle réduit aux appels correspondant est présenté à la figure 23(3.3) [Badri 05, St-Yves 08].

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. 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 [Badri 05, St-Yves 08].

CHAPITRE 3

MÉTHODOLOGIE DE L'APPROCHE

3.1 Introduction à la méthodologie

L'approche que nous proposons ne requiert aucune intervention manuelle autre que la sélection, via une interface utilisateur, des programmes P et P' à analyser. Elle est organisée en cinq étapes principales :

- La première étape constitue l'analyse des changements effectués dans P' à partir des codes sources Java de P et P'. Cette étape nous permet d'obtenir l'ensemble M' des méthodes modifiées.

- La seconde étape consiste à effectuer, en parallèle à la première étape, le compactage des chemins d'exécutions possibles de P' à partir, encore une fois, du code source de P'. Cette étape nous donne en sortie l'ensemble C' des chemins d'exécution possibles de P'.

- La troisième étape consiste à identifier l'ensemble I' des chemins d'exécution impactés par les modifications. Elle utilise en entrée les ensembles M' et C'.

- Grâce à l'ensemble I', la quatrième étape consiste à sélectionner, parmi la batterie de tests existants, l'ensemble de tests S couvrants les chemins d'exécution impactés de P'. Elle retourne également l'ensemble E des chemins d'exécutions impactés non couverts par S.

- Lors de la dernière étape, l'ensemble E est utilisé pour générer l'ensemble des nouveaux cas de test N afin de couvrir les chemins d'exécution non couverts par S. La combinaison de N et S constituera l'ensemble T' qui sera utilisé pour tester P'.

Vous constaterez, dans les sections qui suivent, que notre approche ne requiert aucune instrumentation du code, ni spécification. Elle procède uniquement par analyse statique du code du programme et de sa version modifiée. L'aspect principal qui la distingue des autres approches est qu'elle travaille à partir de l'ensemble des chemins d'exécution possibles de P' plutôt que de se limiter à l'ensemble des chemins d'exécution de P' couverts par les tests existants. Cette caractéristique nous permet d'être en mesure de générer de nouveaux cas de test pour couvrir les chemins d'exécution non couverts par la batterie de tests existants. Ceci constitue une des contributions importantes de l'approche.

Vous constaterez aussi qu'une sixième étape figure dans nos explications ultérieures; la génération des séquences de tests de P. Cette étape ne fait pas réellement partie intégrante de l'approche puisque dans une situation réelle, l'ensemble T devrait déjà exister et pouvoir être réutilisé. Cette étape a été introduite afin de simuler l'existence de séquences de test intégrées ayant servies initialement à tester P. Le schéma présenté à la figure 24 illustre les différentes étapes de notre approche. Chacune de ces étapes sera abordée plus en détail dans les sections qui suivent.

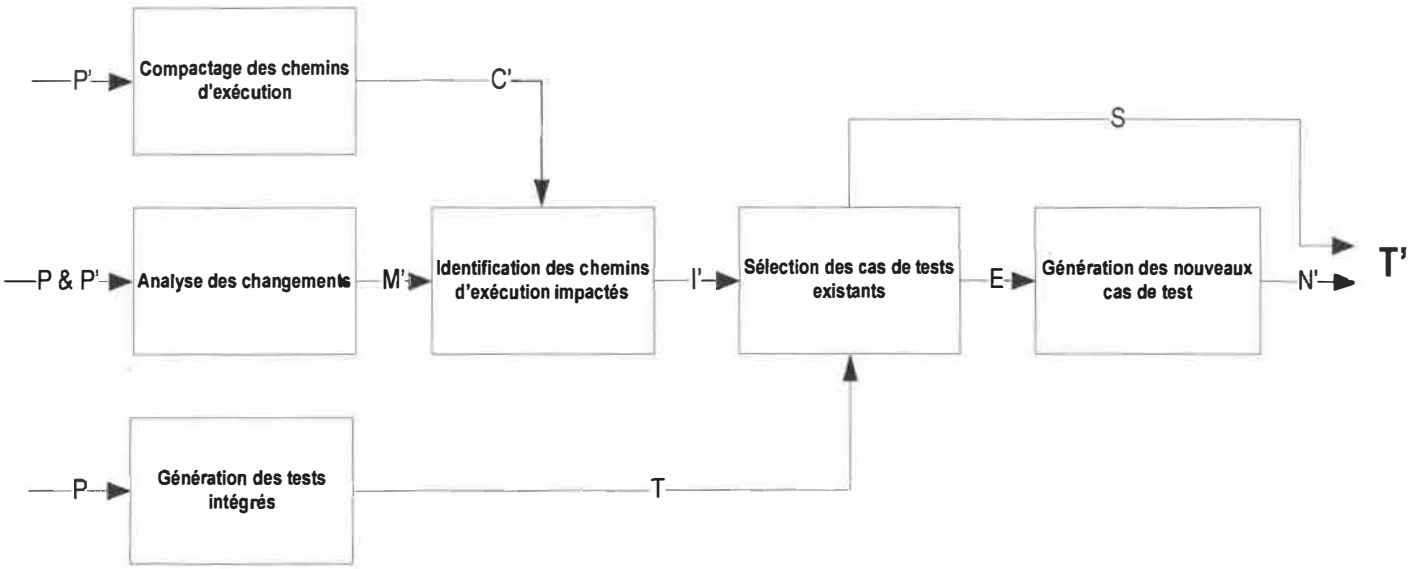


Figure 24. Schéma de la méthodologie proposée.

3.2 Analyse des changements

La première étape de l'approche s'intéresse à déterminer les changements qui ont été effectués au niveau des méthodes lors du passage de P à P'. Nous proposons d'effectuer l'analyse du changement entre P et P' afin d'obtenir la liste des méthodes modifiées. Lors de cette étape, nous comparons chaque méthode du programme modifié avec sa correspondante dans la version originale afin de retenir les méthodes ajoutées, modifiées ou supprimées. L'analyse est effectuée de la façon suivante :

Soit un programme P et sa version modifié P', V l'ensemble des méthodes de P, V' l'ensemble de méthodes de P'. Nous nous intéressons à déterminer l'ensemble des méthodes modifiées M' tel que : Pour toute méthode v' élément de V', v' est un élément de M' si v' est différent de v ou si v' n'est pas un élément de V. À l'ensemble M' s'ajoutent aussi les méthodes supprimées, c'est à dire les éléments v de V telles que : Pour tout v élément de V, v est élément de M' si v n'est pas un élément de V'.

Trois types de modifications sont considérés au niveau des méthodes : l'ajout, la modification et la suppression. Il est à noter que l'expression "méthodes modifiées" réfère aux méthodes ajoutées, modifiées et supprimées dans le reste du présent document.

Le processus d'analyse du changement prend en entrée les répertoires et sous répertoires sources des programmes P et P' afin de les parcourir en entier et d'insérer dans les listes de fichiers lv1 et lv2, le nom absolu de chacun des fichiers des deux projets. Seuls les fichiers portant l'extension ".java" seront comparés. Les fichiers ne répondant pas à ce critère seront ignorés. Les listes lv1 et lv2 constituent l'ensemble des fichiers de P et P' susceptibles de faire l'objet d'une comparaison.

Les fichiers contenus dans ces deux listes sont analysés afin de déterminer les fichiers qui devront faire l'objet d'une comparaison. Dans un premier temps, les fichiers présents à la fois dans lv1 et lv2 sont ajoutés à la liste finale lf des fichiers à comparer. Ensuite, nous vérifions si certains fichiers sont présents dans lv2 et absents dans lv1 afin

de les ajouter aussi à la liste. Ces fichiers constituent des classes ajoutées. L'interface utilisateur (présentée à la section 5.1), qui est introduite à Éclipse sous forme de plug-in accessible directement dans l'outil de développement, prend en entrée le nom absolu des deux répertoires à comparer.

Chaque nom de fichier figurant dans lf est ensuite utilisé afin de comparer le fichier correspondant dans P' à son équivalent dans P à partir de la librairie d'Eclipse 3.3 "org.eclipse.compare.CompareUI". La librairie permet d'effectuer une opération de comparaison qui résulte en l'ouverture d'un éditeur de comparaison dans lequel les détails peuvent être parcourus et consultés. L'exemple présenté à la figure 25 illustre l'analyse des fichiers "testNextGen.java", en provenance de P et P' respectivement, impliquant la détection d'une différence dans la méthode main de la classe testNextGen suite à l'ajout de 2 lignes de code. Dans ce cas, la méthode main serait ajoutée à l'ensemble M' des méthodes modifiées.

```
package NextGen;

import java.util.Calendar;

public class testNextGen {


    public static void main (String[] args)
    {
        Calendar c = null;
        Location l = new Location(c, 2);
        l.getSolde();
    }

}
```

```
package NextGen;

import java.util.Calendar;

public class testNextGen {


    public static void main (String[] args)
    {
        Calendar c = null;
        Location l = new Location(c, 2);
        l.getSolde();

        Employe e = new Employe(1, "", "", "", "", "", ""); //Modif #2
        e.getAdresseComplete();
    }

}
```

Figure 25. Éditeur de comparaison org.eclipse.compare.CompareUI.

Le résultat de la comparaison est obtenu sous forme d'arbre qui sera ensuite parcouru de façon récursive, nœud par nœud afin d'extraire les méthodes modifiées ou ajoutées. Les arbres constituent une représentation hiérarchique des objets (attributs, méthodes, etc.) qui ont été identifiés comme étant différents dans P' lors de l'analyse effectuée via CompareUI. Les arbres sont instanciés par

"org.eclipse.swt.widgets.TreeItem".

La librairie CompareUI permet, entre autres, de faire la distinction entre trois types de modification au niveau des lignes de code : l'ajout, la modification et la suppression. Les lignes de code se terminant par le caractère ")" sont considérées comme des déclarations de méthodes et sont ajoutées à la liste des méthodes modifiées M' lors de l'analyse récursive. Les méthodes identifiées comme ayant été modifiées sont représentées par leur nom complet incluant le package (ex. : NextGen.Succursale.Succursale(int, String)).

3.3 Compactage des chemins d'exécution

Le compactage des chemins d'exécution est effectué en parallèle à l'étape décrite à la section 3.2. Il consiste à déterminer et à compacter, par analyse statique du code, l'ensemble C' des chemins d'exécution possibles des méthodes du programme P'. L'ensemble des chemins d'exécution compactés de P' sera utilisé plus tard dans le processus afin de sélectionner les chemins d'exécution devant être re-testés. Cette portion de l'approche réutilise les concepts proposés par [St-Yves 08].

On commence par 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 26 donne la synthèse de l'algorithme de la méthode M() et le graphe de contrôle correspondant [St-Yves 08].

```
M()
{
        M1();
        If cond1 Then M2();
              Else M3();

        While cond2 Do
        {
                M4();
                M5();
        }
        M6();
}
```

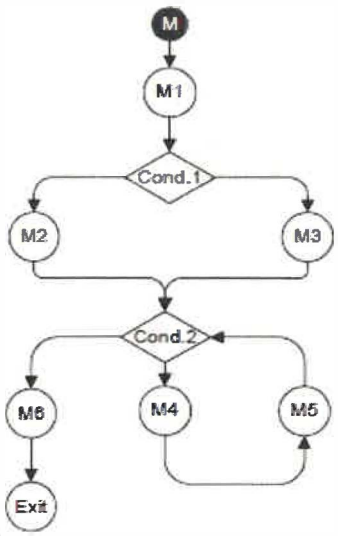


Figure 26. Graphe de contrôle réduit aux appels [St-Yves 08].

On effectue ensuite l'analyse des graphes de contrôle en vue de générer les chemins de contrôle compactés. À cette étape, nous devons nous assurer d'éliminer les chemins d'exécution impossibles. Ces chemins compactés permettent de donner des renseignements à propos du comportement dynamique du programme [St-Yves 08].

```
M()                          M7()                        M6()
{ M1();                      { M7();                     { If cond4 Then M8();
  If cond1 Then M2();          If cond3 Alors M8();        M10();
          Else M3();         }                           }

  While cond2 Do             M3()                        M8()
  {                          {                           { M9(); }
    M4();                      M8();
    M5();                    }
  }
  M6();
}
```

Figure 27. Méthodes M, M1, M2, M3, M4, M5, M6, M7 et M8 [St-Yves 08].

La figure 27 donne la synthèse (contrôle réduit aux appels) de plusieurs méthodes que nous considérons pour illustrer la création des chemins d'exécution compactés [St-Yves 08].

1. $M : M_1 (M_2 / M_3) \{ M_4, M_5 \} M_6$
2. $M_2 : M_7 [M_8]$
3. $M_3 : M_8$
4. $M_6 : [M_8] M_{10}$
5. $M_9 : M_9$

a

$M, M_1, M_2, M_7, M_8, M_9, M_6, M_8, M_9, M_{10}$
$M, M_1, M_3, M_7, M_8, M_9, M_6, M_{10}$
$M, M_1, M_3, M_8, M_9, M_6, M_{10}$
$M, M_1, M_2, M_7, M_8, M_9, M_4, M_5, M_6, M_8, M_9, M_{10}$
$M, M_1, M_3, M_8, M_9, M_4, M_5, M_6, M_8, M_9, M_{10}$
$M, M_1, M_2, M_7, M_8, M_9, M_4, M_5, M_4, M_5, M_6, M_8, M_9, M_{10}$
---

b

Figure 28. Chemins d'exécution compactés (a) et complets (b) [St-Yves 08].

La figure 28.a présente les chemins de contrôle compactés correspondants aux méthodes de la figure 27. 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 28.b illustre une partie des chemins possibles pouvant être déduits des chemins de contrôle compactés.

Dans une implémentation réelle, les méthodes sont représentées par leur signature à l'intérieur des chemins d'exécution. 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 [St-Yves 08].

3.4 Identification des chemins d'exécution impactés

L'identification des chemins d'exécution impactés utilise en entrée les résultats obtenus aux étapes 3.2 et 3.3. Elle consiste à déterminer, à partir de l'ensemble C' des chemins d'exécution compactés du programme P' et de l'ensemble M' des méthodes modifiées, l'ensemble I' des chemins d'exécution complet contenant des éléments ayant subit une modification. Les chemins susceptibles d'être affectés qui seront identifiés à

cette étape devront être re-testés au final. Cet ensemble peut aussi inclure des chemins d'exécution qui n'existaient pas dans l'ensemble des chemins d'exécution possibles du programme P (nouveaux chemins).

On procède d'abord à la génération des chemins d'exécution complets pour chacun des éléments de C'. Le processus analyse ensuite chacun des chemins d'exécution complet contenu dans la liste C' et compare la signature de chacune des méthodes y figurant avec chacune des signatures incluses dans l'ensemble des méthodes modifiées M'. Les éléments de la signature qui sont comparés sont les paramètres en entrée, le nom de la méthode (incluant le package) et les paramètres en sortie. La démarche suivante est appliquée :

Soit M' l'ensemble des méthodes modifiées de P', J' l'ensemble des chemins d'exécution complets de P'. Nous nous intéressons a déterminer I' qui est un sous-ensemble de J' tel que : Pour tout j' élément de J', j' est élément de I' si et seulement si la signature d'une des méthodes de j' est identique à celle d'une méthode de M'.

Soient les méthodes M, M1, M2, M3, M4, M5, M6, M7 et M8 présentées à la figure 27 et leurs chemins d'exécution présentés à la figure 28. La figure 29 illustre l'identification des chemins d'exécution dans le cas d'une modification effectuée à la méthode M5.

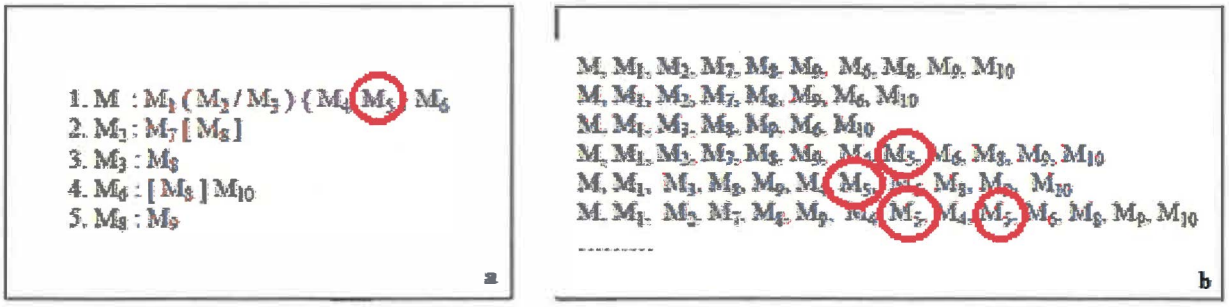


Figure 29. Identification des chemins d'exécution impactés.

Puisqu'ils impliquent tous un appel à la méthode M5 et que les signatures sont identiques, les chemins d'exécution 4, 5 et 6 sont sélectionnés pour être re-testés.

3.5 Génération des séquences de test

L'étape de génération des séquences de test est indépendante des autres étapes. Elle consiste à générer l'ensemble T des séquences de test permettant de valider l'ensemble des chemins d'exécution du programme P. Cette étape ne fait pas réellement partie intégrante de l'approche puisque dans une situation réelle, l'ensemble T existe. Cette étape a été introduite afin de simuler l'existence de séquences de test ayant servies initialement à tester le programme P. Les tests qui seront éventuellement sélectionnés pour valider le programme P' seront sélectionnés au final à partir de l'ensemble T. Le processus de génération de séquences de test se fait de la même façon que celle présentée à la section 3.3.

Les séquences de test générées à cette étape sont formées d'appels séquentiels de méthodes qui seront testées unitairement avec le Framework Junit [Chen 05]. L'outil JUnit est une plateforme qui facilite la production et l'exécution de tests unitaires en Java. Cette plateforme sera utilisée afin d'automatiser l'exécution des tests. Suite à la génération des séquences de test, une classe test est créée. Cette classe hérite de la superclasse TestCase, ce qui indique l'utilisation de JUnit. Pour chaque méthode au sein des séquences, une méthode de test unitaire est créée. La vérification est effectuée par l'exécution automatique via des méthodes assertMi() de JUnit à partir de paramètres fournis par le testeur [Chen 05].

Soient une classe sous test donnée MyClass et sa classe test testMyClass [Chen 05]. Le code de test inclus dans testMyClass est présenté à la figure 30. Le fichier formé par ce code de test est appelé fichier de code de test (ou tout simplement fichier de tests) de la classe MyClass [Chen 05].

```
import JUnit.extensions.TestSetup;
import JUnit.framework.Test;
import JUnit.framework.TestCase;
import JUnit.framework.TestSuite;

public class testMyClass extends TestCase
{ //set local context for each test case.
  public void setUp() { ... }

  //clear local context for each test case.
  public void tearDown() { ... }

  //testMethod1 is the testing method
  // corresponding to the given method Method1
  // in MyClass.
  public void testMethod1() throws
      ApplicationException1
  {
      //generate some test data manually to
      // validate the test result by calling
      // assertXXX method provided by JUnit
      ... ...
  }
  //testMethod2 is the testing method
  // corresponding to the given method Method2
  // in MyClass.
  public void testMethod2() throws
      ApplicationException2 { ... }

  //testMethod3 is the testing method
  // corresponding to the given method Method3
  // in MyClass.
  public void testMethod3() throws
      ApplicationException3 { ... }

  //other testing method
  ... ...
  //Assemble all the testing methods to
  // create a test suite.
  public static Test suite()  //Where Test is
      // an interface provided by JUnit
  {
      TestSetup setup = new TestSetup(new
          TestSuite(TestKlass.class))
      {
          //set the global test context
          protected void setUp() throws Exception { ... }

          //clear the global test context
          protected void tearDown() throws Exception
              { ... }
      };
      return setup;
  }
}
```

Figure 30. Fichier de test de la classe MyClass [Chen 05].

Par la suite, pour chaque séquence, une méthode est créée et cette dernière ne fait qu'appeler dans l'ordre les différentes méthodes de test unitaire créées précédemment. Suite à cette génération automatique, le développeur peut compléter le corps des méthodes de test unitaire et lancer l'exécution en sachant que ces tests couvrent l'ensemble des séquences générées.

Notre approche suppose qu'une méthode de test est disponible pour chacune des méthodes du programme P. Puisque la génération de tests unitaires est exclue de la portée des présents travaux, nous procédons à la génération de tests unitaires de type "coquille vide", c'est à dire qu'une méthode vide est générée pour chacune des méthodes du programme P. Le nom de chaque méthode de test unitaire générée est représentatif de la

signature de la méthode qu'elle valide. La figure 31 présente la méthode "NextGen.Transaction.creerPaiement(double)" pour laquelle un test intégré doit être généré.

```
public void creerPaiement(double montantPresente)
{
    paiement=new Paiement();
    paiement.setMontant(montantPresente);
}
```

Figure 31. Méthode NextGen.Transaction.creerPaiement(double).

Cette méthode contient deux appels de méthode : "NextGen.Paiement.Paiement()" et "NextGen.Paiement.setMontant(double)". Nous constatons aussi que cette méthode possède un seul chemin d'exécution possible, donc un seul test intégré est requis. La figure 32 présente les méthodes de test unitaire de "NextGen.Paiement.Paiement()" et "NextGen.Paiement.setMontant(double)" ainsi que la méthode de test générée afin de tester la méthode "NextGen.Transaction.creerPaiement(double)" soit "testSequence1".

```
public class TestsSequences extends TestCase {

    public static void main(String[] args) {
    }

    public void setUp() {

    }

    public void tearDown() {

    }

    public void NextGen_Paiement_Paiement____() {

    }

    public void NextGen_Paiement_setMontant__double__() {

    }

    public void testSequence1() {

        NextGen_Paiement_Paiement____();
        NextGen_Paiement_setMontant__double__();
    }

}
```

Figure 32. Classe de tests intégrés.

La façon de procéder consiste à générer et compacter les chemins d'exécution possibles du programme P de la même façon que ce qu'on fait pour le programme P' (voir section 2.3). À partir des séquences compactées obtenues, nous procèdons à la génération des séquences d'exécution complètes. Chaque séquence d'exécution complète est ensuite analysée afin d'obtenir la liste des méthodes uniques. Nous construisons ensuite une classe de test dans laquelle nous générerons une méthode de test unitaire sous le format "public void [package]_[classe]_[méthode]___[argument1]_[argument2]_[...]__() {}" pour chacune des méthodes uniques ainsi qu'une méthode de test intégré sous le format "public void testSequence[n]() {[méthode test unitaire1];[méthode test unitaire2];}" pour chacune des séquences d'exécution complètes. Le résultat final de cette étape constitue une classe de test contenant l'ensemble des tests intégrés T tel qu'illustrée par la figure 32.

Notons que T est conçu à la base pour tester les méthodes et les différents chemins d'exécution de P. En effet, les nouvelles méthodes et les nouveaux chemins d'exécution de P' ne sont pas couverts par l'ensemble T. De plus, il est possible que l'ensemble de tests T présente initialement des lacunes, c'est-à-dire qu'il est possible qu'il ne permette pas de couvrir tous les chemins d'exécution et méthodes de P. Pensons, par exemple, à une situation où un oubli ou une erreur aurait été effectuée de la part des développeurs lors de la création de la suite de tests originale T. Les sections 3.5 et 3.6 présentent les étapes qui permettent de corriger cette situation.

3.6 Sélection des cas de tests existants

En pratique, la réexécution systématique de tous les cas de tests existants pour tester P' suite à une modification peut s'avérer coûteuse. La sélection des tests existants permet d'économiser de précieux efforts de test en évitant aux testeurs d'avoir à reprendre l'ensemble des cas de tests existants pour tester P'. Cette approche s'avère très avantageuse dans des contextes tels que l'industrie aérospatiale où les tests peuvent s'avérer complexes et très dispendieux.

Soit un programme P qui possède une batterie de tests T, P' la version modifiée de P et C' l'ensemble des chemins d'exécution impactés de P'. Nous nous intéressons a déterminer S qui est un sous-ensemble de T tel que S correspond à l'ensemble de séquences de test couvrant les chemins impactés C'.

Pour tout t élément de T, t est un élément de S si et seulement si P'(t) peu être différent de P(t). Ce qui revient à dire que pour tout t élément de T, t est élément de S si et seulement si t implique un c' élément de C'.

Notre approche utilise en entrée les résultats obtenus à l'étape 3.4 ainsi que la batterie de tests existante de P (en l'occurrence les tests générés à l'étape 3.5). La façon de procéder consiste à sélectionner, à partir des ensembles T et I', l'ensemble S des séquences de tests existantes au niveau de la batterie et couvrant les chemins d'exécution impactés du programme P'. L'objectif est d'assurer une sélection conservatrice (safe), c'est-à-dire qui sélectionne tous les tests susceptibles de révéler un comportement différent lorsqu'exécuté sur P et P'. À la fin du processus, l'ensemble S et l'ensemble N des nouveaux cas de test générés seront jumelés pour former l'ensemble de tests T' permettant de couvrir tous le chemins d'exécution impactés de P'.

En plus de sélectionner les tests qui doivent être ré-exécutés, cette étape du processus permet aussi d'identifier l'ensemble E des chemins d'exécution de l'ensemble I' qui ne sont pas couverts par les tests existants de l'ensemble T. L'ensemble E sera utilisé ultérieurement afin de générer les nouveau cas de test qui couvrent les séquences d'exécution non couvertes par les tests existants.

Pour identifier les nouveaux cas de test nous procèdons ainsi : soit un programme modifié P', C' l'ensemble des chemins d'exécution impactés de P', S l'ensemble des tests sélectionnés à partir de la batterie de tests existante pour tester C'. Nous nous intéressons à déterminer E qui est un sous-ensemble de C' tel que : pour tout c' élément de C', c' est un élément de E si et seulement si c' n'est pas couvert par S.

La façon de faire consiste à générer les séquences d'exécution complètes de chacune des séquences d'exécution compactées de l'ensemble I'(séquences impactées). Pour chaque séquence d'exécution complète, le fichier contenant l'ensemble T est analysé afin de déterminer les tests la couvrant. Lorsqu'un cas de test est retenu, il est ajouté à l'ensemble S des tests existants couvrant les chemins d'exécution impactés du programme P'. Lorsqu'une séquence d'exécution n'est pas couverte par un cas de test existant dans l'ensemble T, elle est ajoutée à l'ensemble E des séquences non couvertes pas les tests existants.

```
public void creerPaiement(double montantPresente)
{
    if (paiement != null)
    {
        System.out.println(paiement.getMontant());
    }
    else
    {
        paiement=new Paiement();
        paiement.setMontant(montantPresente);
    }
}
```

Figure 33. Méthode NextGen.Transaction.creerPaiement(double).

Soient la version de la méthode "creerPaiement()" illustrée par la figure 33 et l'ensemble T illustré par la figure 32. Nous constatons que la méthode "creerPaiement()" possède deux chemins d'exécution possibles: [NextGen.Paiement.Paiement(),NextGen.Paiement.setMontant(double)] et [NextGen.Paiement.getMontant()]. Puisque le cas de test intégré "testSequence1" de l'ensemble T couvre la séquence d'exécution [NextGen.Paiement.Paiement(),NextGen.Paiement.setMontant(double)], celui-ci serait ajouté à l'ensemble des chemins impactés I'. Puisqu'aucun test de l'ensemble T ne couvre la séquence d'exécution [NextGen.Paiement.getMontant()], celle-ci serait ajoutée à l'ensemble E.

3.7 Génération des nouveaux cas de test

Tel que mentionné précédemment, T est conçu à la base pour tester les différents chemins d'exécution. Ceci implique que si de nouvelles fonctionnalités sont ajoutées à P, les nouveaux chemins d'exécution de P' ne sont pas couverts par l'ensemble T. Cet aspect est critique dans le cadre des systèmes industriels qui sont en constante évolution. La génération de nouveaux cas de test est donc nécessaire afin d'effectuer une couverture complète de P'. Par couverture complète, on entend par couvrir au moins chaque chemin d'exécution possible de P'.

Il faut aussi garder à l'esprit que si l'ensemble T présentait initialement des lacunes, c'est-à-dire qu'il ne permettait pas de couvrir tous les chemins d'exécution et méthodes de P, l'utilisation seule de T afin de couvrir P' est insuffisante même si aucun chemins d'exécution n'est ajouté dans P'. Dans ce cas-ci, la génération de nouveaux cas de test joue un rôle important dans l'amélioration de la batterie de tests originale et ainsi permettre une couverture complète de P'.

Les deux problématiques mentionnées ici-haut sont observées dans les approches similaires à la notre; c'est à dire les approches basées sur les arcs dangereux [Harrold 94, Harrold 00, Harrold 01] et celles basées sur le concept de firewall [Chen 97, White 97]. Contrairement à ces deux types d'approches, la notre supporte l'identification des chemins d'exécution non couverts par les tests existants et permet la génération automatique de nouveaux cas de test afin de les couvrir.

La génération des nouveaux cas de test requiert le résultat obtenu à l'étape 3.6; la liste des chemins d'exécution impactés non couverts par les tests existants. Cette étape consiste donc à générer, à partir de l'ensemble E, l'ensemble N' des nouveaux cas de test couvrant les séquences d'exécution non couvertes par les tests existants de l'ensemble T. L'ensemble N' et l'ensemble S généré à l'étape précédente seront jumelés pour former l'ensemble de tests T' qui servira à tester l'ensemble du programme P'.

Soit un programme P et sa version modifiée P', I' l'ensemble des chemins d'exécution impactés de P', T l'ensemble des tests couvrant l'ensemble des chemins d'exécution de P, S l'ensemble des tests de T couvrant I', E l'ensemble des éléments de I' non couverts par S, T' l'ensemble des tests couvrant I', nous nous intéressons à générer l'ensemble N' des nouveaux cas de test qui couvriront les chemins d'exécution de l'ensemble E précédemment obtenus. Il est à noter que T' est obtenu au final en additionnant S à N'.

On constate que la possibilité de générer des nouveaux cas de test provient de la capacité de notre approche à générer l'ensemble des chemins d'exécution possibles de P' par analyse statique du code. Ceci permet d'avoir une vue globale de l'ensemble des comportements possibles de P' sans être limité aux comportements couverts par la batterie de tests existante. Les approches similaires à la notre se contentent d'utiliser des données recueillies dynamiquement par instrumentation lors de l'exécution de la batterie de tests originale sur P. Cette façon de faire ne permet pas d'identifier les nouveaux chemins d'exécution et les chemins d'exécution existants qui ne sont pas couverts par la batterie de tests existante. Rappelons que dans le cas où un oubli est survenu initialement lors de la génération de l'ensemble de tests T, il est possible que celle-ci n'assure par une couverture totale du programme original P.

La génération des nouveaux cas de test utilise les séquences d'exécution impactées de P' qui ne sont pas couvertes par les tests existants sélectionnés à l'étape précédente. La figure 34 illustre le format des séquences d'exécution qui sont utilisées. Chaque séquence d'exécution, générée selon le processus décrit à la section 3.5, figurant dans cet ensemble fera l'objet d'un nouveau cas de test.

$M, M_1, M_2, M_7, M_8, M_9, M_6, M_8, M_9, M_{10}$
$M, M_1, M_2, M_7, M_8, M_9, M_6, M_{10}$
$M, M_1, M_3, M_8, M_9, M_6, M_{10}$
$M, M_1, M_2, M_7, M_8, M_9, M_4, M_5, M_6, M_8, M_9, M_{10}$
$M, M_1, M_3, M_8, M_9, M_4, M_5, M_6, M_8, M_9, M_{10}$
$M, M_1, M_2, M_7, M_8, M_9, M_5, M_5, M_4, M_5, M_6, M_8, M_9, M_{10}$
.....

Figure 34. Séquences d'exécution.

Supposons, dans l'exemple suivant, la version de la méthode "creerPaiement()" illustrée par la figure 33 et l'ensemble T illustrée par la figure 32.

```
public class TestsSequences extends TestCase {

    public static void main(String[] args) {
    }

    public void setUp() {

    }

    public void tearDown() {

    }

    public void NextGen_Paiement_getMontant____() {

    }

    public void testSequence1() {

        NextGen_Paiement_getMontant____();
    }

}
```

Figure 35. Classe de tests intégrés.

Puisqu'aucun test de l'ensemble T ne couvre la séquence d'exécution [NextGen.Paiement.getMontant()] et que celle-ci est ajoutée à l'ensemble E, la classe de test illustrée par la figure 35 serait générée.

CHAPITRE 4

ÉVALUATION DE L'APPROCHE

4.1 Comparaison avec les approches similaires

Comme notre approche, l'approche proposée par Harrold et al. [Harrold 01] est considérée comme étant conservatrice; c'est à dire qu'elle sélectionne tout les tests existants susceptibles de démontrer des erreurs. La technique décrite dans le papier [Harrold 01] consiste en une adaptation pour Java de l'approche basée sur les arcs dangereux présentée précédemment [Harrold 97]. L'approche d'Harrold et al., contrairement à la notre, requière l'instrumentation du code (processus assez couteux) afin de capturer une matrice de couverture de l'ensemble des cas de test existants sur le programme P. L'instrumentation peut être effectuée de deux façons. La première façon consiste à insérer du code au niveau du programme P afin d'inscrire, dans un fichier ou une base de donnée, les arcs couverts par chacun des cas de test lors de leur exécution. La deuxième façon consiste à modifier l'environnement d'exécution, plutôt que le programme P, afin d'effectuer le même traitement. Ces tâches requièrent des efforts considérables et peuvent s'avérer couteuses surtout dans le cadre de programmes complexes.

Dans certaines situations, cette approche a tendance, due à sa granularité, à sélectionner moins de cas de tests que notre approche. Par exemple, si une méthode modifiée comporte plusieurs chemins d'exécution et que seulement un des chemins d'exécution est modifié, l'approche d'Harrold et al. [Harrold 01] sélectionnera les cas de tests qui couvrent uniquement le chemin d'exécution modifié. Nous avons préféré dans notre approche adopter une démarche qui consiste à re-tester toute la méthode par précaution. Cet aspect est démontré dans un exemple théorique à la section 4.5.

L'approche d'Harrold et al. [Harrold 01] s'intéresse uniquement à la réutilisation des cas de test existants. Elle ne supporte pas la génération des nouveaux cas de test. La

technique consiste à réutiliser les traces d'exécution des cas de test existants. Elle permet de couvrir uniquement les chemins d'exécution qui existaient dans P sans couvrir les chemins d'exécution modifiés (structure) ou ajoutés dans P', contrairement à notre approche qui permet de couvrir les nouveaux chemins d'exécution.

L'approche orientée objet de White et al. [White 92] constitue une adaptation de l'approche par firewall supportant les programmes procéduraux [White 90]. Contrairement à notre approche, l'approche de White et al., suppose que les spécifications fonctionnelles du programme original ainsi que celles des changements effectués sont disponibles. Cet aspect peut s'avérer problématique dans un contexte industriel où les systèmes sont souvent composés de plusieurs applications et où la documentation n'est pas toujours mise à jour et parfois même non disponible.

L'approche de White & al. [White 92] s'applique à un niveau de granularité plus élevé et ne tient pas compte des chemins de contrôle. Elle est donc moins précise que la notre. Elle a tendance à sélectionner plus de cas de test que notre approche. Par exemple, si une méthode etait modifiée dans un programme, l'approche de White et al., sélectionnerait tous les cas de test impliquant la classe modifiée tandis que notre approche sélectionnerait seulement les cas de test impliquant la méthode modifiée. Cet aspect est présenté dans les sections 4.2, 4.3, 4.4 et 4.5. L'approche par firewall s'intéresse uniquement à la réutilisation et ne supporte pas la génération de nouveaux cas de test.

Notre approche est dite conservatrice (safe) puisque, sous certaines conditions, tous les tests existants susceptibles de démontrer des erreurs sont sélectionnés. En d'autres mots, l'approche n'éliminera jamais des tests qui pourraient révéler des erreurs découlant des modifications. Les conditions requises pour le bon fonctionnement de la technique sont les suivantes : les changements doivent être effectués au niveau des méthodes (non sur les attributs ou variables globales) et la batterie de tests d'origine doit respecter le format décrit à la section 3.4.

Notre approche ne requiert aucune instrumentation du code source original. Elle utilise plutôt l'analyse statique automatique du code afin de déterminer les chemins d'exécution couverts par chaque test. Les tests sont sélectionnés à partir de l'ensemble des chemins d'exécution possibles de P', plutôt que de se limiter aux tests existants relatifs à P. Elle permet, également, d'identifier les nouveaux chemins d'exécution dans P' et de générer des nouveaux cas de test relatifs à ces chemins, ce qui permettra d'augmenter de façon évolutive la batterie de tests relative à un programme donné et de la maintenir à jour.

Supposons une petite application à laquelle nous ajouterions une nouvelle fonctionnalité. Dans ce contexte, les tests existants ne couvrent pas les nouveaux chemins d'exécution introduits par la fonctionnalité ajoutée puisqu'elle n'existait pas à l'origine, et encore moins les liens qui peuvent exister entre la nouvelle fonctionnalité et les fonctionnalités du programme (impact). L'approche d'Harrold et al. et celle de White et al. seraient alors inefficaces pour détecter les nouveaux chemins d'exécution à tester alors que notre approche permettrait non seulement d'identifier les nouveaux chemins d'exécution mais également de générer les nouveaux cas de test pour les couvrir.

Cette caractéristique constitue un avantage important dans le cadre d'applications industrielles complexes qui subissent régulièrement des modifications suites aux changements de spécification. La couverture des nouveaux chemins d'exécution et la génération de nouveaux cas de test constituent un apport majeur par rapport aux techniques par firewall et celles basées sur les arcs dangereux. Cet aspect permet, entre autres, de couvrir de nouvelles fonctionnalités et de compléter une suite de tests originale qui ne couvrent pas l'ensemble des chemins d'exécution possibles de la version P'. Des exemples théoriques d'augmentation de batterie de tests sont présentés dans les sections 4.3 et 4.4.

Par ailleurs, étant donné que notre approche sélectionne les tests en termes de

méthodes modifiées et qu'elle considère les chemins de contrôle, elle est plus précise que les approches par firewall qui effectuent l'analyse en termes de classes. Par contre, nous constatons dans certaines situations que les approches basées sur les arcs dangereux sont d'une plus grande précision que notre approche. Cet aspect est présenté à la section 4.5.

Voici un tableau comparatif présentant les différences entre les 3 approches. La colonne "Conservatrice" indique que l'approche permet la sélection conservatrice (safe) de cas de test existants, c'est-à-dire qu'elle permet de sélectionner tous les tests susceptibles de démontrer une faute dans la version modifiée du programme P. La colonne "Supporte la génération" indique que l'approche permet de générer des nouveaux cas de test afin de couvrir les chemins non couverts par la batterie de tests existante. La colonne "Instrumentation" indique que l'approche requiert l'instrumentation de P et la colonne "Spécifications" indique que l'approche requiert les spécifications de P.

Approche	**Conservatrice**	**Supporte la génération**	**Requiert l'instrumentation**	**Requiert les spécifications**
Harrold et al.	x		x	
White et al.	x		x	x
Notre approche	x	x		

Tableau 1. Comparaison entre les 3 approches.

La section suivante présente différents exemples théoriques permettant d'illustrer (et comparer) le comportement de notre approche par rapport aux approches similaires.

4.2 Exemple théorique #1

L'exemple qui suit présente une situation typique où l'approche d'Harrold et al. ainsi que la notre donnent les mêmes résultats. Soit le code original présenté à la figure 36 et la batterie de test T (t1 … t4) présentée dans le tableau 2 :

```
Class A
{
        [...]
        Public static void main(String[] args)
        {
                B.m1();    //1
                If (var4 = 1)    //2
                {
                        B.m2();    //3
                }
                else    //4
                {
                        B.m3();    //5
                }
        }
}
Class B
{
        [...]
        Public static void m1()
        {
                If (var2 > 0)   //6
                {
                        m2();    //7
                }
                else    //8
                {
                        m3();    //9
                }
        }
        Public static void m2()
        {
                var1 = var1 + 1;    //10
        }
        Public static void m3()
        {
                var1 = var1 - 1;    //11
        }
}
```

Figure 36. Classes originales A et B.

La première colonne du tableau 2 présente les tests ti de la suite de tests originale dont nous disposons. La deuxième colonne indique les chemins d'exécution en termes d'arcs (selon l'approche d'Harrold et al.) correspondant à chacun des cas de test. La troisième colonne indique les chemins d'exécution en termes de méthode (selon notre approche) et la quatrième colonne indique les classes impliquées (selon l'approche de White et al.).

Puisque l'approche d'Harrold et al. utilise les arcs dangereux pour déterminer les chemins d'exécution à couvrir, la colonne "Arcs" correspond aux arcs impliqués dans chacun des cas de test ti de la batterie existante. Ces informations sont obtenues par instrumentation du code. Un arc est représenté par un couple d'instructions. Par exemple, l'arc (1,6) signifie que lors de son exécution, le programme passe de l'instruction #1 à l'instruction #6, tel que c'est annoté dans la figure 36.

L'approche de White et al. utilise la notion de classes afin de déterminer les chemins d'exécution à couvrir. Ces données sont obtenues encore une fois par instrumentation du code original. La colonne "Classes" correspond donc à chacune des classes impliquées par un cas de test ti.

Notre approche utilise les chemins d'exécution en termes de méthodes pour déterminer les chemins d'exécution à couvrir. Ces informations sont obtenues à partir de l'analyse statique des cas de test de la batterie existante. La colonne "Chemin d'exécution" correspond donc à chacun des chemins d'exécution obtenus pour chaque cas de test ti lors de l'analyse statique.

Test	**Arcs**	**Chemin d'exécution**	**Classes**
t1	(1,6) (6,7) (7,2) (2,3) (3,10)	m1, m2, m2	A, B
t2	(1,6) (6,7) (7,10) (10,2) (2,4) (4,5) (5,11)	m1, m2, m3	A, B
t3	(1,6) (6,8) (8,9) (9,2) (2,3) (3,10)	m1, m3, m2	A, B
t4	(1,6) (6,8) (8,9) (9,2) (2,4) (4,5) (5,11)	m1, m3, m3	A, B

Tableau 2. Ensemble de tests original de l'exemple #1.

Soit le code original de la figure 36 auquel nous avons apporté une modification (figure 37) :

```
class A
{
        [...]
        Public static void main(String[] args)
        {
                B.m1();   //1
                If (var4 = 1)   //2
                {
                        B.m2();   //3
                }
                else   //4
                {
                        B.m3();   //5
                }
        }
}
class B
{
        [...]
        Public static void m1()
        {
                If (var2 > 0)  //6
                {
                        m2();   //7
                }
                else   //8
                {
                        m3();   //9
                }
        }
        Public static void m2()
        {
                var1 = var1 + 2;   //10
        }
        Public static void m3()
        {
                var1 = var1 - 1;   //11
        }
}
```

Figure 37. Version modifiée de figure 36

L'approche d'Harrold et al. sélectionne les tests impliquant des arcs dangereux. Un arc dangereux constitue un arc impliquant une instruction modifiée. Suite à la modification effectuée à la méthode m2 figurant en rouge, la méthode d'Harrold et al. détectera donc les arcs dangereux (3,10), (7,10), (10,2). Suite à cette identification, l'approche sélectionnera donc les tests impliquant ces arcs dangereux : t1, t2 et t3. Puisque l'approche de White et al. sélectionnera les cas de test en termes de classes et que la classe B a été modifiée, les tests t1, t2, t3 et t4 seront sélectionnés puisqu'ils impliquent tous la classe B. Notre approche cherche à identifier les tests impliquant des méthodes modifiées. Puisque l'approche ne détecte que m2 comme méthode modifiée, elle sélectionnera donc les tests impliquant cette méthode, soient : t1, t2 et t3.

Dans cet exemple, nous remarquons que les tests existants qui sont réellement susceptibles de démontrer des erreurs découlant des modifications sont les tests t1, t2 et

t3 puisqu'ils sont les seuls à exécuter du code modifié. Les 3 approches ont sélectionné ces tests. Elles peuvent donc être considérées comme étant conservatrices. Par contre, bien que dans ce cas notre approche et celle d'Harrold et al. donnent le même résultat en termes de sélection, nous constatons que celle de White et al. est moins précise puisqu'elle sélectionne plus de cas de test pour assurer le même niveau de confiance.

4.3 Exemple théorique #2

L'exemple qui suit présente une situation typique où notre approche, contrairement à celle d'Harrold et al., permet de corriger une suite de tests originale présentant initialement des lacunes. Soit le code original présenté à la figure 38 et la batterie de test T présentée dans le tableau 3 :

```
Class A
{
        [...]

        Public static void main(String[] args)
        {
                B.m1();    //1
                If (var4 = 1)    //2
                {
                        B.m2();    //3
                }
                else    //4
                {
                        B.m3();    //5
                }
        }
}
Class B
{
        [...]

        Public static void m1()
        {
                If (var2 > 0)  //6
                {
                        m2();    //7
                }
                else    //8
                {
                        m3();    //9
                }

        }
        Public static void m2()
        {
                var1 = var1 + 1;    //10
        }

        Public static void m3()
        {
                var1 = var1 - 1;    //11
        }
}
```

Figure 38. Classes originales A et B.

Test	Arcs	Chemin d'exécution	Classes
t1	(1,6) (6,7) (7,2) (2,3) (3,10)	m1, m2, m2	A, B
t3	(1,6) (6,8) (8,9) (9,2) (2,3) (3,10)	m1, m3, m2	A, B
t4	(1,6) (6,8) (8,9) (9,2) (2,4) (4,5) (5,11)	m1, m3, m3	A, B

Tableau 3. Ensemble de tests originale de l'exemple #2.

Soit le code original de la figure 38 auquel nous avons apporté une modification (figure 39):

```
Class A
{
        [...]
        Public static void main(String[] args)
        {
                B.m1();   //1
                If (var4 = 1)   //2
                {
                        B.m2();   //3
                }
                else   //4
                {
                        B.m3();   //5
                }
        }
}
Class B
{
        [...]
        Public static void m1()
        {
                If (var2 > 0)  //6
                {
                        m2();   //7
                }
                else   //8
                {
                        m3();   //9
                }
        }
        Public static void m2()
        {
                var1 = var1 + 2;   //10
        }
        Public static void m3()
        {
                var1 = var1 - 1;   //11
        }
}
```

Figure 39. Version modifiée de la figure 38

Dans cet exemple, une faille est présente au niveau de la suite de tests originale, c'est à dire que le chemin d'exécution m1, m2, m3 n'est pas couvert par les tests existants. Suite à la modification de la méthode m2 figurant en rouge, l'approche

d'Harrold et al. se basera sur les données obtenues par instrumentation du code pour déterminer les chemins d'exécution à couvrir et elle sélectionnera les tests t1 et t3. L'approche de White et al. identifiera la classe B comme étant modifiée, les tests t1, t3 et t4 seront sélectionnés. Notons que les deux approches laisseront le chemin d'exécution m1, m2, m3 non couvert suite au changement.

Dans cette situation, notre approche sélectionnera les tests t1 et t3 mais permettra aussi de détecter que le chemin m1, m2, m3 n'est pas couvert par les tests existant et ainsi générer un nouveau cas de test t5 pour le couvrir.

Même si le chemin d'exécution m1, m2, m3 n'était pas couvert par la suite de tests originale, notre approche permet dans ce cas de la compléter et de pallier à cette lacune. Ce n'est pas le cas des deux autres approches. Cet exemple démontre donc que la génération de nouveaux cas de test n'est pas seulement utile dans le cas où de nouveaux chemins d'exécution sont introduits. Elle permet de couvrir des chemins d'exécution existants omis lorsque la première version du programme a été testée. Voici la batterie de tests suite à l'augmentation :

Test	**Arcs**	**Chemin d'exécution**	**Classes**
t1	(1,6) (6,7) (7,2) (2,3) (3,10)	m1, m2, m2	A, B
t3	(1,6) (6,8) (8,9) (9,2) (2,3) (3,10)	m1, m3, m2	A, B
t4	(1,6) (6,8) (8,9) (9,2) (2,4) (4,5) (5,11)	m1, m3, m3	A, B
t5	---	m1, m2, m3	---

Tableau 4. Ensemble de tests augmenté de l'exemple #2.

Dans cet exemple, nous remarquons que les tests existants qui sont réellement susceptibles de démontrer des erreurs découlant des modifications sont les tests t1 et t3. Puisque les 3 approches ont dans ce cas sélectionné ces tests, elles peuvent être considérées comme étant conservatrices. Bien que dans ce cas notre approche et celle d'Harrold et al. donnent le même résultat en termes de sélection, nous constatons que

celle de White et al. est moins précise puisqu'elle sélectionne plus de cas de test pour assurer le même niveau de confiance. Il est à noter que, contrairement aux deux autres approches, notre approche a l'avantage de couvrir des chemins d'exécution qui n'étaient pas couverts par la batterie de tests existante.

4.4 Exemple théorique #3

L'exemple qui suit présente une situation typique où notre approche, contrairement à celles d'Harrold et al. et celle de White et al, permet de couvrir de nouveaux chemins d'exécution. Soit le code original présenté à la figure 40 et la batterie de test T présentée par le tableau 5 :

```
Class A
{
        [...]
        Public static void main(String[] args)
        {
                B.m1();    //1
                If (var4 = 1)    //2
                {
                        B.m2();    //3
                }
                else    //4
                {
                        B.m3();    //5
                }
        }
}
Class B
{
        [...]
        Public static void m1()
        {
                If (var2 > 0)   //6
                {
                        m2();    //7
                }
                else    //8
                {
                        m3();    //9
                }
        }
        Public static void m2()
        {
                var1 = var1 + 1;    //10
        }
        Public static void m3()
        {
                var1 = var1 - 1;    //11
        }
}
```

Figure 40. Classes originales A et B.

Test	Arcs	Chemin d'exécution	Classes
t1	(1,6) (6,7) (7,2) (2,3) (3,10)	m1, m2, m2	A, B
t2	(1,6) (6,7) (7,10) (10,2) (2,4) (4,5) (5,11)	m1, m2, m3	A, B
t3	(1,6) (6,8) (8,9) (9,2) (2,3) (3,10)	m1, m3, m2	A, B
t4	(1,6) (6,8) (8,9) (9,2) (2,4) (4,5) (5,11)	m1, m3, m3	A, B

Tableau 5. Ensemble de tests originale de l'exemple #3.

Soit le code original de la figure 40 auquel nous avons apporté une modification (figure 41):

```
Class A
{
        [...]
        Public static void main(String[] args)
        {
                B.m1();    //1
                If (var4 = 1)    //2
                {
                        B.m2();    //3
                }
                else    //4
                {
                        B.m3();    //5
                }
        }
}
Class B
{
        [...]
        Public static void m1()
        {
                If (var2 > 0)  //6
                {
                        m2();    //7
                }
                else    //8
                {
                        m3();    //9
                }
        }
        Public static void m2()
        {
                var1 = var1 + 1;    //10
        }
        Public static void m3()
        {
                If (var1 > 0)    //11
                {
                        var1 = var1 - 1;    //12
                }
                Else    //13
                {
                        m2();    //14
                }
        }
}
```

Figure 41. Version modifiée de la figure 40

Suite à la modification apportée à la méthode m3 figurant en rouge, la méthode d'Harrold et al. détecte l'arc dangereux (5,11) et sélectionnera donc les tests impliquant cet arc dangereux : t2 et t4. Puisque l'approche de White et al. sélectionne les tests en termes de classes et que la classe B a été modifiée, les tests t1, t2, t3 et t4 seront sélectionnés. Notre approche détecte que m3 est une méthode modifiée et sélectionnera donc les tests impliquant cette méthode modifiée : t2, t3 et t4.

Contrairement à l'approche de Harrold et al, et à celle de White et al., notre approche sera en mesure de déterminer que les nouveaux chemins d'exécution (m1, m2, m3, m2) et (m1, m2, m3, m2) doivent être testés puisqu'ils impliquent un appel à la méthode modifiée m3. Puisqu'aucun des tests existants ne couvrent ces chemins d'exécution, elle générera 2 nouveaux cas de test t5 et t6 qui couvriront ces 2 nouveaux chemins d'exécution. Au final, voici la batterie de tests suite à l'augmentation :

Test	Arcs	Chemin d'exécution	Classes
t1	(1,6) (6,7) (7,2) (2,3) (3,10)	m1, m2, m2	A, B
t2	(1,6) (6,7) (7,10) (10,2) (2,4) (4,5) (5,11)	m1, m2, m3	A, B
t3	(1,6) (6,8) (8,9) (9,2) (2,3) (3,10)	m1, m3, m2	A, B
t4	(1,6) (6,8) (8,9) (9,2) (2,4) (4,5) (5,11)	m1, m3, m3	A, B
t5	---	m1, m2, m3, m2	---
t6	---	m1, m2, m3, m2	---

Tableau 6. Ensemble de tests augmenté de l'exemple #3.

Dans cet exemple, nous remarquons que les tests existants qui sont réellement susceptibles de démontrer des erreurs provoquées par les modifications sont les tests t2 et t4. Puisque les 3 approches ont, dans ce cas, sélectionné ces tests, elles peuvent être considérées comme étant conservatrices. Bien que l'approche d'Harrold et al. soit plus précise que la notre en termes de sélection, puisqu'elle sélectionne moins de cas de test pour assurer le même niveau de confiance, notre approche à l'avantage de couvrir de

nouveaux chemins d'exécution qui ne sont pas couverts par les tests existants.

4.5 Exemple théorique #4

L'exemple qui suit présente une situation typique où l'approche d'Harrold et al. est plus précise que la notre. Soit le code original présenté à la figure 42 et la batterie de tests T présentée par le tableau 7 :

```
Class A
{
        [...]
        Public static void main(String[] args)
        {
                B.m1();   //1
                If (var4 = 1)   //2
                {
                        B.m2();   //3
                }
                else   //4
                {
                        B.m3();   //5
                }
        }
}
Class B
{
        [...]
        Public static void m1()
        {
                If (var2 > 0)  //6
                {
                        m2();   //7
                }
                else   //8
                {
                        m3();   //9
                }
        }
        Public static void m2()
        {
                var1 = var1 + 1;   //10
        }
        Public static void m3()
        {
                If (var1 > 0)   //11
                {
                        var1 = var1 - 1;   //12
                }
                Else   //13
                {
                        m2();   //14
                }
        }
}
```

Figure 42. Classes originales A et B.

Test	Arcs	Chemin d'exécution	Classes
t1	(1,6) (6,7) (7,2) (3,2) (3,10)	m1, m2, m2	A, B
t2	(1,6) (6,7) (7,2) (2,4) (4,5) (5,11) (11,12)	m1, m2, m3	A, B
t3	(1,6) (6,7) (7,2) (2,4) (4,5) (5,11) (11,13) (13,14) (14,10)	m1, m2, m3, m2	A, B
t4	(1,6) (6,8) (8,9) (9,11) (11,12) (12,2) (2,3) (3,10)	m1, m3, m2	A, B
t5	(1,6) (6,8) (8,9) (9,11) (11,13) (13,14) (14,10) (10,2) (2,3) (3,10)	m1, m3, m2, m2	A, B
t6	(1,6) (6,8) (8,9) (9,11) (11,12) (12,2) (2,4) (4,5) (5,11) (11,12)	m1, m3, m3	A, B
t7	(1,6) (6,8) (8,9) (9,11) (11,13) (13,14) (14,10) (10,2) (2,4) (4,5) (5,11) (11,12)	m1, m3, m2, m3	A, B
t8	(1,6) (6,8) (8,9) (9,11) (11,12) (12,2) (2,4) (4,5) (5,11) (11,13) (13,14) (14,10)	m1, m3, m3, m2	A, B
t9	(1,6) (6,8) (8,9) (9,11) (11,13) (13,14) (14,10) (10,2) (2,4) (4,5) (5,11) (11,13) (13,14) (14,10)	m1, m3, m2, m3, m2	A, B

Tableau 7. Ensemble de tests original de l'exemple #4.

Soit le code original de la figure 42 auquel nous avons apporté une modification (figure 43):

```
Class A
{
        [...]
        Public static void main(String[] args)
        {
                B.m1();
                If (var4 = 1)
                {
                        B.m2();
                }
                else
                {
                        B.m3();
                }
        }
}
Class B
{
        [...]
        Public static void m1()
        {
                If (var2 > 0)
                {
                        m2();
                }
                else
                {
                        m3();
                }
        }
        Public static void m2()
        {
                var1 = var1 + 1;
        }
        Public static void m3()
        {
                If (var1 > 0)
                {
                        var1 = var1 - 2;
                }
                Else
                {
                        m2();
                }
        }
}
```

Figure 43. Version modifiée de la figure 42

Suite à la modification apportée à la méthode m3 figurant en rouge, la méthode d'Harrold et al. détecte l'arc dangereux (11,12) et sélectionnera donc les tests impliquant cet arc dangereux : t2, t4, t6, t7 et t8. L'approche de White et al. identifiera la classe B comme état modifiée, les tests t1, t2, t3, t4, t5, t6, t7, t8 et t9 seront sélectionnés.

Notre approche détecte que m3 est une méthode modifiée et sélectionnera donc les tests impliquant cette méthode modifiée : t2, t3, t4, t5, t6, t7, t8 et t9. Nous remarquons donc que dans cette situation, l'approche d'Harrold et al. est plus précise en sélectionnant 3 cas tests en moins due à sa granularité (instructions vs méthodes).

Dans cet exemple, nous remarquons que les tests existants qui sont réellement susceptibles de démontrer des erreurs découlant des modifications sont les tests t2, t4, t6, t7 et t8. Puisque les 3 approches ont dans ce cas sélectionné ces tests, elles peuvent être considérées comme étant conservatrices. Par contre, l'approche d'Harrold et al. est plus précise que la notre et que celle de White et al. en termes de sélection. En effet, elle sélectionne moins de cas de test pour assurer le même niveau de confiance. Notons aussi que notre approche est plus précise que celle de White et al. puisqu'elle retient moins de cas de test.

CHAPITRE 5

PRÉSENTATION DE L'OUTIL

5.1 Principales composantes

Notre approche est supportée par un outil entièrement développé en Java 1.4 sous Eclipse 3.3. L'application est composée de différents modules dont certains ont été récupérés et adaptés à partir de travaux d'autres membres de l'équipe (projets connexes). Pour effectuer chacune des étapes décrites à la section 3, un module indépendant, avec ses paramètres en entrée et en sortie, a été développé. La figure 44 présente un aperçu de chacun des modules composant notre implémentation.

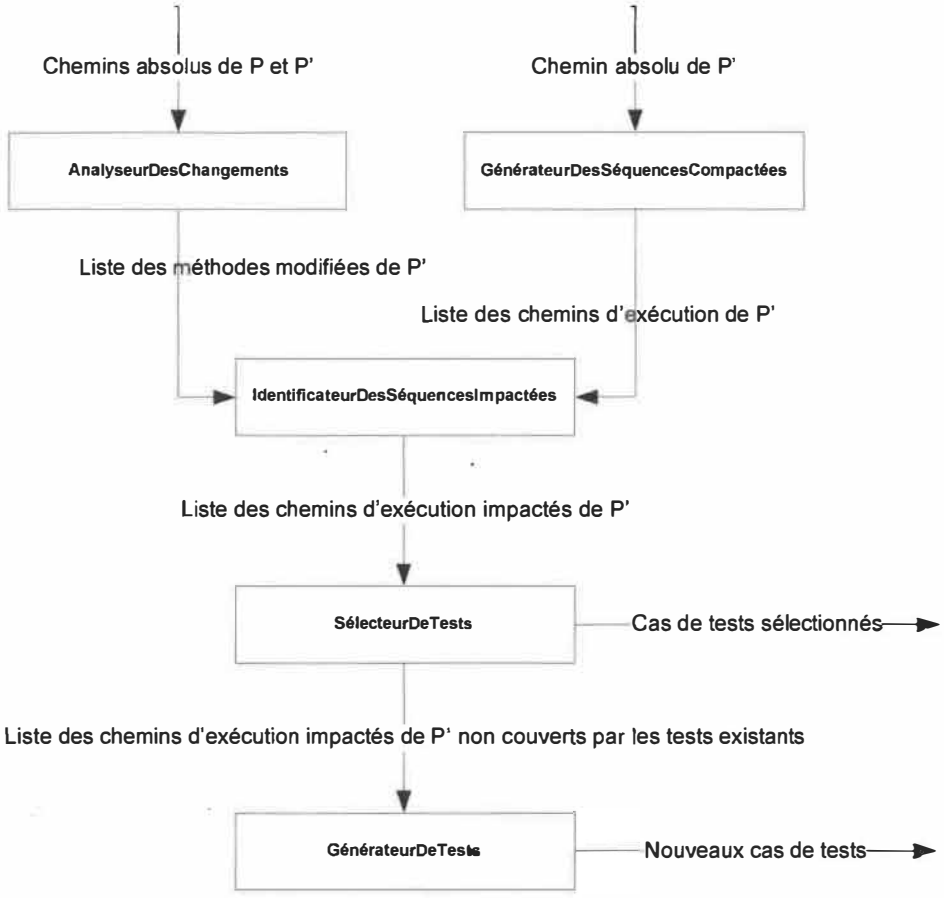


Figure 44. Principales composantes.

Le module d'analyse des changements (AnalyseurDesChangements) prend en entrée le nom absolu (ex : "C:\workspace\NextGen") des deux répertoires contenant les programmes P et P'. La comparaison entre chaque fichier est effectuée à partir de la librairie "org.eclipse.compare.CompareUI" d'Eclipse 3.3. Le résultat de la comparaison est analysé et les méthodes modifiées sont retenues dans une liste pour constituer les données en sortie de ce module. Le résultat sera utilisé par le module d'identification des chemins d'exécution impactés (IdentificateurDesSéquencesImpactées).

Le module de compactage des chemins d'exécution (GénérateurDesSéquencesCompactées) a été récupéré des travaux de D. St-Yves [St-Yves 08] et adapté afin d'être intégré à notre outil. Il prend en entrée le nom absolu du répertoire contenant le programme P' et retourne la liste de tous les chemins d'exécution possible de P'. Les données en sortie de ce module seront utilisées par le module d'identification des chemins d'exécution impactés (IdentificateurDesSéquencesImpactées).

Le module d'identification des chemins d'exécution impactés (IdentificateurDesSéquencesImpactées) utilise en entrée la liste des méthodes modifiées et la liste des chemins d'exécution impactés générés par les modules AnalyseurDesChangements et GénérateurDesSéquencesCompactées respectivement. L'algorithme effectue l'identification par analyse textuelle des signatures de méthodes et retourne les chemins d'exécution impactés sous forme de liste. La liste est utilisée par le module de sélection des tests existants.

Le module de sélection des tests existants (SélecteurDeTests) utilise les classes de test existantes de P et la liste des chemins d'exécution impactés. La sélection effectue l'analyse statique des classes de test et permet d'identifier les tests qui couvrent les chemins d'exécution impactés. En plus de retourner en sortie une liste des tests sélectionnés, elle retourne la liste des chemins d'exécution impactés non couverts par les

tests existants. Le résultat sera utilisé par le module de génération des nouveaux cas de tests.

Le module de génération des nouveaux cas de tests (GénérateurDeTests) a été récupéré des travaux de P.L. Vincent [Vincent 09] et adapté afin d'être intégré à notre outil. Il utilise en entrée la liste des chemins d'exécution impactés non couverts par les tests existants et permet de générer des cas de test unitaire et intégrés sous le format Junit pour chacune des méthodes de la liste. Ce Module générera en sortie des fichiers de tests qui compléteront, avec la liste des tests existants sélectionnés, la suite de tests requise pour tester P'.

5.2 Fonctionnement de l'application

Le fonctionnement de l'outil est très simple. L'interface utilisateur, présentée à la figure 45, permet de sélectionner le répertoire des deux programmes à analyser. L'utilisateur appuie ensuite sur le bouton "Start" pour démarrer le traitement.

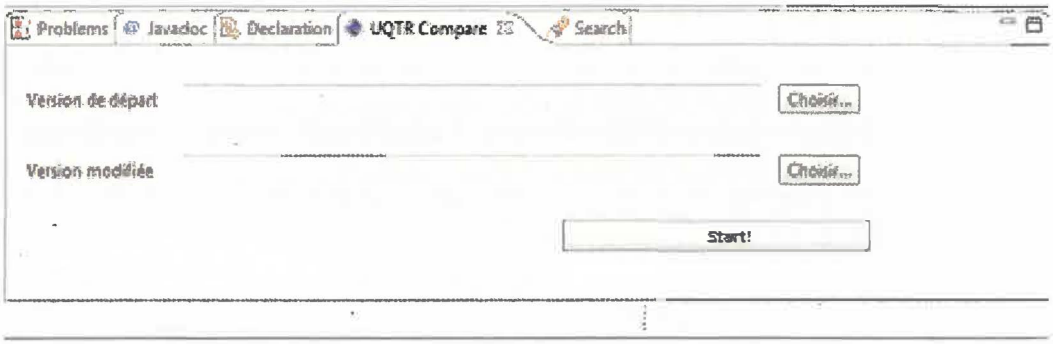


Figure 45. Interface utilisateur.

Au fur et à mesure que le traitement progresse, les résultats sont affichés dans la console Eclipse. À la fin du processus, les tests sélectionnés sont affichés (figure 46) et le fichier des nouveaux cas de test généré en format Junit, nommé "TestsSequences.java", est déposé dans le répertoire source du programme P' (ex. : "c:\workspace\NextGen\TestsSequences.java").

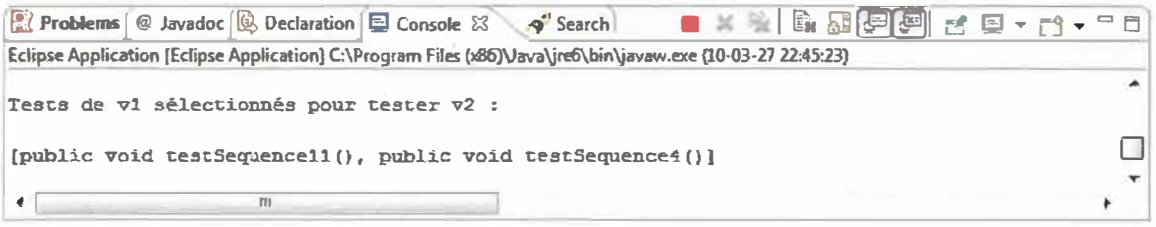


Figure 46. Résultats des tests sélectionnés.

CHAPITRE 6

ÉVALUATION EMPIRIQUE

6.1 Objectif

L'objectif de l'évaluation empirique est de mesurer et d'interpréter les résultats obtenus, à partir de critères d'évaluation, lors de l'application de notre approche sur de véritables études de cas. Pour évaluer notre approche, nous avons retenus un certain nombre de critères définis dans plusieurs travaux de la littérature. Notre objectif était aussi de proposer une sorte d'intégration de ces critères dans le but de procéder à une évaluation objective la plus large possible tenant compte de plusieurs facettes complémentaires des tests de régression. Nous avons retenu, par ailleurs, différentes métriques permettant de valider ces critères. Pour évaluer les études de cas, une démarche et un protocole d'expérimentation ont été établis.

6.2 Critères d'évaluation

Les travaux d'Harrold et al. relatifs à la technique de sélection de tests de régression supportant le langage Java [Harrold 01] proposent une métrique de réduction en termes de nombre de cas de test afin de déterminer l'efficacité de leur approche. L'évaluation de l'approche qui est présentée dans [Harrold 01] s'intéresse aussi à la granularité de l'approche proposée. Dans ce cas, l'étude s'intéresse à savoir si une réduction additionnelle en termes de nombre de cas était obtenue en basant l'approche sur les arcs dangereux (instructions) plutôt que sur les méthodes dangereuses. Il est à noter que la métrique de réduction en termes de nombre de cas est aussi utilisée dans l'étude empirique présentée par White et al. [White 03].

L'étude d'efficacité présentée par Wong & al. [Wong 97] propose de son côté, l'utilisation de trois métriques: la réduction, la précision et le rappel. La métrique de réduction en termes de nombre de cas est similaire à celle décrite précédemment mais les

métriques de précision et de rappel constituent un apport intéressant qui n'a pas été considéré dans les travaux d'Harrold et al. [Harrold 01].

Nous avons évalué les résultats à partir de la métrique de réduction, telle qu'utilisée par Harrold & al. [Harrold 01], White et al. [White 03] et Wong & al. [Wong 97] mais aussi à partir des métriques de précision et de rappel présentées par Wong & al. [Wong 97]. L'évaluation de la granularité a été ignorée dans notre cas puisque l'approche que nous proposons se situe au niveau des méthodes uniquement. Le niveau méthode qui a été adopté dans le cadre de notre approche constitue, en fait, un niveau de granularité intermédiaire, entre le niveau classe et le niveau instruction. Il permet d'être plus précis que le niveau classe et moins complexe et moins coûteux (au niveau de son utilisation) que le niveau instruction. Pour ce qui est de l'évaluation de la portion génération de nouveaux cas, Ammann, Black & al. [Ammann 02] qui ont présenté une approche utilisant le model checking pour générer des ensembles de cas de test basés sur l'analyse des changements, évaluent l'efficacité de leur approche à partir d'une adaptation de la métrique de couverture présentée par Wu et al. [Wu 88]. Dans le cas de notre évaluation, nous utiliserons une adaptation de cette même métrique de couverture.

6.3 Métriques

La présente section défini le détail des métriques que nous avons retenues et utilisées pour évaluer les résultats de notre expérimentation. Elle aborde, de façon théorique, les différents aspects mesurés lors de l'évaluation empirique.

La **réduction** de la taille en termes de nombre de cas permet de démontrer le gain d'efficacité obtenu en utilisant notre approche de sélection plutôt que d'effectuer l'exécution de tous les tests de T [Wong 97]. Le pourcentage de réduction de l'ensemble T peut être obtenu à partir de l'équation : (1 - (Tselect / T)) * 100 [Wong 97]. Notez bien que la réduction de la taille en termes de nombre de cas peut avoir une valeur négative dans le cas ou plus de tests sont sélectionnés dans le but de couvrir des ajouts de code plutôt que du code modifié [Wong 97]. Soit un ensemble de tests original T contenant 35

tests, et un ensemble de tests sélectionnés Tselect de 20 tests, la réduction est calculée par l'équation : 1 – (20/35) * 100 = 43%.

Une perte peut résulter de la réduction de la taille en termes de nombre de cas [Wong 97]. Celle-ci peut être mesurée en termes de **précision** et de **rappel** [Wong 97]. Supposons que l'ensemble de tests T contient r tests [Wong 97]. Parmi ceux-ci, n tests (n < r) résultent en un comportement différent lorsque exécutés sur P et P' [Wong 97]. L'ensemble Tselect, sous ensemble de T qui contient m tests (m ≠ 0), est déterminé à partir de l'approche de sélection de tests de régression C [Wong 97]. Parmi ces m tests, l tests peuvent distinguer P de P' [Wong 97]. La précision de Tselect relativement à P, P', T et C est le pourcentage donné par l'expression : 100*(l / m) [Wong 97]. Le rappel est le pourcentage donné 100*(l / n) si n ≠ 0 [Wong 97] ou 100% si n = 0.

La précision d'un ensemble de tests est le pourcentage de ces tests pour lesquels l'ancien et le nouveau programme pourraient produire des résultats différents [Wong 97]. Soit un ensemble de tests sélectionnés Tselect contenant 9 tests, dans lequel 7 tests peuvent se comporter différemment sur P et P', la précision de Tselect relativement à la modification est calculée par l'équation 100 * (7/9)= 77.78% [Wong 97].

Le rappel d'un ensemble de tests est le pourcentage de tests de régression sélectionnés parmi ceux qui doivent être ré-exécutés [Wong 97]. Soit un ensemble de tests T contenant 16 tests qui permettent de démontrer un comportement différent lorsqu'exécutés sur P et P' et parmi lesquels 7 tests font partie de l'ensemble des tests sélectionnés Tselect, le rappel de Tselect sur la modification est calculé par l'équation : 100 * (7/16) = 43.75% [Wong 97].

Un bas pourcentage de rappel indique que plusieurs cas de tests qui auraient du être ré-exécutés n'ont pas été sélectionnés. Il est vrai que dans la plupart des cas, un bas pourcentage n'est pas souhaitable mais ce n'est pas nécessairement inapproprié dans tous les cas. Supposons un contexte où, due à des contraintes de temps ou de coût, nous serions restreint au niveau du nombre de cas de test à ré-exécuter et où la situation ne

permettrait pas d'exécuter l'ensemble des tests sélectionnés, un bas pourcentage de rappel peut indiquer une économie significative en termes de nombres de tests si la précision correspondante est raisonnablement élevée. C'est à dire que dans le cas où la majorité des tests sélectionnés sont susceptibles de démontrer un comportement différent sur P et P', même si les tests existants qui démontraient réellement des erreurs n'ont pas tous été exécutés, nous avons l'assurance que l'approche a quand même permit un gain significatif en effort de tests, car elle a permis d'éliminer des tests inutiles et de mettre le focus sur les cas qui étaient susceptibles de démontrer des erreurs.

Pour ce qui est de la génération de nouveaux cas de test destinés à couvrir les séquences d'exécution de P' non couvertes par TSelect, nous nous intéressons à déterminer l'ensemble Tnew des nouveaux cas de test. L'ensemble de tests final T', qui sera utilisé pour tester P', est donc représenté par l'expression : T' = Tselect (cas existants sélectionnés par l'approche) + Tnew (nouveaux cas générés par l'approche). Soit Texclu l'ensemble des tests de l'ensemble T qui ont été exclus pour former Tselect, l'ensemble T' peut aussi être représenté de la façon suivante : T' = (T - Texclu) + Tnew.

Afin de démontrer l'efficacité de notre approche en termes de génération de nouveaux cas, nous évaluerons l'ensemble Tnew à partir d'une métrique de **couverture.** Nous avons défini cette métrique, qui s'inspire des travaux de Black & al. [Ammann 02] qui eux–mêmes s'inspiraient des travaux de Wu et al. [Wu 88], en adaptant celle-ci à notre contexte. L'idée de base consiste à mesurer la couverture d'un ensemble de tests en termes de chemins d'exécution non couverts par les tests existants.

Soit N le nombre de séquences d'exécution impactées du programme P' non couvertes par Tselect, soit K le nombre séquences d'exécution impactées du programme P' non couvertes par Tselect qui sont couverts par Tnew. Le pourcentage de couverture de Tnew sur un ensemble de modifications est obtenu à partir de l'expression : C = 100*(K/N). Soit un programme P' pour lequel 12 chemins d'exécution impactés ne sont pas couverts par Tselect et un ensemble Tnew qui couvre 9 des chemins d'exécutions impactés de P' non couverts par Tselect, la couverture C de Tnew sur l'ensemble des

modifications est calculée par l'équation : 100 * (9/12) = 75%.

6.4 Études de cas

Afin d'appliquer l'approche dans le but de l'évaluer, nous utilisons 3 applications Java : NextGenV2, GestionAgence et GestionBibliotheque. L'application NextGenV2 est une extension de l'application NextGen récupérée du livre "UML et les Design Patterns" de Craig Larman [Larman 02]. L'application originale qui comptait 6 classes a été étendue pour les besoins de notre étude pour totaliser 15 classes au final. Nous y avons ajouté des fonctionnalités de gestion des comptes clients, des fournisseurs et des employés. Nous avons aussi ajouté des fonctionnalités de facturation en permettant de supporter les locations ainsi que les paiements par débit et crédit. L'extension de NextGen a engendré l'ajout de 73 méthodes au total.

GestionAgence est une application de gestion récupérée à partir du site javafr.com [java fr]. Elle permet de gérer, à l'aide de la console, les différentes activités d'une agence de voyage. Elle offre, entre autres, des fonctionnalités de gestion des forfaits, des vols, du personnel, des passagers, etc.

L'application GestionBibliotheque est aussi une application de gestion récupérée à partir du site javafr.com [java fr]. Elle a été conçue pour gérer les activités courantes d'une bibliothèque. Parmi les fonctionnalités offertes, nous retrouvons la gestion des livres, des documents, des abonnés, des prêts, etc. Le tableau 8 présente quelques statistiques à propos des applications.

Stats	NextGenV2	GestionBibliothequeV2	GestionAgenceV2
Nombre total de classes	16	10	20
Nombre de total méthodes	100	78	293
Nombre total de tests existants	13	58	104

Tableau 8. Statistiques sur les applications.

6.5 Démarche

Nous avons simulé des modifications au niveau des trois applications afin de produire une version modifiée de chacune d'elle. Une instance de chaque type de modification décrite ci-dessous a été appliquée à chacune des applications. Nous avons au total cinq modifications de type différent qui ont été effectuées sur chaque application. Voici les cinq types de modification:

- Modification d'une ligne existante
- Ajout d'une nouvelle ligne dans une méthode existante
- Retrait d'une ligne dans une méthode existante
- Ajout d'une nouvelle méthode
- Ajout d'une nouvelle classe

L'approche a ensuite été appliquée sur les versions originale et modifiée du code afin d'évaluer les résultats. Le tableau 9 fournit quelques statistiques à propos des modifications effectuées aux applications.

Stats	NextGenV2	GestionBibliothequeV2	GestionAgenceV2
Nombres de modifications effectuées à l'application	5	5	5

Tableau 9. Statistiques sur les modifications effectuées aux applications.

La section suivante décrira le protocole d'expérimentation qui a été conçu et appliqué lors de l'évaluation empirique. Le protocole a été appliqué pour chacune des applications.

6.6 Protocole

En supposant que la version originale de l'étude de cas est représentée par P et que sa version modifiée est représentée par P'. Pour chaque P' de P :

- Appliquer l'approche sur P et P';
- Recueillir les résultats;
- Analyser et interpréter les résultats en fonction des métriques retenues.

6.7 Étude de cas #1

La première évaluation a été effectuée sur l'application NextGenV2. Notre approche a été appliquée et les résultats ont été récoltés. Notre outil a détecté 6 méthodes modifiées dans P' impliquées dans 3 séquences d'exécution impactées. Nous avons sélectionné 2 cas de test pour couvrir les séquences impactées qui ont permis de couvrir 2 chemins d'exécution impactés. Une séquence de test restait non couverte par les tests existants alors un nouveau cas de test a été généré pour la couvrir. Le tableau 10 résume les résultats obtenus.

Nombre de méthodes modifiées	**Nombre de séquences d'exécution impactées**	**Nombre de tests sélectionnés**	**Nombre de séquences couvertes**	**Nombre de séquences non couvertes**	**Nombre de tests générés**
6	3	2	2	1	1

Tableau 10. Résultats de l'étude de cas 1.

Le pourcentage de réduction de Tselect par rapport à l'ensemble des modifications de P' est calculé par l'équation 1 - (Tselect / T) * 100 = 1 – (2/13) * 100% = 85%. La précision est calculée par l'équation 100 * (l / m) = 100 * (2 / 2) = 100% et le rappel est calculé par l'équation 100 * (l / n) = 100 * (2 / 2) = 100%. Rappelons que l

représente les tests sélectionnés susceptibles de démontrer un comportement différent sur P et P', m représente le nombre de tests sélectionnés par l'approche et que n représente les tests existants qui résultent en un comportement différent lorsque exécutés sur P et P'. Il à noter que n a été déterminé par analyse du code. La couverture est calculée à partir de l'équation 100 * (K / N) = 100 * (1 / 1) = 100%. Rappelons que N représente le nombre de séquences d'exécution impactées du programme P' non couvertes par les tests existants et que K représente le nombre de séquences d'exécution figurant dans N qui sont couvertes par les nouveaux tests.

Le pourcentage de réduction de 85% obtenu précédemment démontre le gain en termes d'effort et de coût contrairement à l'exécution systématique de tous les tests existants. La précision indique que parmi les tests sélectionnés par notre approche, 100% étaient susceptibles de démontrer un comportement différent sur P et P'. Aucun test n'a donc été exécuté pour rien. Le rappel de 100% indique que tous les tests existant révélant réellement un comportement différent sur P et P' ont tous été sélectionnées. Nous avons dans ce cas l'assurance qu'aucun test utile n'a été éliminé. La couverture indique que 100% des chemins d'exécution non couverts par les tests existants sont maintenant couverts par les nouveau cas de test générés. Ce résultat nous assure que tous les chemins d'exécution de P' sont maintenant couverts par la nouvelle batterie de tests composée des tests existants sélectionnés et des nouveaux tests générés.

6.8 Étude de cas #2

La deuxième étude de cas a porté sur l'application GestionBibliothèqueV2. Notre approche a permis de détecter 5 méthodes modifiées dans P' impliquées dans 9 séquences d'exécution impactées. Nous avons sélectionné 10 cas de test pour couvrir les séquences impactées qui ont permis de couvrir 8 chemins d'exécution impactés. Une séquence de tests restait non couverte par les tests existants alors un nouveau cas de test a été généré pour la couvrir. Le tableau 11 résume les résultats obtenus.

Nombre de méthodes modifiées	Nombre de séquences d'exécution impactées	Nombre de tests sélectionnés	Nombre de séquences couvertes	Nombre de séquences non couvertes	Nombre de tests générés
5	9	10	8	1	1

Tableau 11. Résultats de l'étude de cas #2.

Le pourcentage de réduction de Tselect par rapport à l'ensemble des modifications de P' est calculé par l'équation 1 - (Tselect / T) * 100 = 1 - (11/58) * 100 = 81%. La précision est calculée par l'équation 100 * (l / m) = 100 * (5 / 10) = 50% et le rappel est calculé par l'équation 100 * (l / n) = 100 * (5 / 5) = 100%. La couverture est calculée à partir de l'équation 100 * (K / N) = 100 * (1 / 1) = 100%.

Le pourcentage de réduction de 81% obtenu précédemment démontre le gain en termes d'efforts et de coût contrairement à l'exécution systématique de tous les tests existants. La précision indique que parmi les tests sélectionnés par notre approche, 50% étaient susceptibles de démontrer un comportement différent sur P et P'. La moitié des tests ont réellement été utiles alors que l'autre moitié ne l'est pas. Le rappel de 100% indique que tous les tests existants révélant réellement un comportement différent sur P et P' ont tous été sélectionnés. Nous avons dans ce cas l'assurance qu'aucun test utile n'a été éliminé. La couverture indique que 100% des chemins d'exécution non couverts par les tests existant sont maintenant couverts par les nouveaux cas de test générés. Ce résultat nous assure que tous les chemins d'exécution de P' sont maintenant couverts par la nouvelle batterie de tests composée des tests existants sélectionnés et des nouveaux tests générés.

6.9 Étude de cas #3

La troisième étude de cas a porté sur l'application GestionAgenceV2. Notre approche a été appliquée et les résultats ont été récoltés. Notre outil a détecté 8 méthodes modifiées dans P' impliquées dans 2 séquences d'exécution impactées. Nous avons sélectionné 2 cas de test pour couvrir les séquences impactées qui ont permis de couvrir un chemin d'exécution impacté. Une séquence de test restait non couverte par les tests existants alors un nouveau cas de test a été généré pour la couvrir. Le tableau 12 résume les résultats obtenus.

Nombre de méthodes modifiées	**Nombre de séquences d'exécution impactées**	**Nombre de tests sélectionnés**	**Nombre de séquences couvertes**	**Nombre de séquences non couvertes**	**Nombre de tests générés**
8	2	2	1	1	1

Tableau 12. Résultats de l'étude de cas #3.

Le pourcentage de réduction de Tselect par rapport à l'ensemble des modifications de P' est calculé par l'équation 1 - (Tselect / T) * 100 = 1 - (2/ 104) = 98%. La précision est calculée par l'équation 100 * (1 / m) = 100 * (2 / 2) = 100% et le rappel est calculé par l'équation 100 * (1 / n) = 100 * (2 / 2) = 100%. La couverture est calculée à partir de l'équation 100 * (K / N) = 100 * (1 / 1) = 100%.

Le pourcentage de réduction de 98% obtenu précédemment démontre le gain en termes d'efforts et de coût contrairement à l'exécution systématique de tous les tests existant. La précision indique que parmi les tests sélectionnés par notre approche, 100% étaient susceptibles de démontrer un comportement différent sur P et P'. Aucun test n'a donc été exécuté pour rien. Le rappel de 100% indique que tous les tests existants révélant réellement un comportement différent sur P et P' ont tous été sélectionnés. Nous avons dans ce cas l'assurance qu'aucun test utile n'a été éliminé. La couverture indique

que 100% des chemins d'exécution non couverts par les tests existants sont maintenant couverts par les nouveaux cas de test générés. Ce résultat nous assure que tous les chemins d'exécution de P' sont maintenant couverts par la nouvelle batterie de tests composée des tests existants sélectionnés et des nouveaux tests générés.

6.10 Conclusion

Le tableau 13 présente un récapitulatif des résultats obtenus lors des 3 études de cas effectuées précédemment. Un bref résumé de l'interprétation des résultats est ensuite présenté.

	Cas #1	**Cas #2**	**Cas #3**
Réduction	85%	81%	98%
Précision	100%	50%	100%
Rappel	100%	100%	100%
Couverture	100%	100%	100%

Tableau 13. Résultat des études de cas.

En résumé, l'évaluation empirique a permis de démontrer que l'application de notre approche sur des cas réels a permis une économie en termes d'effort sans perdre en précision pour autant, car dans deux cas sur trois la précision a été de 100% et dans l'autre cas la précision a été de 50%. Cette mesure nous indique que la plupart des cas de tests sélectionnés avaient des chances de révéler des erreurs. De plus, dans les 3 cas notre approche a permis de sélectionner 100% des tests qui révélaient réellement des erreurs. En aucun cas des tests utiles n'ont été écartés.

Les études de cas ont aussi permis de constater que notre approche permet de couvrir l'ensemble des chemins d'exécution non couverts par les tests existants. Cet aspect confirme que tous les chemins d'exécution impactés de P' sont couverts par l'ensemble de tests final composé de l'union des tests existants sélectionnés et des nouveaux tests générés.

CONCLUSION

Dans ce travail, nous avons présenté les principales approches de tests de régression proposées dans la littérature. Le fonctionnement de chacune des approches a été étudié afin de souligner leurs points forts et leurs points faibles. Cette démarche a permis par la suite de situer notre approche par rapport à la littérature.

Nous avons ensuite proposé une approche de tests de régression, conservatrice (safe), pour les systèmes orientés objet écrits en langage Java. Notre approche permet, dans un premier temps, la sélection de tests de régression de façon conservatrice et permet, dans un deuxième temps, la génération de nouveaux cas de tests couvrant les éléments non couverts par les tests existants. Elle ne requiert aucune instrumentation du code ni spécification.

Chaque étape de notre approche a été expliquée de façon systématique sur des modèles théoriques et des exemples. Les moyens techniques permettant d'appliquer la théorie liée à la démarche adoptée ont aussi été présentés.

Nous avons ensuite évalué notre approche en la comparant à des approches similaires proposées dans la littérature. Nous avons aussi comparé le fonctionnement des approches à travers quatre exemples théoriques. En plus de souligner la capacité d'effectuer une sélection précise des cas de tests existants, les exemples ont permis de démontrer l'efficacité de notre approche à couvrir les chemins d'exécution introduits par de nouvelles fonctionnalités ou encore à compléter la suite de tests originale.

L'implémentation de l'outil supportant notre approche a été présentée. Les grandes lignes de chaque module ont été abordées. Le fonctionnement de l'application a aussi été expliqué à travers des prototypes d'écran.

Finalement, une évaluation empirique a été effectuée sur trois études de cas. L'implémentation de l'approche a été appliquée sur les versions successives de 3

applications Java. Les résultats ont ensuite été analysés et interprétés à partir de métriques qui ont été préalablement sélectionnées. Les résultats obtenus ont permis de mettre encore plus en évidence les avantages que procure notre approche en particulier en termes d'effort et de coût.

Des travaux futurs pourraient s'intéresser à étendre notre approche afin de supporter la couverture des modifications effectuées à des niveaux de granularité plus fins. Actuellement, notre approche supporte uniquement la couverture des modifications effectuées au niveau des méthodes.

RÉFÉRENCES BIBLIOGRAPHIQUES

[Ammann 02] P. Ammann, Paul E. Black and W. Ding, Model Checkers in Software Testing, *National Institute of Standards and Technology*, 2002. NIST-IR 6777.

[Ball 98] Ball, T. On the Limit of Control Flow Analysis for Regression Test Selection. *ISSTA 98,* 1998, Clearwater Beach, FL. pp. 134-142.

[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.

[Chen 94] Chen,Y. F., Rosenblum,D. S.0 and K. P Vo. TestTube: A System for Selective Regression Testing. Proc. 16 th International Conference Software Engineering, Sorrento, Italy, May 1994, pp. 211-220.

[Chen 97] H. Pei, L. Xiaolin, D.C. Kung, H. Chih-Tung, L. Liang, Y. Toyoshima, C. Chen, A technique for the selective revalidation of OO software, Journal of Software Maintenance. Research and Practice 9 (4) (1997), 217–233.

[Chen 05] Cheng-hui Huang, Huo Yan Chen. A semi-automatic generator for unit testing code files based on Junit. Systems, Man and Cybernetics, 2005 IEEE International Conference.

[Cots 89] Cots, B.A. and R.G. Sargent. 1989. Automatic lookahead computation for conservative distributed simulation. CASE Center Technical Report 8916, CASE Center, Syracuse University, December 1989.

[Cots 90a] Cots, B.A. and R.G. Sargent. 1990a. A framework for automatic lookahead computation in conservative Distributed Simulation, in Distributed Simulation, D. Nicol, editor, The Society for Computer Simulation, 1990, pp. 56-59.

[Cots 90b] Cots, B.A. and R.G. Sargent. 1990b. Simulation algorithms for control flow graphs. CASE Center Technical Report 9023, CASE Center, Syracuse University, November 1990.

[Cots 90c] Cots, B.A. and R.G. Sargent. 1990c. Control Flow Graphs A method of model representation for parallel discrete event simulation. CASE Center Technical Report 9026, CASE Center, Syracuse University, December 1990.

[Cots 90d] Cota, B.A. and R.G. Sargent, 1990d. Simultaneous Events and Distributed Simulation, in Proceedings of 1990 Winter Simulation Conference, New Orleans, LA, December 1990, pp. 436-440.

[Harrold 94] Rothermel, G. and M. J. Harrold. Selecting Regression Tests for Object-Oriented Software. *Int'l Conf. on Software Maintenance,* 1994, pp. 14-25.

[Harrold 96] Gupta, R., Harrold, M.J., and M. L. Sofia. Program Slicing Based Regression Testing Techniques. *Journal of Software Test-ing, Verification and Reliability,* vol. 6, no.2, June 1996, pp. 83-112.

[Harrold 97] Rothermel, G. and M. J. Harrold. A Safe, Efficient Regression Test Selection Technique. ACM Transactions on Software Engineering and Methodology. Vol. 6, No. 2, April 1997, pp. 173-210.

[Harrold 00] G. Rothermel, M.J. Harrold, J. Dedhia, Regression test selection for C++ software, Journal of Software Testing Verification and Reliability 10 (2) (2000), 77–109.

[Harrold 01] M. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression Test Selection for Java Software. Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp.312-326, October 2001.

[java fr] http://www.javafr.com

[Koju 03] T. Koju, S. Takada, N. Doi, Regression test selection based on intermediate code for virtual machines, in: Conference on Software Maintenance, Institute of Electrical and Electronics Engineers Inc., 2003, pp. 420–429.

[Korel 98] Korel, B. and A. M. Al-Yami. Automated Regression Test Generation. *ISSTA 98,* 1998, Clearwater Beach, FL. pp. 143-152.

[Larman 02] C. Larman, UML et les Design Pattern, Campus Presse, 2002.

[Li 99] Yuejian Li, Nancy J. Wahl, An Overview of Regression Testing, Software Engineering Notes vol 24 no 1, January 1999.

[Li 06] J. Zhao, T. Xie, N. Li, Towards Regression Test Selection for AspectJ Programs, In 2nd Workshop on Testing Aspect-Oriented Programs (WTAOP'06), Portland, Maine, 2006.

[Onoma 98] Onoma, A.K., Tsai, W., Poonawala, M.H., and H. Suganuma. Regression Testing in an Industrial Environment. Communications of the ACM, Vol. 41, No. 5, May 1998, pp. 81-85.

[Rajlich 00] V. Rajlich and K. Bennett. A staged model for the software life cycle. *IEEE Computer*, 33(7):66 – 71, 2000.

[Runeson 03] P. Runeson, C. Andersson, and M. Host. Test processes in software product evolution - a qualitative survey on the state of practice. *Journal of Software Maintenance and Evolution*, 15(1):41–59, 2003.

[Skoglund 04] Mats Skoglund, Per Runeson, "A Case Study on Regression Test Suite Maintenance in System Evolution," icsm, pp.438-442, 20th IEEE International Conference on Software Maintenance (ICSM'04), 2004.

[Skoglund 05] M. Skoglund, P. Runeson, A case study of the class firewall regression test selection technique on a large scale distributed software system, in: 2005 International Symposium on Empirical Software Engineering (IEEE Cat. No. 05EX1213), IEEE, 2005.

[Skoglund 09] Emelie Engström, Per Runeson, Mats Skoglund, A systematic review on regression test selection techniques, Information and Software Technology, Elsevier, 2009.

[St-Yves 08] D. St-Yves, M. Badri, L. Badri, Dépendances et gestion des modifications dans les systèmes orientés objet: utilisation des graphes de contrôle, 2008.

[Vincent 09] Pierre-Luc Vincent, Mourad Badri, Linda Badri. Tests de régression dans les systèmes orientés objet: une approche basée sur les modèles, 2009.

[White 90] Leung, H.K.N. and L. White. A Study of Integration Testing and Software Regression at the Integration Level. *Proc. Conf. Software Maintenance,* San Diego, Nov. 1990, pp. 290-301.

[White 92] White, L.J. and H. K. N. Leung. A Firewall Concept for Both Control-Flow and Data-Flow in Regression Integration Testing. *Proc. Conf. Software Maintenance-92,* 1992, pp. 262-271.

[White 97] L. White, K. Abdullah, A firewall approach for the regression testing of object oriented software, Software Quality Week, 1997.

[White 03] L. White & al., Firewall Regression Testing of GUI Sequences and their Interractions, Proceedings of the International Conference on Software Maintenance Page 398, 2003.

[Wong 97] W. E. Wong, J. R. Horgan, S. London, H. Agrawal, Study of Effective Regression Testing in Practice, Proceedings of the Eighth International Symposium on Software Reliability Engineering (ISSRE'97), 1997.

[Wu 88] Wu, D.; Hennell, M.A.; Hedley, D.; Riddell, I.J, A practical method for software quality control via program mutation, Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on 19-21 July 1988, Page(s):159 - 170.

[Wu 03] Jang-Wu Jo and Byeong-Mo Chang. Constructing control flow graph that accounts for exception induced control flows for java, Proceedings ofthe 7th Korea-Russia International Symposium, KORUS 2003.