Étude bibliographique

par Franck Fleurey

14 avril 2006

Table des matières

1 Introduction

Tout au long du cycle de vie d'un logiciel, il est important de s'assurer de sa correction. Pour cela, on peut avoir recours à la preuve de programme ou au test. Depuis les débuts de l'informatique, beaucoup de travaux se sont intéressés à la preuve de programme mais ils n'ont aboutis que sur des méthodes applicables dans des domaines très particuliers. Dans tous les autres cas, le test reste le seul moyen de s'assurer de la validité d'une implantation.

A l'heure actuelle, le test représente une part importante du travail au cours de tout développement logiciel. Un grand nombre de travaux s'intéresse à l'activité de test afin d'en augmenter la qualité et d'en réduire le coût. La plupart des études menées jusqu'à maintenant, portent sur le test des parties de logiciel dont la fiabilité est primordiale. Il s'agit généralement de test d'algorithmes réalisant du calcul numérique. Dans ce cadre, des méthodes de test ne traitant que du code relativement court n'opérant que sur des variables numériques ont été introduites. Ces méthodes ont ensuite fait l'objet de diverses extensions pour s'adapter au test de programmes plus important ou manipulant des types de données plus complexes.

Cela étant, le test des logiciels généraux[1] est encore réalisé en grande partie à la main. D'une part parce que les méthodes proposées sont peu adaptées au test système. D'autre part parce que ces méthodes sont trop complexes pour s'intégrer facilement dans les outils de développement logiciel les plus utilisés.

Mon stage de DEA se déroulera dans l'équipe TRISKELL qui s'intéresse au développement de composants logiciels fiables. Dans ce cadre nous nous intéressons aux traces d'exécutions comme une technique permettant d'améliorer le processus de test de systèmes orienté-objets. Initialement, notre objectif était d'appliquer des méthodes de diagnostic issues du matériel afin d'assister le diagnostic logiciel. Nous nous sommes rapidement aperçus que de telles techniques avaient déjà été proposées dans [16, 2]. Nous avons donc décidé d'étudier ces méthodes et d'élargir notre travail à d'autres techniques utilisant les traces d'exécution pour faciliter le test. Nous avons choisi de nous intéresser à des techniques de génération de données de test et de minimisation de suites de test utilisant également les traces d'exécution.

Dans la section 2 nous commençons par faire un tour d'horizon rapide de la méthodologie de test. Dans la section 3 nous étudions des techniques de génération de test basées sur les traces d'exécution et utilisant les algorithmes génétiques. Dans la section 4 nous nous intéressons à la réduction de suites de tests et à une méthode dérivée : la prioritisation de suites de tests. Dans la section 5 nous étudions les méthodes d'assistance au diagnostic proposées dans [16, 2]. Enfin la section 6 conclut cette étude par une courte synthèse et la présentation des divers points qui pourront être abordés durant mon stage.

1. Logiciels "généraux", par opposition à logiciels "critiques".

2 Test logiciel

A l'heure actuelle le test est le moyen principal de validation d'un logiciel. La phase de test intervient généralement à la fin du processus de développement d'un logiciel et est réalisée par étapes successives [5, 4].

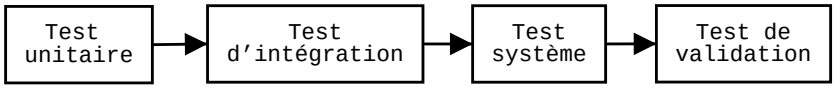

Figure 1. Les étapes du test

La figure 1 présente les quatre étapes principales de la phase de test d'un logiciel. Lors de ces quatre phases, les *objectifs du test* (cf section 2.1) et les méthodes employées sont différents mais à quelque étape que se soit, le test consiste toujours à exécuter le programme sous test avec un jeu d'entrée (les *cas de test*) et à comparer les résultats obtenus avec ceux attendus. La figure 2 illustre le processus de test.

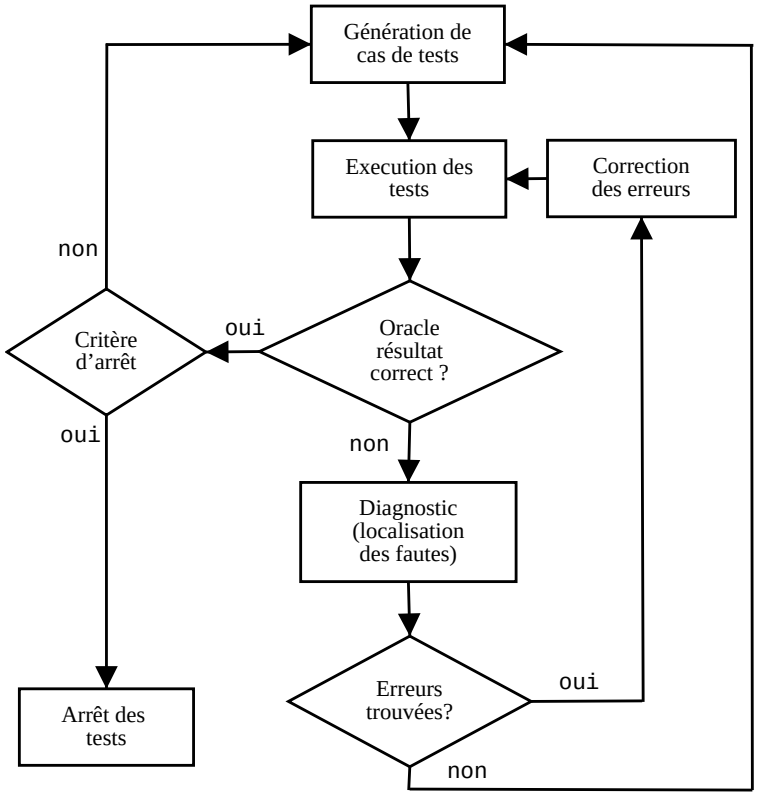

Figure 2. Méthodologie de test

L'activité de test commence toujours par l'élaboration de cas de test. Puis ces cas de test sont confrontés à l'implantation et l'on compare alors les résultats obtenus avec ceux attendus. Il faut donc avoir déterminé au préalable un *oracle*, c'est à dire les sortie attendues. Si cet oracle met en évidence un défaut de l'implantation, il faut localiser l'erreur (c'est le *diagnostic*) et la corriger. Dans le cas contraire, le jeu de test est réussi. En cas de non satisfaction d'un *critère d'arrêt*, il faut alors générer d'autres cas de tests pour tenter de découvrir de nouveaux défauts. L'activité de test ne prend fin que lorsque ce critère d'arrêt est satisfait.

On distingue deux grandes familles de méthodes pour le test : le *test structurel* (ou *boite blanche*) et le *test fonctionnel* (ou *boite noire*). Pour le test structurel, l'implantation du composant sous test est connue et l'objectif premier est de révéler des erreurs dans cette implantation. A contrario, pour le test fonctionnel, l'implantation du composant sous test n'est pas disponible, seules ses entrées et sorties sont accessibles. On s'appuie alors sur la spécification du système pour le tester et l'objectif est de valider l'implantation par rapport à sa spécification.

Nous nous intéressons, ici, tout d'abord aux étapes du test définies sur le figure 1 puis à des techniques pouvant être utilisées dans les phases clés de la méthodologie de test (figure 2).

2.1 Les différentes étapes du test

Comme nous l'avons évoqué précédemment le test d'un système se fait en quatre étapes (figure 1). Tout d'abord le *test unitaire* consiste à tester l'implantation de chaque module logiciel séparément. S'il s'agit d'un logiciel orienté objet cela consistera par exemple à tester chaque classe séparément pour s'assurer de leur bon fonctionnement. Il s'agit généralement de test structurel réalisés conjointement avec le développement du module sous test.

Vient ensuite le *test d'intégration* qui a pour objectif de tester le bon fonctionnement des différents modules logiciel entre eux. Chaque sous-système est testé séparément afin de vérifier le bon fonctionnement des interactions entre les modules unitaires. Une fois le système intégré, l'étape de *test système* permet de vérifier le bon fonctionnement des services rendus par le système. Ces deux dernières étapes sont en général réalisées par du test fonctionnel car la structure du programme sous test se complexifie à mesure que sa taille augmente.

La dernière phase de test est le *test de validation*, qui à pour but de tester la validité du système implanté par rapport à sa spécification initiale.

2.2 La génération de données de test

La génération de cas de test est une phase fondamentale à toute activité de test. A l'heure actuelle une grande partie des tests est encore écrite à la main. Il existe cependant un ensemble de techniques utilisable pour générer automatiquement des ensembles de cas de test (ou *suite de test*). Ces méthodes sont principalement utilisées pour tester des logiciels dit *critiques* c'est à dire dont la défaillance serait lourde de conséquence sur le plan matériel ou humain.

Les méthodes existantes peuvent se classifier en deux catégories. Tout d'abord, les méthodes de génération de test pour le test fonctionnel qui se basent sur un modèle formel de la spécification du composant sous test pour générer des cas de test permettant de vérifier des propriétés sur l'implantation. Et à l'inverse la génération de cas de test pour le test structurel qui se base sur l'analyse statique ou dynamique du programme sous test. Dans la suite nous étudions plus en détail des méthodes de génération de tests dynamiques utilisant les algorithmes génétiques.

2.3 L'oracle

L' oracle est est une fonction qui permet de décider si un cas de test est passé ou a échoué en faisant le lien entre l'implantation et la spécification du programme sous test.

Définition 1. *Soit P une implantation sous test d'une spécification F. Soit* $\mathrm{In}(P)$ *et* $\mathrm{Out}(P)$ *les domaines d'entrée et de sortie de P. L'oracle est une fonction* ok_F *de* $\mathrm{In}(P) \times \mathrm{Out}(P)$ *à valeur booléenne tel que :*

$$\forall x \in \mathrm{In}(P), \mathrm{ok}_F(x, P(x)) = \mathrm{vrai} \Longleftrightarrow P(x) = F(x)$$

Cet oracle peut être, dans la pratique, manuel ou automatique. La décision peut être prise par le testeur en examinant les entrées et sorties du programme mais ce travail peut devenir fastidieux si le testeur utilise de grandes suites de test ou si la spécification est complexe. Dans beaucoup de cas il est donc préférable d'automatiser l'oracle.

Si l'on dispose d'une spécification exécutable du programme (ou d'une autre version du programme) on pourra par exemple comparer ses sorties avec celles de l'implantation sous test pour décider du succès d'un cas de test. L'oracle peut également dans certain cas être embarqué dans le code sous forme d'assertions. Cette idée est implantée dans certain langages orienté-objet sous forme de pré/post-conditions pour les méthodes, d'invariants de classe et de variants et invariants de boucle [17, 6].

Dans certain cas particuliers des oracles automatiques propre au domaine peuvent être utilisés. Si l'on test par exemple un algorithme de compression sans perte, on pourra utiliser l'algorithme de décompression correspondant pour vérifier si l'intégrité est conservée par l'algorithme de compression.

Disposer d'un oracle efficace et automatique est primordial si l'on utilise des méthodes de génération de test nécessitant l'exécution d'un grand nombre de cas de test.

2.4 Le critère d'arrêt

Le test logiciel a pour but de vérifier la correction d'une implantation mais n'est en aucun cas équivalent à une preuve de programme. En effet, le seul moyen d'avoir cette équivalence est de tester le programme sur tout son domaine d'entrée *(test exhaustif)* ce qui est impossible dans le cas général étant donné que le domaine d'entrée d'un programme peut ne pas être fini. D'où l'utilité de définir des critères d'arrêt de test plus faibles permettant d'assurer une certaine confiance dans l'implantation sous test.

Ces critères peuvent être soit fonctionnels, soit structuraux. Les critères fonctionnels sont définis sur la spécification du programme sous test. Si un programme est spécifié par un graphe d'état fini, on pourra par exemple décider d'arrêter le test lorsque tous les états ou transitions de la spécification sont couverts. Les critères structuraux sont quant à eux définis à partir de l'implantation. On peut par exemple définir des critères de couverture de code : appeler au moins une fois chaque procédure, exécuter au moins une fois chaque instruction... On peut également utiliser des représentations plus fines du programme comme par exemple le *graphe de flot de contrôle* (cf section 3.2.1) ou le *graphe Def/Use* et définir des critères à partir de ces représentations (couvrir tous les sommets, tous les arc...).

D'autres approches sont également possibles comme l'analyse de mutation [7] par exemple. Le test par mutation consiste à créer une grande quantité de versions (ou *mutants*) du programme sous test en introduisant dans chacune une erreur élémentaire. L'hypothèse faite est que si une suite de test est capable de rejeter (ou *tuer*) tous les mutants alors cette suite de test est capable de détecter des erreurs plus complexes dans le programme original. Le critère d'arrêt s'exprime ici comme le pourcentage de mutants rejetés par le suite de test, appelé *score de mutation*.

2.5 Le diagnostic

Le diagnostic intervient lorsque des cas de tests ont mis en évidence une défaillance du système sous test. Le diagnostic a pour fonction de localiser l'erreur ayant causé cette défaillance afin de permettre au testeur de la corriger.

La plupart des développeurs utilisent pour cette phase un metteur au point (*debuger*) qui permet de tracer, instruction par instruction, l'exécution du cas de test provoquant la défaillance. Cependant des méthodes plus automatique existent. Ces méthodes emploient pour la plupart des techniques de découpage de programme (ou *program slicing*).

Nous étudions plus en détail cette phase de diagnostic à travers deux méthodes de localisation automatique de faute dans la section 5.

2.6 Conclusion

Nous avons, dans cette section, donné un rapide aperçu des problèmes posés lors du test de logiciel ainsi qu'un aperçu le plus général possible de la méthodologie employée.

Dans la suite nous étudions plus en détail des techniques automatiques pouvant être employées au cours de différentes phases de test.

3 Génération de test et algorithmes évolutionnistes

Les algorithmes génétiques sont généralement utilisés dans des problèmes d'optimisation, lorsque l'espace de recherche est important. Dans cette section, après un rappel rapide du principe des algorithmes génétiques, nous étudions quelques travaux visant à générer des cas de test en utilisant les algorithmes génétiques. Nous étudions tout d'abord la méthode proposée par R.P. Pargas et M.J. Harrold [18] pour le test unitaire, puis une méthode proposée par B. Baudry [3, 4] utilisant un algorithme dérivé des algorithmes génétiques et appliquée au test système.

3.1 Algorithmes génétiques

Les algorithmes génétiques sont une méthode d'optimisation qui imite les processus naturels d'évolutions tel que la reproduction et la mutation. Depuis leurs description par Holand [13] en 1975 ils ont été appliqués avec succès à un grand nombre de problèmes d'optimisation et

d'apprentissage. Ils sont particulièrement intéressants lorsque l'espace de recherche de solution à un problème est trop important pour des algorithmes classiques.

Le principe général des algorithmes génétiques est donné sur la figure 3. Dans un premier temps, une population initiale de solutions au problème considéré est constituée. En général, si le problème le permet, cette population de solutions est générée aléatoirement. Ces solutions, ou *individus*, doivent être représentés par une suite ordonnée de taille fixe d'éléments que nous appellerons *gènes*.

A partir de cette population, une fonction d'utilité (*fitness function*) permet d'évaluer l'efficacité de chaque individu par rapport au problème considéré. Sont alors sélectionnés, dans la population ordonnée ainsi obtenue, les meilleurs individus qui serviront de "parents" à la nouvelle génération de solution. La taille de la population initiale est un paramètre important de l'algorithme car elle fixe la taille des populations des générations suivantes.

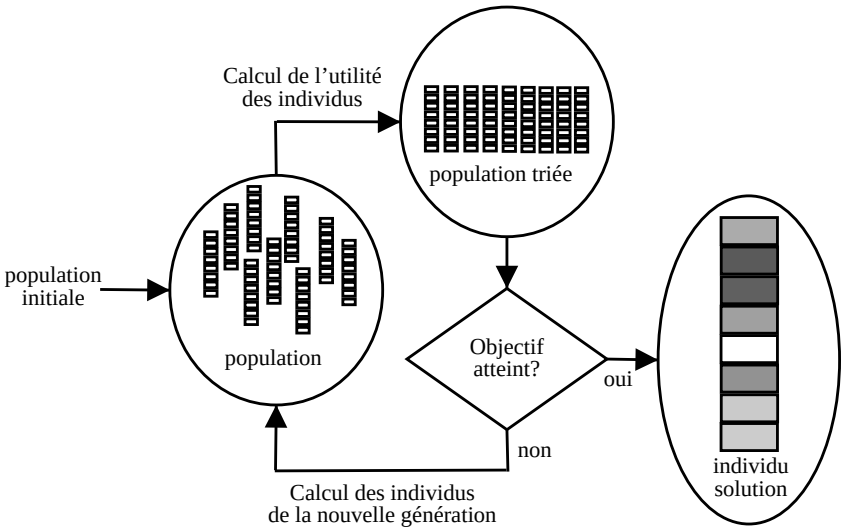

Figure 3. Algorithme génétique

L'hypothèse faite est que les meilleurs éléments d'une population sont des solutions composées d'éléments (ou *gènes*) adaptées au problème. On compose donc à partir de ces solutions des descendants, que l'on espère meilleurs, en utilisant le croisement et la mutation. L'opérateur de croisement produit deux descendants à partir de deux parents. Comme le montre la figure 4, on commence par déterminer aléatoirement un point de croisement dans un individu, puis on construit deux nouveaux individus en recombinant les gènes précédant la coupure d'un parent avec les gènes succédant la coupure de l'autre. L'opérateur de mutation ne produit qu'un descendant à partir d'un parent en substituant un des gènes du parent par un gène aléatoire.

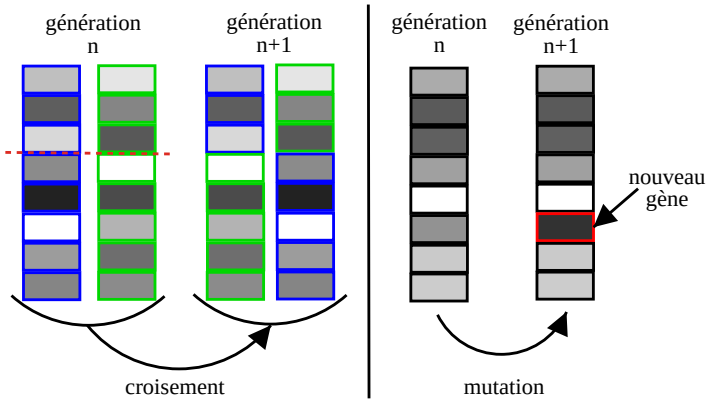

Figure 4. Opérateurs de croisement et mutation

Les parents sont sélectionnés en fonction de leurs utilités dans la population. Statistiquement, ce sont donc les meilleurs parents qui sont sélectionnés (ceux-ci ont d'ailleurs de grande chance d'être sélectionné plusieurs fois) mais les individus ayant une faible utilité ont tout de même une faible probabilité d'être sélectionnés. Les opérateurs de croisement et de mutation sont appliqués sur les parents sélectionnés dans un ratio, paramètre de l'algorithme, appelé *taux de croisement*. Dans la littérature, le *taux de mutation*, parfois utilisé, est défini comme le complémentaire du taux de croisement. La nouvelle population est constituée lorsque qu'elle compte autant d'individus que la génération précédente.

L'opérateur de croisement a un rôle d'optimisation car il permet de créer des individus concentrant les bons gènes alors que l'opérateur de mutation a un rôle d'évolution car il introduit de nouveau gènes dans la population. L'efficacité de l'algorithme est directement liée au taux de mutation et de croisement qui doivent assurer un compromis entre optimisation et évolution. Typiquement on utilise des taux de croisement de l'ordre de 90% à 98%.

Les opérateurs de croisement et mutation étant communs à tout algorithme génétique, ils sont aujourd'hui bien connus. Cependant pour utiliser un algorithme génétique il faut trouver une représentation des solutions du problème à résoudre sous forme de gène et définir une fonction d'utilité. Ces derniers problèmes étant dépendant du domaine d'application il faudra les résoudre à chaque application de l'algorithme.

L'arrêt de l'algorithme peut être de plusieurs nature en fonction du problème : une certaine utilité atteinte, un certain nombre de génération, un certain temps...

Dans la suite nous étudions une méthode de génération de test unitaire s'appuyant sur les algorithmes génétiques tel qu'ils sont décrits ici, puis nous nous intéressons à une méthode de génération de test système s'inspirant des algorithmes génétiques.

3.2 Génération de test et algorithmes génétiques

Plusieurs travaux ont étudié les algorithmes génétiques pour la génération automatique de cas de test. Dans ces travaux, les individus sont des cas de test qui sont optimisés pour atteindre un objectif de test. Les deux difficultés, comme annoncé précédemment, sont d'une part de trouver une représentation sous forme de gènes des cas de test et d'autre part de définir la fonction d'utilité des cas de test par rapport à l'objectif de test. Pour ce qui est de la représentation des données de test, la plupart de ces travaux se placent dans des cas simples ou une donnée de test correspond soit à un entier qui est alors représenté par une chaîne de bits, soit à une liste de valeur numérique. Les fonctions d'évaluation utilisées se basent généralement sur des traces d'exécution des cas de test à évaluer. Ces traces d'exécution sont obtenues en instrumentant le programme sous test afin de collecter, lors de l'exécution d'un cas de test à évaluer, les informations aux points de contrôle du programme.

Dans cette section nous étudions une approche proposée par P. Pargas [18] pour générer des cas de test satisfaisant des objectifs de test simples pour des petit programmes. Nous commençons par quelques pré-requis sur les graphes de contrôle de programme qui seront utiles pour définir les objectifs de test et fonction d'utilité décrits dans [18].

3.2.1 Graphe de flot de contrôle et de dépendance de contrôle

Les graphes de flot de contrôle (GFC) et de dépendance de contrôle (GDC) sont des représentations abstraites de programme. Initialement, ces représentations ont été introduites dans le cadre de l'optimisation de code lors de la compilation. Cependant un grand nombre de travaux utilisent ces représentations dans le domaine du test.

La figure 5 donne le GFC et le GDC d'un petit programme. Dans ces deux graphes, les noeuds représentent les instructions du programme. Dans le GFC les transitions représentent les les enchaînements possibles des instructions. Pour faciliter les traitements le graphe de flot de contrôle est souvent augmenté d'un unique point d'entrée et d'un unique point de sortie.

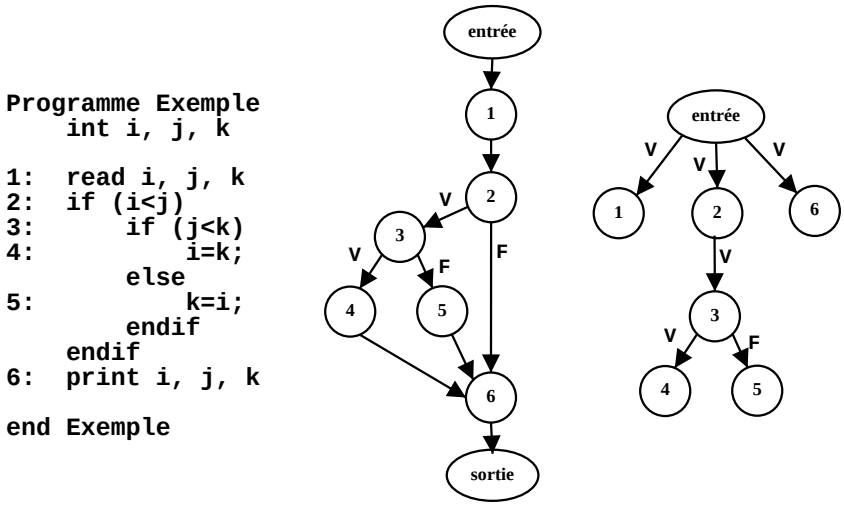

Figure 5. Un exemple de programme et ses graphes de flot de contrôle (au milieu) et de dépendance de contrôle (a droite)

Définition 2. *Soit V et W deux noeuds d'un graphe de contrôle G. V post-domine W si et seulement si pour tout chemin orienté de G entre W et* Sortie *de G (W et* Sortie *exclus) V appartient à ce chemin.*

Définition 3. *Soit X et Y deux noeuds d'un graphe de contrôle G. Y est dit contrôle-dépandant de X si et seulement si :*

1. *Y ne post-domine pas X*
2. *Pour tous chemin orienté γ de G entre X et Y (X et Y exclus), Y post-domine tous les sommets de γ*

Le graphe de dépendance de contrôle se base quant à lui sur la notion de post-domination entre les instructions (définition 2) et les arcs représentent les dépendances de contrôle entre les instructions (définition 3).

Un chemin a-cyclique du GDC allant de l'entrée du graphe à un noeud N contient un ensemble de prédicats qui doivent être satisfaits pour qu'une donnée puisse atteindre le noeud N. Un tel chemin sera appelé un *chemin de prédicat de dépendance de contrôle* (*control-depandance predicate path*).

L'algorithme proposé dans [18] utilise d'un part le graphe de flot de contrôle pour définir des objectifs de test à atteindre et d'autre part le graphe de dépendance de contrôle et les chemins de prédicat de dépendance de contrôle pour évaluer la qualité d'un cas de test vis-à-vis d'un objectif de test.

3.2.2 Génération automatique de cas de tests

L'approche proposée par Pargas et Harrold [18] consiste à utiliser les algorithmes génétiques pour générer une suite de test satisfaisant des critère de test. Les individus des algorithmes génétiques correspondent à des cas de test, la population est donc un ensemble de cas de test. Les critères de test sont définis par des noeud marqués dans le GFC du programme sous test.

Le premier problème, comme nous l'avons vu lorsque nous avons évoqué les algorithmes génétique, est de représenter les individus sous forme d'une chaîne de gène. Dans [18], les auteurs, pour supprimer ce problème se placent dans le cadre de programme simple dont les entrées sont constituées d'un ou plusieurs entiers. Ainsi, les entrées peuvent être vues comme des chaînes de bits représentant les entiers en base 2.

Le second problème est celui de la fonction d'utilité *(fitness function)* permettant d'évaluer la qualité d'un individu vis-à-vis d'un objectif, c'est à dire dans notre contexte la qualité d'un cas de test vis-à-vis de la couverture d'un bloc du programme sous test. L'idée proposée par [18] se base sur la notion de *chemin de prédicat dépendance de contrôle* d'un bloc. Pour couvrir un bloc B, une donnée de test doit rendre vrai tous les prédicats d'un *chemin de prédicat dépen-*

dance de contrôle de B. A partir de ces constatations, la fonction d'évaluation d'un cas de test vis-à-vis de la couverture d'un bloc B est définie comme le nombre de prédicats rendu vrai par l'exécution du cas de test dans les *chemins de prédicat dépendance de contrôle* de B.

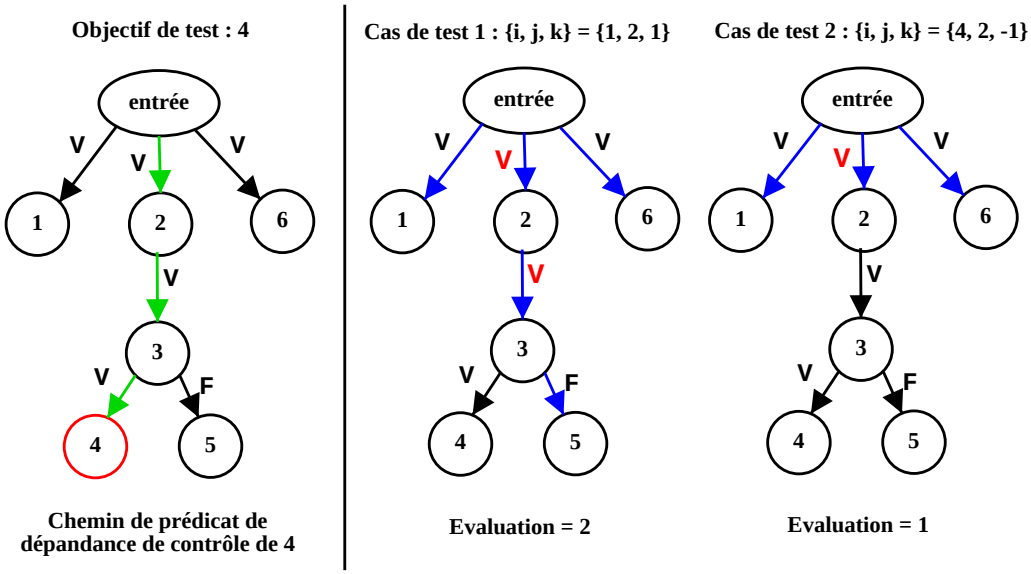

Figure 6. Exemple d'objectif de test (à gauche) et du calcul de l'évaluation de deux cas de test (à droite).

La figure 6 montre, sur l'exemple de programme de la figure 5, l'évaluation de la fonction d'évaluation pour deux cas de test vis-à-vis de l'objectif de test "couvrir le bloc 4".

Remarque 4. La fonction d'évaluation définie ici est le principal apport de cet algorithme par rapport au travaux antérieurs de Jones et al. [14] qui se contentait d'une fonction d'évaluation sur le GFC rendu a-cyclique en limitant le nombre d'itérations dans les boucles.

A partir de cette fonction d'évaluation et des opérateurs de croisement et de mutation classique des algorithmes génétiques, les auteurs de [18] appliquent les algorithmes génétiques sur chaque objectif de test à partir d'une population aléatoire de cas de test.

L'approche est validée expérimentalement grâce à l'outil TGen, implanté par les auteurs, sur des petit programmes C (entre 21 et 82 lignes de code). Les résultats sont comparés en terme de couverture de code avec des test pris aléatoirement.

3.3 Génération de test et algorithmes bactériologiques

Nous présentons ici une autre application des algorithmes génétiques à la génération de données de test. Dans [3, 4], les auteurs s'intéressent à la génération automatique de suite de tests atteignant un bon score de mutation (cf section 2.4). Dans les travaux évoqués précédemment l'algorithme génétique était appliqué pour générer un cas de test atteignant un objectif de test. Ici l'algorithme génétique est utilisé pour générer une suite de test, les individus sont donc des suites de tests et les gènes les cas de test de ces suites de test.

Nous avons évoqué précédemment le problème de la représentation des individus dans l'application des algorithmes génétiques. Dans la section précédente une représentation simple de cas de test constitués d'entiers était utilisée, mais cette représentation n'est utilisable que si les entrées du programme sous test sont très simples. Dans [3, 4], une représentation arborescente des cas de test est proposée pour des programmes dont les entrées sont décrites par une grammaire. Dans ce cadre particulier, mais cependant applicable à un grand nombre de programme (traitement de fichier XML, interpréteurs, compilateur...) la représentation proposée est l'arbre syntaxique abstrait.

3.3.1 Limites des algorithmes génétiques

Le premier algorithme envisagé par les auteurs est un algorithme génétique classique. Les individus sont donc des suites de tests composées d'un nombre constant de cas de tests,correspondant aux gènes des algorithmes génétiques. La fonction d'utilité proposée est le score de mutation de la suite de tests. L'étude de cas menée avec ce modèle sur un pretty-printer C# donne des résultats peu concluants pour plusieurs raisons :

- Les suites de tests étant de taille fixe, rien n'assure que la taille choisie soit suffisante pour atteindre l'objectif. Cependant, choisir une taille trop importante rend plus complexe le calcul de la l'utilité des individus.
- Les cas de test n'étant pas ordonnés dans les suites de test, l'opérateur de croisement usuel perd de son efficacité.

Ces constatations ont poussé les auteurs à définir un autre algorithme évolutionniste prenant en compte ces problèmes : l'algorithme bactériologique.

3.3.2 Les algorithmes bactériologiques

Le modèle bactériologique tiens son nom d'une analogie avec le processus naturel d'adaptation des bactéries [19]. L'objectif de l'algorithme contrairement aux algorithmes génétiques n'est plus de construire un individu solution, mais une population solution à un problème. Une première condition d'application de cet algorithme est que les solutions au problème considéré puissent s'exprimer sous la forme d'ensemble de sous-solutions : les *bactéries*. La grande différence entre ces bactéries et les gènes des algorithmes génétiques est qu'elles ne sont ni ordonnées, ni en nombre fixé.

La figure 7 montre le principe de l'algorithme. L'algorithme dispose d'une mémoire (M) initialement vide. Au cours de l'exécution de l'algorithme, les bactéries dont la fonction d'utilité dépasse un certain seuil sont sauvegardées dans cette me-moire. L'algorithme construit ainsi au cours de son exécution une population de bactéries solution dans sa mémoire. L'utilité d'une bactérie est déterminé en utilisant une fonction d'utilité F, propre au problème considéré, et permettant d'évaluer la qualité d'un ensemble de bactéries vis-à-vis du problème à résoudre. A chaque itération, l'utilité des bactéries est calculée par rapport aux bactéries déjà mémorisées. Ainsi, si b est une bactérie, l'utilité $f(b)$ de b est définie par $f(b) = F(M \cup b) - F(M)$.

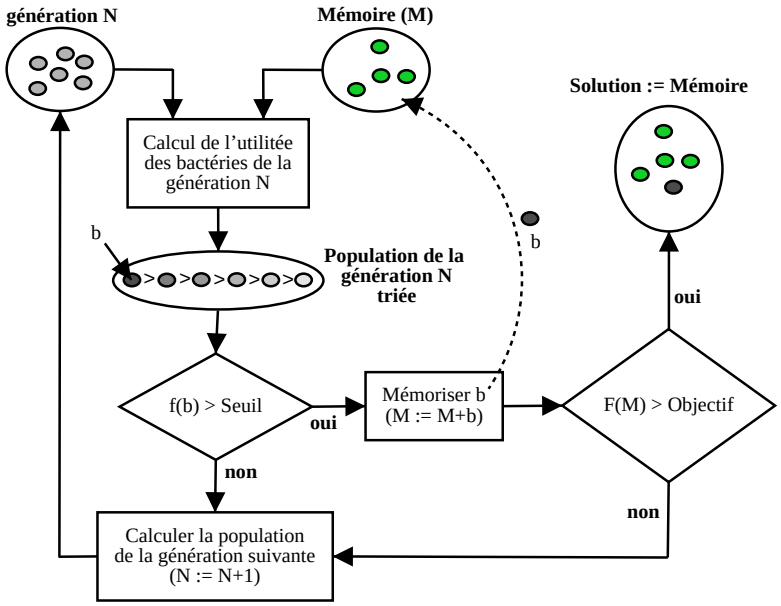

Figure 7. Principe de l'algorithme bactériologique

A chaque itération une nouvelle population de bactéries est calculée. La nouvelle génération est constituée de mutants des meilleurs bactéries de la génération courante. Contrairement aux algorithmes génétiques, aucun opérateur de croisement n'est utilisé. La figure 8 présente le texte de l'algorithme.

```
Entrées :          P_init = {b_0, b_1, ...b_m} (Population initiale)
                   S (Seuil de mémorisation des bactéries)
                   O (Objectif à ateindre)
                   F (Fonction d'utilité d'une population)

Déclarations :     P_n = {b_0^n, b_1^n, ...b_m^n} (Population à l'itération n)
                   M (ensemble de bactéries mémorisées)

Initialisation :   P_0 ← P_init
                   M ← ∅

Progression :      ∀x ∈ P_n, f(x) = F(M ∪ x) − F(M)                    (1)
(itération n)      b ← x ∈ P_n / f(x) = max(f(y), y ∈ P_n)             (2)
                   si f(b) > S alors
                        M ← M ∪ x                                      (3)
                        si F(M) ⩾ O alors SORTIR fsi                   (4)
                   fsi
                   Générer P_{n+1} à partir de P_n                     (5)
                   n ← n + 1                                           (6)

Sortie :           Retourner M
```

Figure 8. L'algorithme bactériologique

Dans [3, 4], après avoir défini cet algorithme, les auteurs appliquent cette méthode à la génération de données de test. Les bactéries sont alors des cas de test et la fonction d'utilité reste le score de mutation d'une suite de test. Les auteurs se placent dans le cadre de programmes dont les entrées peuvent être représentées par une grammaire. Afin de définir l'opérateur de mutation sur les cas de test, ceux-ci sont alors représentés par leur arbre syntaxique. Chaque noeud de l'arbre est vu comme un gène du cas de test. L'opérateur de mutation des bactéries consiste alors à remplacer un noeud de l'arbre syntaxique par un autre généré aléatoirement.

L'étude de cas présentée est réalisée sur un pretty-printer de code C#, et montre de bien meilleurs résultats que l'algorithme génétique. Les auteurs discutent également des réglages de l'algorithme (Seuil de mémorisation, taille des cas de test...).

4 Réduction et prioritisation de suite de test

Dans cette section nous introduisons les notions de réduction et de prioritisation de suite de tests. Ces deux techniques visent à réduire la quantité de cas de test exécutés lors de la maintenance d'un logiciel afin de réduire le coût de cette maintenance.

La *réduction de suite de test* [12] (ou *minimisation de suite de test*) a pour but de sélectionner un sous ensemble de cas de test assurant un pouvoir de détection d'erreur équivalent à la suite de test initiale. La réduction de suite de tests peut être utilisée pour sélectionner des cas de test après en avoir générer un grand nombre, mais aussi et surtout pour limiter le nombre de cas de test à exécuter lors de la maintenance du logiciel sous test.

La *prioritisation de cas de test* [10, 9, 20, 21] consiste à ordonner les cas de test d'une suite de tests afin d'augmenter le *taux de détection de faute* (ou *rate of fault detection*) de cette suite de tests. Le taux de détection de faute d'une suite de tests étant définie comme sa capacité à détecter le plus tôt possible des erreurs dans le composant sous test. Cette technique est particulièrement utile dans le cadre du *test de non-regréssion*. Le test de non-régression intervient lors du test d'un composant ayant été modifié et pour lequel on possédait une suite de tests. Il s'agit alors de ré-exécuter l'ancienne suite de test pour vérifier que les fonctionnalités de la version précédente du composant sont toujours valides.

Les principaux travaux étudiant ces techniques se basent sur des critères de couverture structuraux obtenus grâce à des traces d'exécutions.

4.1 Limites de la réduction de suite de test

M.J. Harrold et al. [12] formalise le problème de la réduction de suite de test ainsi :

Définition 5. *Problème de la réduction/minimisation de suite de test:*
Étant donné une suite de test TS et un ensemble d'objectif de test $r_0, r_1, ..., r_n$ couverts par TS, trouver un ensemble minimale de cas de test, inclus dans TS et couvrant tous les objectifs de test.

Dans la définition 5, les objectifs de test r_i peuvent représenter n'importe quel critère de test : couverture d'instruction, de fonction, des arcs du graphe de contrôle...

L'objectif de la méthode est de trouver un ensemble de cas de test, de cardinalité minimale, satisfaisant le critère de test (c'est à dire un ensemble d'objectifs de test), ce qui dans le cas général est un problème N-P complet. Les méthodes proposés sont donc des heuristiques permettant de réduire une suite de test sans toutefois garantir la minimalité la suite de tests obtenue, mais en conservant le critère de couverture retenu. Dans [12], les auteurs proposent un algorithme permettant de réduire des suites de tests en éliminant les cas de test redondants. L'algorithme est ensuite validé expérimentalement sur un ensemble de petits programmes C comportant entre 19 et 86 lignes de code.

Dans des travaux plus récents, le principe même de la réduction de suite de tests est remis en cause. En effet, comme le montre [21], le fait que la suite de tests réduite satisfasse un même critère de couverture ne garanti en aucun cas que son pouvoir de détection d'erreur est équivalent à celui de la suite de test initial. Dans cet article les auteurs appliquent l'algorithme proposé dans [12] à un ensemble de programme C comportant de 138 à 516 lignes de code et montrent en utilisant un grand nombre de suite de tests et leurs versions réduites que la minimisation peut conduire à réduire le pouvoir de détection d'erreur.

Une autre limitation de certaines méthodes de minimisation vient du fait qu'elles s'appuient sur des critères de couverture de code. Comme l'a montré S.G. Elbaum [11], les critères de couverture de code sont très sensibles aux modifications du code. En effet dans [11], deux études de cas (sur un programme jouet et différentes versions du *bash* unix) sont présentées et montrent que des modifications mineures à un programme induisent des modifications significatives et imprévisibles à la couverture de son code par une suite de tests. Cette constatation limite l'application de la minimisation de suite de tests lors de la maintenance et de l'évolution du programme sous test.

Ces problèmes et limitations pour la minimisation de suite de test ont motivé l'apparition d'une autre technique pour réduire le coût du test de non-regréssion: la prioritisation de cas de test.

4.2 Prioritisation de cas de test

Pour s'abstraire des problèmes précédents, la prioritisation d'une suite de tests conserve tous les cas de test de la suite de test initiale et se contente de les ordonner pour optimiser le taux de détection de faute de la suite de tests. C'est à dire, par exemple, obtenir, au cours de l'exécution de la suite de test, le plus rapidement possible la couverture du code de l'application sous test.

Le problème de la prioritisation de suite de test est défini comme suit par S.Elbaum [10] :

Définition 6. *Problème de prioritisation d'une suite de test :*
Étant donné une suite de test T, P_T l'ensemble des permutations de T et f une fonction de P_T à valeur réelle.
Le problème est de trouver $T' \in P_T$ tel que $\forall T'' \in P_T, T'' \neq T' \Rightarrow f(T') \geqslant f(T'')$

Dans cette définition, P_T représente l'ensemble des ordonnancement de cas de test possible et f est une fonction qui évalue la qualité d'un ordonnancement. Bien évidement un premier problème apparaît avec cette fonction f qu'il reste à définir et dont le rôle est de quantifier le critère que l'on veut couvrir en priorité.

Exemple 7. Si l'objectif de la prioritisation d'une suite de test est d'atteindre le plus rapidement possible la couverture des fonctions du programme sous test, alors f pourra être l'aire verte sous les courbes de la figure 9. Cette figure montre l'évaluation de cette fonction pour deux ordonnancements des cas de test A, B, C, D, E pour couvrir les fonctions $F_1, F_2, ..., F_5$.

Cas de test	F1	F2	F3	F4	F5
A	X				
B	X			X	
C				X	
D	X	X	X		
E	X	X			X

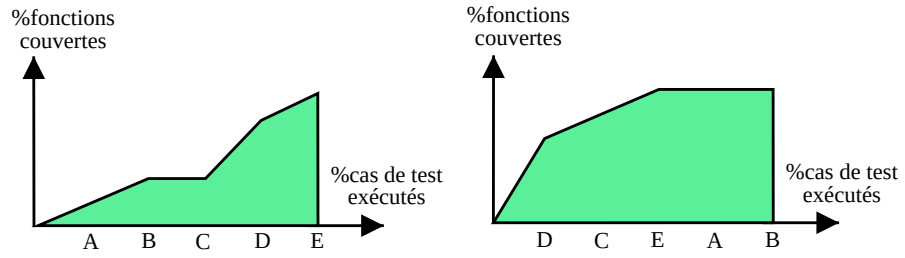

Figure 9. Exemple de critère de prioritisation : Couverture de fonction.

L'idéal serait de définir une fonction f, permettant d'évaluer une suite de tests optimale, qui quantifierait le taux de détection de faute d'une suite de test ordonnée. Mais dans le cas général ce problème est indécidable et il faudra se contenter d'heuristiques tel que celle définie dans l'exemple précédent.

Les travaux [10, 9, 20, 21] définissent un grand nombre d'heuristiques utilisables pour prioritiser des suites de test. Parmi ces heuristiques, deux classes ressortent selon la granularité des informations qu'elles exploitent. La figure 10 présente un sous ensemble de ces heuristiques.

	H0	Aléatoire
	H1	Optimale
Granularité Instruction	H2	Couverture d'instruction
	H3	Couverture d'instruction non encore couverte
	H4	Probabilité de détecter une erreur
	H5	Probabilité de détecter une erreur ajusté selon les cas de test précédents
Granularité Procedure	H6	Couverture de fonction
	H7	Couverture de fonction non encore couverte
	H8	Probabilité de détecter une erreur
	H9	Probabilité de détecter une erreur ajusté selon les cas de test précédents

Figure 10. Quelques heuristiques de prioritisation de cas de test

- H0 : L'heuristique d'ordonnancement aléatoire, équivalente à l'absence de prioritisation, est définie afin de pouvoir évaluer l'utilité des heuristiques définies par la suite.
- H1 : Cette heuristique théorique correspond à l'ordonnancement idéal à atteindre pour maximiser le taux de détection de faute d'une suite de test. Ici encore cette heuristique est définie afin d'évaluer la qualité des autres. En effet, dans le cadre d'une étude de cas, connaissant les versions erronées du programme sous test, il est possible de déterminer l'ordonnancement optimale.
- H2 : Consiste à ordonner les cas de test selon le nombre d'instructions du programme sous test qu'ils couvrent.

- H3 : Consiste à ordonner les cas de test afin de couvrir le plus de code possible, le plus rapidement possible. Pour cela on commence par choisir le cas de test qui couvrent le plus de code, puis, dans les cas de test restant, on choisit celui qui couvre le plus d'instructions non-encore couvertes par les cas de test déjà ordonnés. Et ce, jusqu'à ce que tous les cas de tests soient ordonnés.
- H4 : Cette heuristique utilise une évaluation pour chaque cas de test et pour chaque instruction de la probabilité que le cas de test détecte une erreur des cette instruction. Bien évidement, cette probabilité ne peut être qu'une valeur approchée. Pour calculer cette valeur, [10] propose d'utiliser l'analyse de mutation.
- H5 : Idem H4 en utilisant un algorithme proche de H3.
- H6, H7, H8, H9 : Même type d'heuristiques que H2, H3, H4, H5 mais avec une granularité plus importante.

Les résultats des études de cas menées dans [10, 20] en utilisant ces heuristiques montrent que dans la plupart des cas ces heuristiques sont équivalentes et augmentent de façon notable le taux de détection de faute par rapport à l'ordonnancement aléatoire. C'est le premier résultat intéressant de ces études : on pourra se contenter d'utiliser des algorithmes très simple et peu coûteux tel que H6 ou H7 qui ne s'intéressent qu'à la couverture de procédure.

Toutefois, un pas reste a franchir entre les résultats de ces heuristiques et l'optimal. Pour tenter de franchir ce pas, différentes heuristiques plus complexes sont proposées. Dans [9], les auteurs proposent par exemple des ordonnancements prenant en compte les différentes versions du logiciel, afin de tester en priorité les parties les plus modifiées. Dans [10], les auteurs proposent également d'utiliser des mesures de complexité de procédure pour tester en priorité les procédures les plus complexes, l'hypothèse étant que ces procédures sont celles qui ont le plus de chance de contenir des erreurs.

Les différentes heuristiques proposées dans ce cadre sont en général relativement complexes, et, dans la pratique difficilement utilisables. Toutefois l'apport d'heuristiques simples et facilement applicables tel que celles évoquées précédemment (figure 10) est non négligeable.

5 Assistance au diagnostic grâce aux traces d'exécution

Le diagnostic logiciel consiste à déterminer, à partir de l'observation d'anomalies de fonctionnement d'une implantation, la partie défectueuse de cette implantation. C'est une des phases les plus coûteuses du processus de test/mise-au-point de programme lorsque celui-ci est réalisé entièrement à la main.

Dans la plupart des cas le diagnostic est facilité par l'utilisation de "debuggers" permettant au programmeur de tracer l'exécution du programme qu'il teste. Il peut ainsi visualiser l'évolution de l'état du programme instruction par instruction et localiser finement les cause des anomalies observées. Cela étant cette méthode a ses limites. En effet, un debugger n'offre qu'une vision très locale du programme en cours d'exécution et nécessite que l'utilisateur ait fait un effort de compréhension globale du programme qu'il teste. Pour répondre à ces problèmes des méthodes de diagnostic complémentaires existent. Nous nous intéressons à une méthode de localisation des fautes proposée par H. Agrawal et J.R Horgan [2] puis reprise et améliorée par J.A. Jones et M.J. Harrold [15, 8, 16]. Ces méthodes se basent sur les techniques de *découpage dynamique*.

5.1 Découpage dynamique et diagnostic

Le découpage dynamique (ou *dynamic slicing*) repose sur la notion de trace d'exécution d'un programme. Le découpage dynamique d'un programme consiste à calculer des coupes (ou *slice*) du programme contenant uniquement les instructions du programme affectant un résultat.

La figure 11 présente une fonction et trois coupes dynamiques, c'est a dire les ensembles d'instructions activées pour calculer le résultat lors de trois exécutions avec des paramètres x et y différents.

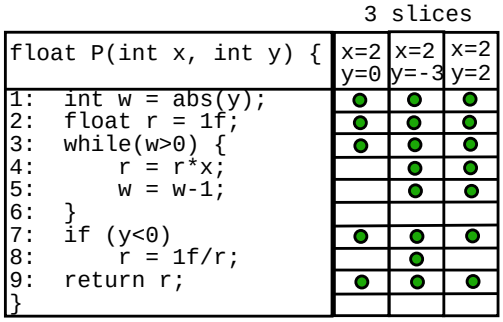

Figure 11. Exemples de coupes dynamiques

L'idée des algorithmes de diagnostic, auquel nous nous intéressons, est de calculer les coupes dynamiques de chaque cas de test, puis de les exploiter en utilisant les verdicts obtenus pour chaque cas de test afin de localiser les erreurs détectés par les cas de test ayant échoués.

5.2 Technique de "dicing "

La première méthode à laquelle nous nous intéressons a été proposée par H. Agrawal [2]. L'idée est ici de calculer des différences (ou *dice*) entre les coupes de cas de test ayant échoués et coupes de cas de test réussis.

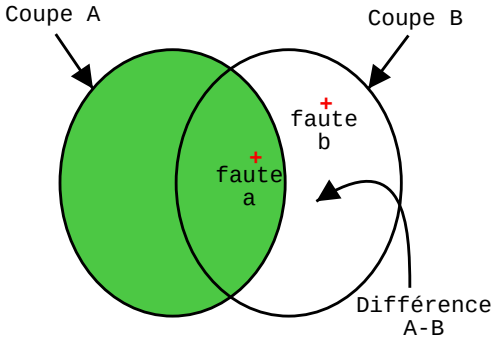

Figure 12. Fautes, coupes et différences

Les hypothèses faites dans le cadre de cette méthode sont les suivantes. Si un cas de test échoue c'est qu'une faute est présente dans la coupe engendrée par ce cas des test. Dans le cas contraire, si le cas de test réussit, on considère que la coupe engendrée par ce cas de test ne contient pas d'erreur. Cette dernière hypothèse est forte car elle suppose que l'oracle utilisé est idéal, c'est a dire que si une instruction erronée est exécutée son verdict est toujours négatif.

Sur la figure 12 ces hypothèses peuvent s'exprimer ainsi: si la coupe A est la coupe engendrée par un cas de test réussit et la coupe B par un cas de test non-réussit alors l'erreur est dans la différence $A - B$. On considère donc que la faute a de la figure, étant dans l'intersection des coupes A et B est improbable. Par contre, la faute b étant uniquement dans la coupe B, elle est présente dans la coupe $A - B$. La coupe $A - B$ étant nécessairement plus petite que la coupe B, l'espace de recherche de la faute a été réduit. Ce procédé permet donc au programmeur de réduire l'espace de recherche d'erreur en réalisant des différences entre cas de test réussis et cas de test non-réussis.

L'équipe d'Agrawal propose l'outil χSlice [1] qui implante la possibilité de calculer des coupes et des différences entre coupes. La figure 13 présente une capture d'écran de cet outil issue de sa documentation.

Un problème reste cependant en suspend. Dans la cas général, on dispose d'un ensemble de cas de test réussis et d'un ensemble de cas de test non-réussis, ce qui conduit à un grand nombre de différences (ou *dices*) possibles. Si une suite de tests comporte t cas de test et que k échouent il existe $k\,t - t^2$ *dices*. Ces *dices* ayant des tailles différentes, l'heuristique présentée dans [2] est d'examiner d'abord les *dices* les plus petit.

L'étude de cas menée avec l'outil χSlice (figure 13) dans [2] sur 32 versions mutantes du programme unix *find* (914 lignes de code C) contenant chacune une erreur montre une réduction notable de la quantité de code suspect en utilisant l'heuristique simple consistant à examiner les coupes les plus petites.

Pour limiter encore les zones de code suspect on pourrait être tenté de faire l'intersection des coupes défavorables et de lui retrancher l'union des coupes favorables. Cette possibilité est évoquée mais pas étudiée car les résultats seraient très mauvais si le programme contenait plusieurs fautes.

Dans le cas de simples différences comme ici, bien que l'étude n'ait pas été faite on peu imaginer que la méthode reste efficace même si le programme contient plusieurs fautes. L'espace de recherche serait alors plus grand mais contiendrait toujours les instructions erronées (en gardant l'hypothèse faite sur l'oracle).

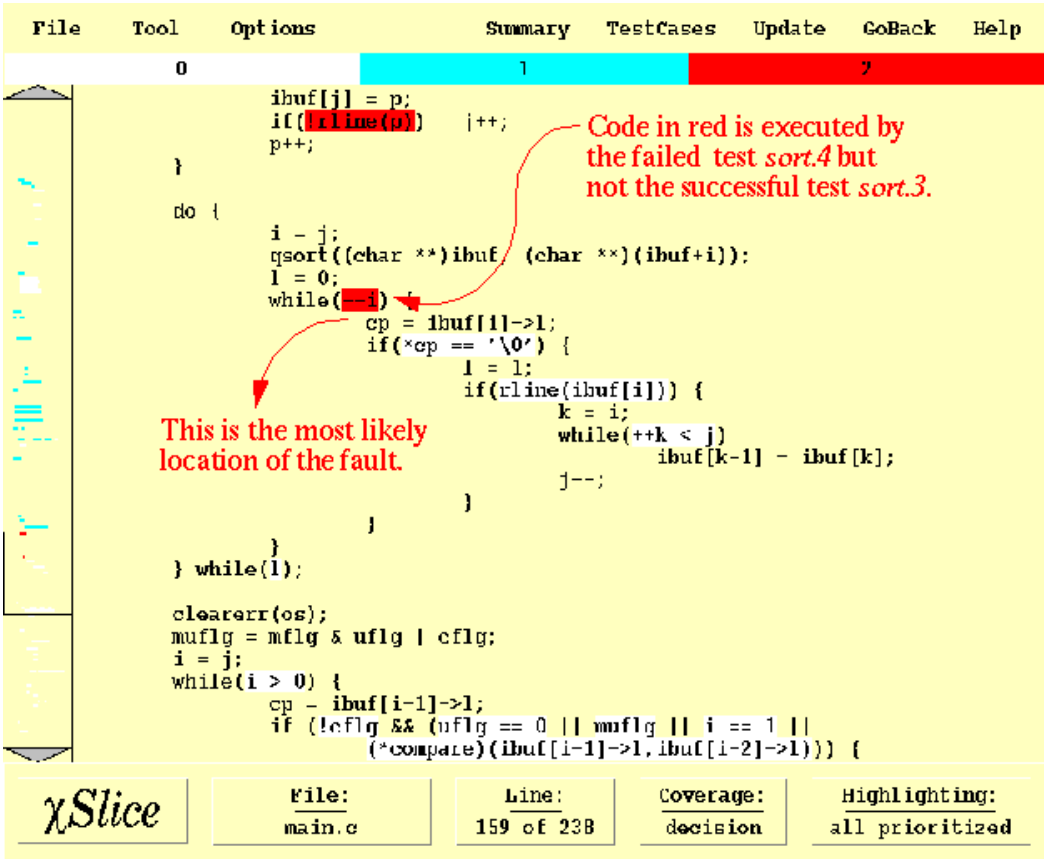

Figure 13. L'outil χSlice

5.3 Recoupement de traces

Dans cette section nous présentons l'approche proposée par Jones et Harrold dans [16]. Comme nous l'avons vu précédemment, l'ensemble des coupes dynamiques (ou traces d'exécution) des cas de test fournissent une grande quantité d'informations utiles au diagnostic. Dans les techniques de *dicing* seule l'information venant de deux traces d'exécution (un *dice*) était présentée au programmeur. L'objectif de la méthode que nous présentons ici est de colorer le code source en incluant les informations venant des traces de tous les cas de test.

Pour illustrer le propos nous allons reprendre l'exemple de programme erroné utilisé dans [16] (figure 14). L'exemple est une fonction qui lit trois entiers et affiche celui qui est compris entre les deux autres. Une erreur a été introduite à la ligne 7 (pour que la fonction soit correcte il faudrait $m = x$). On dispose pour cette fonction de 6 cas de test dont les entrées, traces et verdicts sont donnés sur la figure 14.

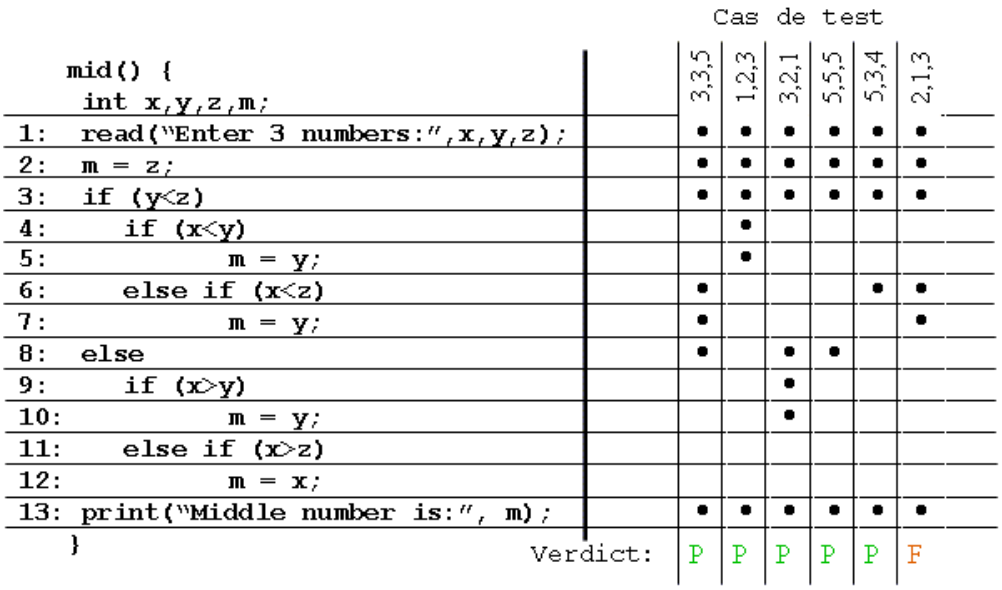

Figure 14. Exemple de programme comportant une erreur (figure tirée de [16])

5.3.1 Approche discrète

La première méthode étudiée par [16], appelée *discrete three-color mapping*, consiste, en utilisant les traces de tous les cas de test, à colorer les instructions suivant qu'elles apparaissent dans des traces de cas de test réussis ou dans des traces de cas de test non-réussis. Si une instruction n'apparaît que dans des traces de cas de test non réussis elle est fortement suspecte et sera colorée en rouge. Si elle n'est exécutée que par des cas de test réussis, elle n'a que peut de chance de contenir des erreurs et elle est colorée en vert. Et enfin dans le cas ou elle est exécutée à la fois par des cas de test réussis et non-réussis, elle est colorée en jaune.

Le code ainsi présenté au programmeur permet de réduire les zones de code à examiner à celles présentées en rouge. Si l'erreur n'est pas présente dans les parties de code rouge ou si comme sur l'exemple de la figure 15 aucune instruction n'apparaît en rouge, le programmeur peut alors élargir son domaine de recherche aux instructions présentées en jaune.

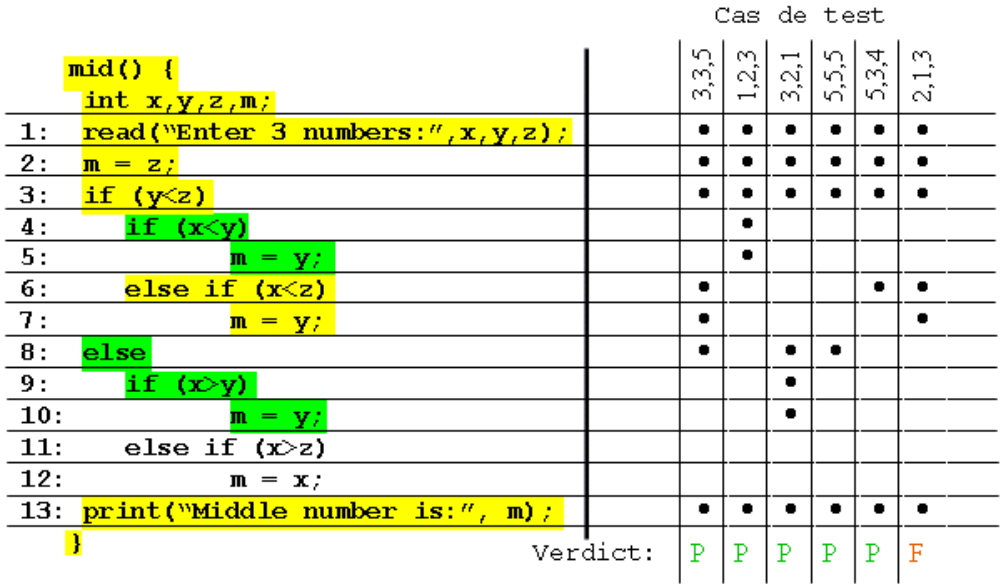

Figure 15. Approche discrète (figure tirée de [16])

La figure 15 montre le résultat de l'application de cette technique sur l'exemple présenté précédemment. Les résultats sont un peu décevants, aucune instruction n'apparaît en rouge et si l'on se limite aux instructions jaunes, l'espace de recherche est aussi important que si l'on devait chercher dans la coupe relative aux cas de test non-réussis.

5.3.2 Approche continue

Comme nous l'avons vu, l'approche discrète ne suffit pas à réduire de façon importante l'espace de recherche des erreurs. Une seconde méthode plus fine est donc proposée par [16] : l'approche continue (ou *continous visual mapping*). La motivation est encore ici d'augmenter la quantité d'information affichée au programmeur en adoptant non plus un modèle avec 3 couleurs mais un dégradé continue de couleur en faisant varier la couleur (*hue*) et la luminosité (*brightness*) de chaque instruction.

La couleur d'une instruction est calculée comme le pourcentage relatif de cas de test réussit qui exécute cette instruction par rapport au cas de test non-réussis qui exécute cette même instruction (équation 1). Ainsi si un plus grand pourcentage de cas de test qui exécute cette instruction sont réussis, l'instruction apparaîtra dans les verts. Dans le cas contraire elle apparaîtra dans une couleur tendant vers le rouge.

L'équation 1 donne l'expression de la couleur d'une instruction i en fonction du pourcentage de cas de test réussit ayant exécuté i (noté $\%P(i)$) et du pourcentage de cas de test non-réussis ayant exécuté i (noté $\%F(i)$).

$$\text{couleur}(i) = \text{rouge} + \frac{\%P(i)}{\%P(i) + \%F(i)} \times (\text{vert} - \text{rouge}) \tag{1}$$

La seconde composante associée à chaque instruction est la luminosité qui est quant à elle calculée en fonction de la couverture de l'instruction considérée par les cas de test. L'équation 2 montre comment est calculée la luminosité d'une instruction i.

$$\text{luminosite}(i) = \max(\%P(i), \%F(i)) \tag{2}$$

Cette composante de luminosité peut être interprétée comme une mesure de la confiance que l'on peut avoir dans la prédiction établie par la couleur. En effet, une instruction apparaîtra comme très lumineuse si $\%P(i)$ est grand ou si $\%F(i)$ est grand, ce qui implique que la couleur a été calculée à partir d'un grand nombre de cas de test. La confiance que l'on peut avoir dans cette couleur en est alors renforcée.

La figure 16 montre les résultats obtenus en utilisant cette méthode sur le même exemple que précédemment. Cette fois ci, l'erreur (ligne 7) est nettement mise en évidence par la coloration des instructions.

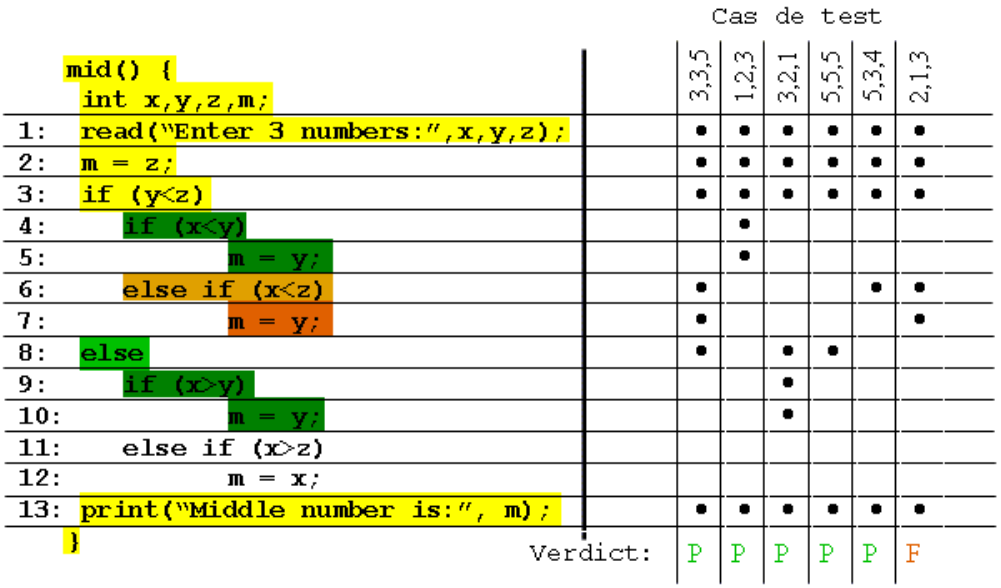

Figure 16. Approche continue (figure tirée de [16])

Dans l'article [16], cette approche est validée expérimentalement par une étude de cas sur le programme *space* (9564 lignes de code C) en utilisant l'outil *Tarentula* (figure 17). Les expérimentations portent aussi bien sur des programmes mutants comportant une seule faute et plusieurs fautes.

Figure 17. Capture d'écran de l'outil Tarentula (figure tirée de [16])

6 Conclusion et perspectives

Nous avons étudié ici un grand nombre de techniques, très différentes, utilisées pour faciliter le test qui ont toutes en commun l'utilisation des traces d'exécution. Dans cette section nous passons rapidement en revue ces méthodes afin de synthétiser ce qu'elles peuvent apporter au test de systèmes logiciels.

Le premier point que nous avons abordé concerne la génération de données de test. La première méthode à laquelle nous nous sommes intéressés utilise les traces d'exécution et les algorithmes génétiques pour générer des cas de test. Cependant cette méthode n'est applicable que pour des systèmes relativement simples dont les entrées sont numériques. Nous avons ensuite introduit les algorithmes bactériologiques au travers d'une méthode de génération de tests traitant de composant logiciels plus complexes. Bien que cette méthode n'utilise pas les traces d'exécution, il serait intéressant de reprendre l'algorithme bactériologique proposé afin de l'appliquer à une méthode de génération de test les utilisant. En effet, dans les travaux traitant des algorithmes bactériologiques, l'utilité des cas de test est calculée en utilisant l'analyse de mutation. On pourrait proposé une fonction d'utilité plus facile à évaluer, basée sur un critère de couverture de code, utilisant les traces d'exécution.

Nous nous sommes ensuite intéressés à la minimisation de suites de test pour rapidement s'apercevoir de sa limite. Nous avons donc étudié plus en détail la prioritisation de cas de tests qui semble d'une part plus facile a mettre en oeuvre et plus fiable. Cette méthode nous intéresse particulièrement parce qu'elle consiste à ordonner des cas de test en utilisant des heuristiques qui sont pour la plupart basées sur les traces d'exécutions. L'étude de ces méthodes nous a montré que des critères structuraux simples permettent de distinguer des cas de test. Notre objectif est, partant de ce constat, de proposer une méthodes de classification de cas de test en utilisant les traces d'exécution. Cette classification des données de test pouvant ensuite être utilisée pour la prioritisation mais aussi lors du diagnostic.

Enfin nous avons étudié deux méthodes d'aide au diagnostic permettant de réduire les zones de code suspects à partir des traces d'exécution des différents cas de tests. Après avoir montré les limites de la première approche, nous nous sommes intéressés à des travaux plus récents qui proposent une technique facilement transposable au test de composants objets. Cependant ces méthodes de diagnostic sont très dépendantes de la suite de test qui est utilisée. Notre objectif dans ce domaine est de proposer un algorithme alliant génération de données de test et diagnostic. En effet, dans le cas ou les zones de code suspectes ne sont pas suffisamment réduites par le diagnostic, il serait intéressant de mettre en oeuvre un générateur de cas de test utilisant les données produites par le diagnostic. Ainsi, les cas de test produits permettraient d'améliorer le diagnostic.

Au cours de mon stage dans l'équipe TRISKELL de l'IRISA nous tenterons de développer et d'expérimenter les différentes techniques évoquées précédemment.

Bibliographie

[1] xslice : A tool for program debugging. http://xsuds.argreenhouse.com/html-man/xslice.html.

[2] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. *Proc. of the 6th IEEE International Symposium on Software Reliability Engineering*, pages 143–151, October 1995.

[3] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. Automatic test cases optimization using a bacteriological adaptation model: Application to .net components. *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, 2002.

[4] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. Genes and bacteria for automatic test cases optimization in the .net environment. In *procedings of the Thirteenth International Symposium on Software Reliability engineering (ISSRE)*, november 2002.

[5] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison Wesley, 1999.

[6] Y. Cheon and G.T. Leavens. A simple and practical approach to unit testing: The jml and junit way. In *ECOOP*, 2002.

[7] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, April 1978.

[8] J. Eagan, M. J. Harrold, J. Jones, and J. Stasko. Visually encoding program test information to find faults in software. Technical report, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, June 2001.

[9] S. Elbaum, A.G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *proceedings of IEEE Transactions on Software Engineering*, pages 159–182, February 2002.

[10] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. Technical report, Computer Science Departmeent, Oregon State University, February 2000.

[11] Sebastian G. Elbaum, David Gable, and Gregg Rothermel. The impact of software evolution on code coverage information. In *ICSM*, pages 170–179, 2001.

[12] Mary Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.

[13] J. Holland. *Adaptation in natural and artificial systems*. The University of Michigan Press, 1975.

[14] B. Jones, H.Sthamer, and D.Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal 11*, pages 299–306, September 1996.

[15] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization for fault localization. *Proceedings of the Workshop on Software Visualization*, 2001.

[16] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. *Proceedings of the 24th International Conference on Software Engineering*, 2002.

[17] B. Meyer. Applying "design by contract". *Computer (IEEE)*, pages 40–51, October 1992.

[18] R. Pargas, M.J. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Journal of software testing, Verification and Reliability*, 9:263–283, September 1999.

[19] M.L. Rosenzweig. *Species Diversity In Space and Time*. Cambridge University Press, 1995.

[20] Gregg Rothermel, Roland Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.

[21] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. *procedings of International Conference on Software Maintenance*, page 179, September 1999.