Modélisation des comportements erronés du logiciel et application à la validation des tests par injection de fautes

Muriel Daran

Année 1996

THÈSE

présentée au
Laboratoire d'Analyse et d'Architecture des Systèmes du CNRS
en vue d'obtenir le titre de
Docteur de l'Institut National Polytechnique de Toulouse
Spécialité : Informatique

par

Muriel DARAN
Maître ès Sciences

MODÉLISATION DES COMPORTEMENTS ERRONÉS DU LOGICIEL ET APPLICATION À LA VALIDATION DES TESTS PAR INJECTION DE FAUTES

Soutenue le 28 octobre 1996 devant le jury :

M.	Jean-Claude	LAPRIE	Président
M.	Jean-Pierre	FINANCE	Rapporteur
M.	Gérard	LADIER	Examinateur
M.	Jean-Cyril	LAPLACE	Examinateur
M.	Larry J.	MORELL	Examinateur
Mme	Pascale	THÉVENOD-FOSSE	Directeur de thèse

Mme Marie-Claude GAUDEL est également rapporteur de cette thèse

Rapport LAAS n° 96497

Cette thèse a été préparée dans le cadre du
Laboratoire d'Ingénierie de la Sûreté de fonctionnement,
Laboratoire coopératif LAAS-CNRS / Aérospatiale / EDF /
Matra Marconi Space / Technicatome / Thomson-CSF

LAAS-CNRS, 7 avenue du Colonel Roche, 31077 Toulouse Cedex 4

AVANT-PROPOS

Les travaux présentés dans ce mémoire ont été réalisés, au sein du Laboratoire d'Ingénierie de la Sûreté de Fonctionnement (*LIS*), grâce à la collaboration entre le laboratoire d'Analyse et d'Architecture des Systèmes (LAAS-CNRS) et la société Technicatome.

Je témoigne ma sincère gratitude à Monsieur Alain COSTES, Directeur du LAAS-CNRS, pour m'avoir accueillie dans ce laboratoire.

J'exprime ma profonde reconnaissance à Jean-Claude LAPRIE, Directeur de Recherche au CNRS, Responsable du groupe *Tolérance aux fautes et Sûreté de Fonctionnement informatique* (TSF) et Directeur du *LIS*, à Olivier DIEUDONNÉ et à Jean-Max GAUBERT, Responsables successifs du service *Contrôle-Commande Electricité* de Technicatome, pour la confiance qu'ils m'ont accordée en me confiant ces travaux. Je leur sais gré de l'environnement scientifique, intellectuel et matériel qui m'a permis d'effectuer ces travaux dans les meilleures conditions.

Je remercie Monsieur Jean-Claude LAPRIE pour l'honneur qu'il me fait en présidant mon Jury de thèse, ainsi que :

- Monsieur Jean-Pierre FINANCE, Professeur et Président de l'Université Henri Poincaré de Nancy,
- Monsieur Gérard LADIER, Responsable du service *Méthodes de Développement* du département *Logiciel* du Centre Opérationnel *Systèmes et Services* de la branche aéronautique de l'Aérospatiale,
- Monsieur Jean-Cyril LAPLACE, Responsable de la section *Logiciel* du service *Contrôle-Commande Electricité* de la direction de l'Ingénierie de Technicatome,
- Monsieur Larry J. MORELL, Professeur associé à l'Université de Hampton, Virginie, USA,
- Madame Pascale THÉVENOD-FOSSE, Directeur de Recherche au LAAS-CNRS,

pour l'honneur qu'ils me font en participant à ce Jury. Je remercie vivement Monsieur Jean-Pierre FINANCE et Madame Marie-Claude GAUDEL, Professeur à l'Université

LAAS-CNRS pour leur aide efficace et d'autant plus appréciée dans les moments de "bourre".

Enfin, mes remerciements s'adressent à l'ensemble des membres du groupe TSF et du *LIS*, qui par leur présence et leurs compétences, m'ont permis de travailler dans un cadre agréable et enrichissant. Que Jean-Paul BLANQUART, Michel CUKIER et Rodolphe ORTALO soient remerciés pour leur lecture attentive de ce mémoire et leurs remarques judicieuses. Un grand merci aux doctorants et aux stagiaires du groupe pour leurs divers coups de pouce, l'amitié qu'ils m'ont témoignée, leur énergie et leur bonne humeur.

Je réserve ici une pensée personnelle et privilégiée à Philippe et Guillaume qui ont supporté avec patience, tolérance et amour mes doutes et mes absences. Je n'oublierai pas dans mes pensées le (la) petit(e) dernier(e) pour sa présence épanouissante mais discrète durant la rédaction de ce manuscrit. Que ma petite famille me pardonne pour le temps que cette thèse leur a volé et qu'il va être doux de la retrouver !

Enfin, que tous les proches, parents et amis, qui m'ont aidée et encouragée et qui ont pallié mes absences maternelles trouvent ici un témoignage de mon affection et de ma reconnaissance.

Paris-Sud, qui ont accepté la charge d'être Rapporteurs, en dépit de leurs nombreuses obligations.

Je remercie vivement Henri JALADIEU, Directeur de la société ISIS, pour m'avoir permis de poursuivre mes travaux afin de les mener à leur terme ces six derniers mois.

Une partie des études expérimentales présentées dans ce manuscrit s'appuie sur un cas réel issu d'un projet développé par la section DI/SCE/LOG de Technicatome. Je remercie l'ensemble des personnels de cette section et du service DI/SCE pour leur contribution et leur accueil chaleureux à chacune de mes visites ou de mes coups de téléphone.

Je remercie sincèrement Jean-Cyril LAPLACE, pour l'intérêt qu'il a porté envers mes travaux et pour m'avoir encadrée au sein de Technicatome. Ses encouragements, sa connaissance des calculateurs étudiés et la pertinence de ses remarques m'ont été d'une grande aide tout au long de ma thèse.

Je ne saurais trop remercier Pascale THÉVENOD-FOSSE, responsable de mes travaux, pour m'avoir encadrée et soutenue (scientifiquement et moralement) tout au long de cette thèse ; ce fut un réel plaisir de travailler avec elle. J'ai pu apprécier, durant ces trois années, sa rigueur intellectuelle, ses compétences, sa patience sans limite et sa grande disponibilité. Ses conseils avisés, ses lectures attentives des différentes versions du manuscrit et son aide dans les moments de doute ont fortement contribué au bon déroulement et à l'aboutissement des travaux présentés dans ce mémoire. Qu'elle trouve ici un témoignage de mon estime et de ma reconnaissance.

Il m'est particulièrement agréable de remercier Yves CROUZET, Chargé de Recherche au CNRS et "père" de l'outil SESAME, sans qui les études expérimentales présentées dans ce mémoire n'auraient pu être menées à bien. Sa disponibilité (même pendant les vacances), ses interventions sur mon Mac et les discussions que nous avons eu sur l'analyse de mutation m'ont été d'un grand secours.

Je souhaite également remercier tous ceux qui m'ont aidée pendant cette thèse et qui ont contribué par leur disponibilité et leur efficacité à son bon déroulement. Un grand merci à Joëlle PENAVAYRE et Marie-Josée FONTAGNE, pour leur sourire, leur présence et leur collaboration dans l'organisation des derniers préparatifs en vue de la soutenance. J'associe à ces remerciements les membres du service technique "Documentation-Édition" et des différents services administratifs et logistiques du

SOMMAIRE

INTRODUCTION GÉNÉRALE

Les systèmes informatiques occupent une place sans cesse croissante dans notre société ; d'une part, parce que les avancées technologiques ont rendu possible le développement d'applications plus complexes et plus importantes, et d'autre part, parce que les domaines d'application de ces systèmes se sont étendus.

Le logiciel est amené à assurer des fonctions de plus en plus complexes, notamment dans le contrôle-commande de systèmes critiques (par exemple, dans les domaines des transports, du nucléaire ou de l'espace). Si l'utilisation de techniques de tolérance aux fautes matérielles est répandue, il n'en est pas de même en ce qui concerne le logiciel. De ce fait, la principale source de défaillance des systèmes tolérants aux fautes devient le logiciel [OFTA 1994]. L'explosion d'Ariane 5, lors de son premier vol expérimental, en est l'exemple le plus récent.

L'accident d'Ariane 5 est dû à une défaillance provoquée par une faute logicielle dans le système de référence inertielle [Lions 1996]. Les nombreux essais et simulations, menés durant le développement de ce système, n'ont pas permis de mettre en évidence la faute correspondante. Cette faute a généré une erreur qui a conduit à l'activation d'un traitement d'exception. Ce traitement d'exception a déclenché, conformément à sa spécification, la mise hors circuit des deux équipements de référence inertielle. Cette faute logicielle n'explique pas à elle seule l'échec de la mission : la spécification et la conception des mécanismes[1] de traitement des exceptions ont également été mis en cause. Cet accident souligne clairement que, malgré des méthodes de développement rigoureuses et des techniques de vérification du logiciel de plus en plus élaborées, on ne peut garantir l'absence de fautes dans un logiciel.

En effet, l'absence d'un modèle des fautes logicielles complet et parfait pose le problème de la confiance que l'on peut accorder aux techniques de validation — et notamment aux tests — par rapport à l'élimination des fautes dans un programme. Généralement, lors du développement d'un logiciel critique, des critères de test (structurels ou fonctionnels) sont utilisés pour guider la sélection a priori ou la validation a posteriori des entrées de test. Or, ces critères ne sont pas directement liés aux fautes

---

1 Ces mécanismes ont été conçus pour tolérer les fautes du matériel ; les fautes du logiciel n'ont pas été prises en compte.

- en proposant des applications de l'analyse de mutation.

Ce mémoire comporte six chapitres.

Le **premier chapitre** décrit les motivations qui nous ont conduites à nous intéresser à la validation des tests du logiciel et à l'analyse de mutation. Nous présentons d'abord certains concepts de la sûreté de fonctionnement (notamment, les définitions associées aux concepts de faute, d'erreur et de défaillance), puis les techniques de vérification du logiciel et en particulier le test. Nous soulignons ensuite les faiblesses liées aux critères de sélection ou de couverture des tests vis-à-vis de la recherche de fautes. Nous décrivons les principes de l'analyse de mutation, qui est une approche expérimentale permettant d'évaluer l'efficacité des tests du logiciel, et basée sur l'injection de fautes. L'utilisation de cette technique pose le problème de la représentativité des fautes injectées vis-à-vis des fautes réelles. Ce problème nous a amené à étudier les comportements erronés du logiciel avant d'envisager des applications de l'analyse de mutation.

Le **deuxième chapitre** décrit, dans un premier temps, la problématique liée à nos études expérimentales. Nous définissons les types d'erreurs étudiés ainsi que les mécanismes de propagation possibles d'une erreur. Dans un deuxième temps, nous décrivons le cadre expérimental utilisé pour l'ensemble de nos expérimentations.

Le **troisième chapitre** présente la première série d'expérimentations menées dans le but d'identifier et de comparer les erreurs et les comportements erronés dus à des fautes réelles d'une part, et à des fautes artificielles de type mutation d'autre part. Ces expérimentations nous ont permis de disposer de deux bases de données conséquentes d'erreurs à partir desquelles ont pu être réalisées des analyses comparatives.

Dans le **quatrième chapitre**, nous tentons d'analyser les comportements erronés précédemment observés par l'étude des dépendances du programme. Nous proposons un formalisme qui nous permet d'expliquer les mécanismes de propagation d'une erreur que sont la création d'erreur, le masquage d'erreur et l'annulation d'erreur.

Le **cinquième chapitre** présente la deuxième série d'expérimentations qui a pour objectif de mettre en œuvre, sur un cas réel, la méthode d'analyse des dépendances proposée au quatrième chapitre. Nous expliquons ainsi les comportements erronés générés par des fautes réelles — fautes qui ont été révélées en milieu industriel, lors des

qu'ils sont censés révéler. Il paraît donc judicieux d'envisager une méthode permettant d'évaluer l'efficacité des entrées de test générées pour une application donnée, vis-à-vis de la recherche de fautes. Une telle méthode existe : c'est l'analyse de mutation (technique d'injection de fautes élémentaires dans le code d'un programme). Mais elle a suscité de nombreuses critiques en raison principalement du problème de la représentativité des fautes injectées. S'il paraît illusoire de démontrer la représentativité des mutations vis-à-vis de fautes réelles, il a cependant été observé, lors de précédentes études expérimentales, que des mutations pouvaient générer des comportements erronés subtils et aussi complexes que ceux créés par des fautes réelles connues.

Nos travaux de recherche s'inscrivent, au sein du *LIS*, dans le thème des "logiciels sûrs de fonctionnement". Ils portent sur la modélisation des comportements erronés en faisant référence à une représentation de l'état interne d'un programme en cours d'exécution. Cette approche est originale puisque nous nous sommes intéressée aux erreurs générées au cours de l'exécution du logiciel, et non pas aux fautes résultant du développement, et parce qu'elle s'appuie sur de nombreuses analyses expérimentales. S'il s'avère que les erreurs et les comportements erronés produits par des mutations sont représentatifs de ceux générés par des fautes réelles connues, alors l'utilisation de l'analyse de mutation pour évaluer l'efficacité d'entrées de test peut être reconsidérée.

Les études expérimentales que nous avons menées portent sur deux programmes issus d'applications critiques du nucléaire qui ont été développées dans des environnements industriels différents. Ces études sont, à notre connaissance, inédites sur des cas réels ; nous avons, en effet, pu disposer sur chacun des programmes étudiés, de fautes réelles, identifiées lors du processus de vérification, et de mutations. Ces programmes sont codés en langage C ; nos analyses ciblent donc le même type d'applications développées à l'aide de langages procéduraux.

Nos résultats permettent de réhabiliter l'analyse de mutation en tant que technique de validation des tests du logiciel :

- en montrant que les erreurs et les comportements erronés produits par des mutations peuvent être représentatifs de ceux générés par des fautes réelles ;
- en expliquant, par l'analyse des dépendances du programme, les similitudes observées entre les comportements erronés produits par des fautes réelles d'une part, et par des mutations d'autre part ;
- en s'interrogeant sur les différents critères utilisés pour modéliser des fautes ;

CHAPITRE I

VALIDATION DES TESTS DU LOGICIEL

I.1 INTRODUCTION

Le rôle du logiciel devient de plus en plus important au sein des systèmes informatiques actuels, tant du point de vue du nombre de fonctions assurées, que du point de vue de la complexité de ces fonctions ; le problème de la sûreté de fonctionnement de tels logiciels se pose alors de manière aiguë.

La **sûreté de fonctionnement** d'un système informatique est la propriété qui permet à ses utilisateurs de *placer une confiance justifiée dans le service qu'il leur délivre* [Laprie et al. 1995][2]. Cette propriété peut être altérée par la présence d'entraves que sont les fautes, les erreurs ou les défaillances ; elles sont les circonstances indésirables — mais non inattendues — causes ou résultats de la non-sûreté de fonctionnement. Une **défaillance** du système survient lorsque le *service délivré dévie de l'accomplissement* de la **fonction** du système, c'est-à-dire de ce à quoi le système *est destiné*. Une **erreur** est la partie de l'état du système qui est *susceptible d'entraîner une défaillance*. La *cause adjugée ou supposée* d'une erreur est une **faute**.

Face à ces entraves, la sûreté de fonctionnement se dote de moyens permettant de fournir au système l'aptitude à délivrer un service conforme à l'accomplissement de sa fonction, et de donner confiance dans cette aptitude. La validation[3] représente un de ces moyens ; elle associe les notions d'élimination des fautes — ou comment réduire la

---

2 Nous nous sommes limitée, dans cette introduction, aux définitions et concepts de la sûreté de fonctionnement nécessaires à la compréhension de nos travaux ; ils sont extraits de [Laprie et al. 1995] qui constitue un guide dans lequel le lecteur intéressé pourra trouver des présentations approfondies des concepts cités ici.

3 Dans d'autres contextes, la validation est souvent associée à la seule notion d'élimination des fautes, comme dans le processus de "Vérification et Validation" [Boehm 1981] ; la vérification permet alors de répondre à la question "a-t-on bien construit le système ?", alors que la validation correspond à la question "a-t-on construit le bon système ?".

différentes phases de vérification du logiciel. Nous montrons comment ces comportements erronés peuvent être reproduits par des mutations. Les résultats de ces nouvelles expériences viennent confirmer ceux obtenus lors de la première série d'expérimentations.

Enfin, dans le **sixième et dernier chapitre**, nous proposons des applications de l'analyse de mutation à des fins industrielles.

I.2.1 Faute, erreur et défaillance

Les mécanismes de création et de manifestation des fautes, erreurs, défaillances au niveau du logiciel sont liés par la chaîne causale suivante :

1) Une **faute** peut être créée tout au long du cycle de vie d'un logiciel, à n'importe quelle phase de son développement (phase de spécification, de conception, de réalisation, etc.) ou pendant la maintenance. Les fautes, ainsi créées, se traduisent de diverses manières au niveau du code : instruction(s) ou donnée(s) incorrecte(s), manquante(s) ou supplémentaire(s). Une faute peut être dormante ou active ; une faute **active** est une faute qui était préalablement **dormante** et qui a été activée lors de l'exécution du programme par des entrées appropriées (lors de l'exécution d'une instruction affectée par la faute). Une faute peut cycler entre ses états dormant et actif, lorsque en particulier cette faute est localisée dans le corps d'une boucle ou lorsque l'exécution du logiciel comprend plusieurs cycles successifs.

2) Une **erreur** (état interne incorrect) peut être créée suite à l'activation d'une faute. Une erreur peut être latente ou détectée ; une erreur est **latente** tant qu'elle n'a pas été reconnue en tant que telle ; une erreur est **détectée** par un test ou un mécanisme de détection. Une erreur peut disparaître sans être détectée. Par propagation, une erreur peut créer de nouvelles erreurs.

3) Une **défaillance** survient lorsque, par propagation, une erreur affecte le service délivré par le logiciel (par exemple, lorsqu'une erreur atteint une donnée observable en sortie), donc lorsqu'elle "passe à travers" l'interface logiciel-utilisateur(s). Dans le cadre de logiciels séquentiels, il y a occurrence d'une défaillance lorsqu'une erreur atteint une donnée observable en sortie[4].

Cette chaîne causale peut s'appliquer de manière récursive à différents niveaux d'abstraction :

... ➞ défaillance ➞ faute ➞ erreur ➞ défaillance ➞ faute ➞ ...

---

4 Dans le cadre de logiciels temps-réel, il peut y avoir également occurrence d'une défaillance lorsque les conditions temporelles de délivrance de service ne permettent pas l'accomplissement de la fonction du système.

présence de fautes — et de prévision des fautes — ou comment estimer la présence, la création et les conséquences des fautes.

L'élimination des fautes comporte trois étapes : la **vérification**, le **diagnostic** et la **correction**. La vérification consiste à déterminer si le système satisfait des propriétés, appelées conditions de vérification. Si tel n'est pas le cas, les deux autres étapes doivent être entreprises : diagnostiquer la ou les fautes qui ont empêché les conditions de vérification d'être remplies, puis apporter les corrections nécessaires. Dans le cadre du développement de logiciels critiques, la vérification constitue une activité cruciale étant donné le niveau de confiance exigé ; plusieurs techniques (analyse statique, vérification formelle, test, ...) sont généralement utilisées de manière combinée dans le but d'une meilleure efficacité vis-à-vis de la recherche de fautes. Cependant, en l'absence d'un modèle de faute représentatif de la réalité, aucune des approches actuelles ne permet de garantir l'élimination de toutes les fautes résiduelles. Dès lors se pose le problème de la confiance que l'on peut avoir dans un logiciel et dans les techniques utilisées pour le vérifier.

Ce premier chapitre a pour but de décrire le contexte de nos travaux de recherche et de définir certains concepts clés nécessaires à la compréhension de ce manuscrit. Le paragraphe I.2 décrit brièvement les différentes techniques de vérification du logiciel en insistant plus particulièrement sur le test. Le paragraphe I.3 expose le problème de la validation des tests du logiciel. L'une des méthodes de validation des jeux de test, appelée analyse de mutation, fait ensuite l'objet du paragraphe I.4.

I.2 VÉRIFICATION DU LOGICIEL

L'objectif de la vérification du logiciel est de révéler les fautes du logiciel ; dans le but d'une meilleure efficacité, différentes techniques sont généralement mises en œuvre au cours du développement d'un logiciel. Avant de présenter ces différentes techniques, nous rappelons, par souci de clarté, les définitions précises associées aux concepts de faute, d'erreur et de défaillance.

- le *test* qui consiste à activer le programme avec des entrées valuées [Beizer 1990] ; il existe de nombreuses méthodes de test, chacune étant caractérisée par le critère de sélection retenu pour le choix des entrées de test, et la méthode de génération utilisée (déterministe ou probabiliste).

Chacune de ces techniques concourt à l'élimination des fautes logicielles sans toutefois résoudre à elle seule le problème de l'existence de fautes dans un logiciel. C'est pourquoi une utilisation combinée de ces techniques est recommandée. L'analyse statique et le test constituent les techniques les plus répandues. Toutefois, l'utilisation de la vérification formelle (preuve mathématique de programme) tend à s'étendre de par le développement croissant d'outils et l'apparition de ces techniques dans des standards de développement [Abrial et al. 1991, Rushby et al. 1991, Rabéjac 1995].

I.2.3 Test du logiciel

Le test consiste à exécuter un programme avec des entrées valuées. Le test exhaustif d'un programme sur toutes ses entrées est pratiquement et théoriquement infaisable, sauf cas très particuliers. On est donc amené à sélectionner de manière pertinente un (petit) sous-ensemble du domaine d'entrée. Cette sélection peut être guidée soit par l'utilisation d'un modèle du logiciel (modèle structurel ou fonctionnel), soit par l'utilisation d'un modèle de faute.

I.2.3.1 Test basé sur un modèle du logiciel

La sélection des entrées de test peut être guidée par l'utilisation de **critères de test** : un critère de test définit un ensemble d'éléments d'un modèle du logiciel, éléments qui devront être activés pendant le test. Ces critères donnent lieu à des **tests fonctionnels** s'ils sont basés sur un modèle correspondant à une représentation "boîte noire" du logiciel, ou à des **tests structurels** si le programme est vu comme une "boîte de verre". Il existe de nombreux critères structurels ou fonctionnels qui peuvent être classés par ordre croissant de sévérité [Myers 1979, Beizer 1990].

Le modèle structurel "standard" est le graphe de contrôle du programme sur lequel sont basés différents critères : on peut citer dans l'ordre croissant de sévérité, par exemple, les critères "toutes les instructions", "toutes les branches", "tous les chemins". Une autre famille de critères consiste à utiliser le flot des données comme modèle structurel ; les critères liés au flot de données permettent de sélectionner des ensembles particuliers de sous-chemins entre les nœuds du graphe de contrôle contenant les

La conséquence de la défaillance d'un composant devient ainsi une faute pour le système qui le contient ou pour le, ou les composants qui interagissent avec lui ; par exemple, une faute dans le code du logiciel peut être considérée comme une défaillance du processus de développement, elle-même due à une erreur du programmeur.

Les flèches dans cette chaîne expriment la relation de causalité entre fautes, erreurs et défaillances. Elles ne doivent pas être interprétées au sens strict : par propagation, plusieurs erreurs peuvent être créées avant qu'une défaillance ne survienne ; une défaillance étant un événement se produisant à l'interface entre deux systèmes ou composants, une erreur peut conduire à une faute sans que l'on observe de défaillance si l'observation de la défaillance n'a pas lieu d'être effectuée, ou si elle ne présente pas d'intérêt.

I.2.2 Techniques de vérification du logiciel

Les techniques de vérification peuvent être classées selon qu'elles impliquent ou non l'exécution du logiciel :

- la **vérification statique** (sans exécution) inclut les techniques telles que :
  - l'*analyse statique* [Myers 1979, Strauss & Ebenau 1994] qui repose (i) sur l'analyse détaillée des documents produits au cours du développement organisée sous forme de lectures croisées, d'inspections ou de revues, ou (ii) sur des contrôles automatiques effectués à l'aide d'outils informatiques tels que des compilateurs ou analyseurs plus évolués (par exemple, contrôle de cohérence de la conception ou analyse du code source) ;
  - la *preuve formelle* qui permet de valider la conception ou le code source d'un programme par rapport à une spécification formelle de ce programme [Gaudel 1993, Rushby 1993] ;
  - l'*analyse de comportement*, basée sur des modèles comportementaux du logiciel (déduits de la spécification, de la conception ou du code source) et qui permet de vérifier des propriétés de cohérence, de complétude, etc. [Davis 1988, Harel et al. 1990].
- la **vérification dynamique** (avec exécution) comprend les deux techniques suivantes :
  - l'*exécution symbolique* qui consiste à activer le programme avec des entrées symboliques [DeMillo et al. 1987] ;

D'autres stratégies ont été développées par Zeil et Morell ; elles sont plus analytiques. Zeil s'intéresse à des classes de fautes affectant des expressions arithmétiques et relationnelles dans des programmes séquentiels [Zeil 1983, Zeil 1984, Zeil 1989] et établit les conditions sous lesquelles chacune de ces fautes génère des erreurs. Morell propose une méthode combinant l'utilisation du test de mutation et l'exécution symbolique [Morell 1988] : cette méthode, appelée test symbolique ("symbolic testing") permet de démontrer l'absence de fautes pré-spécifiées dans un programme en se basant sur l'analyse des effets potentiels des fautes sur le programme si celles-ci y étaient présentes.

Il est intéressant de constater que la plupart de ces méthodes ont conduit leurs auteurs à s'intéresser, dans un premier temps, au comportement erroné d'un programme en cours d'exécution afin de pouvoir dériver des conditions (ou contraintes) permettant d'élaborer un jeu de test ; les jeux de test ainsi conçus conduisent un programme, contenant une faute, à défaillance. A cette occasion, des modèles de faute et de défaillance ont été proposés [Morell 1984, Richardson & Thompson 1988] utilisant les concepts d'activation de faute, de création et de propagation d'erreurs.

A notre connaissance, les méthodes de tests basés sur les fautes n'ont pas été utilisées en milieu industriel. Deux raisons peuvent être mentionnées :

- la controverse sur la représentativité des fautes modélisées ou injectées vis-à-vis de fautes réelles,
- la complexité de l'analyse du programme qu'implique l'élaboration des conditions ou des contraintes permettant de définir un jeu de test ; cette analyse étant en outre peu automatisée.

I.3 VALIDATION DES TESTS DU LOGICIEL

Le problème principal du test se pose sous la forme suivante : quelle confiance peut-on accorder aux tests par rapport à la correction d'un programme ? Une stratégie de test idéale devrait permettre de garantir suite à l'exécution des jeux de tests et à la correction des fautes révélées par les tests, l'absence de fautes résiduelles dans le programme. Comme nous l'avons dit précédemment, une telle stratégie n'existe pas et

définitions des variables, et les nœuds (ou les arcs) référençant ces mêmes variables dans des calculs (ou des prédicats) [Rapps & Weyuker 1985] ; on peut citer, par exemple, les critères "toutes les définitions", "toutes les C-utilisations", "toutes les P-utilisations".

Le test fonctionnel repose sur un modèle décrivant le comportement attendu du programme, mais contrairement au test structurel, il n'existe pas de modèle "standard". Les modèles les plus connus sont : les classes d'équivalence, les tables de décision et les machines à états finis. Pour les classes d'équivalence, le critère de test consiste à sélectionner un élément par classe spécifiée. Pour une table de décision, le test devra assurer l'activation de chacune des règles. Pour une machine à états finis, les critères peuvent cibler la couverture des états, des transitions, voire des séquences de transitions. Enfin, si l'on dispose d'une spécification formelle, on peut s'appuyer sur ce modèle pour construire des entrées de test fonctionnel [Dauchy et al. 1993].

Pour un critère de sélection donné (structurel ou fonctionnel), les jeux de test peuvent être générés selon deux approches :

1) **déterministe** : les jeux de test sont déterminés par un choix sélectif de manière à satisfaire le critère retenu ;
2) **statistique** : les jeux de test sont générés selon une distribution probabiliste du domaine d'entrée, la distribution et le nombre d'entrées étant déterminés en fonction du critère retenu [Thévenod-Fosse 1991, Waeselynck 1993, Mazuet 1994].

I.2.3.2 Test basé sur un modèle de faute

Le **test basé sur des fautes** [DeMillo et al. 1978, Zeil 1983, Morell 1984, Howden 1987, Morell 1990] constitue une autre approche de sélection des entrées de test. L'objectif est alors de construire des jeux de test destinés à révéler des classes de fautes spécifiques. La plus connue de ces classes correspond aux fautes de type mutation : une mutation est une modification syntaxique élémentaire introduite dans le code source d'un programme. Le test de mutation [DeMillo & Offut 1991, DeMillo & Offutt 1993] est une technique de génération automatique de jeux de test permettant de révéler les mutations injectées dans un programme ; elle est basée sur l'utilisation de contraintes algébriques et est supportée par l'outil Mothra (qui permet de générer les mutations et d'exécuter les programmes incorrects ou mutants avec les jeux de test) et par l'outil Godzilla (qui permet de générer automatiquement les jeux de test).

propriétés, de nombreuses études analytiques [Frankl & Weyuker 1991, Frankl & Weyuker 1993a, Frankl & Weyuker 1993b], expérimentales [Girgis & Woodward 1986, Frankl & Weiss 1991] ou basées sur la simulation [Hamlet & Taylor 1988] ont eu pour objectif de comparer différents critères de test vis-à-vis soit de leur coût, soit de leur facilité de mise en œuvre, soit de leur capacité à révéler des fautes. Les résultats de ces comparaisons sont souvent contradictoires et sujets à polémiques [Hamlet 1989, DeMillo et al. 1995]. Ces critiques portent sur le critère de comparaison lui-même (par exemple, relation d'inclusion des critères de test), ou sur les hypothèses expérimentales (nombre et nature des programmes, nombre de jeux de tests, ...). Il paraît ainsi difficile de démontrer la supériorité d'un critère de test par rapport à un autre dans l'absolu, c'est-à-dire indépendamment des programmes et des fautes qui les affectent réellement.

L'un des problèmes posés par les critères de test est qu'ils ne sont pas liés directement aux fautes réelles dans le cas de tests basés sur un modèle du logiciel ou que le modèle de faute servant de base à la sélection reste incomplet et imparfait ; la validité d'un critère ne peut donc être démontrée.

L'autre problème posé par les critères de test est qu'ils ne permettent pas de sélectionner des entrées homogènes quant à leur capacité à révéler les fautes : ils ne sont pas fiables selon la définition donnée par Goodenough et Gerhart. Ainsi par exemple, le critère "tous les chemins" (considéré comme le critère structurel le plus sévère mais souvent impossible à satisfaire) ne permet pas de garantir la détection des erreurs sur un chemin donné [Rapps & Weyuker 1985], quelle que soit l'entrée sélectionnée satisfaisant l'exécution de ce chemin. Supposons qu'un chemin C contienne une faute F, la faute F est révélée si l'entrée sélectionnée pour exécuter C conduit à l'activation de F, à la création et à la propagation d'erreurs suite à cette activation et à l'occurrence d'une défaillance (une ou plusieurs sorties incorrectes). Or toutes les entrées permettant d'exécuter C ne remplissent pas forcément ces conditions. Cette notion de non-fiabilité d'un critère a été constatée expérimentalement à plusieurs reprises [Rapps & Weyuker 1985, Thévenod-Fosse et al. 1991, Waeselynck 1993] .

La non-fiabilité d'un critère peut être compensée en augmentant, dans un jeu de test, le nombre d'entrées sélectionnées selon un critère donné, c'est-à-dire en activant plusieurs fois un élément sélectionné par le critère. C'est le principe du test statistique [Waeselynck 1993, Mazuet 1994] ; cette technique permet de compenser, en partie, l'imperfection des critères de test (vis-à-vis de la non-fiabilité) sans toutefois constituer une méthode parfaite vis-à-vis de la recherche de fautes.

n'existera probablement jamais : elle nécessiterait des jeux de test exhaustifs[5] par rapport à un modèle parfait et complet de toutes les fautes, modèle qui n'existe pas et qui, s'il existait, serait sûrement d'une complexité incompatible avec un test exhaustif.

Nous présentons, dans un premier temps, la difficulté d'évaluer de manière objective et adéquate l'efficacité des critères de tests. Puis, nous soulignons les faiblesses des critères de sélection vis-à-vis de la recherche de fautes. Nous avons vu au paragraphe I.2.3 que ces critères sont utilisés pour construire **a priori** des jeux de test. Ils peuvent également être utilisés pour valider, **a posteriori**, des jeux de test qui ont été conçus selon des stratégies mixtes de sélection et qui ne sont pas exhaustifs vis-à-vis d'un critère de test donné. Cela paraît être la pratique la plus courante dans l'industrie.

Nous présentons, dans un deuxième temps, les techniques de validation des tests du logiciel. L'évaluation de la complétude d'un jeu de test par rapport à un critère basé sur un modèle du programme est en général réalisée à l'aide d'**analyses de couverture structurelle ou fonctionnelle** ; elles sont utilisées fréquemment dans le cas de développement de logiciels critiques. Ces analyses de couverture s'avèrent nécessaires mais non suffisantes pour valider des jeux de tests. Les techniques d'**injection de fautes** qui permettent d'évaluer la complétude d'un jeu de test par rapport à un modèle de faute (même imparfait) constituent, à notre avis, une approche complémentaire digne d'intérêt.

I.3.1 Imperfection des critères de test

Selon Goodenough et Gerhart [Goodenough & Gerhart 1975], un critère de test est idéal s'il est à la fois valide et fiable. Un critère est **valide** si et seulement si, pour toute faute dans un programme, il existe au moins un jeu de test satisfaisant ce critère qui soit capable de révéler la faute. Un critère est **fiable** si tous les jeux de test satisfaisant ce critère sont consistants dans leur capacité à révéler des fautes (tous les révèlent ou aucun ne les révèle). Il est cependant difficile — voire impossible — de démontrer la validité et la fiabilité d'un critère. A défaut de pouvoir démontrer ces

---

5 Dans le cadre de tests fonctionnels basés sur les spécifications formelles d'un programme, la notion de test exhaustif par rapport à tous les comportements du programme a été théoriquement introduite par M.-C. Gaudel ; le test exhaustif ainsi défini s'avère pratiquement impossible à mettre en œuvre et conduit à formuler des hypothèses sur le comportement du programme pour sélectionner un sous-ensemble fini d'entrées de test [Gaudel 1995].

la détection d'erreurs. C'est l'un des objectifs des techniques d'injection de fautes présentées ci-après.

I.3.3 Injection de fautes ou d'erreurs

L'injection de fautes est une technique répandue dans le cadre de la validation des mécanismes de tolérance aux fautes physiques. A ce titre, elle constitue une méthode expérimentale de validation de ces mécanismes [Arlat 1990] par introduction délibérée et contrôlée de fautes dans un système et par observation du comportement de ce système. La plupart des travaux sur l'injection de fautes ont porté sur la prévision des fautes — c'est-à-dire l'évaluation de l'efficacité des mécanismes de tolérance aux fautes, en termes de couverture des fautes tolérées — plutôt que sur l'élimination des fautes. L'injection de fautes peut se faire à différents niveaux d'abstraction du système : sur un prototype du système à analyser ou sur un modèle de simulation [Jenn 1994]. Les fautes peuvent être injectées soit directement sur les composants matériels (altérations physiques ou électriques), soit au niveau logiciel par altération des variables booléennes ou du contenu des mémoires (on cherche alors à simuler l'effet de fautes matérielles sur le système). Les outils d'injection de fautes matérielles par logiciel n'agissent pas sur le code source du logiciel mais plutôt sur la mémoire, sur les registres ou sur les bus de communication ; ils injectent alors des erreurs logicielles.

La technique d'**injection d'erreurs logicielles** est également utilisée dans le cadre de la validation des mécanismes de tolérance aux fautes logicielles. Elle consiste à injecter directement des erreurs au cours de l'exécution du logiciel, et permet ainsi de ne porter l'attention que sur les effets des fautes. Elle est utilisée pour valider des mécanismes de détection d'erreurs en ligne tels que les assertions exécutables [Rabéjac 1995]. Elle permet également d'étudier le comportement du programme lors d'une exécution et en particulier de mesurer la capacité d'un test à propager des erreurs en sortie et à les détecter (observabilité) [Morell & Murrill 1996] ou d'étudier la sensibilité d'une sortie à une erreur sur une variable donnée (mesure d'impact) [Goradia 1993]. Elle ne peut cependant être utilisée pour mesurer l'efficacité de jeux de tests, puisqu'elle occulte l'étape d'activation des fautes (commandabilité).

En ce qui concerne le test du logiciel, à notre connaissance, la seule technique qui existe aujourd'hui est l'injection de mutations. Lorsqu'elle est utilisée pour construire, a priori, des jeux de test, cette technique s'appelle le **test de mutation** (cf. § I.2.3.2).

I.3.2 Analyses de couverture

Les critères de test basés sur un modèle du logiciel peuvent être utilisés, a posteriori, pour évaluer la **couverture des tests**. Dans le cadre de développement de logiciels critiques, deux types d'analyse de couverture des tests peuvent être exigés [RTCA-EUROCAE 1992] :

1) l'*analyse de la couverture des spécifications* du logiciel qui a pour objectif de vérifier la complétude des tests fonctionnels par rapport aux éléments de la spécification du logiciel ;
2) l'*analyse de la couverture structurelle* qui a pour objectif de déterminer quelles parties du code n'ont pas été exécutées au cours des tests.

Différents critères de couverture peuvent être exigés en fonction du niveau de criticité du logiciel. Les critères utilisés pour la couverture structurelle sont les mêmes que les critères de sélection des jeux de tests cités au paragraphe précédent : ils sont basés sur le graphe de contrôle ou le flot de données (critères "toutes les instructions", "toutes les branches", "toutes les conditions/décisions", etc.).

Ces analyses de couverture permettent d'évaluer dans quelle mesure[6], les tests mettent en œuvre, d'une part, chaque élément structurel du code selon les critères spécifiés et, d'autre part, chaque élément fonctionnel avec des entrées normales sélectionnées dans des plages de variations prévues mais également avec des entrées anormales et dans des conditions anormales (tests dans des plages de variation normales et tests de robustesse). Ces analyses sont indispensables ; en effet, si une fonction ou un élément structurel d'un logiciel n'est pas couvert par les jeux de test, le développeur n'a alors aucune information sur la qualité de cette fonction ou de cet élément structurel. Mais elles ne sont pas suffisantes pour valider des jeux de test : le problème de la confiance que l'on peut accorder à ces analyses de couverture, rejoint celui lié à l'imperfection des critères de test et exposé au paragraphe précédent.

Une façon d'augmenter la confiance que l'on peut avoir dans des jeux de tests est d'utiliser une technique permettant d'évaluer l'efficacité de ces jeux de test par rapport à

---

6 On établit alors une mesure de couverture (par exemple, 95 % des branches ont été testées). Si cette mesure ne satisfait pas les exigences spécifiées (critères de couverture), des démonstrations ou des vérifications complémentaires sont alors nécessaires.

notée *sm(P,T)* ; elle correspond à la proportion de mutants non équivalents à P qui sont tués par T. Un score de 100% indique que le jeu T tue tous les mutants non équivalents.

Afin de pallier le problème du nombre restreint de classes de fautes à injecter, l'analyse de mutation repose sur **deux hypothèses** [DeMillo et al. 1988] :

a) l'hypothèse du *programmeur compétent* : "Le programme testé a été écrit par un programmeur compétent. Par conséquent, si le programme est incorrect, il ne diffère du programme correct que par quelques fautes élémentaires".

b) l'hypothèse dite d'*effet de couplage* : "Les fautes complexes sont couplées à des fautes simples. En d'autres termes, des entrées de test qui tuent tous les programmes qui diffèrent du programme original par seulement une faute simple[8], sont suffisamment sensibles pour révéler des fautes plus complexes".

Sous ces hypothèses, le score de mutation peut être considéré comme une mesure représentative de l'efficacité d'un jeu de test à révéler des fautes.

Le premier outil expérimental d'analyse de mutation [DeMillo et al. 1978, DeMillo et al. 1988], appelé Mothra, a été construit pour des programmes en Fortran ; il aide les développeurs à valider leurs tests et à les compléter de manière à révéler toutes les mutations. Le score de mutation sert alors de critère d'arrêt des tests et de mesure de l'efficacité des tests générés. Mothra génère automatiquement des mutations à partir de 22 opérateurs de mutation [Offutt et al. 1996] listés dans le tableau I.1 ; ces opérateurs permettent de décrire la modification syntaxique apportée au programme original.

Dans le cadre de nos expérimentations, nous avons utilisé l'outil SESAME, développé au LAAS-CNRS, par Yves Crouzet [Crouzet 1995] et présenté au chapitre II : nous verrons qu'il utilise une autre méthode pour générer les mutations, basée sur la construction de tables de mutations.

L'analyse de mutation a suscité de nombreuses critiques en raison de son coût et du problème de la représentativité des fautes injectées par rapport aux fautes réelles,

---

8 Une faute est dite "simple" lorsqu'elle peut être corrigée en apportant une seule modification à une instruction du code source. Une faute est dite "complexe" dans le cas contraire. Les mutations sont considérées comme des fautes simples. Si on injecte plusieurs mutations sur une même version d'un programme, on génère alors des mutants complexes (ou d'ordre supérieur à 1 contrairement aux mutations classiques) [Offutt 1989, Offutt 1992].

Lorsqu'elle est utilisée pour valider, a posteriori, des jeux de test, elle est dénommée **analyse de mutation**.

Nous développons au paragraphe I.4, l'état de l'art en matière d'analyse de mutation en exposant ses principes, les utilisations qui en ont été faites et les critiques que cette méthode a suscitées.

I.4 ANALYSE DE MUTATION

I.4.1 Principe

Définie initialement par DeMillo [DeMillo et al. 1978], l'analyse de mutation consiste dans un premier temps à créer, à partir d'un programme original P, un ensemble de programmes P_i — appelés **mutants** — qui diffèrent de P par une et une seule modification élémentaire — syntaxiquement correcte — introduite dans le code source de P. Cette modification est appelée **mutation**. On peut distinguer plusieurs catégories de mutations : remplacement d'un opérateur par un autre opérateur, d'une constante par une autre constante, d'un symbole (nom d'une variable ou d'un tableau, ...) par un autre symbole, etc. (voir par exemple [DeMillo et al. 1988, Thévenod-Fosse et al. 1991]).

Partant d'un programme P, des mutants associés et d'un jeu de test T, l'analyse de mutation consiste dans un deuxième temps à évaluer la proportion des mutations révélées par T. Une mutation m_i est révélée par T si le mutant correspondant P_i fournit au moins une valeur de sortie différente de celle produite par le programme original P en réponse au jeu T : le mutant P_i est alors dit **tué** par T. Dans le cas contraire, c'est-à-dire lorsqu'il fournit les même sorties que le programme original P lors de l'exécution de T, P_i est dit vivant et deux cas sont alors possibles : 1) soit le jeu de test T n'est pas assez sensible[7] pour révéler la mutation m_i, 2) soit le mutant P_i est équivalent au programme P, c'est-à-dire qu'il n'existe aucune entrée de test permettant de les distinguer.

La mesure permettant de mesurer l'efficacité d'un jeu de test à révéler des fautes de type mutation est le **score de mutation**. Pour un programme P et un jeu T, elle est

---

7 Il existe une ou plusieurs entrées de test permettant de révéler m_i mais le jeu de test T n'en contient aucune.

Dans [Voas et al. 1992, Voas & Miller 1995], le comportement du programme est étudié suite à l'injection de mutations : Voas, Morell et Miller analysent les effets des mutations (activation, création d'erreur et propagation jusqu'à la sortie du programme) à l'aide d'une méthode appelée PIE (Propagation - Infection - Exécution). Cette analyse permet d'identifier les parties du code où des fautes si elles existent seront difficiles à révéler. De manière similaire, Morell et Murrill ont étudié le comportement erroné d'un programme suite à injection de mutations. Ils ont pour cela développé un outil (DEFA) qui produit les traces d'erreurs automatiquement [Morell & Murrill 1994]. Les informations recueillies sur chaque chemin d'exécution leur permet ensuite de sélectionner les chemins les plus efficaces pour le test du programme analysé.

I.4.2 Variantes de l'analyse de mutation

Des variantes de l'analyse de mutation ont été développées de manière à réduire son coût. Une première variante de l'analyse de mutation appelée analyse de mutation **faible** a été introduite dans [Howden 1982]. Elle se distingue de l'approche classique dénommée analyse de mutation **forte**, par le fait que pour un mutant P_i obtenu suite à la modification d'un composant élémentaire[9] C du programme original P, le jeu de test doit être seulement capable d'activer le composant muté C_i, de façon à produire un résultat intermédiaire erroné (valeur erronée) suite à l'exécution du composant C_i. L'avantage de l'analyse de mutation faible est son faible coût par rapport à l'analyse de mutation forte car plusieurs mutations peuvent être introduites lors d'une seule exécution de P. Cependant son inconvénient majeur est qu'elle ne mesure que la capacité d'un test à activer une faute et à créer une erreur (commandabilité) ; elle ne mesure donc pas sa capacité à propager les erreurs créées jusqu'à la sortie (observabilité).

Cela a amené d'autres auteurs à proposer une méthode intermédiaire, appelée "firm mutation" [Woodward & Halewood 1988], qui consiste à comparer les états intermédiaires du programme original P et d'un mutant P_i. Quatre types de comparaison d'états intermédiaires ont été mis en œuvre par Offut et par Lee [Offutt et Lee 1991, Offutt et Lee 1994], conduisant à la création de quatre variantes de la "firm mutation". Ces variantes consistent à comparer des états correct et erroné a) suite à la première exécution (dans P et P_i) de l'expression sur laquelle a été injectée la mutation, b) suite à l'exécution de l'instruction contenant le composant muté, c) suite à la première exécution

---

9 Un composant élémentaire peut être une définition de variable, une référence à une variable, une expression arithmétique, une expression relationnelle ou une expression booléenne.

c'est-à-dire de la validité des deux hypothèses sous-jacentes, citées précédemment. Cela a conduit à remettre en cause la validité du score de mutation en tant que mesure de l'efficacité d'un jeu de test. L'analyse de mutation n'a pourtant pas été abandonnée en tant qu'outil expérimental et a donné lieu à d'autres types d'utilisation tels que ceux présentés ci-après.

L'analyse de mutation a été, par exemple, utilisée dans le but de comparer l'efficacité relative de différentes stratégies de test basées sur des critères de sélection différents ou sur des méthodes de génération de test différentes [Ntafos 1984, Girgis & Woodward 1986, Waeselynck 1993, Mazuet 1994, Thévenod-Fosse & Crouzet 1995].

Opérateur de mutation	Description
AAR	Modification d'une référence à un tableau
ABS	Insertion d'une valeur absolue
ACR	Remplacement d'une référence à un tableau par une constante
AOR	Modification d'un opérateur arithmétique
ASR	Remplacement d'une référence à un tableau par une variable scalaire
CAR	Remplacement d'une constante par une référence à un tableau
CNR	Modification d'un nom de tableau
CRP	Modification d'une constante
CSR	Remplacement d'une constante par une variable scalaire
DER	Remplacement d'une instruction DO
DSA	Altération d'une instruction DATA
GLR	Remplacement d'une étiquette GO TO
LCR	Modification d'un opérateur logique
ROR	Modification d'un opérateur relationnel
RSR	Remplacement d'une instruction Return
SAN	Remplacement d'une instruction par une instruction TRAP
SAR	Remplacement d'une variable scalaire par une constante
SCR	Remplacement d'une variable scalaire par une référence à un tableau
SDL	Suppression d'une instruction
SRC	Modification d'une constante source
SVR	Modification d'une variable scalaire
UOI	Insertion d'un opérateur unaire

Tableau I.1 : Opérateurs de mutation de Mothra

mutations par rapport à ceux générés par des fautes réelles connues, nous serons alors en mesure de proposer des applications significatives de l'analyse de mutation (exploitation du score de mutation et diminution du coût de l'analyse).

Le chapitre suivant permet de définir le cadre expérimental de nos travaux et de présenter, en particulier, les programmes analysés, les types de fautes traitées, les types d'erreurs et de comportements erronés que l'on va analyser.

du bloc d'instructions contenant le composant muté et d) suite à chaque exécution du bloc d'instructions contenant le composant muté. Les expériences d'Offutt et de Lee ont consisté à tester 11 petits programmes selon ces différentes variantes ; c'est-à-dire à utiliser comme critère d'arrêt des tests, le score de 100% de mutations révélées pour chacune des variantes. Ils ont ensuite validé, par analyse de mutation forte, les jeux de test obtenus et comparé les scores obtenus vis-à-vis de la mutation forte (inférieurs alors à 100%) ainsi que le coût de l'analyse (en termes de nombre d'exécutions et de nombre de vecteurs de test nécessaires pour atteindre le score de 100%). Les résultats expérimentaux indiquent que les variantes présentant les meilleurs résultats sont les variantes b) ou c) mais que pour les applications critiques, il est préférable de valider les jeux de test par analyse de mutation forte ou par une combinaison des méthodes b) et c).

I.5 CONCLUSION

Au cours de ce chapitre, nous avons présenté le cadre dans lequel s'insèrent nos travaux. Nous avons vu qu'en matière de vérification de logiciel, l'absence d'un modèle de faute complet et représentatif de la réalité pose le problème de la confiance que l'on peut accorder aux tests par rapport à la recherche de fautes dans un programme. Nous avons montré également que les analyses de couvertures structurelles et fonctionnelles ne permettaient pas de garantir l'absence de fautes. Dans cet objectif, il nous semble alors nécessaire d'étudier le comportement des tests vis-à-vis de fautes, c'est-à-dire de mesurer leur capacité à révéler des fautes. L'analyse de mutation nous paraît être une approche pragmatique pour injecter un grand nombre de fautes logicielles.

Le problème reste qu'il semble impossible d'évaluer la représentativité des mutations vis-à-vis des fautes réelles. Cependant, il nous paraît intéressant d'étudier les comportements erronés créés par ces mutations ainsi que leurs similitudes ou leurs différences par rapport à ceux produits par des fautes réelles. En effet, lors de précédents travaux sur le test [Waeselynck 1993, DeMillo & Mathur 1994], les auteurs ont pu constater que les mutations pouvaient engendrer des comportements erronés aussi difficiles à révéler que ceux dus à certaines fautes réelles.

Une grande partie de nos travaux consiste, suite à des études expérimentales, à modéliser les comportements erronés dus à différents types de fautes (réelles ou artificielles) et à évaluer la représentativité des erreurs générées par des mutations par rapport à des erreurs dues à des fautes réelles. Si les résultats obtenus sont en faveur d'une bonne représentativité des erreurs et des comportements erronés générés par les

CHAPITRE II

PROBLÉMATIQUE ET CADRE EXPÉRIMENTAL

II.1 INTRODUCTION

Ce chapitre décrit le cadre méthodologique de nos recherches. Dans nos travaux, tant expérimentaux que théoriques, nous nous sommes intéressée à l'analyse détaillée d'erreurs produites par, d'une part des fautes réelles qui avaient été identifiées lors d'activités de vérification, dans des logiciels du domaine nucléaire, et d'autre part des fautes de type mutation. Notre objectif a été dans un premier temps, de comparer les comportements erronés produits par ces fautes afin d'analyser la représentativité de ceux produits par des mutations. Puis nous nous sommes attachée à l'analyse et à la modélisation des mécanismes parfois très complexes de propagation d'erreur(s).

La première partie de ce chapitre va nous permettre de préciser le niveau de détail auquel se sont situées nos analyses : nous définissons le formalisme retenu pour représenter l'état interne d'un programme en cours d'exécution ; ce formalisme nous permet de présenter les types d'erreurs logicielles qui ont fait l'objet de nos études, ainsi que certaines évolutions possibles au cours de l'exécution du programme et leur impact sur l'état interne de ce programme. Il s'agit ici d'une présentation simplifiée qui vise à sensibiliser le lecteur à la complexité des mécanismes de propagation d'erreurs(s) qui seront analysés et largement détaillés dans la suite de ce mémoire.

La deuxième partie de ce chapitre décrit le cadre expérimental de nos travaux, notamment : les programmes sur lesquels ont porté les expériences, les fautes réelles qui ont été analysées, l'environnement d'analyse de mutation, le principe et la mise en œuvre de nos campagnes expérimentales.

d'une instruction contenant une faute et identifiables après l'exécution de cette instruction, sont appelées **erreurs initiales**.

Les notions d'état interne d'un programme et d'état de donnée (correct et erroné), telles que définies ci-dessus, sont similaires à celles proposées par Murrill [Murrill 1991] et par Voas et Miller [Voas & Miller 1992]. Les deux grands types d'erreur que nous avons distingués correspondent à ceux précédemment définis par Howden [Howden 1991] et par Zeil [Zeil 1984, Zeil 1989]. La distinction entre les erreurs initiales et les erreurs produites par propagation a également été introduite dans d'autres travaux [Thompson 1991, Lee et Iyer 1993, Laski et al. 1995].

II.2.2 Propagation d'une erreur

Considérons deux versions d'un programme, l'une (supposée) correcte et l'autre contenant au moins une faute. Soient PSc_i l'état interne du programme correct à l'instant i et PSe_i celui du programme incorrect au même instant. Prenons un exemple simple, en supposant que PSe_i contient une seule erreur sur le flot des données, c'est-à-dire :

$$PSc_i = \{(V_1, v_1), ..., (V_j, v_j), ..., (V_n, v_n), (PC, x)\}$$
$$PSe_i = \{(V_1, v_1), ..., (V_j, w_j), ..., (V_n, v_n), (PC, x)\}$$

où (V_j, w_j) est l'état de donnée erroné ($w_j \neq v_j$). L'exécution de l'instruction suivante va modifier les états internes des deux programmes, et en particulier l'erreur $\{v_j, w_j\}$ peut ou non propager. Ceci nous conduit à distinguer trois types de propagation, détaillés ci-après : création, annulation, ou masquage d'erreur. Les notations v_j, v_j', w_j et w_j', utilisées dans cet exemple, désignent des valeurs correctes ou erronées, associées à une même variable V_j.

Création d'erreur :

- sur la même variable V_j (précédemment erronée) :

  $PSc_{i+1} = \{ ..., (V_j, v_j'), ...\}$
  $PSe_{i+1} = \{ ..., (V_j, w_j'), ...\}$

  avec $w_j' \neq w_j$, et $w_j' \neq v_j'$ (v_j' peut être égal à v_j)
- sur une autre variable V_k (précédemment correcte) :

  $PSc_{i+1} = \{..., (V_j, v_j), ..., (V_k, v_k'),...\}$
  $PSe_{i+1} = \{..., (V_j, w_j), ..., (V_k, w_k),...\}$

problème au chapitre IV). Ce type d'erreur peut être créé par l'évaluation incorrecte d'une condition de branchement lors de l'exécution d'une instruction conditionnelle de

II.2 PROBLÉMATIQUE

II.2.1 Etat interne d'un programme et types d'erreur

L'état interne d'un programme à un instant donné de son exécution est défini par l'ensemble des variables du programme et de leur valeur (définie ou indéfinie) à cet instant de l'exécution ainsi que par la valeur du compteur de programme, noté PC, indiquant la prochaine instruction à exécuter. Pour un programme utilisant n variables, l'état interne à un instant donné i de son exécution est représenté de la façon suivante :

$$PS_i = \{(var_1, val_1), ..., (var_n, val_n), (PC, x)\}$$

où val_j ($j = 1, ..., n$) représente la valeur courante de la variable var_j et x désigne la valeur du compteur de programme PC. Chaque couple (var_j, val_j) ou (PC, x) représente **l'état d'une donnée** du programme.

Dans nos travaux, nous considérons l'exécution d'une instruction comme étant atomique, c'est-à-dire que l'état interne d'un programme n'est observable qu'avant et après l'exécution d'une instruction. Dans un cycle d'exécution, qui correspond à une exécution complète du programme en réponse à un vecteur d'entrée, l'unité de temps est donc l'exécution d'une instruction. Par convention, l'**instant d'exécution i** ($i \geq 0$) sera associé à la fin de la ième instruction exécutée durant le cycle, l'instant 0 étant celui qui précède l'exécution de la première instruction.

L'état interne d'un programme est incorrect s'il contient au moins un **état de donnée erroné**. Un état de donnée (var_j, val_j') ou (PC, x') est erroné par comparaison au couple (var_j, val_j) ou (PC, x) présent au même instant dans l'état interne du programme correct si $val_j' \neq val_j$ ou $x' \neq x$. Une **erreur** est donc caractérisée par un couple $\{val_j, val_j'\}$ associé à une variable var_j, ou par un couple $\{x, x'\}$ associé au compteur de programme. On peut alors distinguer deux grands types d'erreur :

1. *Erreur sur le flot de données*, lorsqu'un état de donnée (var_j, val_j') est erroné ;
2. *Erreur sur le flot de contrôle*, ou branchement erroné, lorsque l'état de donnée (PC, x') est erroné.

Afin de distinguer l'étape d'activation d'une faute de l'étape de propagation d'erreur, les erreurs produites par activation d'une faute, c'est-à-dire lors de l'exécution

La figure II.1, que nous commentons ci-après, récapitule les évolutions possibles de l'état interne d'un programme incorrect à l'issue de chaque cycle d'exécution, en réponse à une séquence d'entrée $T = (t_1, ..., t_p)$ composée de plusieurs vecteurs t_k ($p>1$).

II.2.3.1 Comportements observables sur un seul cycle d'exécution

Lorsqu'on analyse le comportement d'un programme sur **un seul cycle d'exécution** en supposant l'état interne correct avant le début de l'exécution, seules les transitions tr_1, tr_2 et tr_3 de la figure II.1 sont observables ; elles correspondent aux cas suivants [Voas et al. 1992] :

Cas 1 La faute n'est pas activée durant le cycle d'exécution : l'état interne du programme reste *correct* (tr_1) ;

Cas 2 La faute est activée durant le cycle d'exécution mais aucune erreur n'est créée : l'état interne du programme reste *correct* (tr_1) ;

Cas 3 La faute est activée durant le cycle d'exécution, créant au moins une erreur initiale, mais les résultats en sortie restent corrects. L'état interne du programme est alors :

a. soit *correct* : toutes les erreurs créées suite à l'activation de la faute ont été annulées avant la fin du cycle (tr_1) ;

b. soit *incorrect et non-défaillant* : les erreurs créées et non annulées n'affectent pas de variable de sortie (tr_2) ;

Cas 4 La faute est activée durant le cycle d'exécution, créant au moins une erreur initiale dont la propagation a permis de créer une erreur affectant au moins une variable de sortie : l'état interne du programme est alors *défaillant* et la faute est révélée (tr_3).

A notre connaissance, toutes les études antérieures à nos travaux se sont concentrées sur l'analyse des comportements erronés observables sur un seul cycle d'exécution.

II.2.3.2 Comportements observables sur plusieurs cycles d'exécution

Les transitions tr_4 à tr_9 de la figure II.1, ombrées sur le graphe, ne sont observables que lorsque l'analyse du programme porte sur **plusieurs cycles d'exécution**. Dans ce cas, les états de donnée incorrects à la fin d'un cycle le restent

avec $w_k \neq v_k'$, et v_k' peut être égal à v_k

- sur le compteur de programme PC (précédemment correct) :

$PSc_{i+1} = \{..., (V_j, v_j), ..., (PC, x')\}$
$PSe_{i+1} = \{..., (V_j, w_j), ..., (PC, y')\}$

avec $x \neq x' \neq y'$

Annulation d'erreur :

- par modification de l'état de donnée erroné dans le programme incorrect :

$PSc_{i+1} = \{..., (V_j, v_j), ...\}$
$PSe_{i+1} = \{..., (V_j, v_j), ...\}$

- par modification de l'état de la donnée dans le programme correct :

$PSc_{i+1} = \{..., (V_j, w_j), ...\}$
$PSe_{i+1} = \{..., (V_j, w_j), ...\}$

- par modification des deux états de la donnée, correct et erroné :

$PSc_{i+1} = \{..., (V_j, v_j'), ...\}$
$PSc_{i+1} = \{..., (V_j, v_j'), ...\}$

Masquage d'erreur :

$PSc_{i+1} = \{..., (V_j, v_j), ...\}$
$PSe_{i+1} = \{..., (V_j, w_j), ...\}$

L'erreur $\{v_j, w_j\}$ associée à la variable V_j subsiste mais sans avoir provoqué la création d'autre(s) erreur(s).

Cet exemple nous a permis d'illustrer sur des cas simples les mécanismes de création, d'annulation et de masquage d'erreur, en traduisant certains de leurs effets sur l'évolution de l'état interne du programme.

En pratique, le comportement erroné d'un programme peut être très complexe à analyser car ces divers mécanismes peuvent apparaître de façon simultanée sur une ou plusieurs données, notamment suite à une erreur sur le flot de contrôle c'est-à-dire lorsque PC a une valeur incorrecte. Une erreur sur le flot de contrôle conduisant à exécuter une instruction à la place d'une autre, les deux programmes — correct et incorrect — n'exécutent plus simultanément les mêmes instructions : il est alors parfois difficile de synchroniser la comparaison de leurs états internes (nous reviendrons sur ce

plusieurs anciennes erreurs sans qu'il y ait eu activation de faute ; elle peut aussi être le résultat des propagations combinées de plusieurs erreurs anciennes et nouvelles ; une défaillance peut être observée au cours d'un cycle sans qu'il y ait eu activation de faute ; plusieurs erreurs, anciennes et nouvelles, peuvent propager et en annuler d'autres, etc. Ces processus complexes de création, masquage et annulation d'erreurs dans le cas d'un programme avec mémoire et sur plusieurs cycles d'exécution successifs, ont été observés dans le cadre de notre première série d'expérimentations décrites au chapitre III.

II.3 CADRE EXPÉRIMENTAL

La deuxième partie de ce chapitre a pour objet de décrire le cadre expérimental dans lequel nos expérimentations ont été menées. Il s'agit dans un premier temps de présenter les programmes étudiés ainsi que les fautes réelles révélées lors de la validation de ces programmes et analysées en détail au cours de nos travaux. Nous décrivons ensuite l'environnement d'analyse de mutation qui nous a fourni un large échantillon de fautes artificielles, et les jeux de test dont nous disposions pour étudier les comportements des programmes corrects et incorrects en cours d'exécution. Nous expliquons enfin comment nous avons procédé pour observer et analyser manuellement les comportements erronés générés par les fautes, qu'elles soient réelles ou artificielles.

II.3.1 Fonctionnalités des programmes étudiés

Les études expérimentales ont porté sur deux programmes, appelés ETUD et LOCALES, issus d'applications critiques du nucléaire qui ont été développées dans des environnements industriels différents. Ces deux programmes sont codés en langage C selon des règles de programmation limitant ou interdisant l'utilisation de certaines propriétés du C jugées incompatibles avec la criticité des applications développées, notamment : l'utilisation de pointeurs, l'utilisation de branchements inconditionnels (par exemple, "go to" ou "break"), la concision des expressions qui permet d'écrire l'équivalent de plusieurs instructions d'un langage évolué sous la forme d'une seule instruction en C (par exemple, la pré ou post-incrémentation combinée à une autre opération dans la même instruction).

différente de la valeur existante dans l'état interne du programme incorrect correspondant.

II.2.3 Comportement d'un programme incorrect

Nous nous sommes intéressée dans le paragraphe précédent à l'évolution de l'état interne d'un programme en choisissant comme unité de temps, l'exécution d'une instruction à l'intérieur d'un cycle d'exécution. Nous allons maintenant décrire les évolutions possibles de l'état interne d'un programme entre deux cycles d'exécution successifs.

Un cycle d'exécution correspond à une exécution complète du programme en réponse à un vecteur d'entrée t_k. A la fin d'un cycle d'exécution, l'état interne du programme peut être soit :

(i) *Correct* : tous les états de donnée sont corrects ;

(ii) *Incorrect et non-défaillant* : l'état interne du programme comporte au moins un état de donnée erroné mais aucune variable de sortie n'est affectée ;

(iii) *Défaillant* : l'état interne du programme comporte au moins un état de donnée erroné qui affecte une variable de sortie.

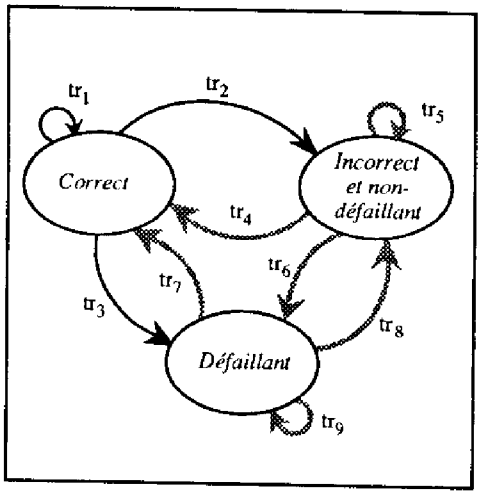


Figure II.1 : Evolutions de l'état interne d'un programme incorrect sur plusieurs cycles d'exécution

Le programme étudié est une version industrielle correspondant aux premières phases de développement. Les fautes réelles analysées ont été révélées au cours des étapes suivantes de vérification du logiciel : lecture croisée, tests unitaires, tests d'intégration. Nous disposons donc, pour ce module, de 4 versions différentes correspondant aux étapes successives de développement : a) la version 1.1 qui est la version initiale avant toute vérification, b) la version 1.2 comportant les corrections dues aux anomalies identifiées par relecture, c) la version 1.3 résultant de la phase des tests unitaires, d) la version 1.4 corrigée après les tests d'intégration.

Les deux fonctions composant le programme LOCALES comportent respectivement 240 et 130 lignes de code source hors commentaires dans la dernière version (version 1.4). Les études concernant ce programme ont été réalisées au niveau unitaire. Le comportement de chaque fonction a donc été analysé en détail à l'aide de tests basés sur la structure de chacune de ces fonctions.

II.3.2 Fautes réelles

L'échantillon de fautes réelles révélées sur chacune des deux applications est varié d'une part, par l'origine des fautes — mauvaise compréhension de la spécification, faute de conception, faute d'initialisation, faute de programmation — et d'autre part, par leur type c'est-à-dire les corrections induites sur le code source du programme.

Elles peuvent affecter de une à quinze instructions (d'affectation ou de contrôle). En effet, dans le cadre de nos études expérimentales, nous n'avons pas décomposé chaque faute réelle en fautes élémentaires c'est-à-dire n'affectant qu'un élément du programme (instruction ou donnée). Ces fautes peuvent conduire aux altérations suivantes dans le code source :

- instruction(s) incorrecte(s) : utilisation d'un (ou plusieurs) opérande(s) ou d'un (ou plusieurs) opérateur(s) incorrect(s) ;
- instruction(s) manquante(s) ;
- instruction(s) supplémentaire(s) ;
- instruction(s) déplacée(s) ;
- combinaison des cas précédents.

au début du cycle suivant. A l'issue du cycle suivant, les évolutions suivantes peuvent alors se produire :

tr_4 : tous les états de donnée sont corrects : les erreurs, anciennes (c'est-à-dire présentes à la fin du cycle précédent) ou nouvelles (créées durant le cycle courant), ont toutes été annulées avant la fin du cycle courant ; l'état interne du programme redevient correct ;

tr_5 : un ou plusieurs états de donnée sont incorrects : les erreurs peuvent être les anciennes erreurs (non annulées) ou des nouvelles erreurs, mais aucune d'entre elles n'affecte une variable de sortie ; une erreur peut ainsi être masquée, ou créer d'autres erreurs pendant plusieurs cycles avant l'occurrence d'une défaillance ;

tr_6 : un ou plusieurs états de donnée sont incorrects : les erreurs peuvent être les anciennes ou des nouvelles, et au moins une d'entre elles affecte une sortie ;

tr_7 : tous les erreurs, anciennes ou nouvelles, ont été annulées avant la fin du cycle courant ;

tr_8 : il subsiste une ou plusieurs erreurs anciennes ou nouvelles, mais aucune d'entre elles n'affecte une variable de sortie ; en particulier toutes les anciennes erreurs affectant les variables de sortie ont été annulées ;

tr_9 : des défaillances successives sont observées ; elles sont dues soit aux anciennes erreurs non annulées sur les variables de sortie, soit à de nouvelles erreurs affectant des variables de sortie (les mêmes ou d'autres).

Pour préciser les mécanismes de création d'une nouvelle erreur au cycle courant, il est nécessaire de faire la distinction entre les programmes *sans mémoire*, et les programmes *avec mémoire*. Un programme est dit *sans mémoire* lorsque, quel que soit le cycle d'exécution, toute variable utilisée est au préalable redéfinie dans le cycle. Dans ce cas, aucune ancienne erreur ne peut propager au cycle courant : toute création d'erreur est due à l'activation de la faute ou à la propagation d'une nouvelle erreur. Une ancienne erreur peut cependant subsister si elle affecte une variable non utilisée et non redéfinie au cycle courant, mais elle sera toujours masquée jusqu'à son annulation (par réinitialisation).

Dans le cas d'un programme *avec mémoire*, certaines variables internes ont pour rôle de mémoriser des résultats issus des cycles précédents dans un but de réutilisation (sans réinitialisation). Les mécanismes de création et de propagation d'erreurs deviennent plus complexes : une erreur peut être créée par propagation d'une ou

Faute	Description	Fonction affectée (version)	Impact sur le code source	# lignes affectées
F1	Test mal placé pour l'élaboration d'une variable de sortie	recopie_capteur (V1.1)	Une instruction de branchement incorrecte, une inst. déplacée et un bloc *else* de 4 inst. manquant	11
F2	Utilisation d'une constante erronée dans un test	recopie_capteur (V1.1)	Une instruction de branchement incorrecte	1
F3	Idem	recopie_capteur (V1.1)	Une instruction incorrecte	1
F4	Initialisation du diagnostic de calcul mal placée (dans le corps d'un *else*)	calcul_d_tsgv (V1.1)	Une instruction déplacée	4
F5	Test dynamique d'une variable de sortie manquant	calcul_d_tsgv (V1.1)	Un bloc de quatre instructions manquant (bloc if-then)	6
F6	Initialisation du diagnostic de calcul mal placée (dans le corps d'un *else*)	calcul_d_tsgv (V1.1)	Une instruction déplacée	4
F7	Test dynamique d'une variable de sortie manquant	calcul_d_tsgv (V1.1)	Quatre instructions manquantes (bloc if-then)	6
F8	Test erroné pour l'élaboration d'une variable de sortie	calcul_d_tsgv (V1.1)	Une instruction de branchement incorrecte	1
F9	Utilisation d'une constante erronée dans le calcul d'une variable	calcul_d_tsgv (V1.2)	Une instruction incorrecte	1
F10	Traitement d'inhibition de défauts des capteurs incorrect	recopie_capteur (V1.3)	Deux blocs de 4 inst. manquants et un bloc de 7 instructions supplémentaire	15
F11	Valeurs par défaut erronées	recopie_capteur (V1.3)	Quatre instructions incorrectes	4
F12	Gestion des erreurs fonctionnelles incorrecte	calcul_d_tsgv (V1.3)	Deux instructions incorrectes	2
F13	Idem	calcul_d_tsgv (V1.3)	Deux instructions incorrectes	2

Tableau II.2 : Caractéristiques des 13 fautes réelles dans le programme LOCALES

II.3.1.1 Programme ETUD

Les fonctionnalités du programme ETUD sont décrites dans [Waeselynck 1993]. Le programme ETUD est une version étudiante d'un composant logiciel destiné à la surveillance des barres d'arrêt d'urgence d'un réacteur nucléaire civil. A chaque cycle d'exécution, 19 positions de barre sont traitées : les informations (positions des barres en code GRAY et données de contrôle) sont lues sur cinq cartes d'interface 32 bits ; elles sont ensuite vérifiées et filtrées, et les mesures en code GRAY sont finalement converties en un nombre de pas mécaniques. Les fonctions de vérification et de filtrage confèrent à ce programme une caractéristique importante pour notre étude : c'est un programme *avec mémoire*. En effet, à chaque cycle d'exécution, les vérifications effectuées sur les positions de barres acquises au début du cycle courant dépendent du résultat des vérifications effectuées au cycle précédent, résultat mémorisé à l'aide de variables internes. Cet effet mémoire nous a permis d'étudier la propagation d'erreurs sur plusieurs cycles d'exécution avant détection.

Ce composant logiciel comprend environ mille lignes de code source hors commentaires. La version ETUD comporte 15 fonctions et ne traite que des variables de type entier représentant soit des nombres, soit des booléens.

Les fautes réelles étudiées ont été révélées lors d'études antérieures sur le test statistique du logiciel [Thévenod-Fosse & Waeselynck 1993, Waeselynck 1993].

II.3.1.2 Programme LOCALES

Le programme LOCALES étudié fait partie d'un calculateur de contrôle-commande de chaufferie nucléaire et plus particulièrement de l'application permettant d'élaborer des mesures à partir d'informations venant des capteurs. Il est composé de deux fonctions :

1. *recopie-capteur* : elle réalise l'acquisition des valeurs des capteurs et détermine les températures de sortie du cœur ainsi que les températures de sortie du générateur de vapeur à partir des grandeurs corrigées des capteurs ; elle calcule également les discordances entre températures.
2. *calcul_d_tsgv* : elle réalise des calculs concernant les températures de sortie du générateur de vapeur (température filtrée, température moyenne et écart entre ces deux températures).

Dans le cadre de nos expérimentations, seuls des programmes écrits en langage C ont été étudiés ; c'est pourquoi nous utilisons, dans ce manuscrit, la syntaxe du langage C pour décrire les mutations étudiées. En outre, certaines mutations analysées sont spécifiques à ce langage, les compilateurs C étant permissifs (sur le contrôle des types notamment).

II.3.3.1 Génération des mutants

Ce module permet à partir d'un code source et de tables de mutations, de générer des **bases de mutants compilables**. Ces bases sont ensuite exécutées avec différents jeux de test par le module d'exécution des mutants.

Les mutations générées sont des mutations du premier ordre : chaque mutation correspond à une et une seule modification syntaxique élémentaire du code source. Trois types de mutation peuvent être introduits dans le code source d'un programme.

1) **Mutations sur les valeurs numériques des constantes** : il s'agit de modifier la valeur numérique d'une constante en lui appliquant un opérateur arithmétique (addition, multiplication, ...), un opérateur de décalage binaire (à droite ou à gauche), ou un opérateur de logique binaire (complément à un, ou exclusif, ...).

   Exemples : l'instruction (i) peut être altérée de diverses manières, notamment par les mutations m1, m2 et m3.

   (i) if (C > 10) → (m1) if (C > 10 + 1)
   (m2) if (C > 10 >> 1)
   (m3) if (C > ~10)

2) **Mutations sur les opérateurs du langage** : il s'agit de remplacer l'opérateur utilisé dans une expression par un opérateur du même type ou d'un autre type ; par exemple, un opérateur arithmétique peut être remplacé par un opérateur logique et un opérateur d'affectation peut être remplacé par un opérateur de comparaison.

   Exemples : l'instruction (i') peut être altérée de diverses manières notamment par les mutations m'1, m'2 et m'3.

   (i') if ((x + y) > 10) → (m'1) if ((x * y) > 10)
   (m'2) if ((x + y) == 10)
   (m'3) if ((x + y) = 10)

Pour l'ensemble des deux programmes, nous avons disposé d'un total de 25 fautes connues : 12 d'entre elles (notées A, ..., L) affectent le programme ETUD, et les autres (notées F1, ..., F13) le programme LOCALES.

Faute	Description	Impact sur le code source	# lignes affectées
A	Opérateur incorrect utilisé dans le calcul d'une valeur de sortie	Trois instructions incorrectes	3
B	L'implémentation du filtrage ne satisfait pas la spécification	Deux instructions manquantes	2
C	Idem	Deux instructions manquantes	2
D	Idem	Une instruction de branchement incorrecte	1
E	Idem	Une instruction manquante	1
F	Idem	Condition et instruction supplémentaires	3
G	Flot de contrôle erroné lors de la lecture des données sur la 5ème carte	Quatre instructions déplacées	6
H	Initialisation manquante pour les variables liées aux données de contrôle lues sur la 5ème carte	Quatre instructions manquantes et une instruction supplémentaire	5
I	L'implémentation du filtrage ne satisfait pas la spécification	Une instruction incorrecte	1
J	Initialisation d'une variable hors boucle et non à chaque itération	Une instruction manquante (dans le corps d'une boucle)	1
K	Etat initial erroné pour le processus de filtrage	Une instruction incorrecte	1
L	Idem	Une instruction incorrecte	1

Tableau II.1 : Caractéristiques des 12 fautes réelles dans le programme ETUD

Les tableaux II.1 et II.2 résument les caractéristiques de ces fautes pour chacun des deux programmes, montrant ainsi leur diversité. Pour chacune d'elles, nous avons mentionné : une description succincte incluant l'origine de la faute, l'impact de la faute sur le code source et le nombre de lignes (hors commentaires) affectées.

Ce module peut fournir des résultats sous différentes formes selon les objectifs de l'analyse effectuée ; il produit, notamment :

- des résultats statistiques sur le nombre de mutants vivants par jeu de test ;
- la liste des mutants vivants par jeu de test ;
- pour chaque mutant exécuté et pour chaque jeu de test, la localisation dans le jeu des vecteurs de test pour lesquels les sorties fournies par le mutant diffèrent des sorties de référence (occurrences de défaillances).

II.3.4 Jeux de test

Chacun des programmes étudiés (programmes originaux et incorrects) a été exécuté avec différents jeux de tests statistiques conçus selon l'approche développée au LAAS-CNRS (voir, par exemple, [Thévenod-Fosse et al. 1995]).

II.3.4.1 Principe du test statistique

Les entrées de test statistique sont générées aléatoirement selon une distribution probabiliste sur le domaine d'entrée déterminée en fonction du critère de test retenu (structurel ou fonctionnel), ce critère permettant de définir un ensemble d'éléments à activer pendant le test (cf. § I.2.3.1).

La mise en œuvre de cette approche comprend deux étapes :

1) la détermination d'un profil de test, c'est-à-dire d'une distribution des entrées permettant de satisfaire le critère retenu ;
2) l'évaluation d'une longueur de test (nombre d'entrées de test par jeu) nécessaire pour garantir une qualité de test donnée, cette qualité étant définie comme la probabilité d'activer l'élément le moins probable du critère retenu.

Cette approche de génération statistique a fait notamment l'objet de travaux de thèse de doctorat menés par H. Waeselynck [Waeselynck 1993] et C. Mazuet [Mazuet 1994].

Une ligne de code source représente ici soit une déclaration de variable interne, soit une instruction d'affectation, soit une instruction de branchement[10], soit un caractère ou un mot-clef désignant le début ou la fin d'un bloc d'instructions.

Pour le programme LOCALES, le nom de la fonction affectée par la faute ainsi que la version du programme correspondant à la phase durant laquelle la faute a été révélée sont également mentionnés dans le tableau II.2.

II.3.3 Environnement d'analyse de mutation

Les mutants étudiés ont été générés à l'aide de l'environnement d'analyse de mutation SESAME (Software Environment for Software Analysis by Mutation Effects) développé par Yves Crouzet, Chargé de Recherche au LAAS-CNRS [Crouzet 1995]. SESAME comprend un module de génération des mutants et un module d'exécution des mutants décrits brièvement dans les paragraphes ci-après. Cet environnement a été conçu pour pouvoir être utilisé sur des programmes développés dans divers langages ; il a déjà permis de conduire de nombreuses campagnes expérimentales sur des applications développées en C, PASCAL, Assembleur et LUSTRE. La figure II.2 représente de manière synthétique cet environnement d'analyse de mutation.

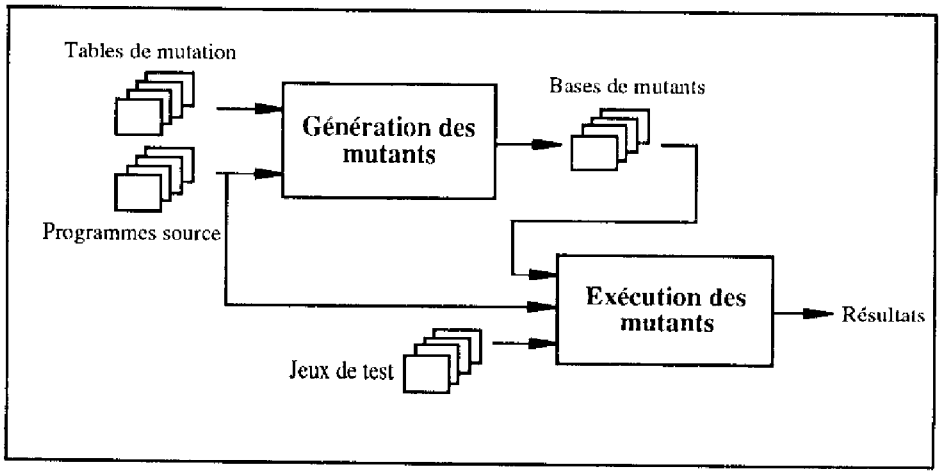


Figure II.2 : Environnement d'analyse de mutation SESAME

10 Ces instructions peuvent être codées sur plusieurs lignes par souci de lisibilité, notamment lorsqu'elles sont composées de plusieurs expressions parfois assez longues.

Pour la fonction *recopie-capteur*, la distribution a été définie à partir du critère de couverture des chemins de la version 1.4, et cinq jeux de 307 entrées de test chacun ont été générés conformément à cette distribution. Pour la fonction *calcul_d_tsgv*, le même critère de couverture a été utilisé sur la version 1.3 (dont la structure n'est pas modifiée par les corrections apportées pour obtenir la version 1.4), et cinq jeux de 97 entrées de test chacun ont été générés. Ces longueurs de test correspondent à une qualité de test de 0,9999 par rapport aux critères retenus, c'est-à-dire, à seulement une chance sur 10 000 de ne pas activer pendant l'exécution d'un jeu de test l'élément le moins probable parmi les éléments sélectionnés par le critère.

II.3.5 Description des expériences

II.3.5.1 Analyse des effets d'une faute

Pour chaque faute (réelle ou de type mutation), l'analyse de ses effets sur l'état interne du programme et l'identification des erreurs générées ont été réalisées de la manière suivante :

1) Choix d'une séquence de test T comportant plusieurs vecteurs de test, permettant d'activer et de révéler la faute : cette séquence est extraite d'un jeu de test statistique complet ;
2) Exécution du programme original puis du programme incorrect avec cette même séquence de test T ;
3) Identification des erreurs par comparaison manuelle entre la trace d'exécution du programme original et celle du programme incorrect ;
4) Identification des défaillances par comparaison manuelle entre les résultats fournis par le programme original et par le programme incorrect.

Une **trace d'exécution** d'un programme est associée ici, à l'exécution d'un programme avec un jeu de test T composé de p (p>1) vecteurs de test, et correspond donc à p cycles d'exécution. Les traces d'exécution des différents programmes ont été obtenues à l'aide de la commande "ctrace" (disponible sous SunOS 4.1) qui permet de suivre l'exécution d'un programme écrit en C ; cette commande permet d'imprimer le texte de chaque instruction exécutée ainsi que les valeurs des variables utilisées ou définies lors de l'exécution de l'instruction.

3) **Mutations sur les symboles** du programme cible : il s'agit de remplacer le nom d'une constante, d'une variable ou d'une fonction par un autre symbole du programme.

Pour chacun de ces types de mutations, des tables sont utilisées : elles permettent (comme les opérateurs de mutation utilisés dans l'environnement Mothra présenté au chapitre I) de spécifier les altérations à appliquer pour chaque constante, opérateur ou symbole. Les tables de mutation sur les constantes et les opérateurs ne dépendent que du langage de programmation. La table de mutation sur les symboles est propre à chaque programme. Les mutations sont réalisées sur chaque constante, opérateur ou symbole du programme à partir de ces tables ; à chaque mutation réalisée correspond un mutant (programme incorrect), qui s'il passe avec succès la phase de compilation, vient enrichir la base de mutants. Cette base contient une description réduite de chaque mutant (différence entre le code source du programme original et du mutant).

En général, les mutations utilisées correspondent à des fautes de type "instruction incorrecte" mais certaines caractéristiques du langage C permettent de créer des mutations de type "instruction manquante" ou "instruction supplémentaire" en remplaçant un opérateur d'affectation "=" par un opérateur de comparaison (par exemple, "==" ou ">") et vice-versa.

II.3.3.2 Exécution des mutants

Le module d'exécution des mutants permet d'exécuter chacun des mutants retenus dans une base avec l'ensemble des jeux de test spécifiés en entrée de ce module. Dans une première phase, ce module établit les sorties de référence en appliquant les différents jeux de test au programme original (non muté). Puis pour chaque mutant, il procède de la manière suivante :

1) Reconstitution du code source du mutant à partir du programme original et de la description du mutant dans la base de mutants ;
2) Génération du code exécutable du mutant ;
3) Exécution du mutant avec chaque jeu de test spécifié et comparaison des résultats obtenus aux sorties de référence.

Nous avons mis en évidence la complexité des comportements erronés d'un programme lorsque ceux-ci sont observés sur plusieurs cycles d'exécution successifs. L'analyse de ces comportements permet d'identifier des mécanismes de création, d'annulation et de masquage d'erreurs beaucoup plus subtils et variés que ceux identifiés lors d'études antérieures (réalisées à partir de l'observation sur un seul cycle d'exécution). Ces mécanismes sont d'autant plus complexes que la propagation d'une erreur peut conduire à la fois à la création, à l'annulation ou au masquage d'une erreur sur une ou plusieurs données ou que la propagation combinée de plusieurs erreurs est parfois nécessaire pour créer, masquer ou annuler une erreur. Ces divers phénomènes ont été observés lors des expérimentations sur le programme ETUD. Ces observations sont commentées au chapitre suivant. L'explication théorique de ces phénomènes fera ensuite l'objet du chapitre IV.

II.3.4.2 Application aux programmes étudiés

La génération statistique permet, à partir d'une distribution probabiliste donnée, d'obtenir automatiquement un grand nombre d'entrées de test. Cette facilité nous a permis de partir d'un large éventail de vecteurs de test pour analyser les effets d'une faute réelle ou artificielle sur le comportement interne des programmes.

a) Programme ETUD

Dans le cadre de nos expériences sur le programme ETUD, nous disposions de jeux de tests statistiques générés lors de précédentes études expérimentales [Waeselynck 1993]. Quatre types de distributions avaient été définis et utilisés pour générer ces jeux :

1) Distribution uniforme sur le domaine d'entrée ; un jeu de 5300 entrées de test avait été ainsi généré.
2) Distribution structurelle dérivée de la structure du programme en utilisant comme critère de test la couverture des instructions ; 5 jeux de 500 entrées de test avaient été ainsi générés.
3) Distributions fonctionnelles dérivées d'une première spécification comportementale du programme déduite de sa spécification informelle, et basée sur les machines à états finis (MEF) et les tables de décisions (TD) ; 5 jeux de 441 entrées de test avaient été ainsi générés.
4) Distributions fonctionnelles dérivées d'une seconde spécification comportementale du programme réalisée dans l'environnement Statemate [Harel et al. 1990], c'est-à-dire basée sur les statecharts ; 5 jeux de 441 entrées de test avaient été ainsi générés.

Ces jeux statistiques nous ont permis d'étudier les comportements erronés du programme sur plusieurs cycles d'exécution successifs. En effet, certaines fautes révélées dans ce programme produisent des erreurs qui doivent propager sur plusieurs cycles d'exécution avant de pouvoir affecter des sorties et donc être détectées.

b) Programme LOCALES

Dans le cadre de nos expériences sur le programme LOCALES, les deux fonctions ont été testées indépendamment (test unitaire). Pour chacune d'elles, des jeux statistiques ont été générés selon une distribution structurelle, basée sur l'analyse du graphe de contrôle.

CHAPITRE III

ANALYSE EXPÉRIMENTALE DE COMPORTEMENTS ERRONÉS

III.1 INTRODUCTION

Ce chapitre décrit les études menées sur le programme ETUD présenté au chapitre II. Ce programme est issu d'une application critique du nucléaire ; ses fonctionnalités ont été décrites au paragraphe II.3.1. Les études expérimentales consistent à analyser les effets d'une part des 12 fautes réelles connues (cf. tableau II.1), et d'autre part de 24 mutations. Leur objectif est, dans un premier temps, d'**identifier les erreurs et les comportements erronés** générés par des fautes d'origines diverses, réelles ou artificielles, et dans un deuxième temps, d'**évaluer la représentativité** sur ce cas concret **des erreurs et des comportements erronés générés par des fautes de type mutation** vis-à-vis de ceux générés par des fautes réelles [Daran & Thévenod-Fosse 1996].

Cette étude, inédite sur un cas réel, a débouché sur la création de deux bases de données d'erreurs conséquentes : l'une relative aux erreurs produites par les fautes réelles et l'autre aux erreurs produites par les mutations. Les erreurs enregistrées dans ces bases sont liées par des relations de cause à effet qui permettent de reconstituer automatiquement les flots d'erreurs relatifs à la propagation d'une erreur initiale (c'est-à-dire créée au moment de l'activation d'une faute). La constitution de ces bases de données et les analyses comparatives menées sur ces deux bases représentent l'essentiel du contenu de ce chapitre.

III.2 CADRE DES EXPÉRIENCES SUR ETUD

Le cadre expérimental général a été décrit au chapitre II (§ II.3). Ce paragraphe a pour but de détailler certains aspects spécifiques aux études menées sur le programme ETUD, et notamment d'apporter des précisions sur la sélection des séquences de test utilisées et des mutations étudiées.

II.3.5.2 Identification des erreurs et des flots d'erreurs

Les erreurs générées au cours de l'exécution d'un programme incorrect ont pu être identifiées par comparaison des deux traces d'exécution, l'une correspondant au programme correct et l'autre correspondant au programme incorrect, exécutés tous deux avec la même séquence de test T ; la séquence d'erreurs identifiées sur l'ensemble d'une exécution est dénommée **trace d'erreurs**. Il est parfois difficile de synchroniser la comparaison des états internes correct et erroné, en particulier lorsque, suite à une erreur de branchement, les deux programmes n'exécutent pas simultanément les mêmes instructions ; ce problème sera abordé au chapitre IV.

Les notions de trace d'exécution ou de trace d'erreurs ont été précédemment définies dans [Murrill 1991, Laski et al. 1995] mais elles étaient alors associées à l'exécution du programme avec une seule entrée de test ; l'analyse des erreurs n'était donc faite que sur un seul cycle d'exécution.

Une même trace d'erreurs peut comporter les erreurs produites suite à plusieurs activations de faute, et une activation de faute peut générer une ou plusieurs erreurs initiales. Nous désignons sous le terme **flot d'erreurs**, une séquence d'erreurs due à la propagation d'**une erreur initiale**. Un flot d'erreurs débute donc par une erreur initiale et comporte toutes les erreurs créées suite à la propagation de cette erreur initiale jusqu'à la fin de l'exécution du programme avec la séquence T. De la même façon, un sous-flot d'erreurs désignera une séquence d'erreurs due à la propagation d'une erreur autre qu'initiale, et donc extraite d'un flot d'erreurs.

II.4 CONCLUSION

Dans ce chapitre, nous avons présenté le cadre méthodologique et expérimental de nos travaux. Ceux-ci reposent sur l'analyse détaillée des erreurs et des comportements erronés induits à la fois par des fautes réelles et des fautes artificielles sur deux logiciels critiques développés dans des contextes industriels différents.

Nous avons précisé la nature des fautes étudiées et les types d'erreurs observés ainsi que les propagations possibles de ces erreurs au cours de l'exécution d'un programme.

Partant de ces résultats, nous avons donc sélectionné un sous-ensemble de séquences de vecteurs en nous basant sur les critères suivants :

1) chaque séquence commence par un vecteur qui permet d'activer une fonction d'initialisation des variables internes ayant pour rôle de mémoriser des résultats issus de cycles d'exécution précédents ; rappelons, en effet, qu'ETUD est un programme *avec mémoire* ; cette contrainte sur le premier vecteur nous permet de garantir un comportement déterministe du programme original pendant l'exécution de toute la séquence, c'est-à-dire un comportement indépendant de l'état interne du programme avant exécution de la première instruction du premier cycle ;
2) chaque séquence révèle au moins une fois une des fautes ;
3) chaque faute est révélée par au moins une séquence.

Nous avons ainsi retenu un premier sous-ensemble de 12 séquences (notées T1, ..., T12), contenant entre 3 et 10 vecteurs de test chacune. Elles ont été utilisées pour analyser des comportements défectueux produits par les fautes A, ..., L ; mais elles ont également conditionné *en partie* le choix des mutations étudiées comme nous allons le voir dans le paragraphe suivant. L'ensemble des séquences de test sera récapitulé au paragraphe III.2.4.

III.2.3 Mutations étudiées

Les 24 mutations étudiées ont été sélectionnées à partir d'un ensemble de 2419 mutations générées lors de précédentes expériences sur l'évaluation de l'efficacité de différents jeux de test statistique par analyse de mutation [Thévenod-Fosse & Crouzet 1995]. Le choix de ces mutations a été fait avant d'étudier (et donc de connaître) les comportements erronés produits par les fautes A, ..., L.

Notre objectif principal étant de répertorier des comportements erronés divers et de comparer les erreurs produites par les mutations et par les fautes réelles, la sélection a été guidée par les contraintes suivantes :

1) se limiter à un échantillon de mutations de taille raisonnable en raison du niveau très détaillé, et donc de la complexité, des analyses que nous nous proposions de faire ; cet échantillon devait cependant être varié en incluant les trois types de mutations (sur les constantes, opérateurs et symboles) générés par l'outil SESAME, et en affectant différentes instructions ;

Mutation	Type	Description	Localisation de la mutation / faute réelle
M1	C	Incrémentation de la valeur d'une constante	A (2ème instruction)
M2	C	Modification de la valeur d'un bit d'une constante	A (3ème instruction)
M3	C	Incrémentation de la valeur d'une constante	B (2ème instruction)
M4	C	Modification de la valeur d'un bit d'une constante	D
M5	C	Modification de la valeur d'un bit d'une constante	H (1ère instruction)
M6	C	Décalage binaire appliqué à une constante	K
M7	C	Décalage binaire appliqué à une constante	L
M8	C	Décalage binaire appliqué à une constante	
M9	C	Modification de la valeur d'un bit d'une constante	
M10	O	Remplacement d'un opérateur "ET bit à bit" par un opérateur "ET logique"	A (1ère instruction)
M11	O	Remplacement d'un opérateur d'affectation par un opérateur de comparaison	E
M12	O	Remplacement d'un opérateur "ET bit à bit" par un opérateur "OU logique"	G (1ère instruction)
M13	O	Remplacement d'un opérateur "ET logique" par un opérateur "OU bit à bit"	
M14	S	Remplacement de la variable définie dans une affectation	B (1ère instruction)
M15	S	Remplacement de la variable définie dans une affectation	C (1ère instruction)
M16	S	Remplacement de la variable définie dans une affectation	E
M17	S	Remplacement de la variable définie dans une affectation	H (1ère instruction)
M18	S	Remplacement d'une variable utilisée dans une affectation	I
M19	S	Remplacement de la variable définie dans une affectation	J
M20	S	Remplacement d'une variable utilisée dans une condition	
M21	S	Remplacement d'une variable utilisée dans une condition	
M22	S	Remplacement d'une variable utilisée dans une affectation	
M23	S	Remplacement de la variable définie dans une affectation	
M24	S	Remplacement d'une variable utilisée dans une condition	

Tableau III.1 : Caractéristiques des 24 mutations
(C : mutation sur la valeur numérique d'une constante ;
O : mutation sur un opérateur ; S : mutation sur un symbole)

III.2.1 Programme original

La version du programme ETUD que nous avons prise comme **référence**, c'est-à-dire supposée correcte, correspond au programme obtenu après correction de l'ensemble des 12 fautes (A, ..., L) identifiées lors de précédents travaux sur le test statistique fonctionnel [Thévenod-Fosse & Waeselynck 1993, Waeselynck 1993] : cette version sera appelée **programme original**. Rappelons que ces fautes, dont les caractéristiques ont été récapitulées dans le tableau II.1 (§ II.3.2), sont d'origines diverses : les fautes A, G et J sont des fautes de codage ; les fautes B à F et la faute I résultent d'une mauvaise compréhension de la spécification du filtrage des mesures ; les fautes H, K et L sont des fautes d'initialisation.

Nos études ont consisté à analyser les comportements de 36 programmes incorrects en les comparant au comportement du programme original. Douze programmes incorrects ont été obtenus en injectant une à une les fautes A, ..., L dans le programme original. Les 24 autres correspondent aux mutants présentés au paragraphe III.2.3.

III.2.2 Choix des séquences de test basé sur les fautes réelles

Nous disposions pour nos analyses de 16 jeux de test statistique générés selon différents types de distribution : uniforme, structurelle ou fonctionnelle (cf. § II.3.4.2).

Chacun des 12 programmes incorrects contenant une des fautes réelles ainsi que le programme original ont été exécutés avec ces 16 jeux de test complets ; cela nous a permis d'identifier pour chaque jeu et pour chaque faute prise isolément, les vecteurs de test pour lesquels les sorties fournies différaient des sorties de référence. Comme indiqué dans [Thévenod-Fosse & Waeselynck 1993]), les 10 jeux statistiques fonctionnels se sont avérés les plus efficaces dans leur capacité à révéler chacune des fautes, en produisant de nombreuses défaillances, et ceci de manière répétitive. Chacun de ces 10 jeux est composé de 441 vecteurs de test : la complexité de la génération et du dépouillement des traces d'exécution associées à des jeux d'une telle longueur, nous a obligée à extraire parmi eux des séquences de vecteurs de test de longueurs réduites.

générées par les fautes réelles, et la seconde de l'analyse des traces d'erreurs générées par les mutations. Elles sont appelées respectivement *BD-Fautes* et *BD-Mutations*.

Dans ce paragraphe, après avoir précisé le contenu d'un enregistrement d'erreur, nous décrivons chacune de ces bases et donnons les principaux résultats quantitatifs issus de leur comparaison.

III.3.1 Enregistrement d'erreur

Chaque erreur identifiée dans une trace d'erreurs donne lieu à un enregistrement dans une des deux bases de données. Un **enregistrement d'erreur** contient les informations suivantes, structurées en 9 champs :

1. le numéro de l'erreur (le principe de la numérotation des erreurs est détaillé ci-après) ;
2. le type de l'erreur (sur le flot de données ou de contrôle) et l'identification de la variable ou de la condition de branchement erronée ;
3. la valeur correcte (fournie par le programme original) ;
4. la valeur erronée ;
5. le numéro de la ligne de code (LOC) dont l'exécution a donné lieu à la création de l'erreur ;
6. la séquence de test Ti ($i \in [1, ..., 13]$) soumise au programme ;
7. le numéro du vecteur de test (dans la séquence Ti) qui a provoqué l'erreur ;
8. l'origine de l'erreur c'est-à-dire la faute, l'erreur ou la combinaison d'erreurs qui l'ont créée ;
9. la conséquence immédiate de cette erreur, c'est-à-dire les erreurs créées, annulées ou masquées par propagation de cette erreur.

Les numéros d'erreur ont été attribués dans l'ordre chronologique d'identification des erreurs durant l'analyse des traces d'erreurs dues d'abord aux fautes réelles, puis aux mutations. Cependant, lorsque deux (ou plus) enregistrements ont leurs champs 2, 3, 4 et 5 identiques, nous leur avons affecté le même numéro, celui correspondant au premier enregistrement dans l'ordre chronologique. Deux enregistrements d'erreur ont donc le même numéro si et seulement s'ils correspondent au même couple {valeur correcte, valeur erronée} associé à la même variable ou condition de branchement et créé lors de l'exécution de la même instruction : les **erreurs** associées à de tels

2) pouvoir comparer les traces d'erreurs produites par les mutations à celles produites par les fautes réelles.

La première contrainte nous a conduit à restreindre notre analyse à 24 mutations, soit deux fois plus que les fautes réelles : 9 d'entre elles (notées M1 à M9) affectent les valeurs numériques de constantes, 4 autres (M10 à M13) portent sur des opérateurs, et les 11 dernières (M14 à M24) sur des symboles.

La seconde contrainte nous a conduit à retenir, de préférence, des mutations détectables par au moins une des 12 séquences de test T1, ..., T12 pour comparer les traces d'erreurs produites dans des conditions d'exécution identiques. De plus, pour faciliter la comparaison des mécanismes d'activation des mutations et des fautes réelles, certaines de ces mutations devaient affecter les mêmes instructions que les fautes réelles. Ainsi 16 des 24 mutations portent sur des instructions impliquées dans la correction d'une faute réelle. Notons cependant qu'aucune mutation ne pouvait a priori être injectée sur les instructions affectées par la faute F puisqu'il s'agit d'instructions qui doivent être supprimées pour corriger F (et donc qui n'existent pas dans le programme original). Les 8 autres mutations ont été choisies sur des lignes de code "éloignées" (structurellement) de celles affectées par des fautes réelles, afin de ne pas trop restreindre la portée des analyses : il s'agit des mutations M8, M9, M13, M20 à M24. Six d'entre elles ont été choisies arbitrairement parmi un ensemble de mutations détectées par au moins une des séquences T1 à T12. Les deux autres, M9 et M23, non révélées par les séquences T1 à T12, nous ont semblé intéressantes à étudier car elles avaient conduit à un nombre très faible de défaillances sur l'ensemble des 10 jeux statistiques fonctionnels complets : une treizième séquence T13 a donc été extraite d'un de ces 10 jeux ; elle est composée de 7 vecteurs et permet d'observer des défaillances dues à M9 et M23.

Le tableau III.1 récapitule les caractéristiques des mutations sélectionnées en donnant, pour chacune d'elles : son type (mutation sur la valeur numérique d'une constante, sur un opérateur ou un symbole), une description succinte de la modification correspondante, et sa localisation dans le code par rapport aux instructions impliquées dans la correction d'une faute réelle (sauf pour les 8 mutations arbitrairement choisies).

L'instant de création d'une erreur durant l'exécution d'un programme est défini par les champs 5, 6 et 7 et éventuellement par un indice de boucle (champ supplémentaire non représenté sur la figure III.1) lorsque le chemin d'exécution comprend plusieurs passages successifs dans une boucle. Les champs 8 et 9 permettent de lier les erreurs entre elles et de reconstituer les traces d'erreurs (et donc les flots d'erreurs) relatives à chaque programme incorrect.

III.3.2 Bases de données complètes et réduites

La base de données des fautes réelles, *BD-Fautes*, contient 1450 enregistrements d'erreur représentant 255 erreurs avec des numéros différents. La base de données des mutations, *BD-Mutations*, contient 2272 enregistrements représentant 348 erreurs. Ces deux bases ne comptabilisent en tout que 395 erreurs avec des numéros différents, ce qui signifie que 208 numéros d'erreur sont communs aux deux bases.

Le nombre d'occurrences d'un même numéro d'erreur dans l'une des bases peut varier de 1 à 81 : ce dernier nombre correspond à une erreur créée par exécution d'une instruction localisée dans une boucle et exécutée quel que soit le chemin sélectionné. En triant chacune des bases de manière à ne garder qu'une occurrence de chaque numéro, nous avons constitué deux bases de données dites **réduites** : la première, associée aux fautes réelles et notée *BD-Fr*, contient donc 255 enregistrements ; la seconde, associée aux mutations et notée *BD-Mr*, en contient 348. Ces bases de données réduites permettent de comparer la nature des erreurs produites par les fautes réelles d'une part, et par les mutations d'autre part, indépendamment de la fréquence de leur création.

III.3.3 Comparaison des erreurs

La comparaison entre deux bases, l'une relative aux fautes réelles et l'autre aux mutations, a pour objectif d'évaluer les pourcentages d'enregistrements dans chaque base qui correspondent à des erreurs communes aux deux bases ; une erreur se différencie d'autres erreurs par les champs 2 à 5 de l'enregistrement correspondant et est identifiée par son numéro figurant au champ 1 (cf. figure III.1). Nous donnons ci-après les résultats de la comparaison entre les deux bases complètes, puis entre les deux bases réduites. Ces seconds résultats sont en effet plus significatifs pour évaluer la représentativité des erreurs générées par les mutations par rapport à celles générées par les fautes réelles, car non biaisés par les occurrences multiples de certaines erreurs identiques au sein d'une même base.

III.2.4 Récapitulatif des expériences

Le tableau III.2 récapitule, pour chacune des 13 séquences de test sélectionnées, le nombre total de vecteurs qu'elle contient, ainsi que la liste des fautes réelles et des mutations dont les comportements erronés ont été analysés par exécution du programme incorrect avec tout ou partie de la séquence.

Séquence de test	# vecteurs	Fautes étudiées	Mutations étudiées
T1	6	A, B	M2, M3, M14
T2	9	A	M10
T3	9	C, D	M4, M8, M15, M20
T4	6	E	M11
T5	8	E	M1, M16
T6	3	F	M12, M21, M22
T7	5	G, H	M5, M13, M17
T8	5	I	M18
T9	4	J	M19, M24
T10	5	K	
T11	5	K	M6
T12	10	L	M7
T13	7		M9, M23

Tableau III.2 : Sélection des séquences de test

III.3. BASES DE DONNÉES D'ERREURS

Chaque exécution d'un programme incorrect avec une séquence Ti ($i \in [1, ..., 13]$) permettant d'activer la faute présente a donné lieu à une **trace d'erreurs** qui est le résultat de la comparaison entre la trace d'exécution du programme incorrect et la trace d'exécution du programme original soumis à la même séquence Ti (cf. § II.3.5). Chacune des traces d'erreurs produites a ensuite été analysée manuellement dans le but d'étudier les effets d'une faute sur l'état interne du programme, et d'identifier précisément les erreurs générées ainsi que leurs liens de cause à effet. Deux bases de données ont ainsi été créées : la première résulte de l'analyse des traces d'erreurs

spécifiques aux mutations, 35 sont des erreurs initiales (soit 10% des 348 erreurs dans *BD-Mr*). Les erreurs initiales sont propres à l'activation d'une faute et, par conséquent, très dépendantes de chaque faute : que des erreurs initiales ne soient pas communes aux deux bases n'a donc rien de surprenant.

L'analyse détaillée des autres erreurs de *BD-Fr* non reproduites par les mutations (12% de *BD-Fr*) montre que ces erreurs sont dues soit à la faute K, lorsque le programme correspondant est exécuté avec la séquence de test T10, soit à la faute D. En ce qui concerne la faute K, aucun mutant n'a été exécuté avec la séquence T10 (cf. tableau III.2) ce qui explique le fait qu'ils n'ont pas reproduit les mêmes erreurs. En ce qui concerne la faute D qui correspond à une expression conditionnelle incorrecte (cf. tableau II.1), elle est en fait similaire à une faute de type mutation et sur l'ensemble des 2419 mutants qui existaient pour le programme ETUD, il aurait été facile de sélectionner a priori une mutation permettant de simuler le comportement erroné produit par D ; mais étant donnés les objectifs de notre étude, les mutations étudiées n'ont pas été sélectionnées dans le but de simuler, par leurs comportements erronés, des fautes réelles.

Les 30% d'erreurs non initiales spécifiques à *BD-Mr* sont dues à des mutations dont les comportements erronés seront décrits au paragraphe III.4.3. Pour la plupart d'entre elles, ces erreurs n'ont donné lieu qu'à un seul enregistrement dans la base complète *BD-Mutations*, c'est-à-dire n'ont été observées qu'une seule fois sur l'ensemble des expériences : c'est ce qui explique l'écart entre ces 30% et les 8% (cités au début de ce paragraphe) issus de la comparaison les deux bases de données complètes.

Ces premiers résultats montrent que, sur un cas réel, les erreurs générées par des mutations peuvent être représentatives d'un nombre conséquent d'erreurs générées par des fautes réelles. Nous allons maintenant poursuivre notre analyse comparative en étudiant les comportements erronés (c'est-à-dire comment une erreur est créée et comment elle propage) produits par les fautes réelles et les mutations.

enregistrements sont **identiques**. Au cours des expériences, nous avons observé des erreurs identiques dans une même trace d'erreurs ainsi que dans différentes traces associées aux fautes réelles et aux mutations : pour un numéro d'erreur donné, *BD-Fautes* et *BD-Mutations* peuvent donc contenir plusieurs enregistrements qui représentent différentes occurrences d'une erreur identique.

La figure III.1 illustre le contenu de quatre enregistrements d'erreur extraits de *BD-Fautes*. Les deux premiers ont le même numéro : en effet, l'erreur E162 a été créée suite à l'activation de la faute H à deux reprises, lors de l'exécution du premier puis du deuxième vecteur de la séquence T7. Cette erreur ne propage que lors de l'exécution du deuxième vecteur. Ces deux enregistrements d'erreur identique diffèrent donc par le numéro du vecteur de test (dans la séquence T7) qui a provoqué l'erreur et par la conséquence immédiate de l'erreur ; ils sont relatifs à la même trace d'erreurs.

1	2	3	4	5	6	7	8	9
Numéro d'erreur	Type et identification	Valeur correcte	Valeur erronée	LOC	Séquence de test	Vecteur de test	Origine	Conséquence
E162	PASSAG	0	1	93	T7	1	H	aucune
E162	PASSAG	0	1	93	T7	2	H	E159, B2, sortie
E159	DEF_TEST	0	1	626	T7	2	E162	Annulée par E20
B2	Cond. 2.6	F	V	638	T7	2	E162	B3

<----------> *Champs identifiant une erreur*

Figure III.1 : Quatre enregistrements d'erreur extraits de *BD-Fautes*

Au cours d'un cycle d'exécution, les programmes incorrect et original peuvent exécuter simultanément des instructions différentes suite à une erreur sur le flot de contrôle (chemins d'exécution disjoints) : la valeur d'une donnée à la fin de l'exécution de deux instructions différentes peut alors n'être visible que sur l'une des deux traces d'exécution, ce qui rend difficile la comparaison des deux traces. Pour traduire ce type de comportement erroné, un enregistrement d'erreur est créé dans la base en utilisant le symbole "*" pour désigner la valeur non observable à cet instant d'exécution dans la trace soit du programme original ("*" figure dans le champ 3), soit du programme incorrect ("*" dans le champ 4).

Sur chaque **nœud** figurent les informations suivantes extraites des champs de l'enregistrement (cf. figure III.1) : le nom de la variable ou de la condition de branchement erroné (champ 2) ; les valeurs correcte et incorrecte (champs 3 et 4), l'une d'entre elles pouvant correspondre au symbole "*" si elle n'est pas observable à l'instant d'exécution correspondant ; le cycle d'exécution durant lequel l'erreur est créée (champ 7). Par exemple, le premier nœud (CO_AQU_DIF = 1/0)$_1$) correspond à l'erreur initiale : il signifie que la variable notée CO_AQU_DIF a la valeur 1 dans le programme correct et 0 dans le programme incorrect au 1er cycle d'exécution, suite à l'activation de K.

Les **arcs** traduisent les liens de cause à effet entre les erreurs, représentés par les champs 8 et 9 dans *BD-Fautes* : le nœud à l'extrémité initiale d'un arc représente une erreur dont la propagation a permis de créer, masquer ou annuler l'erreur figurant à l'extrémité finale de l'arc. Par exemple, le nœud (Cond. 5.5.10 = V/F)$_4$ signifie que la condition de branchement notée 5.5.10 a la valeur vraie dans le programme correct et fausse dans le programme incorrect au 4ème cycle d'exécution ; elle résulte de l'erreur sur la variable notée CO_AQU_DIF (nœud (CO_AQU_DIF = 5/4)$_4$) et conduit à la sélection erronée d'une branche (division en deux sous-flots).

Les deux **nœuds en gras** désignent deux erreurs affectant des variables de sortie différentes et correspondent donc à l'occurrence d'une défaillance du programme puisqu'elles sont créées au même cycle (en réponse au 5ème, et dernier, vecteur de T10). Le **nœud en italique** *(INVALID_PA = 1/1)*$_4$ représente une annulation d'erreur : nous détaillerons ce mécanisme d'annulation par la suite, à titre d'exemple (voir [e] ci-après). Les **nœuds encadrés** désignent des erreurs présentes dans l'état interne du programme incorrect à la fin des 5 cycles d'exécution associés à la séquence de test T10 (et, par conséquent, susceptibles de propager si la séquence avait contenu plus de 5 vecteurs).

Cette représentation graphique d'un flot d'erreurs permet de visualiser les mécanismes de création, masquage et annulation d'erreurs, et de les étudier par analyse structurelle de ce graphe. A titre d'exemples, nous commentons ci-dessous cinq cas différents extraits de la figure III.3, et notés [a] à [e] sur cette figure.

[a] *Création d'une erreur sur la même variable* : durant le premier cycle d'exécution, l'erreur initiale (CO_AQU_DIF = 1/0)$_1$ crée une nouvelle erreur, c'est-à-dire un nouveau couple {valeur correcte, valeur erronée}, qui affecte la même variable en annulant ainsi l'erreur précédente ; cette nouvelle erreur propage de la même manière au cycle suivant, et ce mécanisme continue durant les 3ème et 4ème cycles, les programmes correct et incorrect exécutant le même

La comparaison des bases complètes *BD-Fautes* et *BD-Mutations* conduit aux résultats suivants :

1) sur les 1450 enregistrements d'erreur contenus dans *BD-Fautes*, 1285 (soit **88,6%**) correspondent à des erreurs enregistrées dans *BD-Mutations* ; 165 enregistrements (soit **11,4%**) sont donc relatifs à des erreurs non reproduites par les mutations sélectionnées : 67 (4,6%) d'entre eux représentent des erreurs initiales[11] et 98 (7%) des erreurs créées par propagation ;

2) sur les 2272 enregistrements d'erreur contenus dans *BD-Mutations*, 1930 (soit **84,9%**) correspondent à des erreurs enregistrées dans *BD-Fautes* ; 342 enregistrements (soit **15,1%**) sont spécifiques aux mutations sélectionnées : 158 (7%) d'entre eux représentent des erreurs initiales et 184 (8,1%) des erreurs créées par propagation.

Les résultats quantitatifs de la comparaison des bases réduites *BD-Fr* et *BD-Mr* sont récapitulés sur la figure III.2. Sur un total de 395 erreurs répertoriées au cours de nos expériences sur les 36 programmes incorrects, 208 sont communes aux deux bases, ce qui représente notamment 82% des erreurs relatives aux fautes réelles. Ces résultats, commentés ci-dessous, sont en faveur d'une bonne représentativité des erreurs générées par les mutations étudiées.

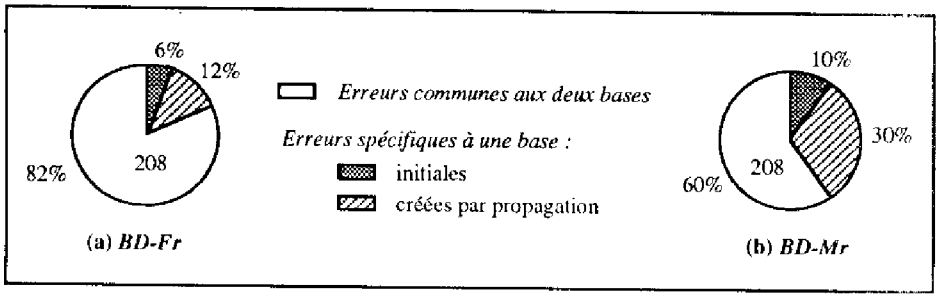


Figure III.2 : Comparaison des erreurs dans les bases de données réduites
(a) Base relative aux fautes réelles (255 erreurs)
(b) Base relative aux mutations (348 erreurs)

Parmi les 47 erreurs dans *BD-Fr* spécifiques aux fautes réelles, 15 sont des erreurs initiales (soit 6% des 255 erreurs dans *BD-Fr*). Parmi les 140 erreurs

---

11 Rappelons qu'une erreur initiale est la conséquence immédiate de l'activation d'une faute (cf. § II.2.1).

[b] *Création de deux erreurs sur deux variables différentes par propagation d'une seule erreur* : l'erreur (VALID_MES = $0/1)_4$ est créée au 4ème cycle, et cette valeur erronée est utilisée à deux reprises dans la suite de ce cycle, pour calculer les valeurs des variables VALID_MESU_VOIE et VALID_DER_VAL qui, à leur tour, deviennent erronées ; cela conduit à une division du flot en deux sous-flots. L'erreur sur VALID_MESU_VOIE propage ensuite, toujours durant le 4ème cycle ; celle sur VALID_DER_VAL est masquée durant la suite du 4ème cycle, mais elle crée des erreurs au cycle suivant.

[c] *Création d'une erreur par passage de paramètres entre deux fonctions* : VALID_TOT_VOIE est une variable de retour d'une fonction appelée ; dans la fonction appelante, sa valeur est mémorisée dans VALID_FILTR_VOIE. Cela conduit à une simple extension du flot d'erreur.

[d] *Création d'une erreur par propagation combinée de deux erreurs* : l'exemple [d] correspond à des instants d'exécution pendant lesquels les programmes correct et incorrect n'exécutent pas simultanément les même instructions, suite à une erreur de branchement ; dans le programme incorrect, la variable MESURE reçoit la valeur 27 (nœud (MESURE = $*/27)_5$) lors de l'exécution d'une instruction incorrectement sélectionnée suite à l'erreur de branchement (Cond. 9.2 = $F/V)_5$; cette valeur 27 résulte d'un calcul qui fait intervenir la variable DER_MESU qui avait reçu à tort la valeur 182 (nœud (DER_MESURE = $*/182)_5$) suite à une précédente erreur de branchement (Cond. 5.5.2 = $F/V)_5$. Cela conduit à la convergence de 2 sous-flots.

[e] *Annulation d'une erreur par propagation combinée de deux erreurs* : l'exemple [e] correspond lui aussi à des instants pendant lesquels les programmes correct et incorrect n'exécutent pas simultanément les même instructions ; dans le programme incorrect la variable MESURE, qui a reçu la valeur 0 à l'instant précédent (nœud (MESURE = $*/0)_4$), est utilisée pour calculer la valeur de INVALID_PA qui passe à 1 ; dans le programme correct, la valeur 1 a été affectée à INVALID_PA à l'instant précédent (nœud (INVALID_PA = $1/0)_4$) et reste inchangée ce qui annule l'erreur {1, 0} créée à l'instant précédent sur cette variable. Cela conduit à la convergence de 2 sous-flots.

Cet exemple de flot d'erreurs montre la complexité de certains comportements erronés au cours de plusieurs cycles d'exécution successifs. En effet, à un instant donné de l'exécution, la propagation d'une erreur ou la propagation combinée de plusieurs erreurs peut conduire à la création, à l'annulation ou au masquage d'autres erreurs. Ces mécanismes peuvent affecter le même état de donnée, ou un ou plusieurs autres. Notons

chemin pendant les 3 premiers cycles. Cela conduit à l'extension simple du flot d'erreurs.

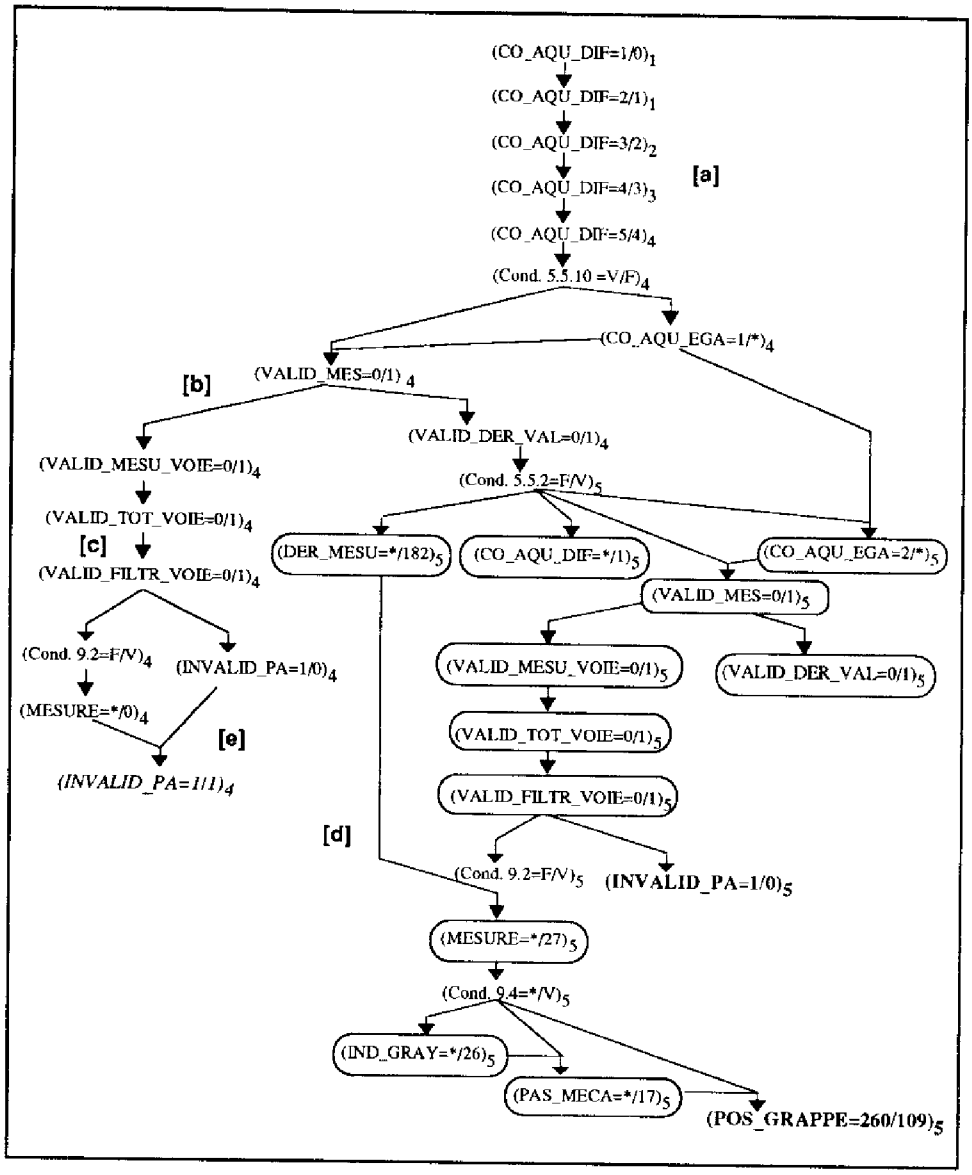


Figure III.3 : Exemple d'un flot d'erreurs dû à la faute K (séquence de test T10)

III.4 ANALYSE DES COMPORTEMENTS ERRONÉS

Faute	Séquence de test (# vecteurs)	# erreurs initiales : # activations (# erreurs/activation)	# flots d'erreurs (FD ou FN : # erreurs/flot)	# total erreurs
A	T1 (6)	3 variables erronées : 1 (3)	3 défaillances	3
	T2 (9)	23 variables erronées : 8 (3)	23 défaillances	23
B	T1 (6)	16 variables erronées : 8 (2)	2 FD : 42 et 22	77
C	T3 (6)	4 variables erronées : 2 (2)	1 FD : 79 2 FN : 25 et 2	107
D	T3 (4)	10 branchements erronés : 10 (1)	4 FD : 52, 33, 23 et 20 6 FN : 28, 25(3), 8 et 5	244
E	T4 (6)	8 variables erronées : 8 (1)	1 FD : 49 7 FN : 2(7)	63
	T5 (7)	10 variables erronées : 10 (1)	1 FD : 60 9 FN : 5(3) et 2(6)	87
F	T6 (3)	25 branchements erronés : 25 (1)	1 FD : 21 12 FN : 22, 16, 7, 4, 3(6) et 2(2)	93
G	T7 (5)	16 variables erronées : 4 (4)	3 défaillances 3 FD : 29 et 23(2) 1 FN : 8	95
H	T7 (5)	10 variables erronées : 2 (5)	1 défaillance 1 FD : 80	89
I	T8 (5)	6 variables erronées : 6 (1)	1 FD : 47 5 FN : 16(5)	127
J	T9 (4)	72 variables erronées : 72 (1)	1 FD : 4 1 FN : 4	78
K	T10 (5)	18 variables erronées : 18 (1)	1 FD : 32 9 FN : 21, 4, 3(4) et 2(3)	83
	T11 (5)	18 variables erronées : 18 (1)	1 FD : 42 4 FN : 4, 3(2), 2	67
L	T12 (10)	18 variables erronées : 18 (1)	1 FD : 86 11 FN : 31, 28, 22, 8, 7, 6(3), 3(2) et 2	214

Tableau III.3 : Analyse des traces d'erreurs générées par les fautes réelles (FD : flot d'erreurs conduisant à défaillance(s) ; FN : flot d'erreurs non détectées)

La base de données complète *BD-Fautes* contient 1450 erreurs réparties dans 15 traces d'erreurs. L'analyse de ces traces a permis d'identifier 86 flots dont 19 seulement

que 5 cycles successifs ont été nécessaires pour révéler la faute K, dont l'activation a pourtant permis de créer une erreur initiale dès le premier cycle.

III.4.1.2 Analyse des traces d'erreurs et des flots d'erreurs

L'analyse d'une trace d'erreurs permet de savoir si une faute est activée une ou plusieurs fois lors de l'exécution du programme avec une séquence de test donnée, quels flots d'erreurs conduisent à défaillance(s), combien d'erreurs peut contenir un flot, etc. Le tableau III.3 récapitule les résultats de ces analyses détaillées. Dans ce tableau, nous indiquons pour chaque faute A, ..., L :

a) les séquences de test avec lesquelles le programme incorrect a été exécuté, chaque exécution avec une séquence de test donnant lieu à une trace d'erreurs ; dans chaque cas, on précise le nombre de vecteurs de test soumis au programme ;

b) le nombre total d'erreurs initiales dans chaque trace d'erreurs ainsi que leur type ; ces erreurs peuvent résulter d'une ou plusieurs activations de la faute : le nombre d'activations qui ont provoqué au moins une erreur initiale[12] ainsi que le nombre d'erreurs créées (en moyenne) à chacune de ces activations sont indiqués ;

c) le nombre de flots d'erreurs dans chaque trace d'erreur ; lorsque ce nombre est inférieur au nombre d'erreurs initiales cela signifie que certaines de ces erreurs ne propagent pas[13]. Nous avons distingué les flots d'erreurs menant à une ou plusieurs défaillances (notés FD), et les flots d'erreurs ne contenant que des erreurs non détectées (notés FN). Le nombre d'erreurs contenu dans chaque flot est précisé : lorsque plusieurs flots, par exemple *m* flots, contiennent un même nombre *n* d'erreurs, on utilise la notation abrégée *n(m)*. Dans le cas particulier où une erreur initiale ne propage pas (pas de flot d'erreurs) mais affecte directement une variable de sortie (sans être annulée avant la fin du cycle d'exécution), la défaillance qui lui correspond est mentionnée ;

d) le nombre total d'erreurs enregistrées pour chaque trace.

---

12 Par construction, les bases de données ne contiennent aucune information relative aux activations de fautes qui n'ont pas généré d'erreur initiale.

13 Rappelons qu'un flot d'erreurs a été défini au chapitre II, comme étant une séquence d'erreurs due à la propagation d'une erreur initiale (cf. § II.3.5.2).

H correspond à la concaténation des 4 flots d'erreurs produits par G. En fait, l'erreur initiale créée par activation de H au 2ème cycle et qui est à l'origine du flot d'erreurs unique produit par H, propage durant les cycles suivants (3, 4 et 5) ; pour G, la même erreur initiale est recréée à chaque activation aux cycles 2 à 5.

Les fautes **J**, **K** et **L** avaient été révélées peu de fois par l'ensemble des 10 jeux de test statistique fonctionnel, et donc qualifiées de fautes subtiles. L'analyse des traces d'erreur montre qu'elles sont cependant souvent activées. Par exemple, la faute J est activée 72 fois : mais seulement deux erreurs initiales propagent et en créant peu d'erreurs (seulement 4 par flot) ; un seul des deux flots conduit à défaillance. Les fautes K et L créent également de nombreuses erreurs initiales ; mais ces erreurs doivent propager durant 5 cycles d'exécution successifs avant défaillance (cf. figure III.3).

III.4.2 Mutations

La base de données *BD-Mutations* contient 2272 enregistrements d'erreur répartis dans 24 traces d'erreurs. L'analyse de ces traces a permis d'identifier 164 flots d'erreurs dont 43 (soit 26%) contiennent au moins une erreur qui correspond à une défaillance ; les 121 autres flots (soit 74%) ne permettent pas de révéler la mutation présente. On remarque que ces deux pourcentages de flots (avec et sans défaillance) sont similaires à ceux déduits des 12 fautes réelles, qui étaient respectivement de 22% et 78%.

Le tableau III.4 récapitule les résultats des analyses des traces d'erreurs produites par les mutations, en indiquant les mêmes informations que dans le tableau III.3 (relatif aux fautes réelles). Le nombre de flots d'erreurs est égal au nombre d'erreurs initiales qui propagent, sauf dans le cas de la mutation M15 pour laquelle c'est une propagation par combinaison de deux erreurs initiales qui est à l'origine de chacun des 3 flots. Les mutations apparaissant en gras sont les 8 mutations sélectionnées sur des lignes de code éloignées de celles affectées par les fautes réelles (cf. § III.2.3).

Ces résultats montrent qu'une mutation peut générer autant de flots d'erreurs qu'une faute dite plus "complexe" (structurellement), et que ces flots d'erreurs peuvent contenir un nombre d'erreurs aussi grand que dans le cas des fautes réelles. Cependant, les mutations ne peuvent générer au plus que deux erreurs initiales par activation[14].

---

14 Seules les mutations sur les symboles ont pu générer deux erreurs initiales. Les autres mutations n'en ont généré qu'une.

(soit 22%) contiennent au moins une erreur qui correspond à une défaillance ; les 67 autres flots (soit 78%) ne permettent pas de révéler la faute présente.

Ces résultats amènent plusieurs commentaires. On constate qu'une faute peut être activée plusieurs fois au cours d'un cycle d'exécution, le nombre d'activations étant supérieur au nombre de vecteurs de test (par exemple, fautes D et E) : de telles fautes affectent des instructions dans le corps d'une boucle. Dans d'autres cas, seul un faible pourcentage d'erreurs initiales propagent (par exemple, 2 sur 16 dans le cas de la faute B) : les autres sont masquées ou annulées. Enfin, la complexité (nombre de flots d'erreurs et nombre d'enregistrements d'erreur par flot) de la trace d'erreurs associée à une faute donnée n'est pas liée au nombre de lignes de code affectées par la faute ; par exemple, considérons les fautes C et D, toutes deux étudiées avec la séquence T10 : les traces d'erreurs produites par C et D contiennent respectivement 3 et 10 flots d'erreurs qui représentent un total de 107 et 244 enregistrements d'erreur ; la trace produite par C est donc beaucoup moins "complexe", bien qu'elle soit relative à 6 cycles d'exécution (les 6 premiers vecteurs de T10) au lieu de 4 seulement pour D (les 4 premiers vecteurs de T10) ; or la faute C affecte 2 instructions alors que D n'en affecte qu'une (cf. tableau II.1).

Le comportement erroné produit par la faute **F** mérite d'être souligné, la faute F étant de type "instructions supplémentaires" (donc, non reproductible par une mutation dans le programme original). Suite à 25 activations, cette faute génère 13 flots d'erreurs dont un seul conduit à défaillance. Les erreurs créées doivent propager sur deux cycles d'exécution successifs avant détection.

Pour la majorité des jeux de test statistique fonctionnel disponibles (8 jeux sur 10), les fautes **G** et **H** avaient produit à chaque fois les mêmes défaillances (mêmes résultats incorrects). Pourtant, ces fautes n'affectent pas les mêmes instructions du code et ne sont pas localisées sur les mêmes fonctions. Dans le cadre de nos expériences, les fautes **G** et **H** ont été analysées avec la même séquence de test T7 de 5 vecteurs, et l'analyse de leurs traces d'erreurs sur les cinq cycles d'exécution nous a permis de comprendre la similitude des défaillances produites par ces deux fautes, bien qu'elles ne soient pas activées par les mêmes vecteurs de test. En effet, la faute H est activée dans une fonction d'initialisation exécutée au 1er et au 2ème cycles et chacune des 2 activations crée 5 erreurs initiales, mais aucune des 5 erreurs créées à la première activation ne propage ; la faute G est activée à chaque cycle à l'exception du premier, et chacune des 4 activations crée 4 erreurs initiales ; les erreurs initiales qui propagent dans les deux traces d'erreurs affectent la même variable, et le flot d'erreurs généré par

Enfin, des mutations peuvent être aussi difficiles à révéler que des fautes qualifiées de subtiles, c'est-à-dire produisant très peu de défaillances bien que très souvent activées. Par exemple, de façon similaire à la faute J, la mutation M19 est activée 71 fois : seulement deux erreurs initiales propagent et en créant peu d'erreurs (seulement 4 par flot) ; un seul des deux flots conduit à défaillance.

La comparaison des flots d'erreurs détaillée au paragraphe suivant permet de mieux apprécier la complexité des comportements erronés produits par les mutations sélectionnées.

III.4.3 Comparaison des flots d'erreurs

Cette analyse comparative a consisté à comparer chacun des flots d'erreurs produits par chaque mutation à chacun de ceux produits par les fautes réelles.

On considère qu'un flot d'erreurs **reproduit complètement** un autre flot lorsqu'ils sont identiques, à l'exception de l'erreur initiale (très spécifique à chaque faute) qui est à l'origine de chacun des deux flots : ces deux flots contiennent donc les mêmes erreurs qui sont liées par les mêmes relations de cause à effet (mêmes mécanismes de création, masquage et annulation). Un flot d'erreurs F1 **reproduit partiellement** un flot d'erreurs F2 lorsque le flot F1 constitue ou contient un sous-flot de F2 conduisant aux mêmes défaillances (dans le cas d'un sous-flot menant à défaillance) ou aux mêmes erreurs masquées ou annulées (dans le cas d'un sous-flot d'erreurs non détectées) à la fin de l'ensemble des cycles d'exécution associés à F1 et à F2.

Le tableau III.5 résume les résultats de ces comparaisons, en indiquant pour chaque faute réelle :

a) les séquences de test avec lesquelles les comportements erronés produits par la faute ont été analysés ;

b) les mutations qui ont complètement ou partiellement reproduit les flots d'erreurs produits par la faute. Pour chaque mutation, nous précisons :
   - la séquence de test qui a permis d'observer les flots reproduits,
   - le nombre de flots d'erreurs (FD ou FN) générés par la faute et reproduits partiellement ou complètement par la mutation, ainsi que le

Mutant	Séquence de test (# vecteurs)	# erreurs initiales : # activations (# erreurs/activation)	# flots d'erreurs (FD ou FN : # erreurs/flot)	# total erreurs
M1	T5 (7)	2 variables erronées : 2 (1)	2 défaillances	2
M2	T1 (6)	1 variable erronée : 1 (1)	1 défaillance	1
M3	T1 (6)	8 variables erronées : 8 (1)	1 FD : 40	47
M4	T3 (2)	28 branchements erronés : 28 (1)	14 FD : 17(3),14(6), 13(3) et 12(2) 14 FN : 9, 8(3), 7(9) et 6	300
M5	T7 (5)	2 variables erronées : 2 (1)	1 FD : 75	76
M6	T11 (5)	18 variables erronées : 18 (1)	1 FD : 42 4 FN : 4, 3(2) et 2	67
M7	T12 (7)	18 variables erronées : 18 (1)	1 FD : 86 11 FN : 31, 28, 22, 8, 7, 6(3), 3(2) et 2	218
M8	T3 (8)	20 variables erronées : 20 (1)	1 FD : 40 13 FN : 4(3), 3(5) et 2(5)	82
M9	T13 (7)	6 variables erronées : 6 (1)	1 FD : 18	23
M10	T2 (8)	3 variables erronées : 3 (1)	3 défaillances	3
M11	T4 (6)	8 variables erronées : 8 (1)	1 FD : 49 et 7 FN : 2(7)	63
M12	T6 (3)	1 variable erronée : 1 (1)	1 FD : 38	38
M13	T7 (5)	17 branchements erronés : 17 (1)	2 FD : 13(2) et 9 FN : 3(9)	59
M14	T1 (6)	16 variables erronées : 8 (2)	2 FD : 40 et 20	74
M15	T3 (9)	10 variables erronées : 5 (2)	1 FD : 71 2 FN : 22 et 6	103
M16	T5 (8)	20 variables erronées : 10 (2)	1 FD : 60 8 FN : 5(3), 4 et 2 (4)	97
M17	T7 (5)	2 variables erronées : 1 (2)	1 FD : 82	83
M18	T8 (4)	10 variables erronées : 10 (1)	1 FD : 47 9 FN : 41, 40, 39, 29, 28, 17, 16(2), 8	281
M19	T9 (4)	141 variables erronées : 71 (2)	1 FD : 4 et 1 FN : 4	147
M20	T3 (4)	5 branchements erronés : 5 (1)	3 FD : 58 et 19(2) 2 FN : 33 et 31	160
M21	T6 (2)	16 branchements erronés : 16 (1)	2 FD : 13 et 17 14 FN : 8, 7(3), 6(9) et 3	116
M22	T6 (2)	15 variables erronées : 15 (1)	2 FD : 18 et 14 2 FN : 9 et 4	56
M23	T13 (7)	12 variables erronées : 6 (2)	1 FD : 18	29
M24	T9 (4)	29 branchements erronés : 29 (1)	4 FD : 8(4) 25 FN : 5(20) et 3(5)	147

Tableau III.4 : Analyse des traces d'erreurs générées par les mutations (FD : flot d'erreurs conduisant à défaillance(s); FN : flot d'erreurs non détectées)

Faute : séquence(s) de test	Mutation : séquence de test	# flots d'erreurs reproduits par la mutation	# flots d'erreurs jamais reproduits par les mutations
A : T1, T2	M10 : T2	3 défaillances	20 défaillances
	M1 : T5	2 défaillances	
	M2 : T1	1 défaillance	
B : T1	M14 : T1	2 FD complètement	aucun
	M3 : T1	1 FD partiellement	
	M8 : T3	1 FD complètement	
C : T3	M15 : T3	1 FD et 1 FN complètement	1 FN
	M20 : T3	1 FN partiellement	
D : T3	M20 : T3	2 FD et 1 FN partiellement	2 FD et 5 FN
E : T4, T5	M11 : T4	1 FD et 7 FN complètement	aucun
	M16 : T5	1 FD et 9 FN complètement	
F : T6	M21 : T6	1 FD et 1 FN partiellement	11 FN
	M22 : T6	1 FD et 1 FN partiellement	
G : T7	M5 : T7	1 FD partiellement	3 défaillances
	M12 : T6	1 FD complètement	
	M13 : T7	2 FD partiellement	
	M17 : T7	3 FD et 1 FN partiellement	
H : T7	M5 : T7	1 FD partiellement	1 défaillance
	M12 : T6	1 FD partiellement	
	M13 : T7	1 FD partiellement	
	M17 : T7	1 FD complètement	
I : T8	M18 : T8	1 FD complètement et 5 FN partiellement	aucun
J : T9	M19 : T9	1 FD et 1 FN complètement	aucun
K : T10, T11	M6 : T11	1 FD et 4 FN complètement	1 FD et 2 FN
	M8 : T3	3 FN complètement	
L : T12	M7 : T12	1 FD et 11 FN complètement	aucun

Tableau III.5 : Similitudes entre les flots d'erreurs produits par les mutations et par les fautes réelles

III.5 CONCLUSIONS

nombre de défaillances sans création de flot d'erreurs[15] reproduites par la mutation ;

c) le nombre de flots d'erreurs et de défaillances (sans création de flot) non reproduits par l'ensemble des mutations étudiées.

Au total, 16 des 19 flots FD et 48 des 67 flots FN contenus dans *BD-Fautes* sont complètement ou partiellement reproduits par les mutations. En ce qui concerne les flots contenus dans *BD-Mutations*, 20 flots FD sur 43 et 64 flots FN sur 121 reproduisent complètement ou partiellement des flots produits par les fautes réelles.

Les 3 flots FD non reproduits par des mutations sont dus aux fautes D et K ; pourtant, ces fautes n'affectent chacune qu'une instruction du code source et sont similaires à des mutations (mais pas à celles étudiées). La faute D, en particulier, porte sur une expression conditionnelle et crée une erreur initiale de type branchement erroné (expression évaluée à faux au lieu de vrai dans le programme original) ; la mutation M4, localisée sur la même instruction que D, génère l'erreur initiale "inverse" (l'expression est évaluée à vrai au lieu de faux). Il n' y a, de ce fait, aucune similarité entre les flots générés par D et ceux générés par M4.

Il était impossible de sélectionner une mutation localisée sur les mêmes instructions que la faute F (de type "instructions supplémentaires") ; cependant 2 mutations reproduisent partiellement 2 des flots d'erreurs produits par F. M22, en particulier, reproduit le même flot FD à partir de la troisième erreur créée. Les 11 flots FN non reproduits par M21 et M22 sont ceux contenant le moins d'erreurs. L'analyse comparative des bases de données montre que toutes les erreurs générées par la faute F apparaissent au moins une fois dans la base de données des mutations.

---

15 Il s'agit des défaillances produites directement par des erreurs initiales affectant les variables de sortie, et qui ont été mentionnées dans le tableau III.3.

CHAPITRE IV

MODÉLISATION DES COMPORTEMENTS ERRONÉS

IV.1 INTRODUCTION

Les études expérimentales présentées au chapitre III nous ont permis d'observer des comportements erronés variés et relativement complexes étant donnée la taille (nombre d'erreurs) des flots d'erreurs associés. Ce chapitre a pour objectif de modéliser ces comportements erronés à l'aide d'un formalisme adapté et notamment d'expliquer les différents mécanismes de création, de masquage et d'annulation d'erreurs par l'étude des dépendances du programme. Les dépendances d'un programme sont des relations établies entre les instructions de ce programme ; leur analyse permet de prévoir, dans une certaine mesure, le comportement du programme en cours d'exécution. Il existe deux classes principales de dépendances : les dépendances structurelles relatives au graphe de contrôle du programme et les dépendances entre variables relatives au flot de données du programme. Ces dépendances sont définies à partir de représentations du programme définies au paragraphe IV.2.

Le paragraphe IV.3 présente les relations de dépendances telles que définies dans la littérature et le formalisme que nous avons déterminé pour modéliser les comportements erronés.

Les paragraphes IV.4 à IV.7 décrivent les mécanismes de création, d'annulation et de masquage d'erreurs à l'aide des relations de dépendances.

Le paragraphe IV.8 s'intéresse aux relations de dépendances liant plusieurs fonctions d'un même programme.

Cinq des 8 mutations sélectionnées sur des lignes de code non affectées par les fautes réelles reproduisent cependant partiellement certains flots d'erreurs (FD ou FN) dus à des fautes réelles : ce sont les mutations M8, M13, M20, M21 et M22 notées en caractères gras dans le tableau III.5. Ceci peut s'expliquer par le fait qu'elles sont analysées avec la même séquence de test, sauf dans le cas de M8.

Les flots d'erreurs générés par les mutations M4, M9, M23 et M24 ne présentent aucune similitude avec les flots d'erreurs dus aux fautes réelles : M9, M23 et M24 sont les 3 autres mutations sélectionnées sur des lignes de code non affectées par les fautes réelles, M4 affecte la même ligne que D (voir commentaire ci-dessus). Ces 4 mutations génèrent au total 81 des 140 erreurs spécifiques à la base de données réduite *BD-Mr* relative aux mutations, soit 58% des 40% indiqués dans la figure III.2.b et 23% du nombre total d'erreurs dans *BD-Mr*.

Seules les mutations M7, M11, M14, M16 et M19 reproduisent complètement l'ensemble des flots d'erreurs générés par des fautes réelles, et donc ont les mêmes traces d'erreurs hormis les erreurs initiales. Cela est possible car les fautes réelles correspondantes (L, E, B et J) créent au plus deux erreurs initiales à chaque activation : elles affectent 1 ou 2 instructions mais sont d'origines diverses, notamment B et E résultent d'une mauvaise compréhension de la spécification.

Pour des fautes qui créent plusieurs erreurs initiales à chaque activation, on peut noter que plusieurs mutations peuvent, en reproduisant chacune des flots différents, simuler les comportements de ces fautes (c'est le cas, par exemple, pour les fautes G et H). Ceci est vrai à condition, toutefois, que les flots d'erreurs n'interagissent pas entre eux, par propagation combinée de plusieurs erreurs : mais ce type d'interaction n'a pas été observé lors de nos études expérimentales ; les cas de propagation combinée de plusieurs erreurs que nous avons pu identifier étaient localisés à l'intérieur d'un même flot. Nous avons cependant observé, à la fois pour des fautes réelles et pour des mutations, des cas où une erreur initiale propageait par combinaison avec une erreur non initiale et donc contenue dans un flot précédemment créé : le flot né de l'erreur initiale s'intégre alors au flot déjà existant.

Pour conclure, il est important de souligner que les comportements erronés dus aux mutations étudiées reproduisent de manière satisfaisante les comportements erronés produits par, d'une part les fautes subtiles J, K et L, et d'autre part, les fautes de type "plusieurs instructions manquantes" (B, C et H).

2. Il existe au plus un arc du nœud m au nœud n, pour tout couple (m,n) appartenant à A_G.
3. Chaque nœud de G apparaît dans au moins un des chemins de n_I à n_F.

Selon l'utilisation faite du graphe, un nœud peut représenter soit un bloc [Rapps & Weyuker 1985], soit une instruction [Thompson 1991] ; nous verrons sur l'exemple présenté dans le paragraphe suivant que dans le cadre de nos analyses, il est nécessaire de considérer chaque instruction comme un nœud distinct du graphe. Les arcs indiquent le flot de contrôle entre les nœuds ; par exemple, si chaque nœud du graphe représente un bloc alors deux nœuds m et n sont liés par un arc si la dernière instruction du bloc m est une instruction de branchement (boucle ou instruction conditionnelle) dont l'une des cibles est la première instruction du nœud n.

Un nœud *d* d'un graphe de contrôle domine le nœud *n*, ce que l'on écrit *d dom n*, si tout chemin partant du nœud initial du graphe de contrôle et arrivant à *n* passe par *d* [Aho et al. 1990]. Par définition, chaque nœud se domine lui-même. Un nœud *m* d'un graphe de contrôle post-domine le nœud *n*, ce que l'on écrit *m post-dom n*, si tout chemin partant du nœud *n* au nœud final du graphe de contrôle passe par *m* [Ferrante et al. 1987]. Par définition, le nœud initial d'un graphe de contrôle domine tous les autres et le nœud final post-domine tous les autres.

Nous utiliserons la notion de nœud *dominant* ou *post-dominant* un autre nœud pour définir les relations de dépendances liées au flot de contrôle.

IV.2.2 Graphe définition-utilisation

Un graphe définition-utilisation se construit à partir du graphe de contrôle du programme que l'on enrichit par des informations relatives au flot des données. Les définitions données dans ce paragraphe sont issues de [Rapps & Weyuker 1985, Ferrante et al. 1987, Aho et al. 1990, Thompson 1991, Johnson & Pingali 1993, Ferguson & Korel 1996].

A chaque nœud n d'un graphe de contrôle sont associés deux ensembles : $D(n)$ qui représente l'ensemble des variables définies au nœud n et $U(n)$ qui représente l'ensemble des variables utilisées (ou référencées) au nœud n. Chaque occurrence de variable est donc identifiée comme étant soit une définition, soit une utilisation de variable. Une **définition** d'une variable x est un nœud qui affecte une valeur à x. De la même façon, une **utilisation** d'une variable x est un nœud qui fait référence à la valeur

phénomènes observés est donc nécessaire. Elle fait l'objet du chapitre IV où nous tentons d'expliquer les différents mécanismes de création, de masquage et d'annulation d'erreurs observés, à l'aide d'un formalisme basé sur l'analyse des dépendances d'un programme. L'étude détaillée des flots d'erreurs, que nous venons de présenter, nous permet de penser que les relations de cause à effet entre erreurs s'expliquent par les dépendances entre les instructions du programme, dépendances liées au flot de contrôle ou au flot de données.

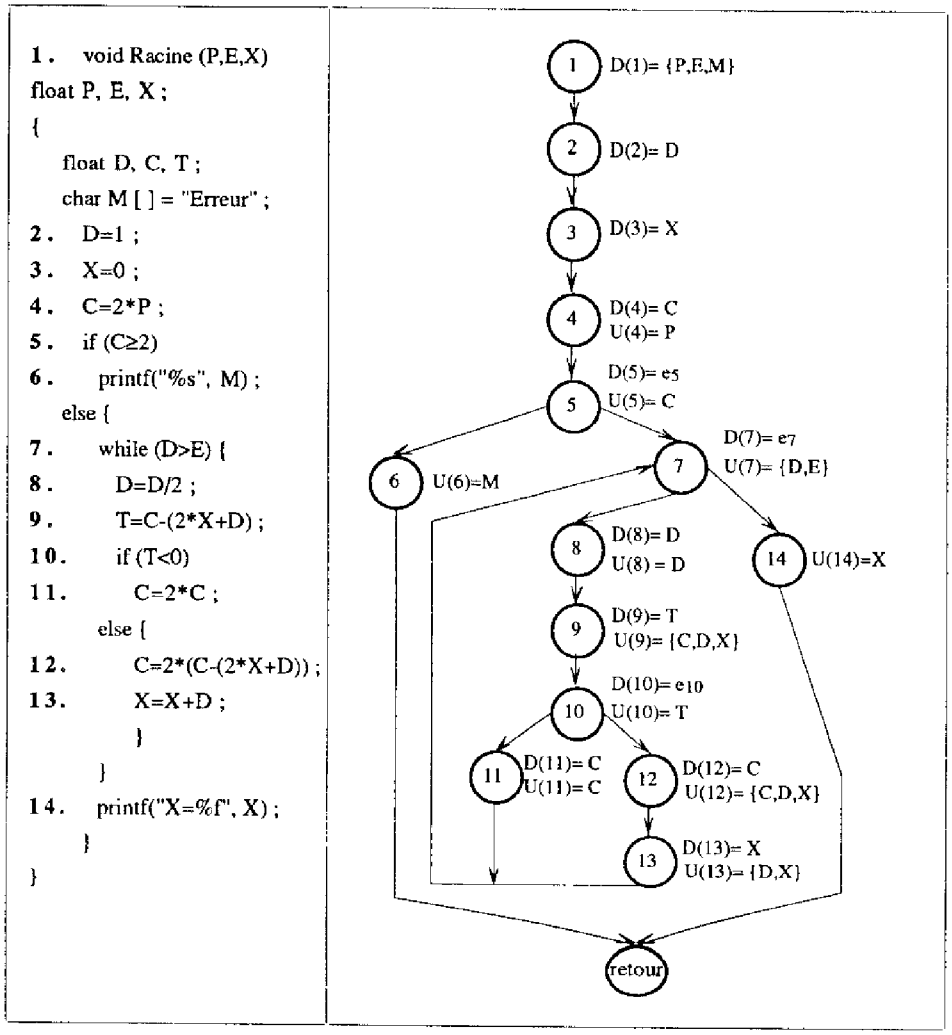


Figure IV.2 : Graphe définition-utilisation du programme Racine (chaque nœud est associé à une instruction du programme)

Dans la figure IV.1, chaque nœud du graphe représente un bloc du programme et on ne s'intéresse qu'au flot des données entre blocs ; pour chaque nœud n du graphe, D(n) représente l'ensemble des variables pour lesquelles n contient une définition globale (c'est-à-dire susceptibles d'être utilisées par un autre nœud), U(n) représente

IV.2 REPRÉSENTATIONS STRUCTURELLES D'UN PROGRAMME

L'analyse des dépendances d'un programme associe l'analyse du flot de contrôle d'un programme avec l'analyse du flot de données qui s'intéresse aux occurrences des variables à l'intérieur du programme. Les relations de dépendances sont définies à partir de l'étude d'un graphe définition-utilisation.

IV.2.1 Graphe de contrôle

Un programme est une séquence finie d'instructions ; nous présumons que les programmes étudiés ne comportent pas de branchement inconditionnel (de type "go to"). Nous distinguons donc, les types d'instruction suivants :

a) les **instructions simples** : instruction d'affectation, instruction d'initialisation, instruction d'entrées/sorties, appel de fonction.

b) les **instructions conditionnelles** : de type "if-then-else" ou "switch" ;

c) les **boucles** : de type "repeat-until", "do while", "while" ou "for".

Les instructions de type b) ou c) contenant des prédicats (conditions de branchement) sont dénommées instructions de contrôle ; l'évaluation des expressions conditionnelles correspondantes a un impact sur le flot de contrôle.

Un programme peut être décomposé en un ensemble fini de blocs ; un bloc étant un ensemble ordonné d'instructions $b = \langle i_1, ..., i_n \rangle$ toujours exécutées en séquence, c'est-à-dire telles que : si $n>1$, pour $k = 2, ..., n$, i_k est l'unique successeur de i_{k-1} et i_{k-1} est l'unique prédécesseur de i_k.

La structure d'un programme est représentée par son *graphe de contrôle* $G = (N_G, A_G)$ qui est un graphe orienté où N_G est l'ensemble des nœuds du graphe et A_G, l'ensemble des arcs (sous-ensemble de $N_G \times N_G$). Un arc (m, n) est un arc orienté allant de m vers n. On dit que n est un *successeur* de m et que m est un *prédécesseur* de n.

Un graphe de contrôle satisfait les conditions suivantes :

1. G comprend un nœud initial n_I qui n'a pas de prédécesseur, et un nœud final n_F qui n'a pas de successeur.

- n est une déclaration de variable comportant une initialisation et x est la variable déclarée et initialisée ;
- n est une instruction d'affectation et x figure dans la partie gauche de cette instruction ;
- n est une instruction de lecture et x est un paramètre de la fonction de lecture appelée ;
- n est une instruction comportant une opération d'incrémentation ou de décrémentation de la variable x ;
- n est une instruction de contrôle de type "for" et x est l'indice de boucle.

Dans le cas où n est une instruction de contrôle de type "if-then-else", "switch", "while", "for" ou "do while", nous considérons que l'expression conditionnelle C évaluée par cette instruction appartient également à $D(n)$.

Une variable x appartient à $U(n)$ dans les cas suivants :

- n est une instruction de contrôle et la variable x apparaît dans l'expression conditionnelle correspondante ("p-use") ;
- n est une instruction de contrôle de type "for" et la variable x est l'indice de boucle ("c-use") ;
- n est ou contient un appel de fonction et la variable x est un des arguments effectifs de cette fonction ;
- n est une instruction simple, x est utilisée dans un calcul ("c-use") :
  - n est une instruction d'affectation et x figure dans l'expression droite de cette instruction ;
  - n est une instruction d'écriture et x est un paramètre de la fonction de sortie appelée ;
  - n est une instruction comportant une opération d'incrémentation ou de décrémentation de la variable x.

IV.3 RELATIONS DE DÉPENDANCES

Deux types de dépendances sont considérées dans la littérature : les **dépendances liées au flot de contrôle** et les **dépendances liées au flot de données**. Elles sont présentées dans les paragraphes suivants. Outre la définition de

de x. On distingue deux types d'utilisation : une utilisation dans un calcul ("c-use") qui affecte directement le calcul d'une variable, et une utilisation dans un prédicat ("p-use") qui affecte le flot de contrôle du programme [Rapps & Weyuker 1985].

Le **graphe définition-utilisation** est un quadruplet $G = (G, V(G), D, U)$ où $G = (N_G, A_G)$ est un graphe de contrôle, V(G) est un ensemble fini de symboles appelés variables et $D : N_G \rightarrow P(V(G))$, $U : N_G \rightarrow P(V(G))$ sont des fonctions. D et U associent respectivement à chaque nœud n de N_G, l'ensemble $D(n)$ des variables définies au nœud n et l'ensemble $U(n)$ des variables utilisées au nœud n.

Comme dans le cas du graphe de contrôle, un nœud peut représenter soit un bloc, soit une instruction, selon l'utilisation faite du graphe. L'exemple ci-dessous illustre les différences entre ces deux types de représentation ; il s'agit de deux graphes définition-utilisation d'une même procédure (écrite en langage C) qui calcule la racine carrée de P ($0 \leq P < 1$) avec la précision E ($0 < E \leq 1$), le résultat étant mémorisé dans la variable X.

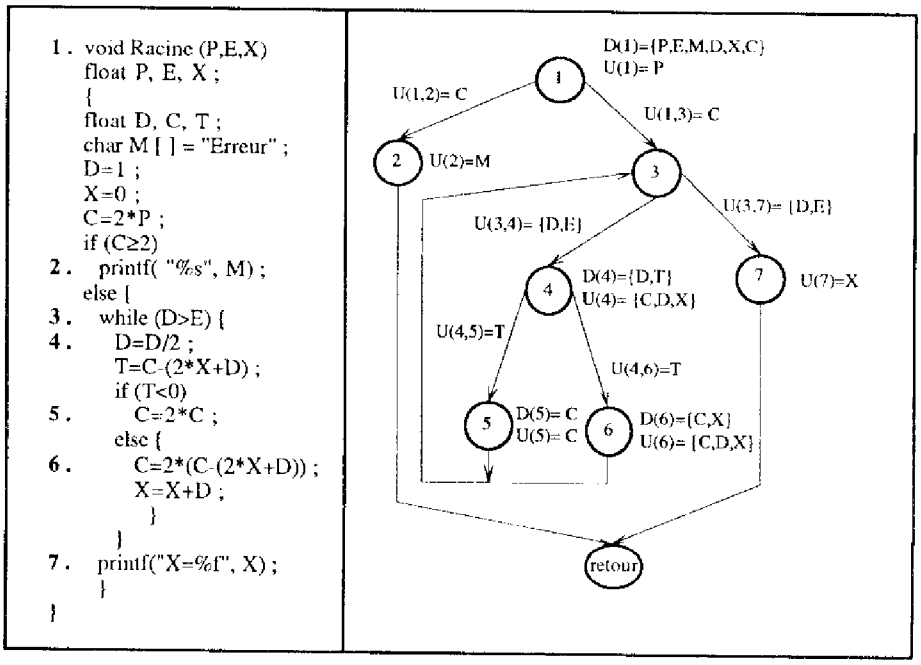


Figure IV.1 : Graphe définition-utilisation du programme Racine (chaque nœud est associé à un bloc du programme)

instructions dans le corps de la boucle et l'instruction de contrôle elle-même ; par exemple dans la figure IV.2, les nœuds 7, 8, 9 et 10 sont dépendants du nœud 7 par le contrôle.

Les dépendances de contrôle telles que définies ici sont équivalentes à celles définies par Podgurski et Clarke sous le terme de "strong control dependence" [Podgurski & Clarke 1990].

Structure **"if -then"** :
if (e_1)
{ v_1 ;
.... ;
v_n ; }
w ;
	Structure **"if-then-else"** :
	if (e_1)
	{ v_1 ;
	... ;
	v_i ; }
	else
	{ v_k ;
	... ;
	v_n ; }
	w ;
		Structure **"switch"** :
		switch (e_1)
		{
		case c_1 : v_1 ; ... ; v_i ;
		case c_2 : v_j ; ... ; v_k ;
		... ;
		case c_p : v_m ; ... ; v_n ;
		}
		w ;

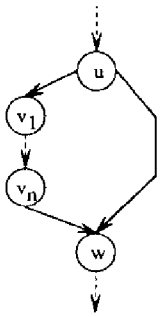
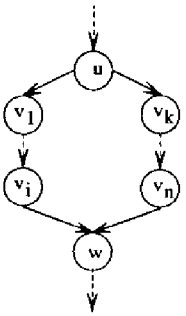
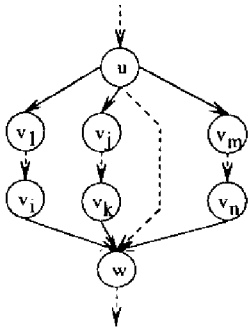
		
Soit u, le nœud de contrôle évaluant e_1 ; v_1 et v_n sont dépendants par le contrôle de u.	Soit u, le nœud de contrôle évaluant e_1 ; les nœuds v_1 à v_i et v_k à v_n sont dépendants par le contrôle de u.	Soit u, le nœud de contrôle évaluant e_1 ; les nœuds v_1 à v_i, v_j à v_k et v_m à v_n sont dépendants par le contrôle de u.

Tableau IV.1 : Dépendances par le contrôle et conditions

l'ensemble des variables utilisées de manière globale au nœud n dans un calcul (c'est-à-dire dont la définition n'est pas locale à ce nœud) ; pour chaque arc allant d'un nœud n à un nœud m, U(n, m) représente l'ensemble des variables utilisées dans un prédicat par l'arc (m, n).

Dans la figure IV.2, chaque nœud du graphe représente une instruction du programme ; pour chaque nœud n du graphe, D(n) représente la variable définie ou l'expression conditionnelle évaluée au nœud n et U(n) représente l'ensemble des variables utilisées au nœud n soit dans un calcul, soit dans un prédicat (dans ce cas, cette dernière information n'est pas reportée sur l'arc comme pour la figure IV.1). Les termes e_5, e_7, e_{10} désignent les expressions conditionnelles évaluées respectivement aux nœuds 5, 7 et 10.

La figure IV.2 indique clairement que la valeur de la variable D, utilisée au nœud 9 pour définir la variable T, a été définie à l'instruction précédente : cette information n'est pas visible sur la figure IV.1 au niveau du bloc 4. La représentation du graphe définition-utilisation au niveau des blocs ne permet pas de connaître précisément les relations de dépendances entre les variables du programme et donc entre les erreurs générées. Dans le cadre de nos études, pour analyser le flux des données et la propagation d'une erreur à chaque instant d'exécution — c'est-à-dire à la fin de l'exécution de chaque instruction, comme nous l'avons précisé au chapitre II (§ II.2.1), il est nécessaire d'utiliser le même niveau de détail pour la représentation du programme et donc d'associer chaque nœud du graphe définition-utilisation à une instruction, comme dans [Thompson 1991].

Dans le reste de ce chapitre, nous considérons donc que chaque nœud du graphe (graphe de contrôle ou graphe définition-utilisation) est associé à une instruction du programme.

Une variable x appartient à $D(n)$ dans les cas suivants [16] :

- n est le nœud initial et x est soit une variable globale, soit un paramètre d'entrée du programme ou de la procédure considérée ;

16 Cette liste n'est pas exhaustive : elle est déduite de l'étude des programmes analysés qui sont écrits en langage C et pour lesquels l'utilisation de certaines instructions ("go to") ou de certaines opérandes (pointeurs) n'est pas autorisée.

l'exécution puisque dans ce cas-là, les séquences d'instructions exécutées par chacun des programmes peuvent être de taille variable ; cependant, la comparaison des traces d'exécution se re-synchronise dès que chacun des chemins exécutés par les programmes correct et incorrect atteint le nœud post-dominant de l'instruction de contrôle affectée par l'erreur de branchement. C'est la solution qui avait été aussi retenue dans [Murrill 1991] lors de l'analyse dynamique de flots d'erreurs.

IV.3.2 Dépendances liées au flot de données

Soient u et $v \in N_G$, v est **dépendant** de u **par les données** ssi il existe une variable x définie au nœud u et utilisée au nœud v et ssi il existe un chemin C du nœud u au nœud v ne contenant pas d'autre définition de x que u.

Les dépendances liées au flot de données sont également utilisées lors du développement de calculateurs parallèles [Padua & Wolfe 1986] c'est-à-dire pour déterminer quand deux instructions peuvent être exécutées en parallèle.

Une **définition** d d'une variable x est **visible** par un nœud u s'il existe un chemin du nœud suivant immédiatement d, au nœud u, tel que x ne soit pas redéfinie le long de ce chemin ; u est alors une **utilisation atteignable** de x depuis d. Si une définition d d'une variable x est visible depuis un nœud u, alors d peut être le nœud où la valeur de x utilisée en u a été définie en dernier.

Pour chaque nœud u et chaque variable x utilisée au nœud u, nous pouvons définir l'ensemble des définitions visibles de x depuis u : Def-vis(x,u). De la même façon nous pouvons définir l'ensemble des utilisations atteignables u d'une variable x définie en d : Util-att(x,d).

IV.3.3 Associations et chaînes de dépendances

A partir des relations définies aux paragraphes précédents, nous avons défini, pour les besoins de nos analyses, les notions d'associations et de chaînes de dépendances.

critères de sélection des entrées de test [Rapps & Weyuker 1985], l'utilisation de ces relations de dépendances a fait l'objet d'études dans le but de localiser des fautes dans un programme [Agrawal et al. 1991, Agrawal et al. 1995, DeMillo et al. 1995b], ou de localiser les instructions du programme affectées par la propagation d'erreur (suite à l'activation d'une faute sur une instruction donnée) [Podgurski & Clarke 1990], ou de définir des "slices"[17] dans le code source d'un programme [Weiser 1981, Weiser 1982, Horwitz et al. 1988, Korel & Yalamanchili 1994] ou encore pour comprendre comment une erreur propage lors de l'exécution d'un programme [Thompson 1991, Thompson et al. 1993].

IV.3.1 Dépendances liées au flot de contrôle

Les dépendances liées au flot de contrôle sont utilisées pour modéliser l'effet des branchements conditionnels sur le comportement d'un programme. Elles sont définies à partir des notions de nœud *dominant* ou *post-dominant* un autre nœud, telles que définies au paragraphe IV.2.1.

Nous définissons également le *post-dominant immédiat* d'un nœud de contrôle u, comme le premier nœud où tous les chemins passant par u se rejoignent.

Soient u et $v \in N_G$, on dit que v est **dépendant de u par le contrôle** ssi il existe un chemin C de u à v, tel que C ne contienne pas le post-dominant immédiat de u. Si v est dépendant par le contrôle de u alors u (qui est un nœud de contrôle) a au moins deux successeurs ; seule l'exécution d'une des branches conditionnées par le résultat de u conduira à l'exécution de v .

Les tableaux IV.1 et IV.2 indiquent pour chaque type de structure de contrôle (conditions ou boucles), le graphe de contrôle correspondant et les dépendances liées au flot de contrôle.

Les instructions qui sont dépendantes par le contrôle d'une instruction conditionnelle sont les instructions dans le corps de l'instruction de contrôle ; par exemple dans la figure IV.2, les nœuds 11, 12 et 13 sont dépendants du nœud 10 par le contrôle. Les instructions qui sont dépendantes par le contrôle d'une boucle sont les

---

17 Il s'agit de la technique du "program slicing" utilisée pour élaborer des systèmes d'aide à la mise au point de programmes.

Les associations de dépendances s'associent les unes aux autres formant ainsi des *chaînes de dépendances* qui modélisent le comportement du programme pendant l'exécution.

Soit G un graphe définition-utilisation, composé des nœuds u_1, ..., u_n et v_1, ..., v_n, et des variables ou prédicats x_1, ..., x_n définis ou utilisés aux différents nœuds du graphe. Une **chaîne de dépendances** est une séquence d'associations de la forme (u_1, v_1, x_1) ... $(u_{i-1}, v_{i-1}, x_{i-1})$ (u_i, v_i, x_i) telle que deux associations se succèdent dans la chaîne, par exemple $(u_{i-1}, v_{i-1}, x_{i-1})$ (u_i, v_i, x_i) si $v_{i-1} = u_i$ et si :

- soit la variable x_{i-1} est utilisée pour définir la variable ou le prédicat x_i ;
- soit l'évaluation de la condition x_{i-1} conduit à exécuter le nœud u_i qui définit la variable ou le prédicat x_i.

Il existe, pour un même chemin d'exécution, plusieurs chaînes de dépendances permettant de modéliser le comportement du programme en exécution ; en effet un nœud peut être relié à plusieurs autres nœuds exécutés par le même chemin par différentes associations de dépendances. A l'inverse, une chaîne de dépendances peut être exécutée par plusieurs chemins d'exécution.

Dans la littérature, des notions similaires aux associations et aux chaînes de dépendances que nous avons définies, ont été utilisées dans le cadre de travaux sur le test. Dans [Frankl & Weyuker 1988], une notion proche des associations de dépendances est utilisée comme critère de test basé sur le flot de données. Dans [Thompson 1991], une autre notion, proche des chaînes de dépendances (appelée "information flow chain"), est utilisée pour représenter la propagation d'une erreur depuis un nœud incorrect jusqu'à l'occurrence d'une défaillance.

Il existe également d'autres types de représentations des relations de dépendances dans un programme. Le plus connu est le graphe de dépendances du programme défini et utilisé pour les techniques d'optimisation de code [Ferrante et al. 1987] : le graphe de dépendances du programme est une représentation graphique rendant explicites les dépendances par le contrôle et les dépendances par les données à l'aide d'arcs de type différent.

Le paragraphe suivant illustre les notions d'associations et de chaînes de dépendances sur l'exemple du programme Racine présenté au paragraphe IV.2.2.

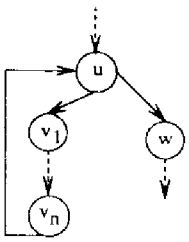
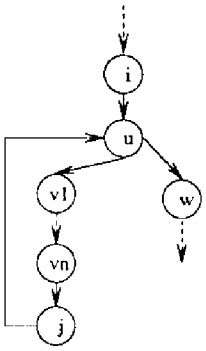
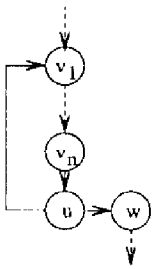
Structure **"while"** : while (e_1) { v_1 ; ... ; v_n ; } w ;	Structure **"for"** : for (e_1 ; e_2; e_3) { v_1 ; ... ; v_n ; } w ;	Structure **"do - while"** : do { v_1 ; ... ; v_n ; } while (e_1) w ;
		
Soit u, le nœud de contrôle évaluant e_1 ; u, v_1 à v_n sont dépendants par le contrôle de u. w est le post-dominant immédiat de u.	Soit i, le nœud associé à e_1 (initialisation), soit u, le nœud de contrôle évaluant e_2 et soit j, le nœud associé à e_3 (incrémentation) ; u, v_1 à v_n et j sont dépendants par le contrôle de u.	Soit u, le nœud de contrôle évaluant e_1 ; u, v_1 à v_n sont dépendants par le contrôle de u.

Tableau IV.2 : Dépendances par le contrôle et boucles

La notion de nœud *post-dominant* est également utilisée pour traiter le problème de re-synchronisation des traces d'exécution des programmes correct et incorrect évoqué aux chapitres II et III. Lorsque, suite à une erreur de branchement, les deux programmes n'exécutent pas simultanément les mêmes instructions, il est difficile d'établir un ensemble des états de données erronés unique[18] à chaque instant de

[18] Les erreurs identifiées peuvent dépendre de l'ordre dans lequel on compare les instructions exécutées par le programme incorrect et celles exécutées par le programme correct. Prenons, par exemple, le cas d'une variable qui est définie avec la même valeur par la branche exécutée par le programme correct et par celle exécutée par le programme incorrect ; on peut dans un cas, enregistrer une erreur sur cette variable (dès qu'on observe la définition dans la trace du programme incorrect) qui est ensuite annulée (suite à l'observation de la définition dans la trace du programme correct) ou dans un autre cas, ne pas enregistrer d'erreur.

IV.3.4 Notations

Les paragraphes IV.4 à IV.7 décrivent les différents mécanismes de création, d'annulation et de masquage d'erreurs observés lors des études expérimentales.

Nous adopterons les notations suivantes :

- P_c, désigne le programme original (considéré comme correct),
- P_i, désigne le programme incorrect (c'est-à-dire contenant la faute),
- G_c, désigne le graphe définition-utilisation du programme correct,
- G_i, désigne le graphe définition-utilisation du programme incorrect,
- A_c, désigne l'ensemble des associations de dépendances du programme correct,
- A_i, désigne l'ensemble des associations de dépendances du programme incorrect,
- $D_c(v)$ et $U_c(v)$ désignent l'ensemble des variables définies et l'ensemble des variables utilisées au nœud v du programme correct,
- $D_i(v)$ et $U_i(v)$ désignent l'ensemble des variables définies et l'ensemble des variables utilisées au nœud v du programme incorrect,
- par extension : $D(v) = D_c(v) \cap D_i(v)$ et $U(v) = U_c(v) \cap U_i(v)$,
- exp-dte$(v)_c$ désigne, lorsque v est une instruction d'affectation, l'expression à droite du signe d'affectation dans le programme correct,
- exp-dte$(v)_i$ désigne, lorsque v est une instruction d'affectation, l'expression à droite du signe d'affectation dans le programme incorrect,
- Def-vis(x,v) désignent l'ensemble des définitions visibles de la variable x au nœud v,
- Util-att(x,v) désignent l'ensemble des utilisations atteignables de la variable x depuis le nœud v,
- Soient une variable x et un nœud v du graphe de contrôle, les valeurs suivantes dépendent du vecteur d'entrée :
  - $def_i(x, v)$ désigne la valeur de la variable x définie au nœud v lors de l'exécution du programme incorrect,
  - $def_c(x, v)$ désigne la valeur de la variable x définie au nœud v lors de l'exécution du programme correct,

IV.3.3.1 Définitions

Une association de dépendances est un triplet **(u, v, x)** représentant les dépendances entre deux nœuds u et v, tel que :

a) soit v est dépendant de u par les données et x est alors une variable définie en u et utilisée en v ($x \in D(u)$ et $x \in U(v)$) ;

b) soit v est dépendant de u par le contrôle et x est un prédicat évalué en u (qui est alors une instruction de contrôle) et dont l'évaluation a conduit à l'exécution de v.

Chaque triplet (u, v, x) est identifié à partir de l'analyse du graphe définition-utilisation.

Les associations[19] qui nous intéressent pour l'étude des comportements erronés sont les **associations exécutables** c'est-à-dire celles qui peuvent être utilisées par un chemin complet exécutable. En effet, une erreur est un concept dynamique associé à une exécution d'un programme et la propagation d'une erreur ne peut s'étudier qu'au travers des associations exécutables du programme.

Il est à noter que l'on peut identifier des associations du type (u, u, x) dans le cas où une instruction u, localisée dans le corps d'une boucle, utilise et définit une même variable x ; x est alors définie au nœud u lors de l'itération i de la boucle et utilisée à ce même nœud lors de l'itération i+j ($j \geq 1$) de cette même boucle.

Nous pouvons ainsi constituer pour chaque programme un ensemble d'associations de dépendances exécutables *A*. A partir de cet ensemble *A*, nous pouvons déduire l'ensemble des définitions visibles d'une variable x utilisée dans un nœud u et l'ensemble des utilisations atteignables d'une variable x définie au nœud d :

$\text{Def-vis}(x,u) = \{d_i \in N_G / (d_i, u, x) \in A\}$

$\text{Util-att}(x,d) = \{u_i \in N_G / (d, u_i, x) \in A\}$

---

19 Pour alléger la rédaction, le terme "association" sera utilisé à la place de la formulation "association de dépendances" dans la suite de ce chapitre.

IV.4.1 Création d'une erreur de type variable erronée

Nous supposons que le nœud v est exécuté dans l'un au moins des programmes correct ou incorrect et qu'il définit une variable x : le nœud v peut être une instruction d'affectation, d'initialisation ou d'entrées/sorties. Une erreur initiale est créée sur la variable x si une faute affecte le nœud v. Plusieurs cas, résumés dans le tableau IV.3, peuvent être considérés suite à l'exécution du nœud v :

a) La valeur affectée à la variable x n'est pas la même dans P_i et P_c (cas 1 du tableau IV.3) : $D_c(v) = D_i(v) = \{x\}$ et $def_i(x, v) \neq def_c(x, v)$.

b) Une valeur est affectée à x dans P_c et non dans P_i ; deux cas peuvent se présenter

- la variable définie est incorrecte (cas 2 du tableau IV.3) : $D_c(v) = \{x\}$, $D_i(v) = \{x'\}$ avec $x \neq x'$ et $def_c(x, v) \neq vis_i(x, v)$ ou $def_i(x', v) \neq vis_c(x', v)$.
- la variable x est définie dans P_c et non dans P_i (cas 3 du tableau IV.3) : $x \in D_c(v)$ et $x \notin D_i(v)$.

c) Une valeur est affectée à x dans P_i et non dans P_c ; la variable x est définie dans P_i et non dans P_c (cas 4 du tableau IV.3) : $x \in D_i(v)$ et $x \notin D_c(v)$.

Le tableau IV.3 indique quelle est l'erreur initiale créée, caractérisée par la variable sur laquelle elle porte et par le couple {valeur correcte, valeur erronée}, et quelles sont les conditions relatives aux dépendances du programme qui permettent de conduir à la création de cette erreur. Ces conditions peuvent ensuite être reliées à différents types de faute.

Les différents cas sont commentés ci-après en fonction du type du nœud v et de la nature de la faute pouvant affecter ce nœud v. Puisqu'il s'avère impossible de définir un modèle de faute logicielle, il paraît également impossible de considérer ici tous les types de fautes. Les exemples commentés ci-après ont donc été observés lors de nos expérimentations sur des cas réels ou sont déduits de ces observations par extrapolation.

IV.3.3.2 Application au programme Racine

Les associations de dépendances associées au programme Racine et définies à partir du graphe définition-utilisation de la figure IV.2 sont les suivantes et forment l'ensemble A_{racine} :

(1, 4, P)	(1, 7, E)	(2, 7, D)	(2, 8, D)	(3, 9, X)	(3, 12, X)
(3, 13, X)	(3, 14, X)	(4, 5, C)	(4, 9, C)	(4, 11, C)	(4, 12, C)
(5, 6, e_5)	(5, 7, e_5)	(7, 8, e_7)	(7, 9, e_7)	(7, 10, e_7)	(7, 7, e_7)
(8, 8, D)	(8, 9, D)	(8, 12, D)	(8, 13, D)	(8, 7, D)	(9, 10, T)
(10, 11, e_{10})	(10, 12, e_{10})	(10, 13, e_{10})	(11, 9, C)	(11, 11,C)	(11, 12, C)
(12, 11, C)	(12, 12, C)	(13, 9, X)	(13, 12, X)	(13, 13, X)	(13, 14, X)

Les expressions conditionnelles sont désignées par les termes e_u, précisant leur localisation au nœud u. Il est à noter que les relations de dépendance telles que (8, 8, D) ou (13, 12, X) sont dues aux itérations de la boucle "while". En effet, la définition de la variable D au nœud 8 lors de l'itération i de la boucle "while" est utilisée à ce même nœud lors de l'itération suivante i+1 ; de même, la définition de la variable X au nœud 13, à l'itération i peut être utilisée au nœud 12 lors de l'une des itérations suivantes i+j (j ≥ 1) puisque la branche 12-13 n'est pas forcément exécutée à chaque itération.

Ces associations se lient les unes aux autres pour former des chaînes de dépendances. Par exemple, lors de l'exécution du chemin 1-2-3-4-5-7-8-9-10-11-7-8-9-10-12-13-7-14, les chaînes de dépendances suivantes sont exécutées (liste non exhaustive) :

CD1 = (1, 4, P) (4, 5, C) (5, 7, e_5) (7, 9, e_7) (9, 10, T) (10, 11, e_{10}) (11, 9, C) (9, 10, T) (10, 12, e_{10}) (12, 12, C)

CD2 = (3, 9, X) (9, 10, T) (10, 11, e_{10}) (11, 9, C) (9, 10, T) (10, 12, e_{10}) (12, 12, C)

CD3 = (3, 9, X) (9, 10, T) (10, 13, e_{10}) (13, 14, X)

CD4 = (2, 7, D) (7, 8, e_7) (8, 7, D) (7, 8, e_7) (8, 9, D) (9, 10, T) (10, 13, e_{10}) (13, 14, X)

CD5 = (4, 5, C) (5, 7, e_5) (7, 8, e_7) (8, 9, D) (9, 10, T) (10, 11, e_{10}) (11, 12, C)

Cas 2 : variable définie au nœud v incorrecte

Le nœud v peut être soit une instruction d'affectation, soit une instruction d'initialisation, soit une instruction de lecture[20].

Si le nœud v est une instruction d'affectation ou une instruction d'initialisation alors il est exécuté par P_i et par P_c. La faute porte soit sur l'expression à gauche de l'affectation, c'est-à-dire sur le nom de la variable affectée, soit sur les expressions à droite et à gauche de l'affectation (cela correspond au remplacement d'une instruction d'affectation par une autre instruction d'affectation). Deux erreurs initiales de type "variable erronée" peuvent être alors créées ; car une variable x' est définie dans P_i à la place d'une variable x qui est, elle, définie dans P_c.

Si le nœud v représente une fonction de lecture, elle est dans ce cas exécutée par P_i et par P_c. On considère que la faute porte sur l'un au moins des paramètres de lecture ; cela signifie que pour chaque paramètre incorrect, une variable x' est lue (c'est-à-dire définie) dans P_i à la place d'une variable x qui est, elle, lue dans P_c, deux erreurs initiales sont alors créées sur x et x'.

Cas 3: variable définie manquante

Le nœud v peut être soit une instruction d'affectation, soit une instruction d'initialisation, soit une instruction de lecture.

Si le nœud v est une instruction d'affectation ou d'initialisation alors nous sommes dans le cas où v est exécuté par seulement P_c, il est dit manquant : $v \in G_c$ et $v \notin G_i$ et $D_c(v) = \{x\}$.

Si le nœud v représente une fonction de lecture, cela peut signifier qu'un des paramètres au moins est manquant ; pour chaque paramètre manquant, une variable x est définie dans P_c, et non dans P_i.

Le cas où la fonction de lecture est manquante est une généralisation de ce cas 3. En effet, l'instruction correspondante est de la forme fonction_de_lecture(format, x, y, z) où les variables x, y, z désignent respectivement les variables lues et le cas commenté

---

20 Seules les instructions représentant des fonctions de lecture sont considérées, ici. En effet, les fonctions d'écriture utilisent des variables mais n'en définissent pas.

- $util_i(x, v)$ désigne la valeur de la variable x utilisée au nœud v lors de l'exécution du programme incorrect,
- $util_c(x, v)$ désigne la valeur de la variable x utilisée au nœud v lors de l'exécution du programme correct,
- $vis_i(x, v)$ désigne la valeur de la variable x visible au nœud v lors de l'exécution du programme incorrect,
- $vis_c(x, v)$ désigne la valeur de la variable x visible au nœud v lors de l'exécution du programme correct.

Dans le cas où v désigne un nœud de contrôle, x désigne alors la variable comprise dans l'expression conditionnelle évaluée.

La notion de valeur visible est liée à celle de définition visible (définie au paragraphe IV.3.2) et est utile lorsqu'une variable est observable dans un seul des programmes correct ou incorrect (lors de l'exécution d'une branche incorrectement sélectionnée).

Pour chaque mécanisme de propagation (création, annulation, masquage) d'erreurs présenté ci-après, nous décrivons quelles sont les conditions réunies durant l'exécution des programmes correct et incorrect qui conduisent à observer ces mécanismes.

En ce qui concerne la création d'erreurs, nous étudions séparément les mécanismes de création d'une erreur initiale (donc, par l'activation d'une faute) et de création d'une erreur par propagation d'une ou plusieurs autres erreurs.

IV.4 CRÉATION D'UNE ERREUR INITIALE

Une ou plusieurs erreurs initiales peuvent être créées suite à l'activation d'une faute ; cela dépend du nombre de nœuds (ou instructions) affectés par la faute et du nombre de variables définies par ces nœuds. Nous ne considérons dans notre étude que les erreurs présentes à la suite de l'exécution d'une instruction, c'est-à-dire observables dans les traces d'exécution ; contrairement à [Thompson 1991, Offutt & Lee 1994], nous ne tenons pas compte des erreurs créées par une expression interne à une instruction et non observables à la fin de l'exécution de cette instruction.

Erreurs initiales				Conditions de création	
Cas	Cond.	Valeur correcte	Valeur erronée	Conditions	Commentaires
1	C	$def_C(C, v)$	$def_i(C, v)$	$D_C(v) = D_i(v) = \{C\}$ et $def_i(C, v) \neq def_C(C, v)$	Expression conditionnelle incorrecte
2	C	$def_C(C, v)$	*	$C \in D_C(v)$ et $D_i(v)=\emptyset$	Expression conditionnelle manquante
3	C	*	$def_i(C, v)$	$C \in D_i(v)$ et $D_C(v)=\emptyset$	Expression conditionnelle ajoutée

Tableau IV.4 : Création d'erreurs initiales de type branchement erroné

Cas 1 : expression conditionnelle incorrecte

Le nœud v est exécuté par P_i et par P_C et la faute porte sur l'expression conditionnelle C (utilisation d'opérandes ou d'opérateurs incorrects). Une erreur initiale de type "branchement erroné" est créée si la valeur de l'expression conditionnelle C est différente dans P_i et dans P_C ; cette erreur conduira à exécuter une branche incorrectement sélectionnée dans P_i.

Cas 2 : expression conditionnelle manquante

Le nœud v est alors exécuté seulement par P_C : $v \in G_C$, $v \notin G_i$ et $D_C(v) = \{C\}$[21]. Une erreur de branchement peut être alors créée puisqu'une valeur est affectée à C dans P_C et pas dans P_i. La création de l'erreur de branchement dépend du type de l'instruction de contrôle et de la valeur définie pour C par P_C :

a) le nœud v est de type "if-then-else" ou "switch" : une erreur de branchement est créée quelle que soit la valeur définie pour C dans P_C ;

b) le nœud v est de type "if-then" : une erreur de branchement est créée si $def_C(C, v) = Faux$. En effet, si $def_C(C, v) = Vrai$ alors les nœuds dépendants de v par le contrôle dans P_C seront exécutés dans P_C et dans P_i ;

---

21 Par abus de langage, les notations "$D_C(v) = \{C\}$" ou "$C \in D_C(v)$" signifient que les variables comprises dans l'expression conditionnelle C appartiennent à $D_C(v)$.

Erreurs initiales				Conditions de création	
Cas	Var.	Valeur correcte	Valeur erronée	Conditions	Commentaires
1	x	$def_C(x, v)$	$def_i(x, v)$	$D_C(v) = D_i(v) = \{x\}$, et $exp\text{-}dte(v)_C \neq exp\text{-}dte(v)_i$	Variables, opérateurs ou constantes utilisés incorrects
2	x x'	$def_C(x, v)$ *	* $def_i(x', v)$	$D_C(v) = \{x\}$, $D_i(v) = \{x'\}$ avec $x \neq x'$ et $def_C(x, v) \neq vis_i(x, v)$ ou $def_i(x', v) \neq vis_C(x', v)$	Variable définie incorrecte
3	x	$def_C(x, v)$	*	$x \in D_C(v)$ et $x \notin D_i(v)$	Variable définie manquante
4	x	*	$def_i(x, v)$	$x \in D_i(v)$ et $x \notin D_C(v)$	Variable définie supplémentaire

Tableau IV.3 : Création d'erreurs initiales de type variable erronée

Cas 1 : variables, opérateurs ou constantes utilisés au nœud v incorrects

Le nœud v peut être une instruction d'affectation exécutée par P_i et par P_c ; elle définit la même variable x, dans P_i et dans P_c. La faute porte sur l'expression à droite de l'affectation, elle peut concerner :

a) soit, une des variables utilisées (différente dans P_i de celle utilisée dans P_c) : $D_C(v) = D_i(v) = \{x\}$ et $U_C(v) \neq U_i(v)$. L'activation de cette faute peut conduire à la création d'une erreur sur x (valeurs attribuées dans P_i et dans P_c différentes) si les valeurs des variables utilisées dans P_i sont différentes des valeurs des variables utilisées dans P_c et si cette différence n'est pas compensée par le calcul effectué.

b) soit, les opérateurs ou les constantes utilisés ; une erreur est créée sur x (valeurs attribuées dans P_i et dans P_c différentes) si les valeurs des constantes numériques ou si les opérateurs utilisés dans P_c et P_i ne sont pas équivalents et si l'opération effectuée ne compense pas ces différences.

Dans le cas où v est une instruction d'initialisation, la faute peut affecter la constante utilisée ; une erreur est alors créée sur x si la valeur numérique de la constante utilisée dans P_i n'est pas équivalente à celle utilisée dans P_c.

IV.4.3 Exemples issus de l'application ETUD

Les exemples issus de l'application ETUD montrent comment l'activation d'une faute conduit à la création d'erreurs initiales telles que décrites dans les deux paragraphes précédents ; nous nous basons pour cela sur l'étude des 12 fautes réelles identifiées, notées A, ..., L, décrites au chapitre II (§ II.3.2) et analysées au chapitre III (§ III.4.1).

La faute A affecte trois instructions (utilisation d'un opérateur incorrect) : son activation conduit à la création de trois erreurs de type "variable erronée" selon le cas 1 du tableau IV.3.

Les fautes B et C correspondent chacune à deux instructions manquantes : leur activation conduit à la création de deux erreurs de type "variable erronée" selon le cas 3 du tableau IV.3.

La faute D affecte une instruction de branchement (utilisation d'opérateurs incorrects dans l'expression conditionnelle) : son activation conduit donc à une erreur de branchement selon le cas 1 du tableau IV.4.

Les fautes E et J correspondent chacune à une instruction manquante ; leur activation conduit à la création d'une erreur de type "variable erronée" selon le cas 3 du tableau IV.3.

La faute F correspond à un bloc supplémentaire dans le code source constitué d'une instruction de contrôle de type "if-then" et d'une instruction dépendant de cette instruction de contrôle. L'activation de F conduit à créer une erreur de branchement (cas 3 du tableau IV.4) qui propage en créant une erreur de type "variable erronée" (la variable est définie dans P_i et non dans P_c) lorsque la valeur du branchement est Vrai dans P_i.

La faute G affecte quatre instructions du code source qui sont placées hors du corps d'un "else" (structure "if-then-else") : son activation conduit à la création de quatre erreurs de type "variable erronée" (cas 4 du tableau IV.3), lorsque la condition de branchement est évaluée à Vrai dans P_c. En effet, dans ce cas, ces 4 instructions sont exécutées dans P_i et non dans P_c.

La faute H correspond, dans le code, au remplacement d'une instruction par quatre autres. Son activation conduit à la création de cinq erreurs de type "variable

pour la variable x, se généralise à tous les paramètres ; une erreur initiale par paramètre manquant est créée.

Cas 4: variable définie supplémentaire

Si le nœud v est une instruction d'affectation ou d'initialisation alors nous sommes dans le cas où v est exécuté par seulement P_i, il est dit supplémentaire : $v \in G_i$ et $v \notin G_c$ et $D_i(v) = \{x\}$.

Si le nœud v représente une fonction de lecture, cela signifie qu'au moins un des paramètres est ajouté ; pour chaque paramètre supplémentaire, une variable x est définie dans P_i, et non dans P_c.

Le cas où le nœud v représentant une fonction de lecture est ajouté est une généralisation de ce cas 4. Une erreur initiale par paramètre lu est alors créée.

Ces différents exemples commentés montrent que des fautes de natures différentes portant sur des nœuds de type différent, peuvent créer suite à leur activation des erreurs de même type.

IV.4.2 Création d'une erreur de type branchement erroné

Supposons que le nœud v est un nœud de contrôle conditionnel du type "if-then", "if-then-else" ou "switch" ou une boucle de type "while", "for" ou "do-while". Il évalue une expression conditionnelle C.

Dans le cas d'une boucle "for", le nœud est décomposé en 3 instructions (cf. tableau IV.2) : initialisation de l'indice de boucle i (nœud i), évaluation de l'expression conditionnelle (nœud v) et incrémentation de l'indice de boucle (nœud j). Une faute affectant les nœuds i et j peut conduire à la création d'une erreur de type "variable erronée" (cf. § IV.4.1). Si une faute affecte le nœud v, alors l'exécution de v peut conduire à la création d'une erreur de branchement (cas qui nous intéresse dans ce paragraphe).

Plusieurs cas peuvent être considérés suite à l'exécution du nœud v, ils sont résumés dans le tableau IV.4 (construit de la même manière que le tableau IV.3).

dans le paragraphe IV.5.3 sur les interactions entre erreurs. Une erreur de type "variable erronée", créée au nœud u et affectant une variable x peut propager à toutes les variables y ou conditions de branchement C définies aux nœuds v dépendants de u par les données.

Pour qu'il y ait possibilité de propagation à un nœud v, les deux conditions suivantes doivent être satisfaites :

1) l'association (u, v, x) appartient à l'ensemble des associations de dépendances A du programme ;
2) cette association intervient dans une chaîne de dépendances utilisée par le chemin en cours d'exécution dans P_c.

L'erreur propage effectivement si le calcul effectué au nœud v ou la valeur de l'expression conditionnelle sont différents dans P_c et dans P_i. Cette erreur peut propager en créant une erreur sur la même variable x, sur une autre variable y ou sur une expression conditionnelle C.

L'analyse des cas détaillés est donnée ci-après ; pour chaque cas, nous décrivons comment une erreur peut propager et quelles sont les conditions de création d'une nouvelle erreur.

Cas 1 : création d'une erreur sur la même variable x

Une erreur sur une variable x créée au nœud u peut propager sur cette même variable à un nœud v lorsque le nœud v est une instruction qui définit et utilise la variable x (instruction d'affectation comprenant une incrémentation de x, par exemple) et si le calcul réalisé en v ne masque pas l'erreur sur x et conduit à une nouvelle valeur différente de x dans P_c et dans P_i.

La propagation conduit à la création d'une nouvelle erreur de type "variable erronée" sur v et à l'annulation de l'ancienne valeur erronée sur x.

$\Rightarrow$ Conditions de création : $x \in D(v) \cap D(u)$ et $x \in U(v)$ telles que $util_i(x, v) \neq util_c(x, v)$ et $def_i(x, v) \neq def_c(x, v)$.

Cas 2 : création d'une erreur sur une autre variable y

Une erreur sur une variable x (définie en u) peut propager à une autre variable y lorsque x est utilisée dans une définition v de y et si v est dépendant de u par les

c) le nœud v est de type "do-while" : une erreur de branchement est créée si $def_c(C, v)$ = Vrai. En effet, si $def_c(C, v)$ = Faux, les nœuds dépendants de v par le contrôle dans P_c ont déjà été exécutés par P_c et par P_i ;

d) le nœud v est de type "while" ou "for", deux cas se présentent :

- si $def_c(C, v)$ = Faux, une erreur de branchement est créée car les nœuds dépendants de v par le contrôle dans P_c seront exécutés dans P_i et non dans P_c ;
- si $def_c(C, v)$ = Vrai, une erreur de branchement est créée seulement à la deuxième exécution de la boucle et cette erreur propage sur l'ensemble des nœuds $u_1, ..., u_n$ dépendants par le contrôle de v, tant que $def_c(C, v)$ = Vrai.

Cas 3 : expression conditionnelle ajoutée

Le nœud v est alors exécuté seulement par P_i : $v \in G_i$, $v \notin G_c$ et $D_i(v) = \{C\}$. On considère alors que certains nœuds existant dans P_c et dans P_i deviennent dépendants par le contrôle de v dans P_i. Une erreur de branchement peut alors être créée puisqu'une valeur est affectée à C dans P_i et pas dans P_c. Comme au cas 2, la création de l'erreur de branchement dépend du type de l'instruction de contrôle et de la valeur définie de C par P_i : on retrouve alors les mêmes sous-cas a) à d) en inversant les termes P_c et P_i.

Autre cas : modification de l'instruction de contrôle

Le nœud v est un nœud de contrôle et la faute porte sur la nature de l'instruction de contrôle. Les dépendances liées au flot de contrôle peuvent alors être modifiées : par exemple, si une instruction "if-then" est remplacée par un "while", le nœud de contrôle v devient dépendant de lui-même. Une erreur de branchement apparaît seulement si l'expression conditionnelle C est vraie et reste vraie après exécution du bloc d'instructions dépendant de v : la première instruction du bloc sera exécutée une nouvelle fois dans P_i et non dans P_c (respectivement dans P_c et non dans P_i) puis toutes les instructions suivantes de ce même bloc. Cette erreur n'apparaît pas immédiatement lors de l'exécution du nœud incorrect v mais seulement suite à la sortie du bloc d'instructions dépendants de v. Selon les modifications affectant le nœud v, les cas 1, 2 ou 3 précédemment commentés peuvent être considérés.

Nous rappelons que les deux exécutions, dans P_c et dans P_i, se re-synchronisent sur le nœud post-dominant immédiat de u.

Plusieurs cas de propagation peuvent être observés en fonction du type de branchement :

a) Si u est un nœud de type "if-then" : l'un des programmes, par exemple P_i (resp. P_c), va exécuter tous les nœuds dépendants par le contrôle de u, tandis que P_c (resp. P_i) n'exécute aucune instruction avant la re-synchronisation sur le nœud post-dominant immédiat de u. L'erreur sur C peut alors générer un sous-flot d'erreurs par propagation.

b) Si u est un nœud de type "if-then-else" ou "switch" : l'un des programmes, par exemple P_i (resp. P_c), va exécuter tous les nœuds correspondant à la branche "if", tandis que P_c (resp. P_i) va exécuter tous les nœuds correspondants à la branche "else". Dans le cas d'un "switch", les deux programmes vont exécuter des nœuds correspondant à des choix différents. L'erreur sur C peut alors générer deux sous-flots d'erreurs, suite à l'exécution de branches différentes dans P_c et dans P_i.

c) Si u est un nœud de type "for", "while" ou "do-while" : l'un des programmes, par exemple P_i (resp. P_c), va exécuter tous les nœuds dépendants par le contrôle de u, c'est-à-dire le corps de la boucle, tandis que P_c (resp. P_i) n'exécute aucune instruction avant la re-synchronisation sur le nœud post-dominant immédiat de u. L'erreur sur C peut alors générer un sous-flot d'erreurs par propagation. Ce comportement erroné peut persister tant que l'évaluation de l'expression conditionnelle au nœud u conduira à une erreur sur C.

Les erreurs créées par propagation peuvent être de divers types selon la nature de l'instruction v dépendant de u par le contrôle. Ces cas sont commentés ci-après.

Cas 1 : création d'erreurs de type "variable erronée"

A un instant donné de l'exécution, une ou plusieurs erreurs de type "variable erronée" peuvent être créées selon la nature du nœud de contrôle u. Nous considérons deux possibilités :

1) L'un des programmes, par exemple P_i (resp. P_c), exécute un nœud v dépendant par le contrôle de u et définissant la variable y, l'autre programme P_c (resp. P_i) n'exécutant aucune instruction avant le nœud post-dominant u (noté

erronée" ; une variable étant définie dans P_i et non dans P_c (cas 4 du tableau IV.3) et quatre autres variables étant définies dans P_c et non dans P_i (cas 3 du tableau IV.3).

La faute I correspond, dans le code, à une instruction incorrecte suite à l'utilisation de variables et d'opérateurs incorrects dans l'expression à droite de l'affectation ; son activation conduit à la création d'une erreur de type "variable erronée" selon le cas 1 du tableau IV.3.

Les fautes K et L correspondent chacune à une instruction erronée (utilisation de constantes incorrectes) ; leur activation conduit à la création d'une erreur de type "variable erronée" selon le cas 1 du tableau IV.3.

Nous pouvons remarquer au travers de ces exemples, qu'ils couvrent à peu près tous les cas considérés de création d'erreurs initiales. Chaque faute réelle pouvant créer plusieurs erreurs initiales, des combinaisons des différents cas présentés dans les tableaux IV.3 et IV.4 ont été observées.

La création d'une erreur initiale a été étudiée dans le modèle RELAY sous le nom d'"origination of a potential failure" [Thompson 1991]. Thompson s'est intéressée au mode de création d'erreur suite à l'activation de fautes élémentaires de type mutation. Les fautes de type "instruction(s) manquante(s)" ou "instruction(s) supplémentaire(s)" ou remplacement d'un opérateur par un opérateur de type différent (permis avec le langage C) n'ont pas été étudiées dans le modèle RELAY.

IV.5 CRÉATION D'UNE ERREUR PAR PROPAGATION

Une erreur peut être créée par propagation d'une ou plusieurs erreurs présentes dans l'état interne du programme. Divers comportements peuvent être observés selon le type d'erreur(s) qui propage(nt) et le comportement à l'exécution de P_c et de P_i. Nous considérons d'abord les cas où une erreur propage seule avant d'aborder les cas où plusieurs erreurs interagissent pour créer une nouvelle erreur.

IV.5.1 Propagation d'une erreur de type "variable erronée"

Nous supposons que P_c et P_i exécutent le même nœud v ; le cas où P_c et P_i exécutent des nœuds différents (suite à une erreur de branchement) est pris en compte

Cas 2 : création d'erreurs de type "branchement erroné"

L'erreur de branchement créée sur C au nœud u peut propager sur une ou plusieurs instructions de contrôle dépendantes de u par le contrôle ; plusieurs cas peuvent être observés :

1) Un seul des programmes, par exemple P_i (resp. P_c), exécute un nœud v dépendant par le contrôle de u et définissant une condition de branchement C', l'autre programme P_c (resp. P_i) n'exécutant aucune instruction avant le nœud post-dominant u. Si v est un nœud de type "if-then", "while", "do-while", ou "for", une erreur de branchement est créée si la condition C' est évaluée à Vrai[22] (cas 1.1). Si v est un nœud de type "if-then-else" ou "switch", une erreur de branchement est créée quelle que soit la valeur définie pour C' (cas 1.2).

⇒ Conditions de création :

1.1. $C' \in D_i(v)$ et $def_i(C', v)$ = Vrai (resp. $C' \in D_c(v)$ et $def_c(C', v)$ = Vrai)

1.2. $C' \in D_i(v)$ et $def_i(C', v)$ = Vrai ou Faux (resp. $C' \in D_c(v)$ et $def_c(C', v)$ = Vrai ou Faux)

2) Les deux programmes exécutent chacun un nœud dépendant par le contrôle de u ; le nœud de contrôle u est alors de type "if-then-else" ou "switch". Supposons que P_c exécute un nœud v et que P_i exécute un nœud v', plusieurs possibilités peuvent être considérées : seul le nœud v est un nœud de contrôle définissant une expression conditionnelle C' (cas 2.1 et 2.2 similaires aux cas 1.1 et 1.2 exposés précédemment), seul le nœud v' est un nœud de contrôle définissant une expression conditionnelle C'' (cas 2.3 et 2.4), les nœuds v et v' sont tous les deux des nœuds de contrôle définissant respectivement les expressions conditionnelles C' et C'' (combinaison des cas 2.1 à 2.4 conduisant à la création de deux erreurs).

⇒ Conditions de création :

2.1. une erreur de branchement est créée si la valeur définie pour l'expression conditionnelle C' au nœud v dans P_i est Vrai, v étant un nœud de type "if-then", "while", "do-while", ou "for" : $C' \in D_i(v)$ et $def_i(C', v)$ = Vrai

---

22 Dans le cas où la condition C' est fausse, l'erreur est immédiatement annulée puisque aucun nœud dépendant de v ne peut être exécuté.

données. La propagation conduit à la création d'une erreur de type "variable erronée" sur v et affectant y, si le calcul utilisant la valeur erronée de x ne compense pas cette erreur et conduit à une valeur différente de y dans P_c et dans P_i.

⇒ Conditions de création : $y \in D(v)$ et $x \in U(v)$ telles que $util_i(x, v) \neq util_c(x, v)$ et $def_i(y, v) \neq def_c(y, v)$.

Cas 3 : création d'une erreur sur une expression conditionnelle C

Une erreur sur une variable x peut propager à une expression conditionnelle C définie au nœud v si v est un nœud de contrôle dépendant de u par les données. La propagation conduit à la création d'une erreur de type "branchement erroné" sur v si le calcul utilisant la valeur erronée de x ne masque pas cette erreur et conduit à une valeur différente de l'expression C dans P_c et dans P_i.

⇒ Conditions de création : $C \in D(v)$ et $x \in U(v)$ tel que $util_i(x, v) \neq util_c(x, v)$ et $def_i(C, v) \neq def_c(C, v)$.

Dans le cas où v est une instruction de contrôle de type "while" ou "do-while", le nœud u peut être une instruction dans le corps de la boucle (u est alors dépendante de v par le contrôle) et la variable x peut être utilisée dans l'évaluation de l'expression conditionnelle C de v (v est alors dépendant de u par les données). L'erreur sur x créée lors de l'exécution de u peut propager en créant une erreur de type "branchement erroné" au nœud v lors de l'évaluation suivante de l'expression conditionnelle C.

IV.5.2 Propagation d'une erreur de type "branchement erroné"

Une erreur de type "branchement erroné" au nœud u, conduit à l'exécution de nœuds différents dans P_c et dans P_i. Cette erreur peut propager à tous les nœuds dépendants par le contrôle de u ; ils sont selon les cas exécutés soit par P_c, soit par P_i. Pour qu'il y ait propagation à un nœud v, les conditions suivantes doivent être satisfaites :

1) l'association (u, v, C) appartient à l'ensemble des associations de dépendances A du programme ;
2) cette association intervient dans une chaîne de dépendances utilisée par le chemin en cours d'exécution dans P_i ou dans P_c.

$C \in D(v)$, $x \in U(v)$ et $z \in U(v)$ telles que $util_i(x, v) \neq util_c(x, v)$ et $util_i(z, v) \neq util_c(z, v)$ et $def_i(C, v) \neq def_c(C, v)$.

2. Une variable erronée x, définie en u, et un branchement erroné, défini en t, interagissent en v sur la définition d'une variable y ou d'une condition C si v est dépendant de u par les données et si v est dépendant de t par le contrôle.

⇒ Conditions de création :

2.1. v est exécuté par P_i alors que P_c exécute un nœud v' définissant y : $x \in U_i(v)$, $y \in D_i(v)$, $y \notin D_c(v')$ et $def_i(y, v) \neq vis_c(y, v')$

2.2. v est exécuté par P_i alors que P_c exécute un nœud v' définissant C : $x \in U_i(v)$, $C \in D_i(v)$, $C \notin D_c(v')$ et $def_i(C, v)$ = Vrai (ou $def_i(C, v)$ = Vrai ou Faux si v' est un nœud de type "if-then-else" ou "switch").

2.3. v est exécuté par P_c alors que P_i exécute un nœud v' définissant y : $x \in U_c(v)$, $y \in D_c(v)$, $y \notin D_i(v')$ et $def_c(y, v) \neq vis_i(y, v')$

2.4. v est exécuté par P_c alors que P_i exécute un nœud v' définissant C : $x \in U_c(v)$, $C \in D_c(v)$, $C \notin D_i(v')$ et $def_c(C, v)$ = Vrai (ou $def_c(C, v)$ = Vrai ou Faux si v' est un nœud de type "if-then-else" ou "switch").

IV.5.4 Détection d'erreur

La détection d'erreur peut être considérée comme un cas particulier de propagation d'erreur. Une erreur est détectée si elle affecte une variable de sortie x sur un nœud u et si cette erreur n'est pas annulée durant l'exécution du chemin allant du nœud u au nœud final f (f doit être dépendant de u par les données). La variable x peut être définie au nœud u soit dans P_c seulement (cas 1), soit dans P_i seulement (cas 2), soit P_c et P_i (cas 3).

⇒ Conditions de détection :

1. $x \in D_c(u) \cap U(f)$ et $x \notin D_i(u)$ telle que $f \in$ Util-$att_c(x, u)$ et $util_c(x, f) \neq util_i(x, f)$

2. $x \in D_i(u) \cap U(f)$ et $x \notin D_c(u)$ telle que $f \in$ Util-$att_i(x, u)$ et $util_i(x, f) \neq util_c(x, f)$

3. $x \in D(u) \cap U(f)$ et $def_i(x, u) \neq def_c(x, u)$ telle que $f \in$ Util-att(x, u) et $util_c(x, f) \neq util_i(x, f)$

w) ; le nœud de contrôle u est alors de type "if-then", "for", "while" ou "do-while". Une erreur de type "variable erronée" peut être créée sur y lors de l'exécution du nœud v si la valeur définie pour y en v est différente de celle visible au même instant de l'exécution dans l'autre programme.

⇒ Conditions de création : v est exécutée soit par P_c (cas 1.1), soit par P_i (cas 1.2) :

1.1. $y \in D_c(v)$, $w \in N_G$ tel que w post-domine u, et $def_c(y, v) \neq vis_i(y, w)$

1.2. $y \in D_i(v)$, $w \in N_G$ tel que w post-domine u, et $def_i(y, v) \neq vis_c(y, w)$

2) Les deux programmes exécutent chacun un nœud dépendant par le contrôle de u ; le nœud de contrôle u est alors de type "if-then-else" ou "switch". Supposons que P_c exécute un nœud v et que P_i exécute un nœud v' : v et v' peuvent définir la même variable y ou deux variables différentes, respectivement y et y'. L'erreur de branchement peut alors propager en créant une erreur sur y si celle-ci est définie seulement au nœud v (cas 2.1), si elle est définie aux nœuds v et v' (cas 2.2), ou sur y et y' si y est définie au nœud v et y' est définie au nœud v' (cas 2.3), ou sur y' si celle-ci est définie seulement au nœud v' (cas 2.4).

⇒ Conditions de création :

2.1. une erreur est créée si la valeur définie pour y au nœud v dans P_c est différente de celle visible au même instant de l'exécution dans P_i :

$y \in D_c(v)$ et $y \notin D_i(v')$, $def_c(y, v) \neq vis_i(y, v')$

2.2. une erreur est créée si la valeur définie pour y au nœud v dans P_c est différente de celle définie au nœud v' dans P_i :

$y \in D_c(v) \cap D_i(v')$ et $def_c(y, v) \neq def_i(y, v')$

2.3. deux erreurs sont créées, l'une sur y définie au nœud v et l'autre sur y' définie au nœud v' si : $y \in D_c(v)$ et $y' \in D_i(v')$ telles que $def_c(y, v) \neq vis_i(y, v')$ et $def_i(y', v') \neq vis_c(y', v)$

2.4. une erreur est créée si la valeur définie pour y' au nœud v' dans P_i est différente de celle visible au même instant de l'exécution dans P_c :

$y' \in D_i(v')$ et $y' \notin D_c(v)$, $def_i(y', v') \neq vis_c(y', v)$.

Dans chacune des possibilités considérées ci-dessus, nous avons supposé que les nœuds dépendants de u par le contrôle étaient des instructions simples. Nous considérons au cas suivant que l'un de ces nœuds peut être une instruction de contrôle.

IV.7 MASQUAGE D'UNE ERREUR

Une erreur est masquée lorsque l'état d'une donnée reste erroné mais n'évolue pas ou n'affecte pas d'autres données. On peut distinguer plusieurs causes relatives au phénomène de masquage :

- une variable erronée n'est pas utilisée dans le reste de l'exécution ; elle ne peut donc affecter d'autres données ;
- une variable erronée est utilisée dans une expression arithmétique mais n'a pas d'impact sur le résultat de cette expression, par exemple dans l'expression ($x_{erronée}$ * y) si y a la valeur 0 ;
- une variable erronée est utilisée dans une expression logique mais n'a pas d'impact sur le résultat de cette expression, par exemple dans l'expression ($x_{erronée}$ OU y) si y a la valeur 1 ou dans l'expression ($x_{erronée}$ ET y) si y a la valeur 0 ;
- une variable erronée est utilisée dans une expression relationnelle mais n'a pas d'impact sur le résultat de cette expression, par exemple dans l'expression ($x_{erronée} \geq y$) lorsque la valeur erronée et la valeur correcte de x appartiennent à la même classe d'équivalence, c'est-à-dire lorsque la différence entre les deux valeurs ne modifie pas la valeur de l'expression relationnelle.

Ces phénomènes ont été étudiés dans d'autres travaux [Bishop & Pullen 1989, Murrill 1991, Goradia 1993]. Bishop et Pullen se sont intéressés aux phénomènes de masquage pour expliquer pourquoi deux fautes non corrélées conduisent à des défaillances coïncidentes. Murril désigne sous le même terme ("cancellation"), les phénomènes d'annulation et de masquage en distinguant par "expression cancellation", le cas où une erreur est utilisée dans une expression mais n'affecte pas le résultat et par "assignment cancellation", le cas où une variable erronée est redéfinie, annulant ainsi l'erreur. Zeil [Zeil 1983, Zeil 1984] s'est intéressé au phénomène de masquage (appelé "blindness") apparaissant dans les expressions arithmétiques et relationnelles.

Nous avons observé une autre cause de masquage liée aux caractéristiques du flot de contrôle : une variable définie suite à une erreur de branchement peut ne pas être erronée si la valeur définie dans le programme incorrect est la même que celle présente dans le programme correct au même instant de l'exécution, l'effet du branchement erroné est alors nul.

2.2. une erreur de branchement est créée quelle que soit la valeur définie pour l'expression conditionnelle C' au nœud v dans P_i, v étant un nœud de type "if-then-else" ou "switch" : C' $\in$ $D_i(v)$ et $def_i(C', v)$ = Vrai ou Faux.

2.3. une erreur de branchement est créée si la valeur définie pour l'expression conditionnelle C" au nœud v' dans P_c est Vrai, v' étant un nœud de type "if-then", "while", "do-while", ou "for" : C" $\in$ $D_c(v')$ et $def_c(C", v')$ = Vrai

2.4. une erreur de branchement est créée quelle que soit la valeur définie pour l'expression conditionnelle C" au nœud v' dans P_c, v' étant un nœud de type "if-then-else" ou "switch" : C" $\in$ $D_c(v')$ et $def_c(C", v')$ = Vrai ou Faux.

Cas 3 : création d'une erreur sur le même branchement

Nous considérons ici le cas où le nœud u est dépendant de lui-même (nœud de type "for", "while" ou "do-while"). Une nouvelle erreur peut être créée sur C, si l'expression conditionnelle C est de nouveau évaluée à Vrai ; le corps de la boucle sera de nouveau exécuté par P_i et non par P_c (respectivement par P_c et non par P_i).

IV.5.3 Interaction entre erreurs

Plusieurs erreurs peuvent interagir, à un instant donné de l'exécution, pour créer une autre erreur. Nous considérons ci-après deux possibilités :

1. Deux variables erronées x et z, définies respectivement en u et t, peuvent être utilisées pour définir une variable y ou une expression conditionnelle C au nœud v exécuté par P_c et par P_i si v est dépendant de u et de t par les données. Les deux erreurs sur x et z propagent à y ou à C si les deux valeurs erronées ne se compensent pas dans le calcul réalisé pour définir y.

$\Rightarrow$ Conditions de création :

1.1 une erreur de type "variable erronée" est créée si :

y $\in$ D(v), x $\in$ U(v) et z $\in$ U(v) telles que $util_i(x, v) \neq util_c(x, v)$ et $util_i(z, v) \neq util_c(z, v)$ et $def_i(y, v) \neq def_c(y, v)$;

1.2 une erreur de type "branchement erroné" est créée si :

Cas 2 : Masquage d'une erreur de type "branchement erroné"

Une erreur de type "branchement erroné" peut être masquée si l'exécution d'un nœud incorrectement sélectionné ne conduit pas à la création d'erreur sur ce nœud. Soit une erreur de branchement créée au nœud u, et supposons que P_i (resp. P_c) exécute le nœud v, dépendant par le contrôle de u alors que P_c (resp. P_i) exécute un autre nœud v' dépendant par le contrôle de u dans le cas d'un branchement de type "if-then-else" ou n'exécute aucun nœud avant le nœud post-dominant u (noté w). Deux cas peuvent être étudiés selon le type du nœud v.

1. v est une instruction simple définissant une variable x : l'erreur de branchement est masquée si la variable x définie en v reçoit la même valeur que celle visible au même instant d'exécution dans P_c (resp. P_i).

⇒ Conditions de masquage :

1.1 P_c (resp. P_i) exécute un autre nœud v' : $x \in D_i(v)$ et $def_i(x, v) = vis_c(x, v')$ (ou resp. $x \in D_c(v)$ et $def_c(x, v) = vis_i(x, v')$) ;

1.2 P_c (resp. P_i) n'exécute pas d'autre nœud avant w : $x \in D_i(v)$ et $def_i(x, v) = vis_c(x, w)$ (ou resp. $x \in D_c(v)$ et $def_c(x, v) = vis_i(x, w)$) ;

2. v est un nœud de contrôle de type "if-then", "while", "do-while", ou "for", définissant une condition C : l'erreur de branchement est masquée si la condition C définie en v est évaluée à Faux dans P_c (resp. P_i).

⇒ Conditions de masquage : $D_i(v) = \{C\}$ et $def_i(C, v) =$ Faux (ou resp. $D_c(v) = \{C\}$ et $def_c(C, v) =$ Faux).

Cas 3 : Masquage par interaction d'erreurs

Deux erreurs (ou plus) peuvent interagir à un instant donné de l'exécution et leur utilisation peut ne créer aucune nouvelle erreur. Plusieurs cas peuvent être considérés en fonction de la nature des erreurs et du nœud sur lequel elles interagissent :

1. Soient 2 variables erronées x et z définies respectivement en u et t, les erreurs sur x et z sont masquées au nœud v exécuté par P_c et P_i, si v est dépendant de u et de t par les données et si l'utilisation de x et z n'affecte pas la valeur de la variable ou de la condition définie en v.

⇒ Conditions de masquage :

1.1 v est une instruction simple définissant une variable y, les conditions de masquage sont les suivantes : $y \in D(v)$, $x \in U(v)$ et $z \in U(v)$ telles que

IV.6 ANNULATION D'UNE ERREUR

L'annulation d'erreur a pour effet de rendre correct un état de donnée précédemment erroné. Une erreur de type variable erronée affectant une variable x est annulée lorsque :

1. P_c et P_i exécutent le même nœud v, la variable x est redéfinie sur ce nœud v et la valeur de x devient la même pour P_c et P_i :

⇒ Conditions d'annulation : $x \in D(v)$ telle que $def_i(x, v) = def_c(x, v)$

Ce cas peut être dû à l'évolution d'une erreur ou à l'interaction entre plusieurs erreurs comme la redéfinition de la variable x (erronée) en utilisant une autre variable erronée.

2. P_c et P_i exécutent des nœuds différents :

   2.1 la variable x est redéfinie au nœud v dans P_i et la valeur affectée à x est la même que celle visible au même instant durant l'exécution de v' dans P_c :

   ⇒ Conditions d'annulation : $x \in D_i(v)$, $x \notin D_c(v')$ tel que $def_i(x, v) = vis_c(x, v')$,

   2.2 la variable x est redéfinie au nœud v dans P_c et la valeur affectée à x est la même que celle visible au même instant durant l'exécution de v' dans P_i :

   ⇒ Conditions d'annulation : $x \in D_c(v)$, $x \notin D_i(v')$ tel que $def_c(x, v) = vis_i(x, v')$,

   Ce cas est dû à la propagation de plusieurs erreurs : redéfinition de la variable x (erronée) dans un branchement erroné.

Une erreur de type branchement erroné est annulée lorsque les instructions dépendantes de ce branchement ont été exécutées et que les deux programmes P_c et P_i se sont re-synchronisés sur un nœud post-dominant. Ce cas peut être dû à l'évolution d'une erreur ou à l'interaction entre plusieurs erreurs comme l'utilisation d'un branchement erroné dans un branchement erroné qui conduit à sortir de ce branchement.

IV.8 ANALYSE INTERPROCÉDURALE DES DÉPENDANCES

L'analyse interprocédurale des dépendances a fait l'objet de nombreux travaux, dans le but d'étudier avec précision l'utilisation des pointeurs et des appels de paramètres par référence. En langage C, les paramètres d'une fonction sont transmis par valeur. Lors de l'appel d'une fonction, une copie des paramètres effectifs (ou arguments) est réalisée et transmise à la fonction appelée ; la fonction appelée ne peut donc les modifier que localement. Par contre, une fonction peut utiliser et définir des variables globales (déclarées en externe à la fonction). L'appel d'une fonction peut se faire de deux manières selon la déclaration de cette fonction :

a) la fonction a une valeur de retour et l'appel de fonction apparaît dans le programme appelant au sein d'une expression dans une instruction simple ou de contrôle. La valeur de retour est alors utilisée pour évaluer l'expression ;

b) la fonction n'a pas de valeur de retour et l'appel de fonction constitue une instruction. L'appel de la fonction peut être alors considéré comme une instruction d'affectation utilisant et définissant des variables globales du programme.

L'analyse interprocédurale des dépendances est basée sur une représentation du programme complet, constitué du programme principal et de l'ensemble des fonctions appelées. On utilise pour cela, un **graphe de contrôle interprocédural GI** [Pande et al. 1991] représenté par le triplet (N_{GI}, A_{GI}, n_0), N_{GI} étant l'ensemble des nœuds composant le programme complet, A_{GI} étant l'ensemble des arcs reliant les nœuds du graphe et n_0 représentant le nœud initial du programme principal. Le graphe GI correspond à l'union des graphes de contrôle représentant le programme principal et chacune des fonctions, auxquels ont été ajoutés certains nœuds et certains arcs représentant les appels et les retours de fonction :

1) N_{GI} contient ainsi un nœud d'entrée et un nœud de sortie pour chaque fonction ainsi qu'un nœud d'appel et un nœud de retour pour chaque appel de fonction ;

2) A_{GI} contient, pour chaque appel de fonction, un arc reliant le nœud d'appel au nœud d'entrée de la fonction appelée et un arc reliant le nœud de sortie de la fonction au nœud de retour dans le programme appelant.

$util_i(x, v) \neq util_c(x, v)$ et $util_i(z, v) \neq util_c(z, v)$ et $def_i(y, v) = def_c(y, v)$.

1.2 v est un nœud de contrôle définissant une expression conditionnelle C, les conditions de masquage sont les suivantes : $C \in D(v)$, $x \in U(v)$ et $z \in U(v)$ telles que $util_i(x, v) \neq util_c(x, v)$ et $util_i(z, v) \neq util_c(z, v)$ et $def_i(C, v) = def_c(C, v)$.

Ce masquage peut avoir des causes diverses : 1) soit les valeurs de x et de z se compensent, 2) soit un troisième opérande masque les valeurs de x et z (par ex. ((x + z) * 0) ou (x ou z ou 1) ou (x et z et 0)), 3) soit l'opérateur utilisé masque l'utilisation des valeurs erronées.

2. Soient une variable erronée x définie en u et un branchement erroné défini en t ; ces deux erreurs peuvent être masquées lors de l'exécution du nœud v dépendant par les données de u et dépendant par le contrôle de t. P_c et P_i exécutent alors des nœuds différents, respectivement, v et v'.

⇒ Conditions de masquage :

2.1 le nœud v est exécuté par P_i (resp. P_c) et définit une variable y, alors que P_c (resp. P_i) exécute un autre nœud v'[23] ; les conditions de masquage sont les suivantes : $x \in U_i(v)$ et $y \in D_i(v)$ telles que $def_i(y, v) = vis_c(y, v')$ (ou resp. $x \in U_c(v)$ et $y \in D_c(v)$ telles que $def_c(y, v) = vis_i(y, v')$) ;

2.2 le nœud v (de type "if-then", "while", "do-while", ou "for") est exécuté par P_i (resp. P_c) et définit une condition C, alors que P_c (resp. P_i) exécute un autre nœud v' ; les conditions de masquage sont les suivantes : $x \in U_i(v)$ et $C \in D_i(v)$ telles que $def_i(C, v) =$ Faux (ou resp. $x \in U_c(v)$ et $C \in D_c(v)$ telles que $def_c(C, v) =$ Faux).

Nous avons considéré, dans les paragraphes IV.5 à IV.7 décrivant les mécanismes de propagation d'erreurs, un programme comme étant une fonction unique. Nous allons maintenant montrer, au paragraphe IV.8, comment plusieurs fonctions dans un programme interagissent les unes avec les autres et peuvent favoriser la propagation d'erreurs.

[23] Ce nœud v' peut être le nœud post-dominant u.

Un phénomène de compensation peut également être observé, lorsque deux erreurs sont utilisées dans la même expression et leur utilisation conduit à une valeur

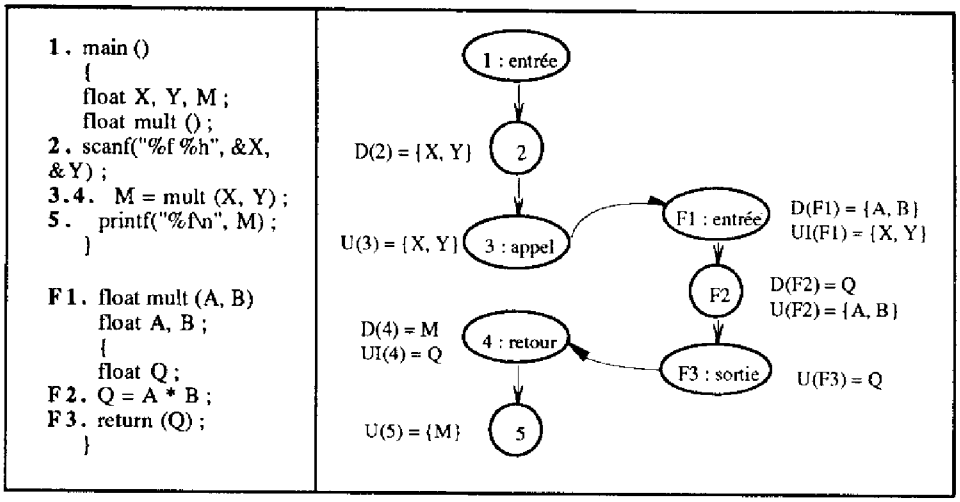


Figure IV.3 : Programme avec appel de fonction et graphe définition-utilisation interprocédural associé

Nous pouvons constituer de la même manière qu'au paragraphe IV.3.3, l'ensemble des associations de dépendances et les chaînes de dépendances.

Si nous prenons, par exemple, le programme analysé dans la figure IV.3, l'ensemble *A* des associations de dépendances est constitué des dépendances suivantes :
A = {(2, 3, X), (2, 3, Y), (2, F1, X), (2, F1, Y), (3, F1, "appel"), (F1, F2, A), (F1, F2, B), (F2, F3, Q), (F2, 4, Q), (F3, 4, "retour"), (4, 5, M)}
Les chaînes de dépendances suivantes peuvent être utilisées lors de l'exécution du programme :
CD1 = (2, 3, X) (3, F1, "appel") (F1, F2, A) (F2, F3, Q) (F3, 4, "retour") (4, 5, M)
CD2 = (2, 3, Y) (3, F1, "appel") (F1, F2, B) (F2, F3, Q) (F3, 4, "retour") (4, 5, M)
CD3 = (2, F1, X) (F1, F2, A) (F2, F3, Q) (F3, 4, "retour") (4, 5, M)
CD4 = (2, F1, Y) (F1, F2, B) (F2, F3, Q) (F3, 4, "retour") (4, 5, M)
CD5 = (2, F1, X) (F1, F2, A) (F2, 4, Q) (4, 5, M)
CD6 = (2, F1, Y) (F1, F2, B) (F2, 4, Q) (4, 5, M)

A chacun de ces nœuds n peut être associé un ensemble D(n) des variables définies au nœud n et un ensemble U(n) des variables utilisées à ce nœud ; nous avons également ajouté un ensemble UI(n) défini pour le nœud d'entrée d'une fonction et le nœud de retour dans le programme appelant, qui contient les variables utilisées pour le transfert "interprocédural" de valeurs ; le contenu de ces divers ensembles est décrit ci-dessous :

- soit n_{appel}, le nœud d'appel d'une fonction dans le programme appelant ; $D(n_{appel})$ est vide et $U(n_{appel})$ est constitué des paramètres effectifs ;
- soit n_{EF}, le nœud d'entrée de la fonction appelée ; n_{EF} dépend par les données de n_{appel}. $UI(n_{EF})$ est constitué des paramètres effectifs tels qu'ils sont définis dans le programme appelant[24] ($UI(n_{EF}) = U(n_{appel})$) et $D(n_{EF})$ est constitué des paramètres formels en entrée de la fonction appelée qui représentent une copie des paramètres effectifs ;
- soit n_{SF}, le nœud de sortie de la fonction appelée ; $D(n_{SF})$ est vide et $U(n_{SF})$ est constitué des variables de retour ;
- soit n_{retour}, le nœud de retour d'une fonction dans le programme appelant ; n_{retour} dépend par les données de n_{SF}. $D(n_{retour})$ est constitué des variables globales modifiées par la fonction appelée et de la variable ou de l'expression affectée par la valeur de retour si elle existe et $UI(n_{retour})$ est constitué des variables de retour définies dans la fonction appelée ($UI(n_{retour} = U(n_{SF}))$.

En ce qui concerne les dépendances par le contrôle, l'appel d'une fonction constitue un branchement inconditionnel ; de ce fait, n_{EF} est dépendant par le contrôle de n_{appel} et n_{retour} est dépendant par le contrôle de n_{SF}.

Cette méthode nous permet de définir un graphe définition-utilisation interprocédural ; la figure IV.3 en est un exemple et illustre toutes les notions définies ci-dessus. Le graphe définition-utilisation interprocédural nous permet donc d'analyser le flot d'informations échangées entre plusieurs fonctions. Les phénomènes de création, de propagation, d'annulation et de masquage d'erreur peuvent ainsi être analysés de la même manière qu'aux paragraphes IV.4 à IV.7 en considérant l'appel et le retour d'une fonction comme une boîte noire puis en analysant les flots de données à l'intérieur de chaque fonction.

---

24 Les paramètres effectifs peuvent être différents à chaque appel de la fonction, $UI(n_{EF})$ contient donc l'ensemble des paramètres effectifs utilisés dans le programme appelant.

CHAPITRE V

APPLICATION DE L'ANALYSE DES DÉPENDANCES

V.1 INTRODUCTION

Ce chapitre a pour objet d'appliquer la méthode d'analyse des dépendances, présentée au chapitre précédent, au programme LOCALES (présenté au paragraphe II.3.1.2). Ce programme a été développé par Technicatome : il est composé de deux fonctions, *recopie-capteur* et *calcul-d-tsgv*, et 13 fautes ont été identifiées et corrigées sur l'ensemble des deux fonctions au cours des activités de vérification menées par Technicatome. Chacune des deux fonctions a été analysée de manière unitaire. Cette étude a pour objectif :

1) d'évaluer l'impact de ces 13 fautes sur le graphe définition-utilisation et les associations de dépendances ;
2) d'expliquer les comportements erronés produits par ces fautes à l'aide des relations de dépendances, c'est-à-dire d'analyser les modifications induites par chaque faute sur les chaînes de dépendances et d'analyser les mécanismes de création, de masquage et d'annulation d'erreurs sur un cas concret en étudiant quelles associations de dépendances ont été utilisées ;
3) de vérifier que les comportements erronés observés pour ces fautes peuvent être partiellement ou totalement reproduits par des mutations injectées à l'aide de l'outil SESAME en s'appuyant sur notre connaissance des relations de dépendances du programme.

V.2 IMPACT DES FAUTES RÉELLES SUR LES ASSOCIATIONS DE DÉPENDANCES

Treize fautes ont été révélées au cours du développement du module LOCALES avant sa mise en service opérationnel : les fautes F1 à F8 ont été révélées par analyse statique, lors de relectures croisées du code source, la faute F9 a été révélée

IV.9 CONCLUSIONS

Dans ce chapitre, nous avons proposé une représentation du programme — *graphe définition-utilisation* — et une méthode d'analyse — détermination des *associations* et des *chaînes de dépendances* — permettant de modéliser le comportement d'un programme séquentiel en cours d'exécution et donc d'expliquer les comportements erronés générés suite à activation d'une faute avec des entrées de test appropriées. Nous avons également décrit les processus de création, d'annulation et de masquage d'erreur en utilisant les relations de dépendances.

Ce formalisme va nous permettre de comprendre les comportements observés lors de nos études expérimentales ; il permet notamment d'expliquer les similitudes observées entre différents flots d'erreurs générés suite à l'activation de fautes de nature "syntaxique" différente. En l'absence d'outils permettant de générer automatiquement le graphe définition-utilisation ou les associations de dépendances, il n'est pas possible d'étudier les relations de dépendances manuellement sur un programme de taille importante, tel que le programme ETUD. Nous avons donc mis en œuvre l'analyse des dépendances sur le programme LOCALES dont chaque fonction a été étudiée au niveau unitaire ; ces études expérimentales font l'objet du chapitre suivant.

La fonction *recopie-capteur* a été affectée par les fautes F1, F2, F3, F10 et F11. La fonction *calcul-d-tsgv* a été affectée par les fautes F4 à F9, F12 et F13.

Les tableaux V.1 et V.2 indiquent pour chacune des fonctions, l'impact de chaque faute sur le graphe définition-utilisation de la version 1.4 et les modifications induites sur les associations de dépendances en terme d'ajout ou de suppression d'associations. Seule la faute F2 ne figure pas dans le tableau V.1 car on ne peut pas évaluer son impact sur la version 1.4 du module LOCALES. Cette faute a été révélée sur la version 1.1 et elle affecte une instruction conditionnelle qui n'existe plus dans la version 1.4 suite à la correction de la faute F10. Cette instruction conditionnelle correspond à la condition A ajoutée par F10 ; la faute F2 n'aurait donc pu être étudiée que combinée à F10.

Faute	Impact sur le graphe définition-utilisation	Impact sur les associations (ajout ou suppression)
F1	Déplacement du nœud 5 entre les nœuds 1 et 2, suppression des nœuds 6, 7 et 8 et modification du nœud 2 (U(2') = U(2) ∪ U(6))	- (2, 5, e_2), (2, 6, e_2), (5, 6, PP0), (6, 7, e_6), (6, 8, e_6), (7, S, PP0) et (8, S, TN[52]) supprimées - (5, 2', PP0) ajoutée
F3	Modification du nœud 68 : U(68') = U(68) - EIO2	- (E, 68, EIO2) supprimée
F10	Suppression des nœuds 15, 16, 17 et 18, ajout d'une condition A avant le nœud 67 et déplacement des nœuds 44, 45, 46, 47 qui deviennent dépendants par le contrôle de A (le bloc 44 à 47 est exécuté si e_A est vraie)	- (15, S, TB), (16, S, TC), (17, S, TN[26]), (18, S, TN[32]), (42, 44, e_{42}), (42, 45, e_{42}), (42, 46, e_{42}) et (42, 47, e_{42}) supprimées - (E, A, TGC), (E, A, TLC)[26], (A, 44, e_A), (A, 45, e_A), (A, 46, e_A) et (A, 47, e_A) ajoutées
F11	Modification des constantes utilisées aux nœuds 17, 18, 46 et 47 qui conduit à des valeurs erronées sur les variables des ensembles D(17), D(18), D(46) et D(47)	- aucune association ajoutée ou supprimée

Tableau V.1 : Impact des fautes réelles sur les associations de dépendances de la fonction *recopie-capteur*

26 La variable TLC n'est ni définie ni utilisée dans le programme correct (c'est-à-dire dans la version 1.4 du programme).

Faute	Impact sur le graphe définition-utilisation	Impact sur les associations (ajout ou suppression)
F4	Déplacement du nœud 6 entre les nœuds 1 et 2 et remplacement de l'arc (5, 16) par un arc (5, 12) : le nœud 12 n'est plus dépendant du nœud 2.	- (2, 12, e_2) et (2, 6, e_2) supprimées - (3,12, NH0) ajoutée
F5	Suppression des nœuds 12, 13, 14 et 15	- (2, 12, e_2), (6, 12, D_CAL), (8, 12, NH0), (11, 12, NH0), (12, 13, e_{12}), (12, 14, e_{12}), (12, 15, e_{12}), (13, 19, NH0), (13, 37, NH0), (13, S, NH0), (14, S, RH0) et (15, S, TN[56]) supprimées
F6	Suppression du nœud 23 et ajout d'un nœud (19') entre les nœuds 19 et 20 avec D(19') = D_CAL (initialisée à 1 au nœud 19' et à 0 au nœud 23) et remplacement de l'arc (22, 35) par un arc (22, 31) : le nœud 31 n'est plus dépendant du nœud 19.	- (19, 23, e_{19}), (19, 31, e_{19}) et (23, 31, D_CAL) supprimées - (6, 31, D_CAL), (19', 31, D_CAL) et (20, 31, NMH0) ajoutées
F7	Suppression des nœuds 31, 32, 33 et 34	- (19, 31, e_{19}), (23, 31, D_CAL), (30, 31, NMH0), (31, 32, e_{31}), (31, 33, e_{31}), (31, 34, e_{31}), (32, 36, NMH0), (32, 37, NMH0), (32, S, NMH0), (33, 35, RMH0), (33, S, RMH0) et (34, S, TN[55]) supprimées
F8	Modification de U(37) : utilisation de la variable D_CAL au lieu des variables NMH0 et NH0	- (3, 37, NH0),(8, 37, NH0), (11, 37, NH0), (13, 37, NH0), (20, 37, NMH0), (25, 37, NMH0), (29, 37, NMH0) et (32, 37, NMH0) supprimées - (E, 37, D_CAL), (6, 37, D_CAL) et (23, 37, D_CAL) ajoutées
F9	Utilisation d'une constante erronée dans la définition de R1_alpha qui conduit à des valeurs erronées sur la variable appartenant à D(28)	- aucune association ajoutée ou supprimée
F12	Modification de D(5), U(5), D(15) et U(15)	- (5, S, TN[56]) et (15, S, TN[56]) supprimées - (5, S, TN[54]) et (15, S, TN[54]) ajoutées
F13	Modification de D(22), U(22), D(34) et U(34)	- (22, S, TN[55]) et (34, S, TN[55]) supprimées - (22, S, TN[54]) et (34, S, TN[54]) ajoutées

Tableau V.2 : Impact des fautes réelles sur les associations de dépendances de la fonction *calcul-d-tsgv*

par les tests unitaires, et les fautes F10 à F13 ont été révélées pendant la phase d'intégration. Nous avons présenté, au paragraphe II.3.2, une description de ces fautes ainsi que leur impact sur le code en précisant le nombre d'

La comparaison des deux tableaux V.1 et V.2, évaluant l'impact sur les associations de dépendances, avec le tableau II.2 décrivant l'impact de chaque faute réelle sur le code source montre que ces deux "mesures" ne sont pas forcément corrélées. Dans le cas de la faute F1, par exemple, six instructions sont affectées dans le code source et huit associations de dépendances sont soit supprimées, soit ajoutées. Par contre, dans le cas de la faute F8, une seule instruction est affectée dans le code source alors que onze associations de dépendances sont soit supprimées, soit ajoutées.

V.3 ETUDE DES MODIFICATIONS SUR LES CHAÎNES DE DÉPENDANCES

Nous avons analysé en détail les modifications apportées par les fautes aux chaînes de dépendances. Il serait long et fastidieux de décrire ces modifications pour chacune des 13 fautes. Dans ce paragraphe, nous avons donc sélectionné quelques fautes pour lesquelles les modifications sur les chaînes de dépendances nous semblent intéressantes à décrire et représentatives de celles observées pour d'autres fautes. L'objectif de ces descriptions est avant tout de montrer l'intérêt d'une telle analyse ; elle permet en effet de mettre en évidence les mécanismes de création, de masquage et d'annulation d'erreurs, et donc de prévoir les comportements erronés qui peuvent être générés par la faute au cours de l'exécution du programme. Cette étude montre l'apport de l'analyse des dépendances dans ce domaine.

Nous désignons comme dans le chapitre IV, la version de référence (version 1.4 considérée comme correcte) de chacune des fonctions par P_c et la version affectée par une faute F_j par P_j (j = 1 à 13).

Les résultats des analyses, présentées ci-après, seront synthétisés dans des tableaux de la forme suivante :

1) la première colonne indique le chemin exécuté (voir la liste des chemins exécutables en annexe A), les lettres a à k permettent de distinguer les différents cas identifiés pour un même chemin ;
2) dans la deuxième colonne, on précise la chaîne de dépendances utilisée par P_c suivie (après la flèche) de la chaîne de dépendances utilisée par P_j. Les variables ou expressions conditionnelles en gras sont erronées et celles en gras et en italique permettent d'indiquer que les erreurs correspondantes ont été annulées ou masquées.

V.3.1 Fonction *recopie-capteur*

V.3.1.1 Faute F1

L'activation de la faute F1 crée une erreur initiale sur la variable PP0 qui est annulée par la suite quel que soit le chemin exécuté, $C_1$1 à $C_1$3 (voir annexe A, § A.2.2). Le tableau V.3 récapitule pour chaque chemin, les modifications induites sur les chaînes de dépendances lors de l'exécution du programme incorrect, ce qui permet de comprendre les mécanismes de création et d'annulation d'erreur décrits ci-dessous :

- si le chemin exécuté est $C_1$1 (séquence de nœuds 1 - 2 - 3 - 4 - 9) dans P_c, la faute conduit à exécuter le chemin 1 - 5 - 2' - 3 - 4 - 9 dans P_1 ; en effet, e_2 dans P_c et $e_{2'}$ dans P_1 sont toutes deux évaluées à Vrai. La variable PP0 est définie dans P_1 au nœud 5 (car ajout de l'association (5, 2', PP0)) puis redéfinie au nœud 3, ce qui annule l'erreur créée au nœud 5 ; aucune erreur ne peut propager vers des variables de sortie (cf. cas $C_1$1-a et b dans tableau V.3) ;
- si le chemin exécuté est $C_1$2 (1 - 2 - 5 - 6 - 7 - 8 - 9) dans P_c , la faute conduit à exécuter le chemin 1 - 5 - 2' - 3 - 4 - 9 dans P_1 (les nœuds 6, 7 et 8 ont été supprimés). Si e_6 est vraie dans P_c alors $e_{2'}$ est également vraie dans P_1. Une erreur initiale est créée puis annulée sur la variable PP0. Les associations (3, S, PP0) et (4, S, TN[52]) dans P_1 sont utilisées à la place des associations (7, S, PP0) et (8, S, TN[52]) supprimées par la faute F1. Or les nœuds 3 et 7 définissent la variable PP0 à la même valeur constante et les nœuds 4 et 8 définissent la variable TN[52] également à une même valeur constante. Aucune erreur ne propage donc vers des variables de sortie (cf. cas $C_1$2- a et b dans tableau V.3) ;
- si le chemin exécuté est $C_1$3 (1 - 2 - 5 - 6 - 9) dans P_c , la faute conduit à exécuter le chemin 1 - 5 - 2' - 9 dans P_1 ; en effet, si e_2 et e_6 sont fausses dans P_c alors $e_{2'}$ est également fausse dans P_1. Une seule erreur est créée puis annulée sur la variable PP0 car le nœud 5 est exécuté par P_1 et P_c.

La faute F1 ne peut donc jamais conduire le programme à défaillance, quel que soit le chemin exécuté. Les annulations d'erreurs sont dues aux redondances inhérentes au programme. Cette analyse détaillée des modifications apportées aux chaînes de dépendances par F1 est intéressante. En effet, F1 qui a fait l'objet, dans un contexte industriel, d'un rapport d'anomalie lors d'une étape de relecture du code et qui a donc conduit à des corrections dans le code n'aurait pas pu être révélée en phase de test. Les

corrections effectuées peuvent se justifier pour satisfaire des règles de programmation mais cette analyse montre qu'elles ne modifient pas le comportement de la fonction.

Cas	Chaîne de dépendances observée dans P_c → chaîne de dépendances observée dans P_1
$C_1$1-a	(E, 2, PP) (2, 3, e_2) (3, S, ***PP0***) → (E, 5, PP) (5, 2', **PP0**) (2', 3, $e_{2'}$) (3, S, ***PP0***)
$C_1$1-b	(E, 2, PP) (2, 3, e_2) (3, S, ***PP0***) → (E, 2', PP) (2', 3, $e_{2'}$) (3, S, ***PP0***)
$C_1$2-a	(E, 2, PP) (2, 5, e_2) (5, 6, ***PP0***) (6, 7, e_6) (7, S, ***PP0***) → (E, 5, PP) (5, 2', **PP0**) (2', 3, $e_{2'}$) (3, S, ***PP0***)
$C_1$2-b	(E, 2, PP) (2, 5, e_2) (5, 6, ***PP0***) (6, 8, e_6) (8, S, ***TN[52]***) → (E, 5, PP) (5, 2', **PP0**) (2', 4, $e_{2'}$) (4, S, ***TN[52]***)
$C_1$3-a	(E, 2, PP) (2, 5, e_2) (5, S, ***PP0***) → (E, 5, PP) (5, S, ***PP0***)

Tableau V.3 : Modifications des chaînes de dépendances dues à la faute F1

V.3.1.2 Faute F10

La faute F10 consiste tout d'abord à supprimer les nœuds 15, 16, 17, 18. Des erreurs sont créées sur les variables TB, TC, TN[26] et TN[32] à chaque exécution dans P_c du bloc contenant les nœuds 15 à 18 (chemins $C_2$2, $C_2$3 et $C_2$4).

La faute F10 consiste également à déplacer les nœuds 44, 45, 46, 47 qui deviennent dépendants d'une condition A. Or la condition A est évaluée à Faux quel que soit le chemin exécuté et les nœuds dépendants par le contrôle de A (c'est-à-dire les nœuds 44, 45, 46 et 47) ne sont jamais exécutés. Tout se passe comme si ces nœuds étaient supprimés ainsi que les associations correspondantes : (44, S, TGB), (45, S, TGC), (46, S, TN[27]) et (47, S, TN[33]). Or, il n'existe pas d'autres associations définissant les variables TGB, TGC, TN[27] et TN[33]. Comme ci-dessus, des erreurs sont créées sur les variables TGB, TGC, TN[27] et TN[33] à chaque exécution dans P_c du bloc contenant les nœuds 44 à 47.

Les erreurs créées conduisent toutes à défaillance puisqu'elles affectent des variables de sortie. Le tableau V.4 indique les modifications apportées aux chaînes de dépendances par la suppression des nœuds 15, 16, 17 et 18 (celles apportées par la suppression des nœuds 44, 45, 46 et 47 sont similaires).

Cas	Chaîne de dépendances observée dans P_c → chaîne de dépendances observée dans P_{10}
C_22-a, C_23-a, C_24-a	(E, 13, TE) (13, 15, e_{13}) (15, S, TB) → (E, S, TB)
C_22-b, C_23-b, C_24-b	(E, 13, TE) (13, 16, e_{13}) (16, S, TC) → (E, S, TC)
C_22-c, C_23-c, C_24-c	(E, 13, TE) (13, 17, e_{13}) (15, S, (17, S, TN[26]) → (E, S, TN[26])
C_22-d, C_23-d, C_24-d	(E, 13, TE) (13, 18, e_{13}) (18, S, TN[32]) → (E, S, TN[32])

Tableau V.4 : Modifications des chaînes de dépendances dues à la faute F10

V.3.1.3 Fautes F11 et F3

L'activation de la faute F11 conduit à la création d'erreur sur les variables TN[26], TN[27], TN[32] et TN[33] ; les quatre erreurs initiales créées ne propagent pas et affectent directement des variables de sortie. Les associations et les chaînes de dépendances ne sont pas modifiées.

Pour la faute F3, les modifications apportées aux chaînes de dépendances sont peu nombreuses, l'erreur initiale créée affectant directement une variable de sortie.

La fonction *recopie-capteur* effectue l'acquisition et le test de valeurs et réalise très peu de calculs, les variables ne sont pour la plupart définies qu'une seule fois et peu utilisées, ce qui explique les modifications simples et peu nombreuses apportées aux chaînes de dépendances par les fautes F1, F3, F10 et F11. La fonction *calcul-d-tsgv*, par contre, réalise beaucoup de calculs et les variables sont utilisées et définies de nombreuses fois favorisant ainsi la propagation d'erreur ; les modifications observées sur les chaînes de dépendances peuvent donc être plus complexes, comme nous allons le voir dans le paragraphe suivant.

V.3.2 Fonction *calcul-d-tsgv*

V.3.2.1 Faute F4

L'activation de la faute F4 crée une erreur initiale sur la variable D_CAL qui n'affecte jamais de variable de sortie et une erreur de type branchement erroné car le

nœud 12 n'est plus dépendant du nœud 2 et peut être exécuté par tous les chemins et non plus seulement par ceux pour lesquels la condition e_2 au nœud 2 était évaluée à Faux.

Le tableau V.5 récapitule les modifications induites par l'activation de F4 sur les chaînes de dépendances :

- si le chemin exécuté dans P_c est C1 (voir annexe A, § A.3.2), l'activation de la faute conduit à exécuter la séquence de nœuds 1 - 6 - 2 - 3 - 4 - 5 - 12 - 13 - 14 - 15 - 16 ... dans P_4 (au lieu de 1 - 2 - 3 - 4 - 5 - 16 ...). Les variables NH0, RH0 et TN[56] sont redéfinies respectivement par les nœuds 13, 14 et 15 mais avec les mêmes valeurs qu'aux nœuds 3, 4 et 5. Aucune erreur n'est donc créée sur ces variables et l'erreur sur D_CAL est masquée car non utilisée durant la suite de l'exécution (cf. cas C1-a, b et c) ;
- si le chemin exécuté dans P_c est C2, l'activation de la faute conduit à exécuter la séquence 1 - 6 - 2 - 7 - 8 - 9 - 12 dans P_4 (au lieu de 1 - 2 - 6 - 7 - 8 - 9 - 12) ; le déplacement du nœud 6 n'a aucun impact sur le flot des données, il est exécuté dans P_c et P_4 et l'erreur sur D_CAL est annulée (cf. cas C2-a) ;
- si le chemin exécuté dans P_c est C3, l'activation de la faute conduit à exécuter la séquence 1 - 6 - 2 - 7 - 10 - 11 - 12 dans P_4 (au lieu de 1 - 2 - 6 - 7 - 10 - 11 - 12) annulant ainsi l'erreur sur D_CAL (cf. cas C3-a)
- si le chemin exécuté dans P_c est C4 ou C5 ou ... C11, l'activation de la faute conduit à exécuter la séquence 1 - 6 - 2 - 7 - ...- 12 - 16 ... dans P_4, car e_{12} est toujours évaluée à Faux, l'erreur sur D_CAL est annulée (cf. cas C4-a à C11-a).

Cas	Chaîne de dépendances observée dans P_c → chaîne de dépendances observée dans P_4
C1-a	(2, 3, e_2) (3, 19, NH0) → (2, 3, e_2) (3, 12, NH0) (12, 13, e_{12}) (13, 19, ***NH0***)
C1-b	(2, 4, e_2) (4, S, RH0) → (2, 3, e_2) (3, 12, NH0) (12, 14, e_{12}) (14, S, ***RH0***)
C1-c	(2, 5, e_2) (5, S, TN[56]) → (2, 3, e_2) (3, 12, NH0) (12, 15, e_{12}) (15, S, ***TN[56]***)
C2-a C3-a	(2, 6, e_2) (6, 12, D_CAL) (12, 13, NH0) ... → (6, 12, ***D_CAL***) (12, 13, NH0) ...
C4-a à C11-a	(2, 6, e_2) (6, 12, D_CAL) → (6, 12, ***D_CAL***)

Tableau V.5 : Modifications des chaînes de dépendances dues à la faute F4

Cette analyse des modifications sur les chaînes de dépendances dues à F4 nous permet de conclure que les erreurs initiales sur D_CAL et e_{12} ne conduisent jamais à défaillance. De la même manière que la faute F1 (cf. § V.3.1.1), F4 ne peut être révélée en phase de test ; elle a d'ailleurs été identifiée lors de relectures du code source. Les comportements erronés générés par F6 sont similaires à ceux produits par F4.

V.3.2.2 Faute F5

La faute F5 consiste à supprimer les nœuds 12 à 15 du graphe ainsi que les associations correspondantes. F5 est activée lorsqu'un des chemins de C2 à C11 est exécuté dans P_C (exécution du nœud 12) mais ne crée des erreurs que lors de l'exécution des chemins C2 ou C3 pour lesquels l'erreur initiale au nœud 12 conduit à l'exécution de la branche 13-14-15 dans P_C. Les modifications apportées aux chaînes de dépendances sont alors importantes (cf. tableau V.6).

On observe que l'erreur créée sur la variable NH0, redéfinie au nœud 13 dans P_C et non dans P_5 et ensuite utilisée aux nœuds de contrôle 19 et 37 propage en créant de nombreuses erreurs. Si le chemin exécuté dans P_C est C2 :

- des erreurs sont créées sur les variables NH0, RH0 et TN[56] qui ne sont pas redéfinies par la suite ; ces erreurs sont détectées en sortie du programme (cas C2-a à c) ;
- l'erreur créée sur la variable NH0 propage sur la condition e_{19} ; cette erreur de branchement propage en créant des erreurs sur les variables NMH0, RMH0, D_CAL, R_alpha, R1_alpha et TN[55] (cas C2-d à f). L'erreur créée sur RMH0 (au nœud 29) est due à la combinaison de l'erreur créée sur RH0 au nœud 9 et de l'erreur de branchement au nœud 24 (cf. cas C2-d). Les erreurs sur NMH0, RMH0 et TN[55] sont annulées suite à la redéfinition de ces variables respectivement aux nœuds 32, 33 et 34 dans P_4 ; en effet, les nœuds 20, 21 et 22 exécutés dans P_C les définissent avec les mêmes valeurs ;
- aucune erreur n'est créée sur les variables N1 et R1 puisque les erreurs sur les variables utilisées dans leur définition, NMH0 et RMH0, ont été annulées (cf. cas C2-g et C2-h) ;
- pour la même raison, aucune erreur n'est créée sur e_{37}, ni sur les variables NDH0 et TN[51] (cas C2-i et C2-j).

Cas	Chaîne de dépendances observée dans P_c → chaîne de dépendances observée dans P_5
C2-a	(8, 12, NH0) (12, 13, e_{12}) (13, S, **NH0**) → (8, S, **NH0**)
C2-b	(8, 12, NH0) (12, 14, e_{12}) (14, S, **RH0**) → (8, 9, NH0) (9, S, **RH0**)
C2-c	(8, 12, NH0) (12, 15, e_{12}) (15, S, **TN[56]**) → (E, S, **TN[56]**)
C2-d	(8, 12, NH0) (12, 13, e_{12}) (13, 19, **NH0**) (19, 20, e_{19}) (20, S, ***NMH0***) → (8,19, **NH0**) (19, 23, e_{19}) (23, 31, **D_CAL**) (31, 32, e_{31}) (32, S, ***NMH0***) → (8,19, **NH0**) (19, 24, e_{19}) (24, 27, e_{24}) (27, 28, **R_alpha**) (28, 29, **R1_alpha**) (29, 30, **RMH0**) (30, 31, **NMH0**) (31, 32, e_{31}) (32, S, ***NMH0***) → (8, 9, **NH0**) (9, 29, **RH0**) (29, 30, **RMH0**) (30, 31, **NMH0**) (31, 32, e_{31}) (32, S, ***NMH0***)
C2-e	(8, 12, NH0) (12, 13, e_{12}) (13, 19, **NH0**) (19, 21, e_{19}) (21, S, ***RMH0***) → (8,19, **NH0**) (19, 24, e_{19}) (24, 27, e_{24}) (27, 28, **R_alpha**) (28, 29, **R1_alpha**) (29, 30, **RMH0**) (30, 31, **NMH0**) (31, 33, e_{31}) (33, S, ***RMH0***)
C2-f	(8, 12, NH0) (12, 13, e_{12}) (13, 19, **NH0**) (19, 22, e_{19}) (22, S, ***TN[55]***) → (8,19, **NH0**) (19, 24, e_{19}) (24, 27, e_{24}) (27, 28, **R_alpha**) (28, 29, **R1_alpha**) (29, 30, **RMH0**) (30, 31, **NMH0**) (31, 34, e_{31}) (34, S, ***TN[55]***)
C2-g	(8, 12, NH0) (12, 13, e_{12}) (13, 19, **NH0**) (19, 20, e_{19}) (20, 36, ***NMH0***) (36, S, N1) → (8,19, **NH0**) (19, 24, e_{19}) (24, 27, e_{24}) (27, 28, **R_alpha**) (28, 29, **R1_alpha**) (29, 30, **RMH0**) (30, 31, **NMH0**) (31, 32, e_{31}) (32, 36, ***NMH0***) (36, S, N1)
C2-h	(8, 12, NH0) (12, 13, e_{12}) (13, 19, **NH0**) (19, 21, e_{19}) (21, 35, ***RMH0***) (35, S, R1) → (8,19, **NH0**) (19, 24, e_{19}) (24, 27, e_{24}) (27, 28, **R_alpha**) (28, 29, **R1_alpha**) (29, 30, **RMH0**) (30, 31, **NMH0**) (31, 33, e_{31}) (33, 35, ***RMH0***) (35, S, R1)
C2-i	(8, 12, NH0) (12, 13, e_{12}) (13, 19, **NH0**) (19, 20, e_{19}) (20, 37, ***NMH0***) (37, 38, e_{37}) (38, S, NDH0) → (8,19, **NH0**) (19, 24, e_{19}) (24, 27, e_{24}) (27, 28, **R_alpha**) (28, 29, **R1_alpha**) (29, 30, **RMH0**) (30, 31, NMH0) (31, 32 e_{31}) (32, 37, ***NMH0***) (37, 38, e_{37}) (38, S, NDH0)
C2-j	(8, 12, NH0) (12, 13, e_{12}) (13, 19, **NH0**) (19, 20, e_{19}) (20, 37, ***NMH0***) (37, 39, e_{37}) (39, S, TN[51]) → (8,19, **NH0**) (19, 24, e_{19}) (24, 27, e_{24}) (27, 28, **R_alpha**) (28, 29, **R1_alpha**) (29, 30, **RMH0**) (30, 31, **NMH0**) (31, 32 e_{31}) (32, 37, ***NMH0***) (37, 38, e_{37}) (39, S, TN[51])
C3-a	(11, 12, NH0) (12, 13, e_{12}) (13, S, **NH0**) → (11, S, **NH0**)
C3-b	(11, 12, NH0) (12, 14, e_{12}) (14, S, **RH0**) → (10, S, **RH0**)
C3-c	(11, 12, NH0) (12, 15, e_{12}) (15, S, **TN[56]**) → (E, S, **TN[56]**)
C3-d à C3-j	idem que C2-d à C2-j, en remplaçant l'association (8, 12, NH0) en début de chaque chaîne par l'association (11, 12, NH0)

Tableau V.6 : Modifications des chaînes de dépendances dues à la faute F5

L'exécution du chemin C3 conduit à des comportements erronés similaires : il suffit de remplacer l'association (8, 12, NH0) par (11, 12, NH0).

On constate donc que les erreurs créées par l'activation de F5 propagent beaucoup, pouvant ainsi créer des flots d'erreurs complexes ; mais peu d'erreurs sont détectées et beaucoup sont annulées en cours d'exécution.

V.3.2.3 Faute F9

La faute F9 est activée lors de l'exécution du nœud 28, c'est-à-dire lors de l'exécution des chemins C5, C6, C7, C9, C10 ou C11. L'activation de F9 génère une erreur initiale de type "variable erronée" sur R1_alpha. La faute F9 est une faute simple, similaire à une mutation, et le tableau V.6 montre qu'elle a cependant des effets importants sur les chaînes de dépendances du programme.

L'erreur initiale propage en créant une erreur sur RMH0 puis NMH0 définies respectivement au nœud 29 et au nœud 30. Ces erreurs sur RMH0 et NMH0 propagent ensuite de diverses manières suivant les chemins exécutés (cf. tableau V.6) :

- si le chemin exécuté dans P_c est C5 ou C9, le tableau V.6 récapitule les modifications apportées aux chaînes de dépendances et montre quelles variables sont affectées par la propagation d'erreur. Certaines erreurs sont annulées suite à des redéfinitions ; c'est le cas des erreurs sur NDH0 et TN[51] qui sont redéfinies dans P_9 aux mêmes valeurs que celles attribuées par d'autres nœuds dans P_c.
- si le chemin exécuté dans P_c est C6 ou C10, la propagation d'erreur est similaire à celle observée pour les chemins C5 et C9 jusqu'à l'exécution du nœud 43 (cas C6 ou C10-a à f). Ensuite les modifications apportées aux chaînes de dépendances sont différentes car les nœuds 44 et 45 ne sont pas exécutés ; les erreurs sur NDH0 et TN[51] ne sont donc pas annulées (cf. tableau, cas C6 ou C10-g et h).
- si le chemin exécuté dans P_c est C7 ou C11, l'erreur initiale propage en créant des erreurs sur les variables RMH0 et NMH0 (aux nœuds 29 et 30) qui sont ensuite annulées (redéfinitions). Aucune erreur ne propage donc au nœud de contrôle 37 et l'erreur initiale n'affecte pas de variable de sortie : aucune défaillance n'est donc observée (cf. cas C7 ou C11-a et b).

Cas	Chaîne de dépendances observée dans P_c → chaîne de dépendances observée dans P_9
C5-a, C9-a C6-a, C10-a	(28, 29, **R1_alpha**) (29, S, **RMH0**) → (28, 29, **R̄1_alpha**) (29, 30, **RMH0**) (30, 31, **NMH0**) (31, 33, e_{31}) (33, S, **RM̄H0**)
C5-b, C9-b C6-b, C10-b	(28, 29, **R1_alpha**) (29, 30, **RMH0**) (30, S, **NMH0**) → (28, 29, **R̄1_alpha**) (29, 30, **RMH0**) (30, 31, **NMH0**) (31, 32, e_{31}) (32, S, **NM̄H0**)
C5-c, C9-c C6-c, C10-c	(E, S, **TN[55]**) → (28, 29, **R1_alpha**) (29, 30, **RMH0**) (30, 31, **NMH0**) (31, 33, e_{31}) (33, S, **TN[55]**)
C5-d, C9-d C6-d, C10-d	(28, 29, R1_alpha) (29, 35, **RMH0**) (35, S, **R1**) → (28, 29, **R1_alpha**) (29, 30, **RMH0**) (30, 31, **NMH0**) (31, 33, e_{31}) (33, 35, **RM̄H0**) (35, S, **R1**)
C5-e, C9-e C6-e, C10-e	(28, 29, **R1_alpha**) (29, 30, **RMH0**) (30, 36, **NMH0**) (36, S, N1) → (28, 29, **R̄1_alpha**) (29, 30, **RMH0**) (30, 31, **NMH0**) (31, 32, e_{31}) (32, 36, **NM̄H0**) (36, S, N1)
C5-f, C9-f C6-f, C10-f	... (30, 37, **NMH0**) (37, 41, e_{37}) (41, S, **RDH0**) → (E, S, **RDH0**)
C5-g C9-g	... (30, 37, **NMH0**) (37, 41, e_{37}) (41, 42, **RDH0**) (42, 43, **NDH0**) (43, 44, e_{43}) (44, S, ***NDH0***) → ... (30, 31, **NMH0**) (31, 32, e_{31}) (32, 37, **NMH0**) (37, 38, e_{37}) (38, S, ***NDH0***)
C5-h C9-h	... (30, 37, **NMH0**) (37, 41, e_{37}) (41, 42, **RDH0**) (42, 43, **NDH0**) (43, 45, e_{43}) (45, S, ***TN[51]***) → ... (30, 31, **NMH0**) (31, 32, e_{31}) (32, 37, **NMH0**) (37, 39, e_{37}) (39, S, ***TN[51]***)
C6-g C10-g	... (30, 37, **NMH0**) (37, 41, e_{37}) (41, 42, **RDH0**) (42, S, **NDH0**) → ... (30, 31, **NMH0**) (31 32, e_{31}) (32, 37, **NMH0**) (37, 38, e_{37}) (38, S, **NDH0**)
C6-h C10-h	(E, S, **TN[51]**) → ... (30, 31, **NMH0**) (31, 32, e_{31}) (32, 37, **NMH0**) (37, 39, e_{37}) (39, S, **TN[51]**)
C7-a, C11-a	(28, 29, **R1_alpha**) (29, 30, **RMH0**) (30, 31, **NMH0**) (31, 32, e_{31}) (33, S, ***RMH0***) → (28, 29, **R1_alpha**) (29, 30, **RMH0**) (30, 31, **NMH0**) (31, 32, e_{31}) (32, S, ***RM̄H0***)
C7-b, C11-b	(28, 29, **R1_alpha**) (29, 30, **RMH0**) (30, 31, **NMH0**) (31, 33, e_{31}) (33, S, ***RMH0***) → (28, 29, **R1_alpha**) (29, 30, **RMH0**) (30, 31, **NMH0**) (31, 33, e_{31}) (33, S, ***RM̄H0***)

Tableau V.7 : Modifications des chaînes de dépendances dues à la faute F9

V.3.3 Conclusions

Cette étude nous a permis sur un cas réel d'appliquer l'analyse des dépendances pour identifier et commenter les modifications apportées par une faute aux chaînes de dépendances. L'analyse de ces modifications permet de mettre en évidence les mécanismes de création d'erreur (propagation d'un nœud du graphe à un autre), d'annulation d'erreur (redéfinition) et de masquage d'erreur (non utilisation) ainsi que d'interaction entre erreurs.

Nous allons, dans le paragraphe suivant, utiliser notre connaissance des relations de dépendances sur le programme LOCALES et des modifications apportées aux associations et aux chaînes de dépendances par des fautes réelles pour sélectionner a priori des mutations susceptibles de reproduire totalement ou partiellement les comportements erronés générés par les fautes réelles.

V.4 COMPORTEMENTS ERRONÉS REPRODUITS PAR LES MUTATIONS

Les expériences commentées dans ce paragraphe ont consisté à :

1) sélectionner des mutations parmi un ensemble de mutations générées à partir des tables jointes en annexe B et selon le principe décrit au paragraphe V.4.1.1 ;
2) comparer, selon la méthode présentée au paragraphe V.4.1.2, les comportements erronés (c'est-à-dire les flots d'erreurs) produits par, d'une part, 9 des 13 fautes réelles étudiées et d'autre part, chacune des mutations sélectionnées.

Rappelons que parmi les treize fautes réelles identifiées sur le programme LOCALES, une faute (F2) n'a pu être étudiée sur la version 1.4 du programme (considérée comme la version correcte) et trois programmes incorrects associés aux fautes F1, F4 et F6 sont équivalents au programme original. Nous n'avons pas cherché à sélectionner des mutations reproduisant, dans le cas des fautes F1, F4 et F6, les mécanismes de masquage et d'annulation d'erreur bien que ce soit possible.

Les mutations sélectionnées et les résultats de la comparaison des comportements erronés sont détaillés au paragraphe V.4.2.

V.4.1 Cadre des expériences

V.4.1.1 Principe de sélection

La sélection des mutations consiste d'abord à cibler certains nœuds du graphe définition-utilisation (qui ne sont pas forcément ceux affectés par la faute réelle), puis certains types de mutation : l'objectif de cette sélection est de montrer que des fautes simples de natures syntaxiques différentes et localisées à divers endroits du graphe peuvent reproduire partiellement ou complètement les comportements erronés dus aux fautes réelles étudiées. Soient F une faute réelle et v un des nœuds affectés par cette faute, les nœuds correspondant aux critères suivants ont été sélectionnés pour y injecter des mutations :

a) nœud v correspondant soit à une définition de variable, soit à un branchement affecté lors de l'activation de la faute réelle F ;

b) nœud u correspondant à une définition en amont d'une variable utilisée au nœud v ;

c) nœud w correspondant à une utilisation en aval d'une variable définie au nœud v.

Le choix des mutations a ensuite porté :

- dans le cas a), sur une mutation susceptible de créer une des erreurs initiales générées par l'activation de la faute F au nœud v ;
- dans le cas b), sur une mutation susceptible de créer une erreur qui propagera jusqu'au nœud v et qui reproduira un ou plusieurs flots d'erreurs générés par la faute F à partir du nœud v ;
- dans le cas c), sur une mutation susceptible de reproduire, à partir du nœud w, la propagation d'erreur due à une des erreurs initiales créée par la faute F.

V.4.1.2 Comparaison des comportements erronés

Pour confirmer le fait que notre sélection a priori a permis d'identifier des mutations capables de reproduire partiellement ou totalement les comportements erronés

générés par les fautes réelles, des analyses similaires à celles menées pour l'application ETUD (cf. § III.4) ont été menées.

Chacun des programmes incorrects a été obtenu en injectant une à une les fautes réelles ou les mutations sélectionnées dans le programme original (correspondant à la version 1.4 du programme LOCALES). Les **traces d'exécution** des programmes incorrects et du programme original ont été obtenues en exécutant ces programmes avec des **séquences de test** comportant chacune entre 20 et 50 vecteurs pour la fonction *recopie-capteur* et entre 11 et 30 vecteurs pour la fonction *calcul-d-tsgv*. Ces séquences de test ont été sélectionnées parmi les jeux de test statistique dont nous disposions : cinq jeux de test pour la fonction *recopie-capteur* comportant chacun 307 vecteurs et cinq jeux de test pour la fonction *calcul-d-tsgv* comportant chacun 97 vecteurs (cf. § II.3.4.2). Chaque séquence de test permet d'exécuter au moins une fois chacun des chemins identifiés dans chacune des fonctions.

Afin de confirmer la similitude des comportements erronés, nous avons procédé, pour chaque mutation susceptible de reproduire les comportements erronés d'une faute réelle, de la manière suivante :

i) chaque mutant et le programme incorrect contenant la faute réelle ont été exécutés avec la même séquence de test ;

ii) les traces d'exécution et les traces d'erreurs correspondantes ont été générées ;

iii) les flots d'erreurs et les défaillances produits par chaque mutation ont été comparés à ceux produits pour la faute réelle, par analyse manuelle des traces d'erreurs et des résultats de test.

V.4.2 Résultats expérimentaux

Une centaine de mutations ont été sélectionnées selon le principe exposé au § V.4.1.1. Nous ne détaillerons ici qu'une trentaine d'entre elles, représentatives de nos observations. Pour chacune d'elles, nous allons préciser la raison de son choix et les comportements erronés qu'elle a reproduits. L'ensemble des résultats est synthétisé dans les tableaux V.8 (pour la fonction *recopie-capteur*) et V.9 (pour la fonction *calcul-d-tsgv*) qui comprennent les informations suivantes :

- un numéro permettant d'identifier la mutation ;
- le nœud du graphe définition-utilisation sur lequel la mutation a été injectée (cf. graphes dans l'annexe A) ainsi que la modification apportée (modification d'un

opérateur, d'une constante[27] ou d'un symbole représentant le nom d'une variable). Par exemple, la modification "EIO2" → "EIO1" signifie que le symbole "EIO2" a été remplacé par le symbole "EIO1" et la modification "=" → "==" signifie que l'opérateur d'affectation "=" a été remplacé par l'opérateur de comparaison "==" dans le mutant correspondant ;

- les comportements erronés reproduits par la mutation, c'est-à-dire soit les flots d'erreurs relatifs à la faute réelle considérée que la mutation a reproduits, soit les défaillances observées en l'absence de propagation d'erreurs.

V.4.2.1 Fonction recopie-capteur

Dans le cadre de la fonction *recopie-capteur*, les fautes étudiées créent des erreurs initiales qui affectent directement des variables de sortie sans propager au cours de l'exécution du programme ; cela est dû à la sémantique de la fonction. De plus, pour chacune de ces fautes, il n'existe pas d'autre définition de ces variables erronées que les nœuds affectés par ces fautes. On peut donc chercher à reproduire les mêmes comportements erronés, en l'occurence les mêmes erreurs initiales et les mêmes défaillances, en injectant une mutation sur chacun des nœuds affectés.

Ainsi, l'activation de F3 crée une erreur initiale sur la variable ED0 définie au nœud 68 ; les mutations sélectionnées sont des mutations de type remplacement d'un symbole (M1-F3) ou d'un opérateur (M2-F3 et M3-F3) qui créent la même erreur initiale pour tout ou partie des entrées de test.

L'activation de F10 crée des erreurs initiales sur les variables définies aux nœuds 15 à 18 et 44 à 47, suite à la suppression de ces nœuds ; ces erreurs ne propagent pas et affectent directement des variables de sortie. Une mutation seule ne peut reproduire l'ensemble de ces erreurs mais une mutation sélectionnée sur chacun de ces nœuds permet de reproduire chacune des erreurs initiales individuellement. Chacune des mutations ainsi sélectionnées est soit de type remplacement de l'opérateur d'affectation par un opérateur d'un autre type (M1-F10, M3-F10, M4-F10 et M5-F10), reproduisant ainsi la suppression d'une instruction, soit de type modification de la constante utilisée pour définir la variable au nœud considéré (M2-F10). Toutes les mutations sélectionnées ont généré en partie les erreurs initiales créées par F10 et qui affectaient directement des variables de sortie.

---

27 Les constantes dans le programmes LOCALES sont codées "K_XXX".

L'activation de F11 crée des erreurs initiales sur les variables TN[26], TN[27], TN[32] et TN[33] par utilisation de constantes erronées aux nœuds les définissant. Il suffit pour reproduire chacune des erreurs initiales d'injecter sur chacun de ces nœuds, une mutation de type modification de la constante symbolique qui reproduit la même erreur initiale (M1-F11 à M4-F11).

Le tableau V.8 donne le résultat de l'analyse comparative des comportements erronés dus aux fautes réelles et aux mutations sélectionnées.

Faute	Mutations injectées		Comportements erronés reproduits
	N°	Description	
F3	M1-F3	nœud 68 : "EIO2" → "EIO1"	Mêmes défaillances
	M2-F3	nœud 68 : "|" → "&"	Mêmes défaillances pour certains vecteurs de test
	M3-F3	nœud 68 : "|" → "^"	Mêmes défaillances pour certains vecteurs de test
F10	M1-F10	nœud 17 : "=" → "=="	Mêmes défaillances relatives à TN[26]
	M2-F10	nœud 17 :"K_F_ER" → "0"	Mêmes défaillances relatives à TN[26]
	M3-F10	nœud 18 : "=" → "=="	Mêmes défaillances relatives à TN[32]
	M4-F10	nœud 46 : "=" → "=="	Mêmes défaillances relatives à TN[27]
	M5-F10	nœud 47 : "=" → "=="	Mêmes défaillances relatives à TN[33]
F11	M1-F11	nœud 17 : "K_F_ER" → "K_ER"	Mêmes défaillances relatives à TN[26]
	M2-F11	nœud 18 : "K_F_ER" →"K_ER"	Mêmes défaillances relatives à TN[32]
	M3-F11	nœud 46 : "K_F_ER" → "K_ER"	Mêmes défaillances relatives à TN[27]
	M4-F11	nœud 47 : "K_F_ER" → "K_ER"	Mêmes défaillances relatives à TN[33]

Tableau V.8 : Similitudes entre les flots d'erreurs produits par les fautes réelles et par les mutations sur la fonction *recopie-capteur*

V.4.2.2 Fonction calcul-d-tsgv

Dans le cadre de la fonction *calcul-d-tsgv*, les fautes étudiées génèrent des comportements plus complexes en terme de création, d'annulation et de masquage d'erreurs. Pour reproduire ces comportements erronés, nous avons pu injecter des mutations sur divers nœuds du graphe définition-utilisation et non pas seulement sur les nœuds sur lesquels une erreur initiale a été créée.

La faute F5 consiste à supprimer les nœuds 12 à 15, le nœud 12 étant un nœud de contrôle et les nœuds 13 à 15 étant dépendants par le contrôle du nœud 12. On peut chercher à reproduire les mêmes comportements erronés de différentes manières :

- soit en injectant une mutation de type remplacement d'un opérateur, sur le nœud 12 conduisant à évaluer e_{12} à Faux (cf. M1-F5), ce qui revient à supprimer les associations (12, 13, e_{12}), (12, 14, e_{12}) et (12, 15, e_{12}) ;
- soit en ciblant les définitions de NH0 au nœud 13 et en amont du nœud 13 (nœuds 8 et 11) : la propagation d'erreur étant due essentiellement à l'erreur initiale sur NH0 créée au nœud 13, on peut chercher à reproduire les flots d'erreurs correspondants. Les mutations ainsi sélectionnées consistent donc, soit à modifier la variable utilisée pour définir NH0 (par exemple, M5-F5 au nœud 11), soit à supprimer la définition de NH0 au nœud 8 (M4-F5) ou au nœud 13 (M2-F5 et M3-F5) par une mutation de type remplacement de l'opérateur d'affectation ou par une mutation du type remplacement du symbole de la variable définie. Dans ce dernier cas, la mutation sélectionnée (M3-F5) conduit à remplacer la variable définie (NH0) par une autre variable (NDH0) créant ainsi une erreur sur la variable non définie dans le mutant (NH0) et sur la variable non définie dans le programme original (NDH0) ; cette dernière erreur ne propage pas, et est annulée par redéfinition de NDH0 dans les nœuds qui suivent.

La faute F7 consiste à supprimer les nœuds 31 à 34, le nœud 31 étant un nœud de contrôle et les nœuds 32 à 34 étant dépendants par le contrôle du nœud 31. De la même manière que pour F5, on peut reproduire les comportements erronés en sélectionnant une mutation sur le nœud 31 (M1-F7). La propagation d'erreur étant due essentiellement à l'erreur initiale sur NMH0, nous avons ciblé cette fois la définition de NMH0 au nœud 32 (M2-F7) et l'utilisation de NMH0 au nœud 37 (M3-F7).

L'activation de F8 crée une erreur initiale de type "branchement erroné" au nœud 37 ; nous avons sélectionné des mutations sur ce nœud susceptibles de créer ce même branchement erroné (M1-F8 et M2-F8). Ces mutations sont de type remplacement d'un opérateur ; elles reproduisent partiellement les comportements erronés dus à F8 (c'est-à-dire pour certains chemins exécutables). Il aurait également été possible de reproduire partiellement certains flots d'erreurs dus à F8 en sélectionnant des mutations soit sur les nœuds définissant les variables utilisées au nœud 37 et visibles depuis ce nœud (par exemple, sur les nœuds 20, 25, 30 ou 32 pour la variable NMH0 et sur les nœuds 3, 8,

11 ou 13 pour la variable NH0), soit sur les nœuds dépendants par le contrôle du nœud 37 (par exemple les nœuds 38, 39, 41, 42 ou 43) afin de reproduire les erreurs propageant sur certaines variables.

Faute	Mutations injectées		Comportements erronés reproduits
	N°	Description	
F5	M1-F5	nœud 12 : " || " → "&&"	Tous les flots d'erreurs.
	M2-F5	nœud 13 : " = " → "=="	Les flots d'erreurs relatifs à la propagation de l'erreur initiale sur NH0.
	M3-F5	nœud 13 : "NH0" → "NDH0"	Les flots d'erreurs relatifs à la propagation de l'erreur initiale sur NH0.
	M4-F5	nœud 8 : " = " → "=="	Les flots d'erreurs produits lors de l'exécution par P_C du chemin C2.
	M5-F5	nœud 11 : " RH0 " → "RMH0"	Les flots d'erreurs produits lors de l'exécution par P_C du chemin C3.
F7	M1-F7	nœud 31 : " || " → "&&"	Tous les flots d'erreurs.
	M2-F7	nœud 32 : "NMH0" → "NDH0"	Les flots d'erreurs relatifs à la propagation de l'erreur initiale sur NMH0.
	M3-F7	nœud 37 : "K_D" → "K_MIN"	Flots d'erreurs relatifs à la propagation de l'erreur initiale sur NMH0 partiellement à partir du nœud 37.
F8	M1-F8	nœud 37 : " || " → "&&"	Les flots d'erreurs produits lors de l'exécution par P_C des chemins C7 ou C11.
	M2-F8	nœud 37 : " || " → " ^ "	Les flots d'erreurs produits lors de l'exécution par P_C des chemins C1 ou C2 .
F9	M1-F9	nœud 28 : "1.0" → "10.0"	Tous les flots d'erreurs.
	M2-F9	nœud 29 : "*" → "+"	Les flots d'erreurs produits lors de l'exécution par P_C des chemins C7 ou C11 mais valeurs erronées différentes aux nœuds 29 et 30.
	M3-F9	nœud 30 : "10.0" → "100.0"	Les mêmes flots d'erreurs lors de l'exécution par P_C des chemins C5, C6, C9, C10 ou C11 mais valeurs erronées différentes pour NMH0 (nœud 30).
F12	M1-F12	nœud 5 : "TN[56] → "TN[54]"	Défaillances sur TN[56] mais valeur erronée différente.
	M2-F12	nœud 15 :"TN[56] → "TN[54]"	Défaillances sur TN[56] mais valeur erronée différente.
F13	M1-F13	nœud 5 : "TN[55] → "TN[54]"	Défaillances sur TN[55] mais valeur erronée différente.
	M2-F13	nœud 15 :"TN[55] → "TN[54]"	Défaillances sur TN[55] mais valeur erronée différente.

Tableau V.9 : Similitudes entre les flots d'erreurs produits par les fautes réelles et par les mutations sur la fonction *calcul-d-tsgv*

ont été sélectionnées sur ce nœud (M1-F9) et sur une utilisation de cette variable (M2-F9 au nœud 29) mais également sur un nœud dépendant du nœud 29 par les données (M3-F9 au nœud 30) pour reproduire les comportements erronés dus à la faute F9. Les mutations M1-F9 et M3-F9 sont de type remplacement d'une constante numérique dans des calculs réalisés pour définir respectivement R1_alpha et NMH0. M1-F9 crée la même erreur initiale que F9 sur R1-alpha. M3-F9 crée une erreur initiale sur NMH0 qui sera ensuite annulée au nœud 32, cette erreur conduit à un branchement erroné au nœud 31 ; les flots d'erreurs dus à F9 sont alors reproduits à partir de ce nœud 31. La mutation M2-F9 est de type remplacement d'un opérateur ; les flots d'erreurs dus à F9 sont également reproduits à partir du nœud 31 pour certains des chemins exécutés.

La faute F12 consiste à définir la variable TN[54] à la place de TN[56] au nœud 5 (lors de l'exécution du chemin C1) et au nœud 15 (lors de l'exécution des chemins C2 et C3) et à utiliser dans ces définitions des constantes erronées. Ces variables constituant des variables de sortie et étant seulement définies par ces nœuds, les mutations n'ont pu être sélectionnées que sur ces nœuds. Il en a été de même pour la faute F13. Les mutations destinées à reproduire les erreurs créées par les fautes F12 et F13 ont permis de créer des erreurs sur les mêmes variables (qui ont pu être détectées par les mêmes vecteurs de test) mais avec des valeurs erronées différentes.

Le tableau V.9 donne le résultat de l'analyse comparative des comportements erronés dus aux fautes réelles et aux mutations sélectionnées.

V.4.2.3 Commentaires

L'étude des tableaux V.8 et V.9 montre que toutes les mutations étudiées reproduisent totalement ou partiellement les comportements erronés générés par les fautes réelles. En effet :

1) soit la mutation sélectionnée est activée par les mêmes entrées de test que la faute réelle considérée et elle reproduit les mêmes flots d'erreurs et les mêmes défaillances que ceux produits par la faute ; c'est le cas des mutations M1-F3, M1-F5, M1-F7, M1-F9.

2) soit la mutation sélectionnée n'est activée que par un sous-ensemble des entrées de test permettant d'activer la faute réelle mais elle reproduit pour ces entrées (c'est-à-dire pour certains chemins exécutés) les mêmes flots d'erreur et les mêmes défaillances que ceux produits par la faute réelle ; c'est le cas des mutations M2-F3, M3-F3, M4-F5, M5-F5, M1-F8 et M2-F8.

3) soit la mutation étudiée génère des comportements erronés similaires à ceux produits par la faute réelle et ceci pour les mêmes entrées de test : même structure du flot d'erreurs mais valeurs erronées différentes. C'est le cas des mutations M2-F9, M3-F9, M1-F12, M2-F12, M1-F13 et M2-F13.

4) soit la mutation étudiée reproduit partiellement les comportements erronés produits par la faute réelle considérée (c'est-à-dire un sous-ensemble des flots d'erreurs ou une partie d'un flot d'erreurs menant à la même défaillance) : c'est le cas pour les fautes réelles qui créent des erreurs initiales sur plusieurs variables, comme les fautes F10 ou F11 (cf. M1-F10 à M5-F10 et M1-F11 à M4-F11). C'est également le cas des mutations injectées en aval du nœud affecté par la faute, elles reproduisent alors un sous-flot d'erreurs à partir d'une erreur donnée (cf. M2-F5, M3-F5, M2-F7 et M3-F7.

Ces expériences sont complémentaires de celles menées sur le programme ETUD, pour lequel les mutations étudiées n'avaient pas été sélectionnées a priori pour reproduire les comportements erronés générés par les fautes. En sélectionnant a priori des mutations susceptibles de reproduire tout ou partie des comportements erronés créés par des fautes, nous avons obtenu de nouveaux arguments favorables à la représentativité des erreurs générées par les mutations.

Nous présentons à titre indicatif, au paragraphe suivant, les résultats obtenus par l'ensemble des jeux de test vis-à-vis des fautes réelles et des mutations étudiées.

V.4.2.4 Résultats vis-à-vis des jeux statistiques

De façon plus globale, nous avons comparé les résultats obtenus par l'ensemble des jeux de test statistique complets vis-à-vis des fautes réelles et des mutations étudiées en termes de pourcentage de vecteurs d'entrée révélant la faute ou la mutation.

En ce qui concerne les fautes F1, F4 et F6, nous avons pu vérifier qu'elles ne sont révélées par aucun des jeux de test. Nous avons également observé que la combinaison des fautes F2 et F10 conduit au même résultat que F10 seule.

Pour les autres fautes et les mutations, les résultats sont rassemblés dans le tableau V.10 (pour la fonction *recopie-capteur*) et dans le tableau V.11 (pour la fonction *calcul-d-tsgv*) qui donnent le pourcentage moyen des vecteurs de test révélant la faute ou la mutation sur l'ensemble des cinq jeux de test statistique élaborés pour chacune des fonctions *recopie-capteur* et *calcul-d-tsgv*.

Faute		Mutations	
F3	25,99 %	M1-F3	25,99 %
		M2-F3	52,90 %
		M3-F3	24,23 %
F10	99,80 %	M1-F10	98,76 %
		M2-F10	98,76 %
		M3-F10	98,76 %
		M4-F10	99,15 %
		M5-F10	99,15 %
F11	99,80 %	M1-F11	98,76 %
		M2-F11	98,76 %
		M3-F11	99,15 %
		M4-F11	99,15 %

Tableau V.10 : Résultats des jeux de test statistique vis-à-vis des fautes et des mutations sur *recopie-capteur*

Faute		Mutations	
F5	16,70 %	M1-F5	16,70 %
		M2-F5	16,70 %
		M3-F5	16,70 %
		M4-F5	53,40 %
		M5-F5	49,48 %
F7	23,50 %	M1-F7	23,50 %
		M2-F7	23,50 %
		M3-F7	23,50 %
F8	43,30 %	M1-F8	23,51 %
		M2-F8	30,53 %
F9	65,77 %	M1-F9	65,77 %
F12	15,46 %	M1-F12	7,83 %
		M2-F12	13,61 %
F13	35,87 %	M1-F13	22,06 %
		M2-F13	14,85 %

Tableau V.11 : Résultats des jeux de test statistique vis-à-vis des fautes et des mutations sur *calcul-d-tsgv*

En ce qui concerne les fautes réelles, ces résultats varient beaucoup d'une faute à l'autre, ce qui peut indiquer une variété des comportements erronés. Il est alors intéressant de souligner que les mutations étudiées ont pu reproduire aussi bien les comportements erronés dus à des fautes réelles difficiles à révéler par l'ensemble des vecteurs de test telles que F5 ou F12 que ceux dus à des fautes révélées par pratiquement tous les vecteurs de test, telles que F10 et F11. De plus, la majorité des mutations qui ne reproduisent que partiellement les flots d'erreurs dus aux fautes réelles ont produit moins de défaillances et semblent donc plus difficiles à révéler.

V.5 CARACTÉRISATION DES FAUTES

Les travaux sur la modélisation des fautes se sont principalement intéressés à la caractérisation des fautes par leur nature syntaxique. Par exemple, dans [DeMillo & Mathur 1994], DeMillo classe les fautes en deux catégories principales : 1) les fautes simples de type mutation et 2) les fautes complexes qui sont elles-même classées selon leur impact sur le code du programme (code manquant, code déplacé, algorithme incorrect, etc.). La différence entre une faute dite simple et une faute dite complexe est généralement déterminée par la taille syntaxique de la faute, définie par le nombre d'instructions ou d'éléments du code source qui doivent être modifiés pour corriger le programme. Dès lors se pose le problème de la représentativité des mutations vis-à-vis des fautes réelles : puisqu'une mutation est une faute simple, il semble évident qu'elle ne peut pas modéliser des fautes complexes.

Pour répondre à cette limitation, Offut, dans [Offutt & Hayes 1996], a introduit la notion de taille sémantique qui est définie par la taille relative du sous-domaine des entrées pour lesquelles sont observées, lors de l'exécution du programme incorrect, des sorties incorrectes. Une faute est alors de taille sémantique large si de nombreuses entrées de test permettent de la révéler. Offut a constaté qu'il n'y avait pas de corrélation entre la taille syntaxique d'une faute et sa taille sémantique.

Nos résultats expérimentaux nous amènent à nous interroger sur la manière dont ont été modélisées les fautes jusqu'à présent et à proposer une caractérisation des fautes plus appropriée au contexte de la validation des tests.

Dans nos études expérimentales sur le module LOCALES, nous avons caractérisé les fautes réelles, en nous basant sur deux types de représentation du programme : 1) en évaluant le nombre d'instructions du code source affectées par la faute (cf. tableau II.2),

c'est-à-dire le nombre d'instructions corrigées entre les versions incorrecte et correcte du programme, et 2) en évaluant le nombre d'associations de dépendances ajoutées ou supprimées (cf. tableaux V.1 et V.2). Nous avons remarqué que ces deux types de mesure n'étaient pas corrélés : une faute qui affecte une seule instruction du code source peut affecter de nombreuses associations de dépendances (par exemple, la faute F8). De plus, l'analyse détaillée des fautes réelles sur les modules ETUD et LOCALES nous a permis de nous rendre compte expérimentalement que la nature syntaxique d'une faute ne permettait pas de préjuger de la complexité ou de la subtilité des comportements qu'elle peut générer. Ainsi, une faute de complexité syntaxique faible peut être difficile à révéler car soit son activation crée rarement des erreurs initiales, soit son activation crée des erreurs qui sont souvent annulées ou masquées au cours de leur propagation. Le cas contraire est également possible : une faute de complexité syntaxique faible peut créer une erreur initiale qui va propager en créant de nombreuses autres erreurs conduisant à des sous-flots d'erreurs plus ou moins complexes.

Dans le cas du test dont le but est de détecter des erreurs, il nous paraît plus judicieux de caractériser une faute en fonction de la complexité des comportements erronés qu'elle peut générer. Schématiquement, une faute dont l'activation génère un comportement erroné simple est une faute qui, lorsqu'elle est activée, crée une erreur initiale sur une variable de sortie, ne propage pas en créant d'autres erreurs et n'est ni masquée ni annulée durant le reste de l'exécution du programme ; l'erreur initiale créée conduit donc à une défaillance sur la même variable à chaque fois. A l'opposé, une faute dont l'activation génère un comportement erroné complexe est une faute qui, à chaque activation peut créer ou non une ou plusieurs erreurs initiales différentes selon les conditions d'activation et telles que leurs propagations créent d'autres erreurs qui peuvent être annulées ou masquées en cours d'exécution. Cette caractérisation des comportements erronés peut être faite par analyse de l'impact des fautes sur les chaînes de dépendances. Cette analyse, comme nous l'avons expérimenté au paragraphe V.3, permet d'estimer la complexité des comportements erronés qui pourront être générés lors de l'exécution du programme correspondant. Cet axe de recherche nous semble intéressant à poursuivre sur d'autres applications pour caractériser un sous-ensemble de mutations susceptible de produire un échantillon représentatif de comportements erronés.

V.6 CONCLUSIONS

Dans ce chapitre, nous avons mis en pratique l'analyse des dépendances définie au chapitre IV, sur un cas réel : le programme LOCALES. Nous avons pu au travers de l'étude des chaînes de dépendances, mettre en évidence, avant toute exécution du programme, les mécanismes de création, d'annulation et de masquage d'erreur susceptibles d'être produits par l'activation des fautes réelles. Nous avons pu également en ciblant certaines définitions ou utilisations reproduire partiellement ou totalement les comportements erronés générés par les fautes réelles par le biais de mutations. Les mutations sélectionnées montrent qu'une faute simple qui est localisée soit en amont dans le graphe définition-utilisation soit en aval d'un nœud affecté par une faute donnée peut reproduire une partie des flots d'erreurs générés par cette faute. Cela s'explique par les relations de dépendances qui lient le nœud affecté par la faute réelle et le nœud affecté par la mutation sélectionnée.

Ces résultats nous ont amenée à nous interroger sur les critères qui permettaient de juger de la représentativité de fautes artificielles vis-à-vis de fautes réelles. La taille syntaxique d'une faute nous apparaît comme étant un critère insuffisant. La taille sémantique et la complexité des comportements erronés nous apparaissent comme des critères plus adéquats mais difficiles à évaluer de manière systématique.

Les résultats positifs concernant la représentativité des erreurs et des comportements erronés produits par les mutations sur cette seconde étude de cas, nous permettent maintenant de considérer l'analyse de mutation telle que nous l'avons appliquée, c'est-à-dire l'analyse de mutation forte (cf. § I.4.2), comme une technique expérimentale appropriée pour évaluer l'efficacité de jeux de test à détecter des erreurs. L'application de cette technique dans un contexte industriel, ses avantages et ses contraintes font l'objet du chapitre suivant.

CHAPITRE VI

VALIDATION DES TESTS PAR ANALYSE DE MUTATION

VI.1 INTRODUCTION

Les résultats des expériences menées sur les programmes ETUD et LOCALES montrent que les erreurs et les comportements erronés générés par des mutations peuvent être représentatifs de ceux dus à des fautes réelles de nature syntaxique complexe, c'est-à-dire dont la correction concerne plusieurs instructions du code réparties sur plusieurs blocs. Ainsi, l'analyse de mutation permet de disposer d'un large échantillon de comportements erronés et suffisamment variés pour être représentatif d'un nombre non négligeable de comportements erronés générés par des fautes réelles ; il est alors possible d'évaluer l'efficacité de jeux de tests vis-à-vis de la détection de ces comportements erronés.

Rappelons que la mesure associée aux résultats d'une analyse de mutation est le score de mutation (cf. § 1.4.1). Pour un programme P et un jeu de test T, elle correspond à la proportion de mutants non équivalents à P qui sont tués par T. Un score de 100% indique que le jeu de test T tue tous les mutants non équivalents mais, en pratique, ce score idéal est difficilement atteignable.

Dans le cadre de son utilisation dans un contexte industriel, l'inconvénient majeur de cette technique est qu'elle n'est pas entièrement automatisable dans l'état de l'art actuel. L'établissement du score de mutation implique une analyse des mutants non tués (c'est-à-dire vivants) par le jeu de test T pour identifier, parmi eux, ceux qui sont équivalents au programme original. Or cette analyse des mutants vivants doit être faite manuellement, ce qui est coûteux.

De plus, les étapes d'ores et déjà automatiques (génération et exécution des mutants) peuvent être coûteuses en temps d'exécution en raison du nombre important de mutants qui peuvent être construits par génération exhaustive à partir de types de mutations pré-définis. D'autres techniques connues d'injection de fautes (telles que

l'analyse de mutation faible) ou les techniques d'injection d'erreurs, bien que moins coûteuses en temps d'exécution, ne nous semblent pas suffisantes pour la validation des tests (en particulier, sur des programmes critiques). En effet, elles occultent soit l'étape d'activation de la faute (dans le cas d'injection d'erreurs en cours d'exécution), soit l'étape de propagation d'une erreur jusqu'en sortie du programme (dans le cas de l'analyse de mutation faible) et nos résultats expérimentaux ont clairement montré que l'observation d'une erreur à un instant donné de l'exécution ne suffit pas pour conclure que les entrées de test vont permettre de révéler la faute correspondante par observation des sorties à la fin du cycle d'exécution.

Dans ce chapitre, nous allons examiner ces problèmes de coût, liés d'une part à l'analyse des mutants vivants et d'autre part au temps d'exécution prohibitif, et proposer des voies d'investigation qui nous paraissent prometteuses pour faciliter l'intégration de l'analyse de mutation dans des projets industriels.

VI.2 ANALYSE DES MUTANTS VIVANTS

L'analyse des mutants vivants consiste à identifier les raisons pour lesquelles certaines mutations n'ont pas été révélées par un jeu de test. Deux cas sont à considérer :

1) soit le mutant est équivalent au programme original et aucune entrée de test ne permet de le distinguer ;
2) soit le jeu de test n'est pas assez sensible pour révéler la mutation.

VI.2.1 Caractéristiques des mutants équivalents

Les résultats obtenus sur plusieurs analyses de mutation menées sur le programme ETUD [Waeselynck 1993] d'une part, et sur le programme LOCALES d'autre part, nous permettent de distinguer trois catégories principales d'équivalence :

- **équivalence fonctionnelle :** la mutation introduite ne génère pas d'erreur initiale et le comportement du programme n'est pas affecté. C'est le cas, par exemple des mutations suivantes :
  - remplacement d'un OU logique par un OU binaire lorsque les opérandes sont de type booléen ;

- remplacement d'un ET logique par un ET binaire lorsque les opérandes sont de type booléen ;

- remplacement dans une expression de la forme "x == a", de l'opérateur de comparaison "==" par un opérateur "≥" lorsque a représente la valeur maximale du domaine de définition de la variable x.

- **équivalence due à la structure du programme** : la mutation introduite dans le programme peut générer des erreurs qui sont systématiquement annulées ou masquées en cours d'exécution quel que soit le chemin exécuté, et ne peuvent donc jamais conduire à défaillance. Ce type d'équivalence peut être dû :
  - à l'existence de redondances dans le programme ; par exemple, une erreur de branchement peut conduire à exécuter une branche incorrectement sélectionnée qui redéfinit des variables aux mêmes valeurs que la branche qui est exécutée dans le programme original.
  - au flot de données qui lie des variables entre elles de telle sorte que le remplacement d'une variable par une autre ne crée pas d'erreur observable en sortie du programme. Par exemple, lorsque deux variables x et y sont liées par une instruction de la forme "x = y" (au nœud u), le remplacement de x, utilisée au nœud v, par y, v étant dépendant de u par les données, ne crée pas d'erreur au nœud v, si quel que soit le chemin exécuté la seule définition visible en v de x est le nœud u.
- **équivalence due à l'environnement d'exécution** : la mutation introduite ne peut être révélée dans l'environnement d'exécution utilisé mais elle pourrait l'être dans un autre environnement. Deux cas distincts peuvent se produire :
  - la portion de code affectée par la mutation n'est pas exécutable dans la phase de test considérée ;
  - les initialisations réalisées par défaut par le compilateur ou l'environnement de test conduisent à un comportement du mutant identique à celui du programme original. C'est le cas, par exemple, du remplacement d'un opérateur d'affectation "=" par un opérateur de comparaison "==" sur un nœud initialisant une variable à 0 (ce qui revient à supprimer l'instruction d'initialisation) si l'environnement d'exécution effectue une remise à 0 systématique avant chaque

exécution. Nous avons pu également observer ce type d'équivalence lors du remplacement d'une constante a par une constante b sur un nœud de contrôle de la forme "if (x == a)" lorsque la variable x est toujours initialisée à une valeur c ($c \neq a$ et $c \neq b$) par l'environnement de test et n'est pas redéfinie dans la fonction ni observée en sortie.

A titre indicatif, nous exposons ci-après les résultats de l'analyse des mutants vivants issus des expériences menées sur le programme LOCALES pour préciser, sur cet exemple, quelle a été la proportion de mutants équivalents par rapport au nombre de mutants vivants.

VI.2.2 Résultats expérimentaux sur le programme LOCALES

Les tableaux VI.1 et VI.2 donnent, pour chacune des fonctions *recopie-capteur* et *calcul-d-tsgv*, les résultats de l'analyse des mutants vivants. Ces résultats ont été obtenus suite à la génération exhaustive des mutants selon les tables présentées en annexe B et à l'exécution de ces mutants avec les jeux de test statistique présentés au paragraphe II.3.4. Les mutations ont été générées à l'aide de quatre tables de mutations : 1) une table de mutations sur les valeurs numériques des constantes, 2) une table de mutations sur les opérateurs du langage, 3) deux tables de mutations sur les symboles (pour les symboles représentant les constantes utilisées dans le programme et pour les symboles représentant les variables du programme). Les tableaux VI.1 et VI.2 indiquent pour chaque type de mutations généré[28], le nombre de mutants restés vivants par rapport au nombre de mutants exécutés et parmi ces mutants vivants, le nombre de mutants équivalents au programme original quel que soit l'environnement d'exécution utilisé (équivalence fonctionnelle ou due à la structure du programme), le nombre de mutants équivalents en test unitaire (équivalence due à l'environnement d'exécution) et le nombre de mutants non équivalents.

Ces résultats montrent que le nombre de mutants équivalents peut être non négligeable et difficile à évaluer a priori sans une analyse détaillée. Si l'identification des mutants équivalents ne peut être automatisable dans l'état de l'art actuel, l'acquisition de certaines informations utiles au dépouillement peut l'être, comme nous allons le voir maintenant.

---

28 Dans le cadre des expériences sur le programme LOCALES, aucun mutant sur les valeurs numériques des constantes n'est resté vivant.

Types de mutations	# mutants vivants / # mutants exécutés	# mutants équivalents	# mutants équivalents en test unitaire	# mutants non équivalents
sur les opérateurs	202 / 990	101	84	17
sur les symboles représentant des constantes	78 / 516	37	11	30
sur les symboles représentant des variables	25 / 419	10	15	0
Totaux	305 / 1925	148	110	47

Tableau VI.1 : Analyse des mutants vivants pour la fonction *recopie-capteur*

Types de mutations	# mutants vivants / # mutants exécutés	# mutants équivalents	# mutants équivalents en test unitaire	# mutants non équivalents
sur les opérateurs	122 / 681	45	69	8
sur les symboles représentant des constantes	12 / 349	0	5	7
sur les symboles représentant des variables	5 / 226	5	0	0
Totaux	139 / 1256	50	74	15

Tableau VI.2 : Analyse des mutants vivants pour la fonction *calcul-d-tsgv*

VI.2.3 Aides à l'analyse des mutants vivants

L'analyse des mutants vivants, lorsqu'ils sont nombreux, peut s'avérer très fastidieuse si on ne procède pas par étapes successives et en s'aidant d'utilitaires, ainsi que nous le proposons ci-après.

Avant d'analyser un par un les mutants laissés vivants par un jeu de test, il paraît judicieux d'identifier ceux qui sont manifestement liés à la faiblesse du jeu de test. Il est,

tout d'abord, évident de s'assurer que les mutants vivants ne sont pas localisés sur des parties de code qui n'ont jamais ou très peu été exécutées. Cela peut se faire simplement par une analyse de couverture structurelle classique (réalisée par exemple par une option du compilateur C ou par des outils spécifiques).

Ensuite des informations relatives à chaque instruction du code, telles que le nombre de mutants exécutés et la liste des mutants vivants[29], permettent d'identifier des parties de code sur lesquelles les mutations semblent plus difficiles à révéler par le jeu de test considéré, bien qu'elles aient été exécutées un nombre raisonnable de fois. Ce type d'informations facilite l'identification des instructions propices au masquage d'erreur et donc des mutants non équivalents. Il est alors conseillé de compléter les jeux de test avant de poursuivre inutilement l'analyse des mutants vivants.

A l'issue de ces premières étapes, les mutants laissés vivants par le jeu de test (éventuellement complété) doivent être analysés manuellement pour déterminer si des entrées de test auraient permis de les tuer ou non. Certaines informations dont l'acquisition peut être automatisée constituent une assistance pour cette analyse ; elles ne permettent pas de savoir si un mutant vivant est équivalent au programme original ou non mais elles aident à comprendre les raisons pour lesquelles une mutation n'a pas été révélée, c'est-à-dire à savoir si elle a pu créer une erreur initiale et si cette erreur a propagé. L'utilisation d'un outil de mise au point du programme ou de génération des traces d'exécution permet d'acquérir ce type d'informations. Dans le cadre de nos expériences, nous avons utilisé un outil de génération des traces d'exécution ; l'emploi d'un tel outil peut paraître cependant coûteux dans un contexte industriel. L'utilisation d'un outil de mise au point permet, par l'insertion de points d'observation dans le code, de savoir si l'exécution du mutant a conduit à la création d'une ou plusieurs erreurs et si ces erreurs ont propagé. Une approche qui nous paraît intéressante à étudier et à développer serait l'insertion automatique par l'outil d'analyse de mutation de points d'observation sur le nœud affecté par la mutation (dans le but de fournir les valeurs des variables définies par l'instruction mutée pour savoir si une erreur initiale a été créée) et sur des nœuds du programme préalablement sélectionnés d'après l'analyse des dépendances (en ciblant par exemple, les associations de dépendances utilisées le plus

---

29 Un utilitaire associé à l'environnement d'analyse de mutation SESAME (appelé "mcov") nous a permis d'acquérir ces informations dans le cadre de nos expériences ; il s'inspire des outils d'analyse de couverture structurelle classique et fournit, pour chaque instruction du code source, le nombre de mutants générés et la liste des mutants vivants.

souvent lors de l'exécution). Dans le cas où il y a eu création d'erreur initiale, les points d'observation suivants permettent d'indiquer où la propagation d'erreurs s'est arrêtée.

Les informations apportées par l'analyse de dépendances du programme (analyse du graphe définition-utilisation ou analyse des chaînes de dépendances) nous ont permis, dans certains cas, d'expliquer pourquoi un mutant était équivalent ou difficile à tuer. Il existe de nombreux travaux récents ayant pour objet l'acquisition automatique d'informations relatives aux dépendances du programme (par exemple, [Harrold et al. 1993, Harrold & Rothermel 1996]) mais il n'y a pas encore, à notre connaissance, d'outil disponible permettant d'automatiser de telles analyses.

L'approche pragmatique présentée ici est issue des analyses de mutation menées dans le cadre de nos expériences sur le programme LOCALES et il s'est avéré que les utilitaires mentionnés ont facilité de manière appréciable l'analyse des mutants vivants. Le nombre de mutants équivalents peut également être diminué par l'approche introduite au paragraphe suivant qui est la mutation sélective.

VI.3 MUTATION SÉLECTIVE

Pour diminuer le coût d'exécution de l'analyse de mutation forte, on peut chercher à diminuer le nombre de mutants exécutés. Cette approche, dénommée mutation sélective, a été développée et expérimentée par Mathur et Offutt [Mathur & Wong 1993, Offutt et al. 1996] en utilisant l'outil Mothra construit pour des programmes en Fortran. Leurs études ont porté essentiellement sur la sélection des opérateurs de mutation à utiliser pour générer des mutants (cf. description des opérateurs de mutation de Mothra au § I.4.1). En particulier dans [Offutt et al. 1996], les auteurs préconisent l'utilisation de seulement 5 des 22 opérateurs de mutation disponibles.

La mutation sélective doit, à notre avis, cibler en priorité des mutations qui ne sont révélées que par seulement un petit sous-domaine des entrées soit parce que :

a) leur activation crée rarement une erreur initiale ;

b) les erreurs créées font l'objet de nombreux masquages ou annulations ; elles propagent rarement ou leur propagation affecte rarement (c'est-à-dire seulement sous certaines conditions) des variables de sortie.

Ces mutations correspondent à des fautes dites subtiles, c'est-à-dire des fautes qui ne peuvent être révélées que par certaines entrées de test particulières ou encore par

certaines séquences d'entrées de test (cf. fautes J, K et L identifiées sur le programme ETUD et décrites au § III.4.2). On peut citer quelques exemples-type de mutations correspondant au cas a) présenté ci-dessus :

- remplacement d'un opérateur de comparaison "==" par un opérateur ">" dans une expression de la forme "x == borne supérieure" : une erreur initiale n'est créée que pour des entrées de test correspondant à un point singulier ;
- remplacement d'un opérateur OU logique, " || ", par un opérateur OU exclusif bit à bit , " ^ ", entre deux expressions booléennes. Soient, A et B, deux opérandes dont les valeurs sont booléennes, l'expression "A ^ B" produit un résultat différent de "A || B" (et crée donc une erreur initiale) seulement dans le cas où A = B = 1 ;
- remplacement dans une expression conditionnelle de la forme " x < a ", d'une constante a par une constante b dont la valeur est proche de a : seules des entrées de test particulières permettent de distinguer l'écart entre a et b et donc de créer une erreur initiale.

Ces types de mutations peuvent être identifiés par analyse du code source. Alors que les mutations dont le comportement erroné correspond au cas b) ne peuvent être identifiées a priori ; une analyse des chaînes de dépendances du programme peut s'avérer nécessaire pour déterminer ce type de mutations. En effet, d'après nos observations expérimentales, ces masquages ou annulations d'erreur peuvent être dus notamment à des utilisations rares (ou dans des conditions particulières) des variables erronées ou à des redéfinitions fréquentes des variables erronées.

La sélection des mutations peut également chercher à diminuer le coût de l'analyse des mutants vivants, en évitant si possible de générer des mutants équivalents au programme original. Par exemple, on peut retirer de la table de mutations sur les opérateurs, le type de mutation qui consiste à remplacer un OU logique par un OU binaire si, pour un programme donné, toutes les opérandes utilisées avec cet opérateur, sont de type booléen.

Divers critères de sélection des mutations peuvent être intéressants à étudier : la mutation sélective peut cibler certains types de mutation (par le biais des tables de mutations ou des opérateurs de mutation selon l'environnement utilisé) mais également

certaines instructions du code[30]. Cette dernière voie nous paraît prometteuse étant donné nos observations sur les relations de dépendances : il s'agirait, par exemple, de sélectionner des mutations sur des instructions selon les dépendances qui les lient aux autres instructions du programme. La sélection peut également porter sur des instructions dont la syntaxe est propice à la non création d'erreur initiale ou aux masquages d'erreurs ; il s'agit, par exemple, d'instructions comprenant plusieurs expressions relationnelles liées par des OU logique ou comprenant des opérateurs de décalage binaire.

La mutation sélective est à notre avis une approche prometteuse ; elle nécessite cependant des investigations supplémentaires afin de caractériser plus précisément et pour des applications diverses les mutations intéressantes à injecter.

VI.4 APPLICATION DE L'ANALYSE DE MUTATION

L'intérêt d'utiliser l'analyse de mutation forte, pour des applications industrielles, nous paraît à présent certain. Cette technique devrait être utilisée en complément des analyses de couverture structurelle, dans le cadre de logiciels critiques, afin d'évaluer l'efficacité des jeux de test par rapport à la détection d'erreurs et ainsi d'accroître la confiance que l'on peut avoir dans ces jeux.

La seule mesure permettant d'évaluer l'efficacité des jeux de test vis-à-vis des fautes injectées est le score de mutation. Or, dans un contexte industriel, le développement d'un logiciel nécessite la production de plusieurs versions successives de ce logiciel (créées suite à des corrections ou à des évolutions de spécification), ce qui implique également la conception de plusieurs jeux de test permettant de vérifier chacune de ces versions. En raison du coût actuel du dépouillement des résultats, le score de mutation ne peut être établi que pour certaines versions de ce logiciel. Dans ce contexte, il n'est pas utile d'établir un score de mutation sur une version préliminaire dont on sait qu'elle ne prend pas en compte toutes les corrections ou évolutions envisagées à un moment donné du développement. Par contre, il nous paraît souhaitable de réaliser une analyse de mutation complète et donc d'analyser les mutants vivants, sur une version considérée comme finale et devant faire l'objet d'une certification ou d'une qualification.

---

30 L'environnement SESAME permet facilement d'expérimenter ce type de sélection.

Si l'analyse de mutation sur tout le programme s'avère trop coûteuse, on peut se contenter de cibler les parties de code les plus critiques.

Sur les versions préliminaires successives pour lesquelles l'analyse des mutants vivants n'est pas envisageable pour des raisons de coût, il nous semble cependant intéressant d'évaluer les pourcentages des mutants vivants (équivalents ou non) par rapport aux mutants exécutés et de les comparer. L'évolution de ces pourcentages peut servir d'indicateur de dégradation éventuelle de l'efficacité des jeux de tests d'une version à une autre. Si le pourcentage des mutants vivants par rapport aux mutants exécutés baisse de manière significative d'une version à une autre, il est nécessaire de s'interroger sur la pertinence des jeux de test pour la nouvelle version.

VI.5 CONCLUSIONS

Les résultats des études conduites et présentées dans ce manuscrit nous autorisent à penser que l'analyse de mutation est une méthode digne d'intérêt et pertinente pour évaluer l'efficacité de jeux de test générés pour une application donnée, vis-à-vis de la recherche de fautes. Nous avons montré, en particulier, que les comportements erronés générés par les mutants ne sont pas toujours triviaux mais peuvent être représentatifs de fautes subtiles, c'est-à-dire difficiles à révéler par des jeux de test et pouvant dès lors subsister dans le programme en cours de validation.

Le coût de l'analyse de mutation peut cependant faire l'objet de réticences, quant à son utilisation pour des projets industriels. Nous avons exposé dans ce chapitre les raisons de ce coût (coût d'exécution et coût d'analyse des mutants vivants) et envisagé plusieurs méthodes permettant de le réduire. Deux axes d'amélioration de cette technique nous semblent particulièrement intéressants à étudier :

1) l'élaboration et l'utilisation d'outils d'aide au dépouillement des résultats et en particulier d'aide à l'analyse des mutants vivants ;
2) la sélection des mutations à injecter (mutation sélective) en ciblant certains types de mutations ou certaines parties du code de manière à diminuer le coût d'exécution (c'est-à-dire le nombre de mutants à exécuter) et le nombre de mutants équivalents. La mutation sélective pourrait également permettre de cibler certaines mutations en fonction des comportements erronés qu'elles sont susceptibles de générer.

Les analyses de mutation conduites sur le programme LOCALES ont permis d'identifier certaines caractéristiques de mutants subtils ou équivalents. Mais les travaux de recherche, illustrés par des expérimentations sur d'autres cas réels, doivent encore se poursuivre pour affiner les critères permettant de sélectionner un sous-ensemble pertinent de mutations à injecter.

CONCLUSION GÉNÉRALE

Les travaux présentés dans ce mémoire ont été motivés par le besoin d'élaborer une méthode expérimentale d'évaluation de l'efficacité des jeux de test générés pour un programme donné, vis-à-vis de la détection des comportements erronés du logiciel. Il est important de rappeler que nos résultats concernent des logiciels séquentiels développés à l'aide de langages procéduraux (tels que Pascal, C, Fortran).

Dans le premier chapitre, nous avons souligné l'imperfection des critères de couverture (structurelle ou fonctionnelle) utilisés pour guider la sélection a priori ou la validation a posteriori des entrées d'un jeu de test. En effet, ces critères ne sont pas directement liés aux fautes qu'ils sont censés révéler. Le problème qui se pose alors, est celui de la confiance que l'on peut accorder aux entrées de test utilisées pour un programme donné vis-à-vis de la recherche de fautes. L'**analyse de mutation**, en tant que technique d'injection de fautes logicielles, nous semble être une approche expérimentale appropriée à la validation des tests du logiciel. Cependant, en l'absence d'un modèle de fautes représentatif de la réalité, il paraît difficile de définir les types de fautes à injecter. Nous nous sommes donc intéressée aux erreurs et aux comportements erronés générés au cours de l'exécution du logiciel, et non pas aux fautes résultant du développement.

Nous avons mené des études expérimentales, inédites à notre connaissance, sur l'analyse détaillée des erreurs et des comportements erronés induits à la fois par des fautes réelles et par des fautes de type mutation. Ces études ont porté sur deux logiciels distincts, appelés ETUD et LOCALES, issus d'applications critiques du nucléaire développées dans des environnements industriels différents. Nous avons pu disposer dans les deux cas, de fautes identifiées au cours du développement du logiciel (au nombre de 12 pour le programme ETUD et de 13 pour le programme LOCALES) et de mutations générées à l'aide de l'environnement SESAME (développé au LAAS-CNRS).

La première série d'expérimentations, sur le programme ETUD, a eu pour objectif d'analyser de manière détaillée des erreurs produites par 12 fautes réelles d'une part, et par 24 mutations de premier ordre d'autre part. ETUD est un programme avec mémoire, il se distingue ainsi d'autres programmes étudiés dans des travaux relatifs à la modélisation d'erreurs. Les analyses effectuées nous ont permis de définir, d'un point de vue théorique, les types d'erreurs observés ainsi que les évolutions de l'état interne

d'un programme incorrect sur plusieurs cycles d'exécution. Ces observations ont mis en évidence des **mécanismes de création et de propagation d'erreur** complexes. L'analyse détaillée des **traces d'erreurs** produites lors de l'exécution de chacun des 36 programmes incorrects nous a permis de constituer deux bases de données d'erreurs conséquentes : l'une relative aux erreurs produites par les fautes réelles et comprenant 1450 enregistrements d'erreurs et l'autre relative aux erreurs produites par les mutations et comprenant 2272 enregistrements d'erreurs. Ces bases de données ont été construites de manière à pouvoir comparer la nature des erreurs enregistrées ainsi que les flots d'erreurs (représentant la propagation d'une erreur initiale c'est-à-dire créée suite à l'activation d'une faute). Ces analyses comparatives entre les deux bases de données ont produit des résultats quantitatifs et qualitatifs favorables à une bonne représentativité des erreurs produites par des mutations par rapport aux erreurs produites par des fautes réelles puisque, notamment : 85% des 2272 erreurs produites par les mutations analysées ont également été produites par les fautes réelles ; 63 des 86 flots d'erreurs générés par les fautes réelles ont été reproduits complètement ou partiellement par les mutations.

Sur le plan théorique, une **modélisation des comportements erronés** a été déduite de ces premières observations. A partir d'une représentation adéquate du programme (graphe définition-utilisation), nous avons mis en évidence les **relations de dépendances** entre les instructions et les données du programme ; en effet, ces relations de dépendances permettent d'expliquer les mécanismes complexes de création, de masquage et d'annulation d'erreur. Nous avons proposé une méthode d'analyse des dépendances, basée sur la détermination d'**associations** et de **chaînes de dépendances** ; cette méthode permet d'expliquer les comportements erronés produit par une faute.

La deuxième série d'expérimentations, sur le programme LOCALES, a eu pour objectif d'appliquer cette méthode d'analyse des dépendances pour comprendre et expliquer les similitudes entre les comportements erronés produits par des fautes réelles et ceux produits par des mutations. Nous avons, dans un premier temps, analysé les modifications induites par chacune des 13 fautes réelles sur les chaînes de dépendances du programme. Cela nous a conduite notamment, à identifier trois de ces fautes comme étant impossibles à révéler par des jeux de test en montrant que les programmes incorrects correspondants sont fonctionnellement équivalents à la version correcte. Dans un deuxième temps, nous avons sélectionné a priori des mutations susceptibles de reproduire partiellement ou totalement les mêmes comportements erronés que les fautes réelles en se basant sur les relations de dépendances identifiées dans le programme.

L'analyse expérimentale des comportements erronés générés par ces mutations a permis de vérifier leur similitude par rapport à ceux dus aux fautes réelles.

Concernant la représentativité des comportements erronés dus à des mutations, les principales conclusions issues de l'ensemble de nos observations expérimentales sont les suivantes :

- les mutations de premier ordre peuvent produire des erreurs et des comportements erronés similaires, voire identiques, à ceux produits par des fautes réelles de nature syntaxique plus complexe (c'est-à-dire dont la correction implique plusieurs instructions du code source éventuellement réparties sur plusieurs blocs) ;
- les mutations peuvent être aussi difficiles à révéler par des jeux de test que des fautes réelles ;
- les similitudes observées entre les comportements erronés produits par des fautes réelles et par des mutations s'expliquent à l'aide des relations de dépendances ;
- les comportements erronés générés par les mutations sont suffisamment variés pour être représentatifs d'un bon nombre de comportements erronés induits par des fautes réelles.

Dans la littérature, les critiques concernant la représentativité des mutations vis-à-vis de fautes réelles reposent essentiellement sur une caractérisation des fautes en fonction de leur taille syntaxique. Nos travaux montrent clairement que cette mesure n'est pas adéquate pour évaluer la complexité ou la subtilité des comportements erronés qui peuvent être générés par une faute ; elle ne permet donc pas de prévoir si les erreurs dues à une faute seront facilement détectables ou non par un jeu de test. Il nous paraît dès lors plus judicieux de caractériser la nature des comportements erronés que la nature des fautes elles-mêmes.

Nos résultats permettent de réhabiliter l'analyse de mutation en tant que technique de validation des tests du logiciel. A notre avis, l'utilisation de cette technique est justifiée en complément des analyses de couverture structurelle menées sur une application critique ; elle permet d'accroître la confiance que l'on peut avoir en des jeux de test vis-à-vis de leur capacité à détecter des erreurs. Nous avons proposé au chapitre VI des applications qui nous semblent réalistes dans un contexte industriel étant donné les coûts inhérents à l'analyse de mutation.

D'un point de vue pratique, le fait d'avoir expérimenté l'analyse de mutation sur des études de cas nous a permis :

- d'une part de faire évoluer l'environnement de mutation SESAME utilisé afin de faciliter son industrialisation, et de proposer l'intégration d'utilitaires d'aide au dépouillement des résultats ;
- d'autre part d'identifier certaines caractéristiques des mutations non révélées ou difficiles à révéler qui devraient permettre de dégager des critères de sélection d'un sous-ensemble de mutations, et ainsi de diminuer le coût de l'analyse de mutation.

Ces travaux ouvrent des axes d'investigation supplémentaires et prometteurs, permettant soit de faciliter l'utilisation de l'analyse de mutation en réduisant en particulier son coût, soit de généraliser les résultats obtenus à d'autres types d'application. Parmi ces axes, nous pouvons citer en particulier :

- le développement d'utilitaires permettant d'automatiser l'analyse des dépendances d'un programme et donc de faciliter l'identification de mutants équivalents au programme original ou difficiles à révéler ;
- l'étude des relations de dépendances sur des applications plus complexes pour affiner la sélection des mutations intéressantes à injecter (mutation sélective) ;
- la caractérisation des fautes en fonction des comportements erronés qu'elles peuvent générer à l'aide d'expérimentations sur d'autres applications industrielles, dans le but de constituer des échantillons représentatifs de comportements erronés et d'approfondir la technique de mutation sélective ;
- l'étude expérimentale des comportements erronés sur d'autres types de logiciels, développés à l'aide d'autres approches (telles que l'approche orientée-objet) ou correspondant à d'autres types d'applications (applications distribuées ou concurrentes) et ceci dans le but d'évaluer la faisabilité d'une approche telle que l'analyse de mutation dans ces domaines.

ANNEXE A

ANALYSE DÉTAILLÉE DES FONCTIONS DU PROGRAMME LOCALES

A.1 INTRODUCTION

Cette annexe regroupe pour chacune des fonctions du programme LOCALES, *recopie-capteur* et *calcul-d-tsgv* :

1) le code source associé,
2) son graphe définition-utilisation (en langage C),
3) la liste des chemins exécutables,
4) la liste des associations de dépendances.

Ces différents éléments permettent d'étudier les comportements erronés générés par les fautes étudiées sur ce programme.

A.2 FONCTION RECOPIE-CAPTEUR

A.2.1 Programme source

La figure A.1 comprend le code source original de la fonction *recopie-capteur* ; pour des besoins de confidentialité, les commentaires ont été retirés.

```
void recopie_capteur ()
  {

/* Declaration des variables locales   */
    entier          N_ACT;
    entier          DIF_T;
    entier          DIF_G;

/* Initialisation de test de signature */
    N_ACT =  K_REC ;

    if (PP ==  K_PP_CD )
      {
        PP0 =  K_PP_D ;
        TN[ 52 ] =  K_PP_ER ;
      }
    else
      {
/* Elaboration de PP0 */
       PP0 = PP;
       if ((PP0 <  K_PP_m ) || (PP0 >  K_PP_M ))
           {
             PP0 =  K_PP_D ;
             TN[ 52 ] =  K_PP_ER ;
           }
       }
/* Elaboration de T0 */
    if ((TE ==  K_TE_CD ) && (TL ==  K_TL_CD ))
      {
        D1 =  ok ;
        T0 =  K_T_D ;
        TN[ 53 ] =  K_T_ER ;
      }
    else
      if (TE ==  K_TE_CD)
        {
          D1 =  ko ;
        /* Inhibition des defauts des capteurs gamme etroite */
        TB =  ok ;
        TC =  ok ;
        TN[ 26 ] = K_F_ER;
        TN[ 32 ] = K_F_ER;
        T0 = TL;
        }
      else
        if (TL ==  K_TL_CD)
          {
            D1 =  ok ;
            T0 = TE;
          }
        else
```

```
          {
            if (D1 ==  ko)
              {
                if ((TE >= K_T_MIN) && (TE <= K_T_MAX))
                  {
                    T0 = TE;
                    D1 =  ok ;
                  }
                else
                  {
                    T0 = TL;
                  }
              }
            else
              {
                T0 = TE;
              }
          }
    if ((T0 <  K_T_m ) || (T0 >  K_T_M ))
      {
        T0 =  K_T_D ;
        TN[ 53 ] =  K_T_ER ;
      }

/* Elaboration  de la discordance sur les entrees */

/* gestion entre bornes MIN et MAX */
    if (((TE >= K_T_MIN) && (TE <= K_T_MAX)) ||
        ((TL >= K_T_MIN) && (TL <= K_T_MAX)))
      {
        if ((TE ==  K_TE_CD ) || (TL ==  K_TL_CD ))
          {
            T_DIS =  declenche ;
          }
        else
          {
            DIF_T = TE - TL;
            if ((DIF_T < -PK) || (DIF_T > PK))
              {
                T_DIS =  declenche ;
              }
          }
      }

/* Elaboration de TG0 */
    if ((TGE ==  K_TGE_CD ) && (TGL ==  K_TGL_CD ))
      {
        DG1 =  ok ;
        TG0 =  K_D ;
        TN[ 54 ] =  K_ER ;
      }
    else
      if (TGE ==  K_TGE_CD)
        {
```

```
          DG1 =  ko ;
/* Inhibition des defauts des capteurs gamme etroite */
          TGB =  ok ;
          TGC =  ok ;
          TN[ 27 ] = K_F_ER;
          TN[ 33 ] = K_F_ER;
          TG0 = TGL;
        }
      else
        if (TGL ==  K_TGL_CD)
          {
            DG1 =  ok ;
            TG0 = TGE;
          }
        else
          {
            if (DG1 == ko)
              {
                if ((TGE >= K_MIN) && (TGE <= K_MAX))
                  {
                    TG0 = TGE;
                    DG1 =  ok ;
                  }
                else
                  {
                    TG0 = TGL;
                  }
              }
            else
              {
                TG0 = TGE;
              }
          }
    if ((TG0 <  K_m ) || (TG0 >  K_M ))
      {
        TG0 =  K_D ;
        TN[ 54 ] =  K_ER ;
      }

/* Elaboration  de la discordance sur les entrees */

/* gestion entre bornes MIN et MAX */
    if (((TGE >= K_MIN) && (TGE <= K_MAX)) ||
       ((TGL >= K_MIN) && (TGL <= K_MAX)))
      {
        if (
            (TGE ==  K_TGE_CD ) || (TGL ==  K_TGL_CD )
           )
          {
            G_DIS =  declenche ;
          }
        else
          {
            DIF_G = TGE - TGL;
```

```
            if (
               (DIF_G < -PK) || (DIF_G > PK)
               )
              {
                G_DIS =  declenche ;
              }
          }
      }

/* Bloc inhibition des capteurs retire */

/* Elaboration de ET0 */
    ET0 = ENF1 | ETM1 | ENF2 | ETM2;

/* Elaboration de ED0 */
    ED0 = EI01 | EI02 ;

/* Verification de la signature */
    if (N_ACT !=  K_REC )
      {
        D_SEQUENC =  err_appel ;
        trap1( k_n_err_seq );
      }
    else
      {
        BIT_SIGN. B_ACTION_RECOPIE_CAPTEUR  =  true ;
      }
    return;
  }
```

Figure A.1 : Programme source de la fonction *recopie-capteur*

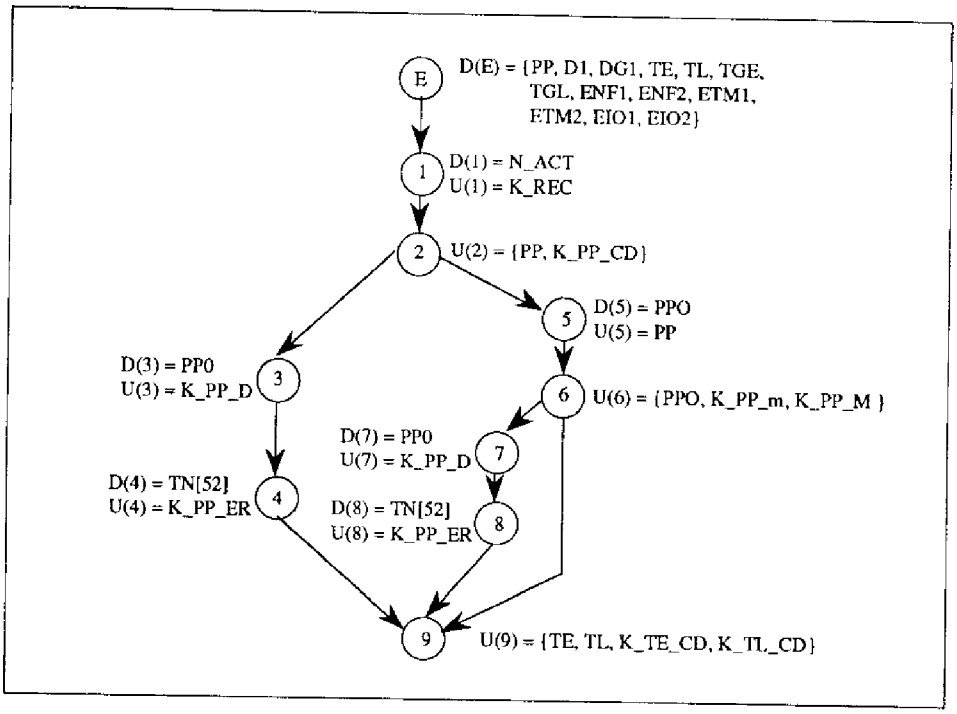
A.2.2 Graphe définition-utilisation

Le graphe définition-utilisation de la fonction *recopie-capteur* comporte 74 nœuds ; les nœuds E et S représentent respectivement le nœud d'entrée et le nœud de sortie du programme. Pour chaque nœud n sont mentionnés l'ensemble D(n) des variables définies au nœud n et l'ensemble U(n) des variables utilisées au nœud n (cf. § IV.2.2).

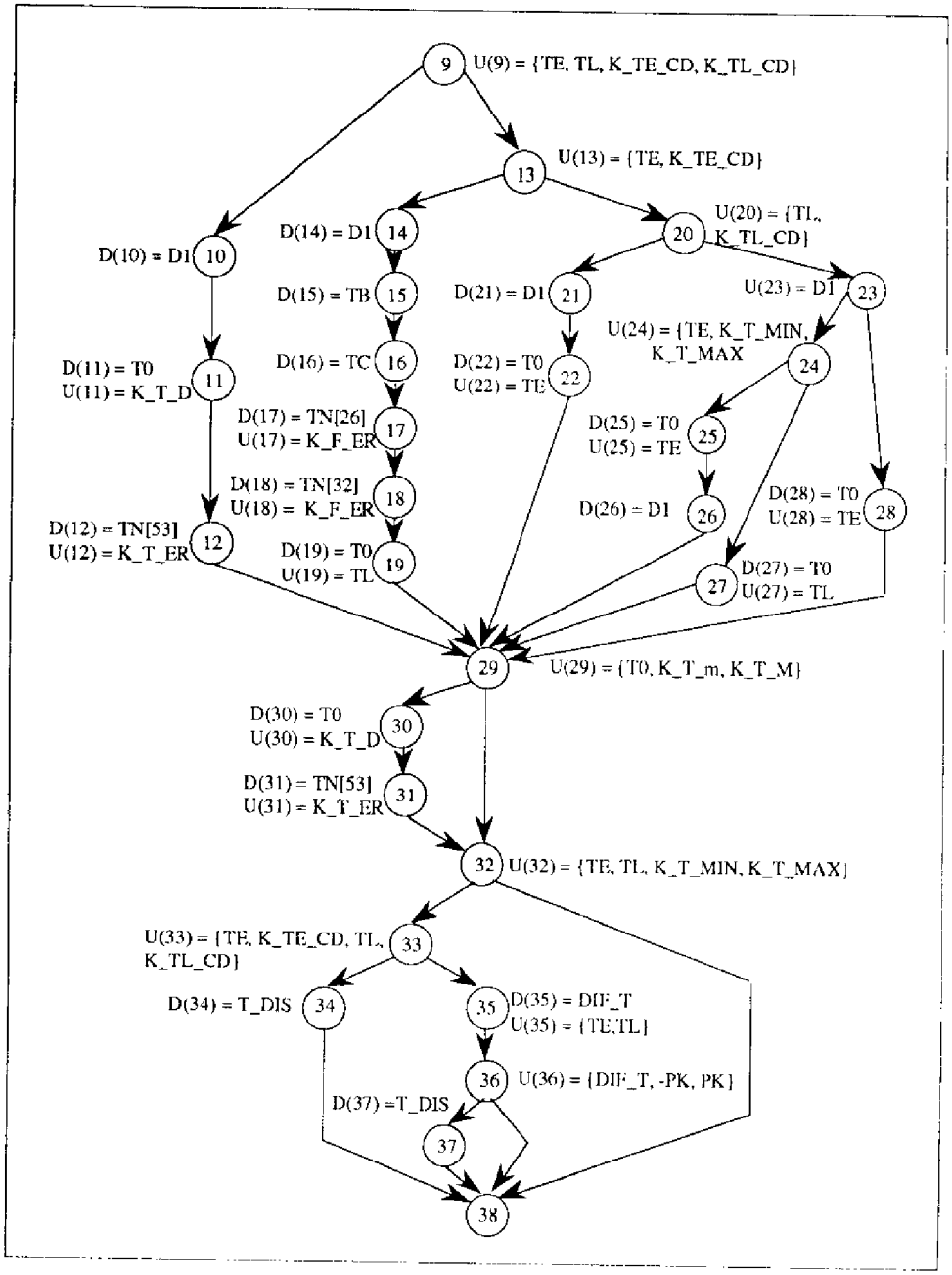
Les variables de la forme K_xxx sont, en fait, des constantes : leurs valeurs sont définies dans un fichier inclus en en-tête du programme source.

Nous avons fait figurer le graphe définition-utilisation de *recopie-capteur* sur quatre pages : la première partie du graphe comprend les nœuds E à 9, la seconde partie

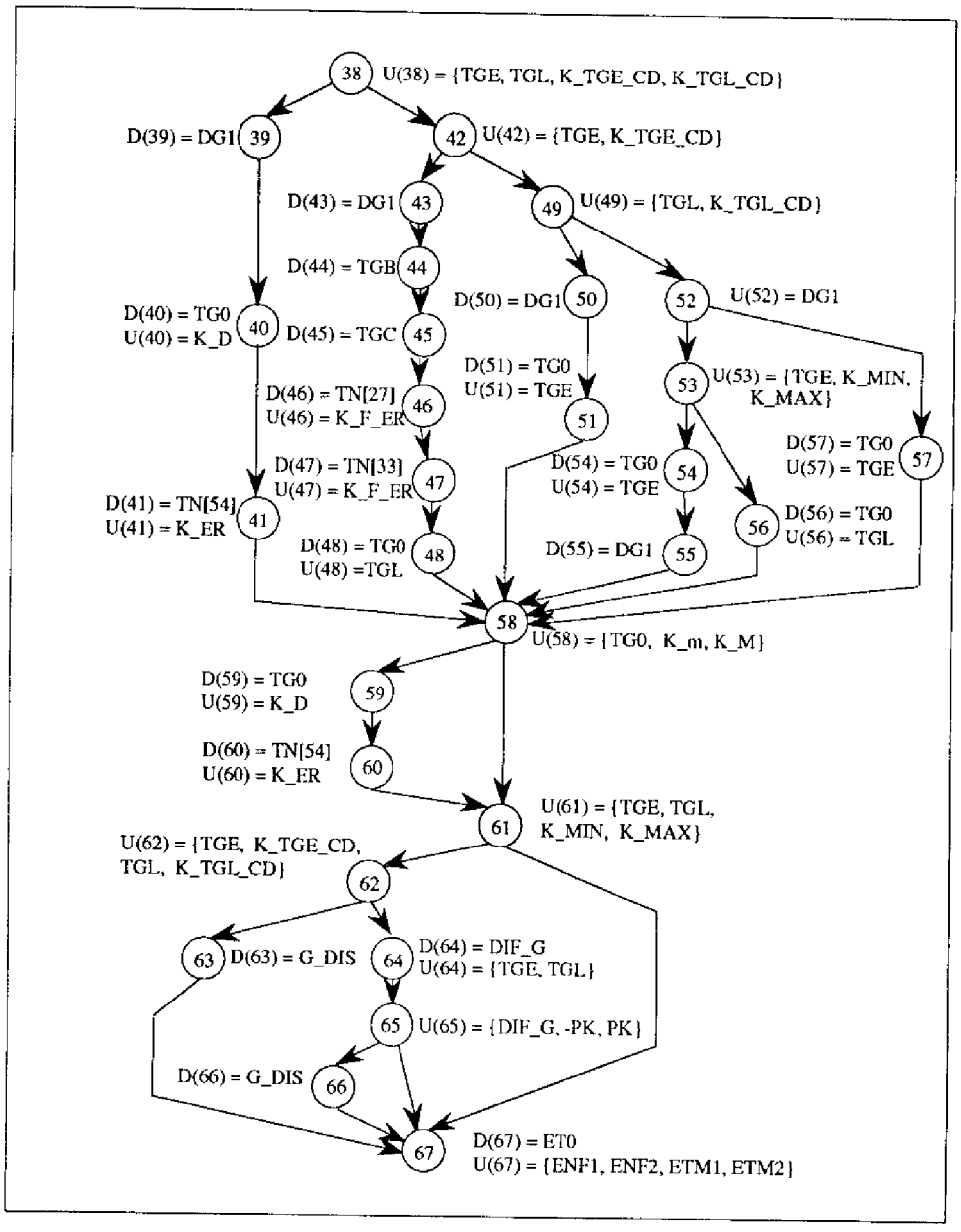
les nœuds 9 à 38, la troisième partie les nœuds 38 à 67, et la dernière partie les nœuds 67 à S.



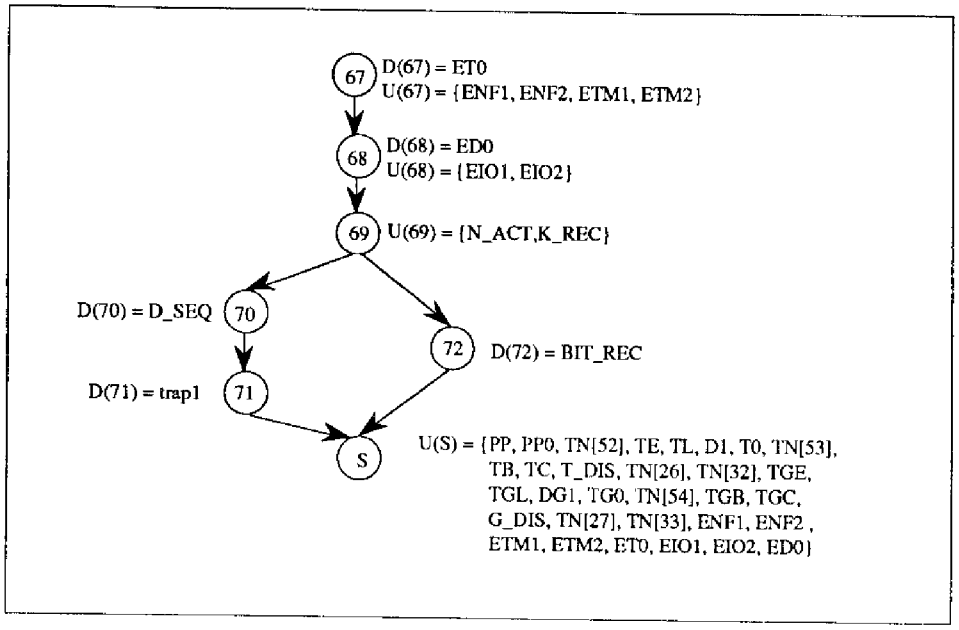
Partie 1 du graphe définition-utilisation de *recopie-capteur*



Partie 2 du graphe définition-utilisation de *recopie-capteur*



Partie 3 du graphe définition-utilisation de *recopie-capteur*



Partie 4 du graphe définition-utilisation de *recopie-capteur*

A.2.3 Chemins exécutables

Lors de l'analyse de la fonction *recopie-capteur*, 972 chemins complets exécutables ont été identifiés. Il serait donc fastidieux de tous les énumérer. Cependant, on constate que *recopie-capteur* peut se décomposer en 3 sous-ensembles fonctionnellement indépendants. Nous avons donc identifié et étudié les chemins exécutables pour chacun de ces sous-ensembles :

- sous-ensemble 1 du nœud 1 au nœud 9 : 3 chemins ont été identifiés (notés C_11 à C_13) ;
- sous-ensemble 2 du nœud 9 au nœud 38 : 18 chemins ont été identifiés (notés C_21 à C_218) ;
- sous-ensemble 3 du nœud 38 au nœud 73 : 18 chemins ont été identifiés (notés C_31 à C_318).

Les chemins exécutables sont listés ci-après sous forme d'une liste de nœuds du graphe définition-utilisation correspondant :

C_11 = 1 - 2- 3 - 4 - 9
C_12 = 1 - 2 - 5 - 6 - 7 - 8 - 9
C_13 = 1 - 2 - 5 - 6 - 9
C_21 = 9 - 10 - 11 - 12 - 29 - 30 - 31 - 32 - 38
C_22 = 9 - 13 - 14 - 15 - 16 - 17 - 18 - 19 - 29 - 30 - 31 - 32 - 38
C_23 = 9 - 13 - 14 - 15 - 16 - 17 - 18 - 19 - 29 - 32 - 33 - 34 - 38
C_24 = 9 - 13 - 14 - 15 - 16 - 17 - 18 - 19 - 29 - 32 - 38
C_25 = 9 - 13 - 20 - 21 - 22 - 29 - 30 - 31 - 32 - 38
C_26 = 9 - 13 - 20 - 21 - 22 - 29 - 32 - 33 - 34 - 38
C_27 = 9 - 13 - 20 - 21 - 22 - 29 - 32 - 38
C_28 = 9 - 13 - 20 - 23 - 24 - 25 - 26 - 29 - 32 - 33 - 35 - 36 - 37 - 38
C_29 = 9 - 13 - 20 - 23 - 24 - 25 - 26 - 29 - 32 - 33 - 35 - 36 - 38
C_210 = 9 - 13 - 20 - 23 - 24 - 27 - 29 - 30 - 31 - 32 - 38
C_211 = 9 - 13 - 20 - 23 - 24 - 27 - 29 - 32 - 33 - 35 - 36 - 37 - 38
C_212 = 9 - 13 - 20 - 23 - 24 - 27 - 29 - 32 - 33 - 35 - 36 - 38
C_213 = 9 - 13 - 20 - 23 - 24 - 27 - 29 - 32 - 38
C_214 = 9 - 13 - 20 - 23 - 28 - 29 - 30 - 31 - 32- 33 - 35 - 36 - 37 - 38
C_215 = 9 - 13 - 20 - 23 - 28 - 29 - 30 - 31 - 32 - 38
C_216 = 9 - 13 - 20 - 23 - 28 - 29 - 32 - 33 - 35 - 36 - 37 - 38
C_217 = 9 - 13 - 20 - 23 - 28 - 29 - 32 - 33 - 35 - 36 - 38
C_218 = 9 - 13 - 20 - 23 - 28 - 29 - 32 - 38
C_31 = 38 - 39 - 40 - 41 - 58 - 59 - 60 - 61- 67 - 68 - 69 - 72- 73
C_32 = 38 - 42 - 43- 44 - 45- 46 - 47 - 48 - 58 - 59 - 60 - 61 - 67 - 68 - 69 - 72- 73
C_33 = 38 - 42 - 43- 44 - 45- 46 - 47 - 48 - 58 - 61 - 62 - 63 - 67 - 68 - 69 - 72- 73
C_34 = 38 - 42 - 43- 44 - 45- 46 - 47 - 48 - 58 - 61 - 67 - 68 - 69 - 72- 73
C_35 = 38 - 42 - 49 - 50 - 51- 58 - 59 - 60 - 61 - 67 - 68 - 69 - 72- 73
C_36 = 38 - 42 - 49 - 50 - 51- 58 - 61 - 62 - 63 - 67 - 68 - 69 - 72- 73
C_37 = 38 - 42 - 49 - 50 - 51- 58 - 61 - 67 - 68 - 69 - 72- 73
C_38 = 38 - 42 - 49 - 52 - 53 - 54 - 55 - 58 - 61 - 62 - 64 - 65 - 66 - 67 - 68 - 69 - 72- 73
C_39 = 38 - 42 - 49 - 52 - 53 - 54 - 55 - 58 - 61 - 62 - 64 - 65 - 67 - 68 - 69 - 72- 73
C_310 = 38 - 42 - 49 - 52 - 53 - 56 - 58 - 67 - 68 - 69 - 72- 73
C_311 = 38 - 42 - 49 - 52 - 53 - 56 - 58 - 61 - 62 - 64 - 65 - 66 - 67 - 68 - 69 - 72- 73
C_312 = 38 - 42 - 49 - 52 - 53 - 56 - 58 - 61 - 62 - 64 - 65 - 67 - 68 - 69 - 72- 73
C_313 = 38 - 42 - 49 - 52 - 53 - 56 - 58 - 61 - 67 - 68 - 69 - 72- 73
C_314 = 38 - 42 - 49 - 52 - 57 - 58 - 59 - 60 - 61 - 62 - 64 - 65 - 67 - 68 - 69 - 72- 73
C_315 = 38 - 42 - 49 - 52 - 57 - 58 - 59 - 60 - 61 -67 - 68 - 69 - 72- 73
C_316 = 38 - 42 - 49 - 52 - 57 - 58 - 61 - 62 - 64 - 65 - 66 - 67 - 68 - 69 - 72- 73
C_317 = 38 - 42 - 49 - 52 - 57 - 58 - 61 - 62 - 64 - 65 - 67 - 68 - 69 - 72- 73
C_318 = 38 - 42 - 49 - 52 - 57 - 58 - 61 - 67 - 68 - 69 - 72- 73

A.2.4 Associations de dépendances

Une association de dépendances (u, v, x) signifie soit que le nœud v est dépendant du nœud u par les données si x est une variable définie en u et utilisée en v, soit que le nœud v est dépendant du nœud u par le contrôle si x est une expression conditionnelle évaluée en u et dont l'évaluation peut conduire à l'exécution de v (cf. § IV.3.3). Les nœuds E et S représentent respectivement le nœud d'entrée et le nœud de sortie de la fonction. Un certain nombre de variables sont des paramètres d'entrée et sont définies au nœud E. D'autres représentent des variables de sortie et sont donc utilisées par le nœud S.

(E, 2, PP)	(E, 5, PP)	(E, 9, TE)	(E, 9, TL)	(E, S, TN[52])
(1, 69, N_ACT)	(2, 3, e_2)	(2, 4, e_2)	(2, 5, e_2)	(2, 6, e_2)
(3, S, PP0)	(4, S, TN[52])	(5, S, PP0)	(5, 6, PP0)	(6, 7, e_6)
(6, 8, e_6)	(7, S, PP0)	(8, S, TN[52])		
(E, 9, TE)	(E, 13, TE)	(E, 22, TE)	(E, 24, TE)	(E, 25, TE)
(E, 28, TE)	(E, 32, TE)	(E, 33, TE)	(E, 35, TE)	(E, 9, TL)
(E, 19, TL)	(E, 20, TL)	(E, 27, TL)	(E, 32, TL)	(E, 33, TL)
(E, 35, TL)	(E, 36, PK)	(E, 65, PK)	(E, 23, D1)	(E, S, TN[53])
(E, S, TN[26])	(E, S, TN[32])	(9, 10, e_9)	(9, 11, e_9)	(9, 12, e_9)
(9, 13, e_9)	(10, S, D1)	(11, 29, T0)	(13, 14, e_{13})	(13, 15, e_{13})
(13, 16, e_{13})	(13, 17, e_{13})	(13, 18, e_{13})	(13, 19, e_{13})	(13, 20, e_{13})
(14, S, D1)	(15, S, TB)	(16, S, TC)	(17, S, TN[26])	(18, S, TN[32])
(19, 29, T0)	(19, S, T0)	(20, 21, e_{20})	(20, 22, e_{20})	(20, 23, e_{20})
(21, S, D1)	(22, 29, T0)	(22, S, T0)	(23, 24, e_{23})	(23, 28, e_{23})
(24, 25, e_{24})	(24, 26, e_{24})	(24, 27, e_{24})	(25, 29, T0)	(25, S, T0)
(27, 29, T0)	(27, S, T0)	(26, S, D1)	(28, 29, T0)	(28, S, T0)
(29, 30, e_{29})	(29, 31, e_{29})	(30, S, T0)	(31, S, TN[53])	(32, 33, e_{32})
(33, 34, e_{33})	(33, 35 e_{33})	(33, 36, e_{33})	(36, 37, e_{36})	(34, S, T_DIS)
(37, S, T_DIS)	(35, 36, DIF_T)	(35, S, DIF_T)		
(E, 38, TGE)	(E, 42, TGE)	(E, 51, TGE)	(E, 53, TGE)	(E, 54, TGE)
(E, 55, TGE)	(E, 57, TGE)	(E, 61, TGE)	(E, 62, TGE)	(E, 64, TGE)
(E, 38, TGL)	(E, 48, TGL)	(E, 49, TGL)	(E, 56, TGL)	(E, 61, TGL)
(E, 62, TGL)	(E, 64, TGL)	(E, 52, DG1)	(E, S, TN[54])	(E, S, TN[27])
(E, S, TN[33])	(38, 39, e_{38})	(38, 40, e_{38})	(38, 41, e_{38})	(38, 42, e_{38})
(39, S, DG1)	(40, 58, TG0)	(42, 43, e_{42})	(42, 44, e_{42})	(42, 45, e_{42})

(42, 46, e_{42})	(42, 47, e_{42})	(42, 48, e_{42})	(42, 49, e_{42})	(43, S, DG1)
(44, S, TGB)	(45, S, TGC)	(46, S, TN[27])	(47, S, TN[33])	(48, 58, TG0)
(48, S, TG0)	(49, 50, e_{49})	(49, 51, e_{49})	(49, 52, e_{49})	(50, S, DG1)
(51, 58, TG0)	(51, S, TG0)	(52, 53, e_{52})	(52, 57, e_{52})	(53, 54, e_{53})
(52, 54, e_{53})	(53, 55, e_{53})	(53, 56, e_{53})	(55, S, DG1)	(54, 58, TG0)
(54, S, TG0)	(56, 58, TG0)	(56, S, TG0)	(58, 59, e_{58})	(58, 60, e_{58})
(59, S, TG0)	(59, S, TN[54])	(61, 62, e_{61})	(62, 63, e_{62})	(62, 64, e_{62})
(62, 65, e_{62})	(65, 66, e_{65})	(63, S, G_DIS)	(64, 65, DIF_G)	(66, S, G_DIS)
(E, 67, ENF1)	(E, 67, ENF2)	(E, 67, ETM1)	(E, 67, ETM2)	(67, S, ET0)
(E, 68, EIO1)	(E, 68, EIO2)	(68, S, ED0)		

A.3 FONCTION CALCUL-D-TSGV

A.3.1 Programme source

La figure A.2 comprend le code source original de la fonction *calcul-d-tsgv* ; pour des besoins de confidentialité, les commentaires ont été retirés.

```
void calcul_d_tsgv ()
  {

/* Declaration des variables locales   */
    entier          N_ACT;
    reel            R_ALPHA;
    reel            R1_ALPHA;

/* Initialisation de test de signature */
    N_ACT =  K_CAL;
     if (TGL ==  K_TGL_CD)
      {
        NH0 =  K_GLF_D ;
        RH0 = 0.0;
        TN[ 56 ] =  K_GLF_ER ;
      }
    else
      {
        D_CAL =  ok ;
        if ((G1 == K_TGL_CD ) || (G2 == K_TGL_CD ) ||
           (G3 == K_TGL_CD))
          {
            NH0 = TGL;
            RH0 = (reel)NH0 / (reel)10.0;
          }
        else
          {
```

```
            RH0 = (reel)(G3 + G2 + G1 + TGL) / (reel)40.0;
            NH0 = (entier)(RH0 * (reel)10.0);
          }
        if ((D_CAL !=  ok ) ||
           (NH0 <  K_GLF_m ) || (NH0 >  K_GLF_M ))
          {
            NH0 =  K_GLF_D ;
            RH0 = 0.0 ;
            TN[ 56 ] =  K_GLF_ER ;
          }
      }
/* Mise a jour des variables */
    G3 = G2;
    G2 = G1;
    G1 = TGL;

/* Calcul de la temperature moyenne (NMH0) */
    if (NH0 ==  K_GLF_D)
      {
        NMH0 =  K_D ;
        RMH0 = 0.0;
        TN[ 55 ] =  K_M_ER ;
      }
    else
      {
        D_CAL =  ok ;
        if (N1 == K_D)
          {
            NMH0 = NH0;
            RMH0 = (reel)NMH0 / (reel)10.0;
          }
        else
          {
            R_alpha = (reel) R_TC  / ((reel) R_TC  + PKM);
            R1_alpha = (reel)1.0 - R_alpha;
            RMH0 = R1_alpha * R1 + R_alpha * RH0;
            NMH0 = (entier)(RMH0 * (reel)10.0);
          }
        if ((D_CAL ==  ko ) ||
            (NMH0 < K_m ) || (NMH0 > K_M ))
          {
            NMH0 =  K_D ;
            RMH0 = 0.0 ;
            TN[ 55 ] =  K_M_ER ;
          }
      }
    R1 = RMH0;
    N1 = NMH0;

/* Elaboration de NDH0 */
    if ((NMH0 == K_D) || (NH0 == K_GLF_D))
      {
        NDH0 =  K_D_D ;
        TN[ 51 ] =  K_D_ER ;
```

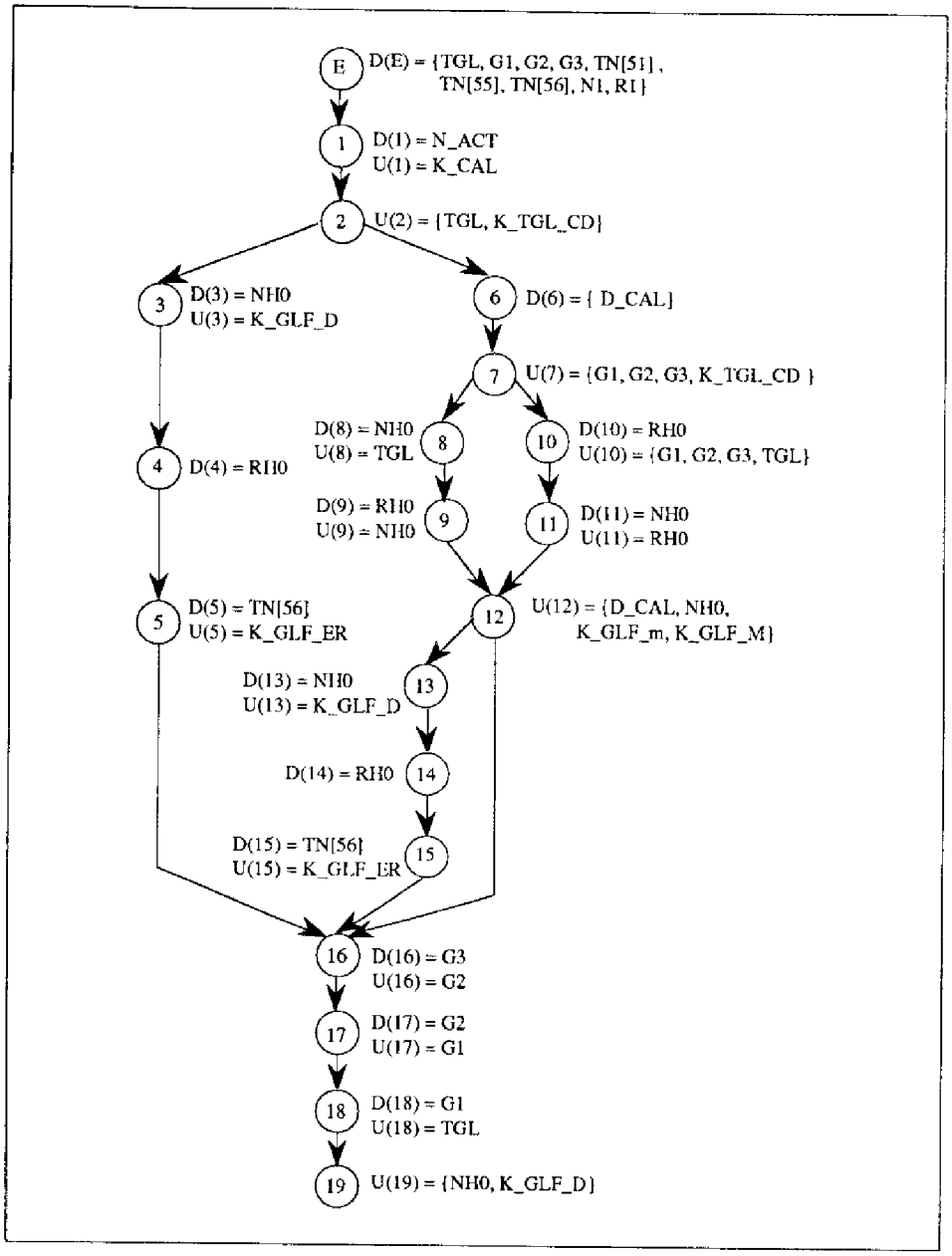
```
      }
    else
      {
        D_CAL =  ok ;
        RDH0 = RMH0 - RH0;
        NDH0 = (entier)(RDH0 * (reel)10.0);
        if ((D_CAL !=  ok ) ||
            (NDH0 < K_D_m ) || (NDH0 > K_D_M ))
          {
            NDH0 =  K_D_D ;
            TN[ 51 ] =  K_D_ER ;
          }
      }

/* Verification de la signature */
    if (N_ACT !=  K_CAL )
      {
        D_SEQUENC =  err_appel ;
        trap1( k_n_err_seq );
      }
    else
      {
        BIT_SIGN. B_ACTION_CALCUL_D_TSGV  =  true ;
      }
    return;
  }
```

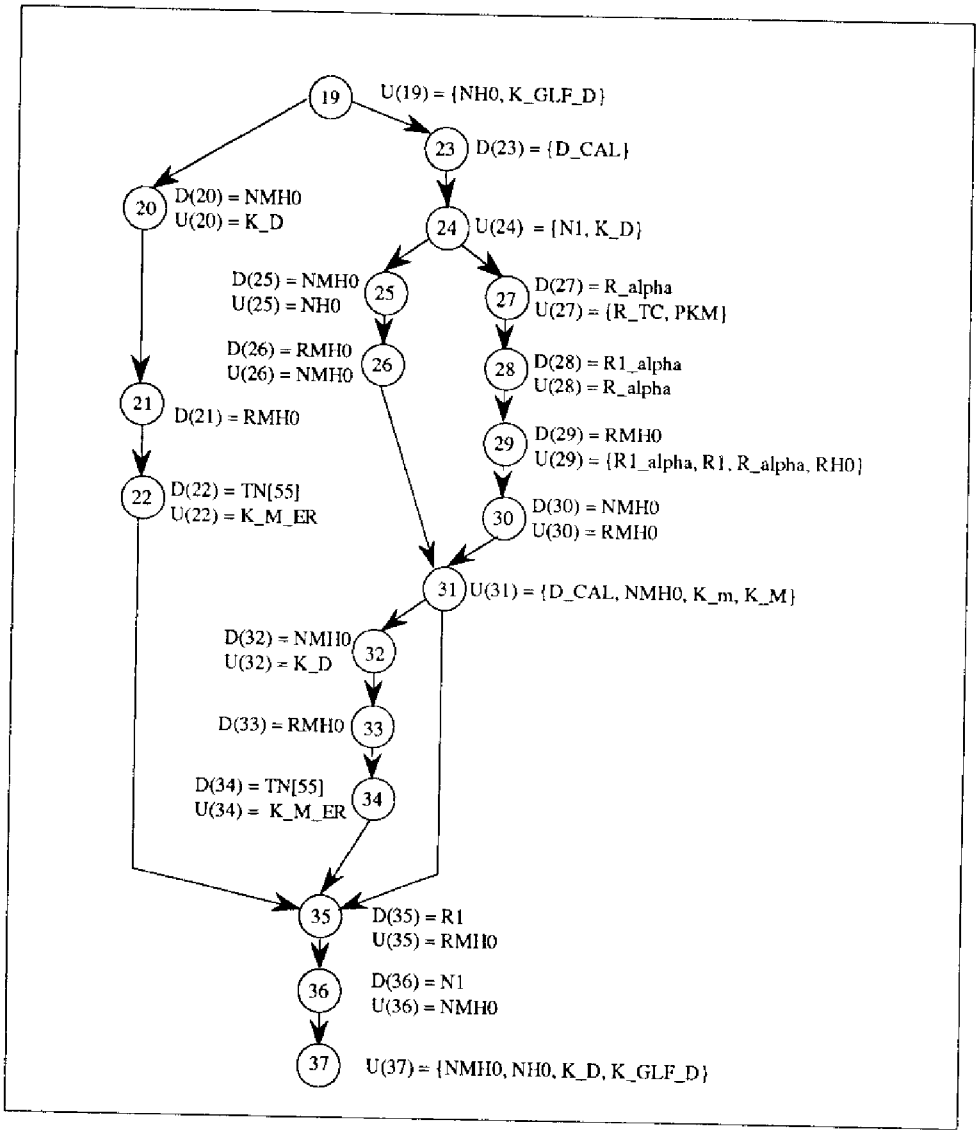
Figure A.2 : Programme source de la fonction *calcul-d-tsgv*

A.3.2 Graphe définition-utilisation

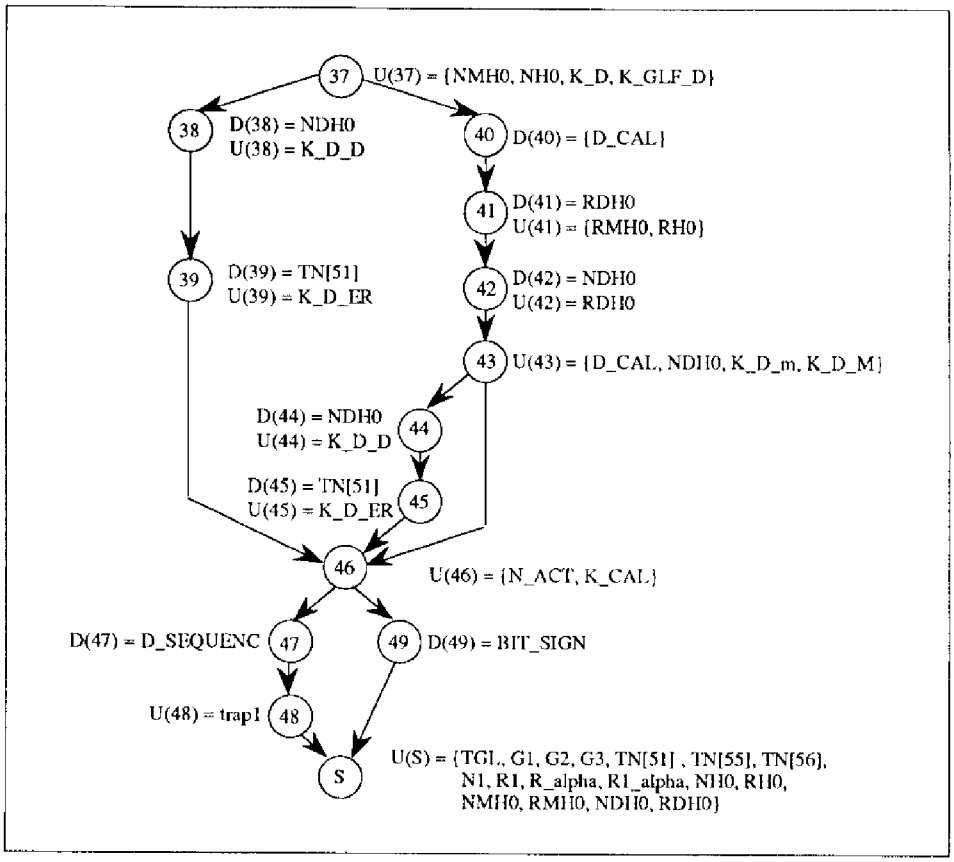
Le graphe définition-utilisation de la fonction *calcul-d-tsgv* comporte 51 nœuds. Nous l'avons fait figurer sur trois pages : la première partie du graphe comprend les nœuds E à 16, la seconde partie les nœuds 16 à 37 et la troisième partie les nœuds 37 à S.



Partie 1 du graphe définition-utilisation de *calcul-d-tsgv*



Partie 2 du graphe définition-utilisation de *calcul-d-tsgv*



Partie 3 du graphe définition-utilisation de *calcul-d-tsgv*

A.3.3 Chemins exécutables

Onze chemins complets exécutables ont été identifiés lors de l'analyse de la fonction *calcul-d-tsgv* ; ils sont listés ci-après sous forme d'une liste de nœuds du graphe définition-utilisation correspondant :

C1 : 1 - 2 - 3 - 4 - 5 - 16 - 17 - 18 - 19 - 20 - 21 - 22 - 35 - 36 - 37 - 38 - 39 - 46 - 49 - 50
C2 : 1 - 2 - 6 - 7 - 8 - 9 - 12 - 13 - 14 - 15 - 16 - 17 - 18 - 19 - 20 - 21 - 22 - 35 - 36 - 37 - 38 - 39 - 46 - 49 - 50
C3 : 1 - 2 - 6 - 7 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 - 18 - 19 - 20 - 21 - 22 - 35 - 36 - 37 38 - 39 - 46 - 49 - 50

C4 : 1 - 2 - 6 - 7 - 8 - 9 - 12 - 16 - 17 - 18 - 19 - 23 - 24 - 25 - 26 - 31 - 35 - 36 - 37 - 40 - 41 - 42 - 43 - 46 - 49 - 50
C5 : 1 - 2 - 6 - 7 - 8 - 9 - 12 - 16 - 17 - 18 - 19 - 23 - 24 - 27 - 28 - 29 - 30 - 31 - 35 - 36 - 37 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 49 - 50
C6 : 1 - 2 - 6 - 7 - 8 - 9 - 12 - 16 - 17 - 18 - 19 - 23 - 24 - 27 - 28 - 29 - 30 - 31 - 35 - 36 - 37 - 40 - 41 - 42 - 43 - 46 - 49 - 50
C7 : 1 - 2 - 6 - 7 - 8 - 9 - 12 - 16 - 17 - 18 - 19 - 23 - 24 - 27 - 28 - 29 - 30 - 31 - 32 - 33 - 34 - 35 - 36 - 37 - 38 - 39 - 46 - 49 - 50
C8 : 1 - 2 - 6 - 7 - 10 - 11 - 12 - 16 - 17 - 18 - 19 - 23 - 24 - 25 - 26 - 31 - 35 - 36 - 37 - 40 - 41 - 42 - 43 46 - 49 - 50
C9 : 1 - 2 - 6 - 7 - 10 - 11 - 12 - 16 - 17 - 18 - 19 - 23 - 24 - 27 - 28 - 29 - 30 - 31 - 35 - 36 - 37 - 40 - 41 42 - 43 - 44 - 45 - 46 - 49 - 50
C10 : 1 - 2 - 6 - 7 - 10 - 11 - 12 - 16 - 17 - 18 - 19 - 23 - 24 - 27 - 28 - 29 - 30 - 31 - 35 - 36 - 37 - 40 - 41 - 42 - 43 - 46 - 49 - 50
C11 : 1 - 2 - 6 - 7 - 10 - 11 - 12 - 16 - 17 - 18 - 19 - 23 - 24 - 27 - 28 - 29 - 30 - 31 - 32 - 33 - 34 - 35 - 36 - 37 - 38 - 39 - 46 - 49 - 50

A.3.4 Associations de dépendances

Un certain nombre de variables sont des paramètres d'entrée et sont définies au nœud d'entrée E. D'autres représentent des variables de sortie et sont donc utilisées par le nœud de sortie S.

(E, 2, TGL)	(E, 7, G1)	(E, 7, G2)	(E, 7, G3)	(E, 8, TGL)
(E, 10, TGL)	(E, 10, G1)	(E, 10, G2)	(E, 10, G3)	(E, 16, G2)
(E, 17, G1)	(E, 18, TGL)	(E, 27, R_TC)	(E, 27, PKM)	(E, S, TN[56])
(E, S, TN[55])	(E, S, TN[51])	(1, 46, N_ACT)	(2, 3, e_2)	(2, 4, e_2)
(2, 5, e_2)	(2, 6, e_2)	(2, 7, e_2)	(2, 12, e_2)	(3, 19, NH0)
(3, S, NH0)	(4, S, RH0)	(5, S, TN[56])	(6, 12, D_CAL)	(7, 8, e_7)
(7, 9, e_7)	(7, 10, e_7)	(7, 11, e_7)	(8, 9, NH0)	(8, 12, NH0)
(8, 19, NH0)	(8, 25, NH0)	(8, 37, NH0)	(8, S, NH0)	(9, 29, RH0)
(9, 41, RH0)	(9, S, RH0)	(10, 11, RH0)	(10, 29, RH0)	(10, 41, RH0)
(10, S, RH0)	(11, 12, NH0)	(11, 19, NH0)	(11, 25, NH0)	(11, 37, NH0)
(11, S, NH0)	(12, 13, e_{12})	(12, 14, e_{12})	(12, 15, e_{12})	(13, 19, NH0)
(13, 37, NH0)	(13, S, NH0)	(14, S, RH0)	(15, S, TN[56])	(16, S, G3)
(17, S, G2)	(18, S, G1)	(19, 20, e_{19})	(19, 21, e_{19})	(19, 22, e_{19})
(19, 23, e_{19})	(19, 24, e_{19})	(19, 31, e_{19})	(20, 36, NMH0)	(20, 37, NMH0)

(20, S, NMH0) (21, 35, RMH0) (21, S, RMH0) (22, S, TN[55]) (23, 31, D_CAL)
(24, 25, e_{24}) (24, 26, e_{24}) (24, 27, e_{24}) (24, 28, e_{24}) (24, 29, e_{24})
(24, 30, e_{24}) (25, 26, NMH0) (25, 31, NMH0) (25, 36, NMH0) (25, 37, NMH0)
(25, S, NMH0) (26, 35, RMH0) (26, 41, RMH0) (26, S, RMH0) (27, 28, R_alpha)
(27, 29, R_alpha) (28, 29, R1_alpha) (29, 30, RMH0) (29, 35, RMH0) (29, 41, RMH0)
(29, S, RMH0) (30, 31, NMH0) (30, 36, NMH0) (30, 37, NMH0) (30, S, NMH0)
(31, 32, e_{31}) (31, 33, e_{31}) (31, 34, e_{31}) (32, 36, NMH0) (32, 37, NMH0)
(32, S, NMH0) (33, 35, RMH0) (33, S, RMH0) (34, S, TN[55]) (35, S, R1)
(36, S, N1) (37, 38, e_{37}) (37, 39, e_{37}) (37, 40, e_{37}) (37, 41, e_{37})
(37, 42, e_{37}) (37, 43, e_{37}) (38, S, NDH0) (39, S, TN[51]) (40, 43, D_CAL)
(41, 42, RDH0) (41, S, RDH0) (42, 43, NDH0) (42, S, NDH0) (43, 44, e_{43})
(43, 45, e_{43}) (44, S, NDH0) (45, S, TN[51]) (46, 47, e_{46}) (46, 48, e_{46})
(46, 49, e_{46})

ANNEXE B

TABLES DE MUTATIONS APPLIQUÉES AU PROGRAMME LOCALES

B.1 INTRODUCTION

Cette annexe regroupe les tables de mutations utilisées lors de nos diverses expérimentations sur le programme LOCALES. Le format des tables est différent selon le type de mutation appliquée : dans le cas des mutations sur les constantes, une mutation consiste à accoler une chaîne de caractères représentant une opération arithmétique à une valeur numérique repérée dans le programme cible, alors que dans le cas des mutations sur les opérateurs du langage ou les symboles utilisés dans le programme cible, une mutation consiste à rechercher et remplacer une chaîne de caractères.

B.2 TABLE DE MUTATIONS SUR LES CONSTANTES

Pour les mutations sur les constantes numériques, chaque ligne de la table de mutation indique la chaîne de caractères (entre guillemets) qui doit être accolée à la chaîne représentant une valeur numérique de constante dans le programme cible. Le caractère suivant la chaîne de caractères indique au module générant les mutants la position de l'insertion : la mutation interviendra en suffixe dans le cas du caractère "s", en préfixe dans le cas du caractère "p" ou ne sera pas insérée dans le cas du caractère "z" si la valeur numérique de la constante est nulle. La figure B.1 représente la table de mutation sur les constantes appliquée dans le cadre des expérimentations sur le programme LOCALES. Il existe peu de constantes apparaissant sous la forme de valeurs numériques dans le code source. La plupart des constantes sont représentées par un symbole et définies dans un fichier inclus en en-tête du code source : les mutations sur ces constantes ont été générées à l'aide d'une table de mutations sur les symboles (cf. B.4).

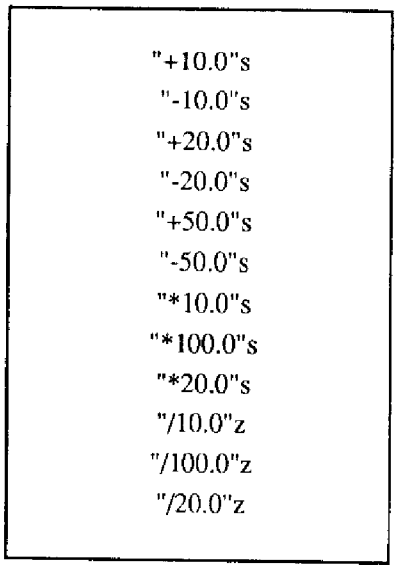


Figure B.1 : Table de mutation sur les constantes

Résultats de la génération des mutants : 44 mutants ont été générés à l'aide de cette table pour la fonction *recopie-capteur* et 101 pour la fonction *calcul-d-tsgv*. Aucun des mutants générés ne s'est avéré équivalent au programme original.

B.3 TABLE DE MUTATIONS SUR LES OPÉRATEURS

Pour les mutations sur les opérateurs du langage, chaque ligne de la table de mutation indique dans sa partie gauche (par rapport au symbole "-->") l'opérateur qui doit être recherché et remplacé dans le code source du programme, successivement par chacun des opérateurs, listés dans la partie droite. Si les parties droite et gauche sont séparées par le symbole "<-->", une permutation est appliquée entre tous les opérateurs ; chaque opérateur à droite et à gauche du symbole "<-->", est alors recherché dans le code source et remplacé par chacun des autres opérateurs figurant sur la même ligne. La figure B.2 représente la table de mutation sur les opérateurs du langage C appliquée dans le cadre des expérimentations sur le programme LOCALES.

Type de mutation	Commentaire
" ^ " <--> " && ", " || "," & ", " | ", " >> ", " << "	permutation op. de traitement des bits et op. logiques entre eux
" ! " --> " ~ ", " "	op. de négation remplacé par op. de complément à un ou supprimé
" ~ " --> " ! ", " "	op. de complément à un remplacé par op. de négation ou supprimé
"== " <--> " != ", " ≤ "," ≥ ", " < ", " > ", " = "	permutation op. de comparaison entre eux et op. d'affectation
" = " <--> " += ", " -= "," *= "," /= ", " <<= "," >>= ", " &= "	permutation op. d'affectation entre eux
" + " <--> " - ", " / ", " * ", " % ", " & "	permutation op. arithmétiques entre eux et op. de ET binaire
"++" <--> "--"	permutation op. d'incrémentation et de décrémentation

Figure B.2 : Table de mutation sur les opérateurs

Résultats de la génération des mutants : 990 mutants ont été générés à l'aide de cette table pour la fonction *recopie-capteur* dont 185 ont été identifiés comme étant équivalents au programme original ; 681 mutants ont été générés pour la fonction *calcul-d-tsgv* dont 114 ont été identifiés comme étant équivalents au programme original.

B.4 TABLES DE MUTATIONS SUR LES SYMBOLES

Les tables de mutation sur les symboles sont traitées par l'outil SESAME de la même manière que les tables de mutation sur les opérateurs. Dans le cadre des expérimentations sur le programme LOCALES, nous avons distingué deux types de symboles : ceux représentant les variables déclarées dans le programme et ceux représentant les constantes symboliques[31] du programme (commençant par la lettre K ou représentant un booléen). Nous avons ainsi établi deux tables de mutation (cf. figure B.3 et B.4).

Dans le cas de mutation sur les variables du programme, nous avons fait figurer sur une même ligne les variables de même type et de même nature (d'un point de vue sémantique).

```
"TE" <--> "TL", "TGE", "TGL", "PP"
"PP" <--> "PP0"
"TE" <--> "T0"
"TL" <--> "T0"
"TGE" <--> "TG0"
"TGL" <--> "TG0"
"D1" <--> "DG1"
"DIF_T" <--> "DIF_G"
"TB" <--> "TC", "TGB", "TGC"
"ET0" <--> "ED0", "T_DIS", "G_DIS"
"PP0" <--> "T0", "TG0"
"ENF1" <--> "ENF2", "ETM1", "ETM2", "EIO1", "EIO2"
"TGL" <--> "G1", "G2", "G3", "N1"
"NH0" <--> "NMH0", "NDH0", "TGL"
"RH0" <--> "RMH0", "RDH0", "R1"
"R_alpha" <--> "R1_alpha"
```

Figure B.3 : Table de mutation sur les symboles appliquées aux variables du programme

31 Ces constantes sont définies par des lignes #define et ne correspondent donc pas à des variables déclarées.

Résultats de la génération des mutants : 419 mutants ont été générés à l'aide de cette table pour la fonction *recopie-capteur* dont 25 ont été identifiés comme étant équivalents au programme original ; 226 mutants ont été générés pour la fonction *calcul-d-tsgv* dont 5 ont été identifiés comme étant équivalents au programme original.

```
"K_TGE_CD" --> "K_MIN", "K_m", "K_MAX", "K_M", "K_D_M", "K_PP_M", "0"
"K_TGL_CD" --> "K_MIN", "K_m", "K_MAX", "K_M", "K_D_M", "K_PP_M", "0"
"K_T_D" --> "K_MIN", "K_m", "K_MAX", "K_M", "K_D_M", "K_PP_M", "0"
"K_D" --> "K_MIN", "K_m", "K_MAX", "K_M", "K_D_M", "K_PP_M", "0"
"K_D_D" --> "K_MIN", "K_m", "K_MAX", "K_M", "K_D_M", "K_PP_M", "0"
"K_PP_CD" --> "K_MIN", "K_m", "K_MAX", "K_M", "K_D_M", "K_PP_M", "0"
"K_PP_D" --> "K_MIN", "K_m", "K_MAX", "K_M", "K_D_M", "K_PP_M", "0"
"K_TE_CD" --> "K_MIN", "K_m", "K_MAX", "K_M", "K_D_M", "K_PP_M", "0"
"K_TL_CD" --> "K_MIN", "K_m", "K_MAX", "K_M", "K_D_M", "K_PP_M", "0"
"K_GLF_D" --> "K_MIN", "K_m", "K_MAX", "K_M", "K_D_M", "K_PP_M", "0"
"K_REC" <--> "K_CAL"
"K_MIN" --> "K_m", "K_D", "K_D_M", "K_PP_M", "K_MAX", "K_M", "0"
"K_T_MIN" --> "K_T_m", "K_T_D", "K_D_M", "K_PP_M", "K_T_MAX", "K_T_M", "0"
"K_m" --> "K_MIN", "K_D", "K_D_M", "K_PP_M", "K_MAX", "K_M", "0"
"K_T_m" --> "K_T_MIN", "K_T_D", "K_D_M", "K_PP_M", "K_T_MAX", "K_T_M", "0"
"K_PP_m" --> "K_MIN", "K_D", "K_D_M", "K_PP_M", "K_MAX", "K_M", "0"
"K_D_m" --> "K_MIN", "K_D", "K_D_M", "K_PP_M", "K_MAX", "K_M", "0"
"K_GLF_m" --> "K_MIN", "K_D", "K_D_M", "K_PP_M", "K_MAX", "K_M", "0"
"K_MAX" --> "K_M", "K_D", "K_D_M", "K_PP_M", "K_m", "K_MIN", "0"
"K_T_MAX" --> "K_T_M", "K_T_D", "K_D_M", "K_PP_M", "K_T_m", "K_T_MIN", "0"
" K_M " --> "K_MAX", "K_D", "K_D_M", "K_PP_M", "K_m", "K_MIN", "0"
" K_T_M " --> "K_T_MAX", "K_T_D", "K_D_M", "K_PP_M", "K_T_m", "K_T_MIN", "0"
"K_D_M" --> "K_MAX", "K_D", "K_M", "K_PP_M", "K_m", "K_MIN", "0"
"K_PP_M" --> "K_MAX", "K_D", "K_D_M", "K_M", "K_m", "K_MIN", "0"
" K_GLF_M " --> "K_MAX", "K_D", "K_D_M", "K_PP_M", "K_m", "K_MIN", "0"
"K_PP_ER" <--> "K_T_ER", "K_ER", "K_D_ER", "K_M_ER", "K_GLF_ER", "K_F_ER", "0"
"PK" <--> "PKM", "PKI"
"false" --> "declenche", "true"
"arme" --> "declenche", "true"
"declenche" --> "arme", "true"
"true" --> "false", "declenche"
"ok" --> "declenche", "ko"
"ko" --> "declenche", "ok"
"0.0" <--> "10.0", "100.0", "20.0", "40.0", "50.0"
"26" <--> "27", "32", "33", "51", "52", "53", "54", "55", "56"
```

Figure B.4 : Table de mutation sur les symboles appliquées aux constantes du programme

Les constantes symboliques définies dans le programme LOCALES représentent souvent la même valeur numérique ; par exemple K_TGE_CD et K_TGL_CD représentent la valeur 32767, "false" et "arme" représentent le booléen 0. C'est pourquoi dans la table de mutation correspondante ne figure pas toutes les permutations possibles ; cela génèrerait un grand nombre de mutants équivalents.

Résultats de la génération des mutants : 516 mutants ont été générés à l'aide de cette table pour la fonction *recopie-capteur* dont 48 ont été identifiés comme étant équivalents au programme original ; 349 mutants ont été générés pour la fonction *calcul-d-tsgv* dont 5 ont été identifiés comme étant équivalents au programme original.

RÉFÉRENCES BIBLIOGRAPHIQUES

[Abrial et al. 1991] J.R. Abrial, M.K.O. Lee, D.S. Neilson, P.N. Scharbach, I.H. Sorensen, "The B-method", Proc. *VDM'91 : Formal Developments Methods*, pp. 398-405, Noordwijkerhout, Pays-Bas, Springer-Verlag, 1991.

[Agrawal et al. 1991] H. Agrawal, R.A. DeMillo, E.H. Spafford, "Dynamic Slicing in the Presence of Unconstrained Pointers", Proc. *Symposium on Testing, Analysis, and Verification (TAV4)*, pp. 60-73, Victoria, Canada, 1991.

[Agrawal et al. 1995] H. Agrawal, J.R. Horgan, S. London, W.E. Wong, "Fault Localization using Execution Slices and Dataflow Tests", Proc. *Sixth International Symposium on Software Reliability Engineering (ISSRE'95)*, pp. 143-151, Toulouse, France, 1995.

[Aho et al. 1990] A. Aho, R. Sethi, J. Ullman, *COMPILATEURS : Principes, techniques et outils*, Paris, InterEditions, 1990, 875 p.

[Arlat 1990] J. Arlat, *Validation de la Sûreté de Fonctionnement par Injection de Fautes : Méthode — Mise en œuvre — Application*, Thèse de doctorat d'état, Institut National Polytechnique de Toulouse, 1990.

[Beizer 1990] B. Beizer, *Software Testing Techniques*, New-York, Van Nostrand Reinhold, 1990, 550 p.

[Bishop & Pullen 1989] P.G. Bishop, F.D. Pullen, "Error Masking: a Source of Failure Dependency in Multi-Version Programs", Proc. *1st Working Conference on Dependable Computing for Critical Applications (DCCA-1)*, pp. 53-73, Santa-Barbara, USA, 1989.

[Crouzet 1995] Y. Crouzet, *Contribution à la Tolérance aux Fautes et à sa Validation*, Habilitation à Diriger les Recherches, LAAS-CNRS, novembre 1995.

[Daran & Thévenod-Fosse 1996] M. Daran, P. Thévenod-Fosse, "Software error analysis : a real case study involving real faults and mutations", Proc. *International Symposium on Software Testing and Analysis (ISSTA'96)*, pp. 158-171, San Diego, USA, 1996.

[Dauchy et al. 1993] P. Dauchy, M.-C. Gaudel, B. Marre, "Using Algebraic Specifications in Software Testing: a case study on the software of an automatic subway", *Journal of Systems and Software*, vol. 21, no. 3, pp. 82-96, juin 1993.

[Frankl & Weyuker 1993] P.G. Frankl, E.J. Weyuker, "Provable Improvements on Branch Testing", *IEEE Transactions on Software Engineering*, vol. 19, no. 10, pp. 962-975, october 1993.

[Davis 1988] A.M. Davis, "A Comparison of Techniques for the Specification of External System Behavior", *Communications of the ACM*, vol. 31, no. 9, pp. 1098-1115, 1988.

[DeMillo et al. 1978] R.A. DeMillo, R.J. Lipton, F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer", *Computer*, vol. 11, no. 4, pp. 34-41, avril 1978.

[DeMillo et al. 1987] R.A. DeMillo, W.M. McCracken, R.J. Martin, J.F. Passafiume, *Software Testing and Evaluation*, Menlo Park, CA, USA, The Benjamin/Cummings Publishing Company, Inc., 1987.

[DeMillo et al. 1988] R.A. DeMillo, D.S. Guindi, W.M. McCracken, A.J. Offutt, K.N. King, "An Extended Overview of the Mothra Software Testing Environment", Proc. *2nd Workshop on Software Testing, Verification and Analysis*, pp. 142-151, Banff, Canada, 1988.

[DeMillo & Offut 1991] R.A. DeMillo, A.J. Offut, "Constraint-Based Automatic Test Data Generation", *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900-910, 1991.

[DeMillo & Offutt 1993] R.A. DeMillo, A.J. Offutt, "Experimental Results from an Automatic Test Case Generator", *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 2, pp. 109-127, 1993.

[DeMillo & Mathur 1994] R.A. DeMillo, A.P. Mathur, *On the Use of Software Artifacts to Evaluate the Effectiveness of Mutation Analysis for Detecting Errors in Production Software*, Rapport , Software Engineering Research Center, Purdue University, W. Lafayette, août 1994.

[DeMillo et al. 1995a] R.A. DeMillo, A.P. Mathur, W.E. Wong, "Some Critical Remarks on a Hierarchy of Fault-Detecting Abilities of Test Methods", *IEEE Transactions on Software Engineering*, vol. 21, no. 10, pp. 858-861, octobre 1995.

[DeMillo et al. 1995b] R.A. DeMillo, H. Span, E.H. Spafford, "Critical Slicing for Software Fault Localization", Proc. *1996 International Symposium on Software Testing and Analysis (ISSTA'95)*, pp. 121-134, San-Diego, USA, ACM Press, 1995b.

[DeMillo et al. 1996] R.A. DeMillo, H. Span, E.H. Spafford, "Critical Slicing for Software Fault Localization", Proc. *1996 International Symposium on Software Testing and Analysis (ISSTA'96)*, pp. 121-134, San-Diego, USA, ACM Press, 1996.

[Ferguson & Korel 1996] R. Ferguson, B. Korel, "The Chaining Approach for Software Test Data Generation", *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 1, pp. 63-86, janvier 1996.

[Ferrante et al. 1987] J. Ferrante, K.J. Ottenstein, J.D. Warren, "The Program Dependence Graph and its Use in Optimization", *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319-349, 1987.

[Frankl & Weiss 1991] P.G. Frankl, S.N. Weiss, "An Experimental Comparison of the Effectiveness of the All-uses and All-edges Adequacy Criteria", Proc. *Symposium on Testing, Analysis and Verification (TAV 4)*, pp. 154-163, Victoria, Canada, 1991.

[Frankl & Weyuker 1988] P.G. Frankl, E.J. Weyuker, "An Applicable Family of Data Flow Testing Criteria", *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1483-1498, octobre 1988.

[Frankl & Weyuker 1991] P.G. Frankl, E.J. Weyuker, "Assessing the Fault-Detecting Ability of Testing Methods", Proc. *ACM Sigsoft'91 conference on Software for Critical Systems*, pp. 77-91, 1991.

[Frankl & Weyuker 1993a] P.G. Frankl, E.J. Weyuker, "A Formal Analysis of the Fault-Detecting Ability of Testing Methods", *IEEE Transactions on Software Engineering*, vol. 19, no. 3, pp. 202-213, mars 1993.

[Frankl & Weyuker 1993b] P.G. Frankl, E.J. Weyuker, "Provable Improvements on Branch Testing", *IEEE Transactions on Software Engineering*, vol. 19, no. 10, pp. 962-975, octobre 1993.

[Frankl & Weyuker 1993] P.G. Frankl, E.J. Weyuker, "A Formal Analysis of the Fault-Detecting Ability of Testing Methods", *IEEE Transactions on Software Engineering*, vol. 19, no. 3, pp. 202-213, mars 1993.

[Gaudel 1993] M.-C. Gaudel, "Développement Formel de Logiciel, Validation et Vérification", Proc. *1er Congrès Biennal de l'Association Française des Sciences et Technologies de l'Information et des Systèmes (AFCET'93)*, pp. 49-58, Versailles, 1993.

[Gaudel 1995] M.-C. Gaudel, "Testing can be formal, too", Proc. *TAPSOFT'95*, Aarhus, LNCS, vol. 915, pp. 82-96, Springer Verlag, 1995.

[Girgis & Woodward 1986] M.R. Girgis, M.R. Woodward, "An Experimental Comparison of The Error Exposing Ability of Program Testing Criteria", Proc. *1st Workshop on Software Testing, Verification and Analysis*, pp. 64-73, 1986.

[Goodenough & Gerhart 1975] J.B. Goodenough, S.L. Gerhart, "Toward a Theory of Test Data Selection", *IEEE Transactions on Software Engineering*, vol. 1, no. 2, pp. 156-173, juin 1975.

[Goradia 1993] T. Goradia, "Dynamic Impact Analysis : A Cost-effective Technique to Enforce Error-propagation", Proc. *International Symposium on Software Testing and Analysis (ISSTA'93)*, pp. 171-181, Cambridge, USA, 1993.

[Hamlet & Taylor 1988] D. Hamlet, R. Taylor, "Partition Testing Does Not Inspire Confidence", Proc. *2nd Workshop on Software Testing, Verification and Analysis*, pp. 206-215, Banff, Canada, 1988.

[Hamlet 1989] R. Hamlet, "Theoretical Comparison of Testing Methods", Proc. *SIGSOFT 3rd Symposium on Software Testing, Analysis and Verification (TAV3)*, pp. 28-37, Key West, USA, 1989.

[Harel et al. 1990] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, M. Trakhtenbrot, "STATEMATE : A Working Environment for the Development of Complex Reactive Systems", *IEEE Transactions on Software Engineering*, vol. 16, no. 4, pp. 403-413, 1990.

[Harrold et al. 1993] M.J. Harrold, B. Malloy, G. Rothermel, "Efficient Construction of Program Dependence Graphs", Proc. *International Symposium on Software Testing and Analysis (ISSTA'93)*, pp. 160-170, Cambridge, MA, USA, 1993.

[Harrold & Rothermel 1996] M.J. Harrold, G. Rothermel, "Separate Computation of Alias Information for Reuse", Proc. *International Symposium on Software Testing and Analysis (ISSTA'96)*, pp. 107-120, San Diego, USA, 1996.

[Horwitz et al. 1988] S. Horwitz, T. Reps, D. Binkley, "Interprocedural Slicing Using Dependence Graphs", Proc. *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pp. 35-46, Atlanta, Georgia, 1988.

[Howden 1982] W.E. Howden, "Weak Mutation Testing and Completeness of Test Sets", *IEEE Transactions on Software Engineering*, vol. 8, no. 4, pp. 371-379, 1982.

[Howden 1987] W.E. Howden, *Functional Program Testing and Analysis*, Computer Science Series, McGraw-Hill International Editions, 1987, 175 p.

[Howden 1991] W.E. Howden, "Program Testing versus Proofs of Correctness", *Journal of Software Testing, Verification and Reliability*, vol. 1, no. 1, pp. 5-15, 1991.

[Jenn 1994] E. Jenn, *Sur la Validation des Systèmes Tolérant les Fautes : Injection de Fautes dans des Modèles de Simulation VHDL*, Doctorat, Institut National Polytechnique de Toulouse, 1994.

[Johnson & Pingali 1993] R. Johnson, K. Pingali, "Dependence-Based Program Analysis", Proc. *ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pp. 78-89, Albuquerque, NM, ACM SIGPLAN Notices, 1993.

[Korel & Yalamanchili 1994] B. Korel, S. Yalamanchili, "Forward Computation of Dynamic Program Slices", Proc. *International Symposium on Software Testing and Analysis (ISSTA'94)*, pp. 66-79, Seattle, Washington, USA, 1994.

[Laprie et al. 1995] J.-C. Laprie, J. Arlat, J.-P. Blanquart, A. Costes, Y. Crouzet, Y. Deswarte, J.-C. Fabre, H. Guillermain, M. Kaâniche, K. Kanoun, C. Mazet, D. Powell, C. Rabéjac, P. Thévenod-Fosse, *Guide de la Sûreté de Fonctionnement*, Cépaduès-Editions, Toulouse, France, 1995, 324 p.

[Laski et al. 1995] J. Laski, W. Szermer, P. Luczycki, "Error Masking in Computer Programs", *Software Testing, Verification and Reliability*, vol. 5, pp. 81-105, 1995.

[Lee & Iyer 1993] I. Lee, R.K. Iyer, "Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System", Proc. *23rd Symposium on Fault-Tolerant Computing (FTCS 23)*, pp. 20-29, Toulouse, France, 1993.

[Lions 1996] J.-L. Lions, *Ariane 5 - Echec du vol Ariane 501*, Rapport de la Commission d'enquête, juillet 1996.

[Mathur & Wong 1993] A.P. Mathur, W.E. Wong, *Reducing the Cost of Mutation Testing : An Empirical Study*, Tech. report, Dept of Computer Sciences, Purdue University, West Lafayette, décembre 1993.

[Mazuet 1994] C. Mazuet, *Stratégies de Test pour des Programmes Synchrones - Application au Langage LUSTRE*, Thèse de Doctorat, Institut National Polytechnique de Toulouse, 1994.

[Morell & Murrill 1994] L. Morell, B.W. Murrill, *Error Flow Analysis*, Rapport, Hampton University and Virginia Commonwealth University, 1994.

[Morell 1984] L.J. Morell, *A Theory of Error-based Testing*, PhD Thesis, University of Maryland, USA, 1984.

[Morell 1988] L.J. Morell, "Theoretical Insights into Fault-based Testing", Proc. *2nd Workshop on Software Testing, Verification and Analysis*, pp. 45-62, Banff, Canada, 1988.

[Morell 1990] L.J. Morell, "A Theory of Fault-Based Testing", *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 844-857, 1990.

[Morell & Murrill 1996] L.J. Morell, B.W. Murrill, "Using Perturbation Analysis to Measure Variation in the Information Content of Test Sets", Proc. *International Symposium on Software Testing and Analysis (ISSTA'96)*, pp. 92-97, San Diego, USA, 1996.

[Murrill 1991] B.W. Murrill, *Error Flow in Computer Programs*, PhD Thesis, College of William and Mary, Virginia, USA, 1991.

[Myers 1979] G.J. Myers, *The Art of Software Testing*, New York, John Wiley & Sons, 1979.

[Ntafos 1984] S.C. Ntafos, "An Evaluation of Required Element Testing Strategies", Proc. *7th International Conference on Software Engineering*, pp. 250-256, Orlando, Florida, 1984.

[Offutt 1989] A.J. Offutt, "The Coupling Effect: Fact or Fiction ?", Proc. *Third Symposium on Testing, Analysis and Verification (TAV 3)*, pp. 131-140, Key West, Florida, 1989.

[Offutt 1992] A.J. Offutt, "Investigations of the Software Testing Coupling Effect", *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 1, pp. 5-20, janvier 1992.

[Offutt et al. 1996] A.J. Offutt, A. Lee, G. Rothermel, R. Untch, C. Zapf, "An Experimental Determination of Sufficient Mutation Operators", *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 2, pp. 99-118, avril 1996.

[Offutt & Hayes 1996] A.J. Offutt, J.H. Hayes, "A Semantic Model of Program Faults", Proc. *International Symposium on Software Testing and Analysis (ISSTA'96)*, pp. 195-200, San Diego, USA, 1996.

[Offutt & Lee 1991] A.J. Offutt, S.D. Lee, "How Strong is Weak Mutation ?", Proc. *Symposium on Testing, Analysis and Verification (TAV 4)*, pp. 200-213, Victoria, British Columbia, 1991.

[Offutt & Lee 1994] A.J. Offutt, S.D. Lee, "An Empirical Evaluation of Weak Mutation", *IEEE Transactions on Software Engineering*, vol. 20, no. 5, pp. 337-344, 1994.

[OFTA 1994] OFTA, *Informatique Tolérante aux Fautes*, ARAGO 15, Paris, Masson, 1994.

[Padua & Wolfe 1986] D.A. Padua, M.J. Wolfe, "Advanced Compiler Optimizations for Supercomputers", *Communications of the ACM*, vol. 29, no. 12, pp. 1184-1201, 1986.

[Pande et al. 1991] H.D. Pande, B.G. Ryder, W. Landi, "Interprocedural Def-Use Associations in C Programs", Proc. *Symposium on Testing, Analysis and Verification (TAV 4)*, pp. 139-149, Victoria, Canada, 1991.

[Podgurski & Clarke 1990] A. Podgurski, L.A. Clarke, "A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance", *IEEE Transactions on Software Engineering*, vol. 16, no. 9, pp. 965-979, 1990.

[Rabéjac 1995] C. Rabéjac, *Auto-surveillance logicielle pour applications critiques : méthode et mécanismes*, Thèse de doctorat, Institut National Polytechnique de Toulouse, 1995.

[Rapps & Weyuker 1985] S. Rapps, E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information", *IEEE Transactions on Software Engineering*, vol. 11, no. 4, pp. 367-375, 1985.

[Richardson & Thompson 1988] D.J. Richardson, M.C. Thompson, "The RELAY Model of Error Detection and its Application", Proc. *2nd Workshop on Software Testing, Verification and Analysis*, pp. 223-230, Banff, Canada, 1988.

[RTCA-EUROCAE 1992] RTCA-EUROCAE, *Considérations sur le Logiciel en vue de la Certification des Systèmes et Equipements de Bord*, DO-178B/ED-12B, "Radio Technical Commission for Aeronautics" (RTCA), "Organisation Européenne pour l'Equipement de l'Aviation Civile" (EUROCAE), 1992.

[Rushby 1993] J. Rushby, *Formal Methods and the Certification of Critical Systems*, Technical report no. CSL-93-7, Computer Science Laboratory, SRI International, décembre 1993.

[Rushby et al. 1991] J. Rushby, F.v. Henke, S. Owre, *An Introduction to Formal Specification and Verification Using EHDM*, Rapport no. SRI-CSL-91-02, SRI International, CA, USA, 1991.

[Strauss & Ebenau 1994] S.H. Strauss, R.G. Ebenau, *Software Inspection Process*, McGraw-Hill, Inc., 1994.

[Thévenod-Fosse 1991] P. Thévenod-Fosse, "Software Validation by Means of Statistical Testing : Retrospect and Future Direction", dans *Dependable Computing for Critical Applications (Proc. 1st IFIP Int. Working Conference on Dependable Computing for Critical Applications : DCCA-1)*, (A. Avizienis, J.-C. Laprie, Éds.), pp. 23-50, Vienne, Autriche, Springer-Verlag, 1991.

[Thévenod-Fosse & Crouzet 1995] P. Thévenod-Fosse, Y. Crouzet, "On the Adequacy of Functional Test Criteria Based on Software Behaviour Models", Proc. *5th IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-5)*, pp. 176-187, Urbana-Champaign, USA, 1995.

[Thévenod-Fosse & Waeselynck 1993] P. Thévenod-Fosse, H. Waeselynck, "STATEMATE Applied to Statistical Software Testing", Proc. *International Symposium on Software Testing and Analysis (ISSTA'93)*, pp. 99-109, Cambridge, Massachussets, USA, 1993.

[Thévenod-Fosse et al. 1991] P. Thévenod-Fosse, H. Waeselynck, Y. Crouzet, "An Experimental Study on Software Structural Testing: Deterministic versus Random Input Generation", Proc. *21st Fault-Tolerant Computing Symposium*, pp. 410-417, Montréal, 1991.

[Thévenod-Fosse et al. 1995] P. Thévenod-Fosse, H. Waeselynck, Y. Crouzet, "Software Statistical Testing", dans *Predictably Dependable Computing Systems (PDCS2)*, (B. Randell, J.-C. Laprie, H. Kopetz, B. Littlewood, Eds.), pp. 253-272, Berlin, Allemagne, Springer-Verlag, 1995.

[Thompson 1991] M.C. Thompson, *An Investigation of Fault-Based Testing Using the RELAY Model*, PhD Thesis, University of Massachusetts, USA, 1991.

[Thompson et al. 1993] M.C. Thompson, D.J. Richardson, L.A. Clarke, "An Information Flow Model of Fault Detection", Proc. *International Symposium on Software Testing and Analysis (ISSTA'93)*, pp. 182-192, Cambridge, MA, USA, 1993.

[Voas & Miller 1992] J.M. Voas, K.W. Miller, "The Revealing Power of a Test Case", *Software Testing, Verification and Reliability*, vol. 2, pp. 25-42, 1992.

[Voas & Miller 1995] J.M. Voas, K.W. Miller, "Software Testability : The New Verification", *IEEE Software*, vol. 12, no. 3, pp. 17-28, 1995.

[Voas et al. 1992] J.M. Voas, L.M. Morell, K. Miller, "Predicting Where Faults Can Hide from Testing", *IEEE Software*, vol. 8, no. 2, pp. 41-48, mars 1991.

[Waeselynck 1993] H. Waeselynck, *Vérification de Logiciels Critiques par le Test Statistique*, Thèse de Doctorat, Institut National Polytechnique de Toulouse, 1993.

[Weiser 1981] M. Weiser, "Program Slicing", Proc. *5th International Conference on Software Engineering*, pp. 439-449, San Diego, USA, 1981.

[Weiser 1982] M. Weiser, "Programmers Use Slices When Debugging", *Communications of the ACM*, vol. 25, no. 7, pp. 446-452, juillet 1982.

[Woodward & Halewood 1988] M.R. Woodward, K. Halewood, "From Weak to Strong, Dead or Alive ? An Analysis of Some Mutation Testing Issues.", Proc. *2nd Workshop on Software Testing, Verification and Analysis*, pp. 152-158, Banff, Canada, 1988.

[Zeil 1983] S.J. Zeil, "Testing for Perturbations of Program Statements", *IEEE Transactions on Software Engineering*, vol. 9, no. 3, pp. 335-346, 1983.

[Zeil 1984] S.J. Zeil, "Perturbation Testing for Computation Errors", Proc. *7th International Conference on Software Engineering*, pp. 257-265, Orlando, California, 1984.

[Zeil 1989] S.J. Zeil, "Perturbation Techniques for Detecting Domain Errors", *IEEE Transactions on Software Engineering*, vol. 15, no. 6, pp. 737-746, 1989.

TABLE DES MATIÈRES

Thèse de Doctorat de Muriel Daran

"Modélisation des Comportements Erronés du Logiciel et Application à la Validation des Tests par Injection de Fautes"

RÉSUMÉ : Les fautes logicielles constituent à l'heure actuelle une des principales sources de défaillance des systèmes informatiques critiques. L'absence d'un modèle des fautes logicielles complet et parfait pose le problème de la confiance que l'on peut accorder aux techniques de vérification — et notamment aux tests — par rapport à l'élimination des fautes dans un programme. Cette confiance serait accrue si on pouvait mesurer la capacité de jeux de tests à révéler des fautes injectées dans un programme. Cependant, dans l'état de l'art actuel, il est illusoire de démontrer la représentativité des fautes injectées (par analyse de mutation, par exemple) vis-à-vis de fautes réelles.
Les travaux présentés dans ce mémoire s'intéressent donc aux erreurs générées au cours de l'exécution du logiciel et non pas aux fautes résultant du développement. Ils s'appuient sur de nombreuses études expérimentales qui ont été menées sur deux programmes séquentiels issus d'applications critiques du nucléaire. Ces études ont eu pour objet d'analyser et de comparer les erreurs et les comportements erronés générés d'une part par des fautes réelles, et d'autre part par des fautes artificielles (de type mutations).
Nos observations expérimentales ont permis de définir un modèle des comportements erronés en faisant référence à une représentation de l'état interne d'un programme séquentiel en cours d'exécution. Ce modèle permet d'expliquer, par l'analyse des dépendances du programme, les mécanismes — parfois complexes — de création, d'annulation et de masquage d'erreurs.
Concernant la représentativité des comportements erronés dus à des mutations, les observations issues des deux études de cas montrent que les mutations de premier ordre peuvent produire des erreurs et des comportements erronés similaires, voire identiques, à ceux produits par des fautes réelles. Ces similitudes s'expliquent à l'aide des relations de dépendances qui lient les instructions d'un programme entre elles.
En conclusion, la bonne représentativité des erreurs générées par les mutations nous permet de réhabiliter l'analyse de mutation en tant que technique de validation des tests du logiciel et d'en proposer des applications à des fins industrielles.

Mots clés : validation du logiciel, faute, erreur, défaillance, propagation d'erreur, flots des données, étude des dépendances, analyse de mutation.

"Software Erroneous Behavior Modeling and Application to Test Data Validation by Fault Injection"

ABSTRACT : Software faults represent nowadays one of the main failure sources of critical systems. The lack of a software fault model raises the problem of confidence in the fault revealing power of verification techniques — and especially test data. The confidence should be increased if the test data efficiency could be assessed with respect to seeded faults. However, in the present state of the art, it is impossible to prove the representativeness of injected faults (by mutation analysis, for instance) with respect to real faults.
Therefore, this dissertation is concerned with software errors generated during software execution and not with faults produced during the software development. Experimental studies have been conducted on two sequential critical programs from the nuclear field. These experiments were performed in order to analyze and compare the errors and erroneous behaviors generated on one hand by the real faults and, on the other hand by artificial faults (mutation type).
Our experimental observations allowed us to define a model of erroneous behaviors, based on a detailed representation of the program internal state during execution. This model helps to explain, through program dependence analysis, the mechanisms of error creation, masking and cancellation — that can be sometimes complex.
Regarding the representativeness of erroneous behaviors due to mutations, the observations from the two case studies show that first order mutations can create similar errors and erroneous behaviors to those produced by real faults. These similarities can be explained by the data and control dependencies between the program statements.
Finally, the high representativeness of the errors due to mutations allows us to rehabilitate mutation analysis as a sound technique for the validation of test data and to suggest some industrial applications.

Keywords : software validation, fault, error, failure, error propagation, data flow, dependence analysis, mutation analysis.