Titre: Title:	Théorie et méthode de Tests des états logiques (TEL), preuve par les tests de logiciels ayant un comportement vectoriel
Auteurs: Authors:	Pierre Véronneau, & Pierre N. Robillard
Date:	1986
Type:	Rapport / Report
Référence: Citation:	Véronneau, P., & Robillard, P. N. (1986). Théorie et méthode de Tests des états logiques (TEL), preuve par les tests de logiciels ayant un comportement vectoriel. (Rapport technique n° EPM-RT-86-49). https://publications.polymtl.ca/10141/

Institution:	École Polytechnique de Montréal
Numéro de rapport: Report number:	EPM-RT-86-49
URL officiel: Official URL:	
Mention légale: Legal notice:	

EPM/RT-86/49

THÉORIE ET MÉTHODE DE "TESTS DES ÉTATS LOGIQUES" (TEL), PREUVE PAR LES TESTS DE LOGICIELS AYANT UN COMPORTEMENT VECTORIEL.

Pierre Véronneau, étudiant, Ph.D.
Pierre -N. Robillard, Directeur de thèse

Département de Génie Électrique

École Polytechnique de Montréal
Octobre 1986

Dépôt légal, 2e trimestre 1984
Bibliothèque nationale du Québec
Bibliothèque nationale du Canada

Pour se procurer une copie de ce document, s'adresser au:

Éditions de l'École Polytechnique de Montréal
École Polytechnique de Montréal
Case postale 6079, Succursale A
Montréal (Québec) H3C 3A7
(514) 340-4000

Compter 0,10$ par page (arrondir au dollar le plus près) et ajouter 3,00$ (Canada) pour la couverture, les frais de poste et la manutention. Régler en dollars canadiens par chèque ou mandat-poste au nom de l'École Polytechnique de Montréal. Nous n'honorerons que les commandes accompagnées d'un paiement, sauf s'il y a eu entente préalable dans le cas d'établissements d'enseignement, de sociétés ou d'organismes canadiens.

Table des matières.

Tableaux

Résumé

Dans ce rapport, une approche appelée "Test des Etats Logiques" est proposée. Dans cette stratégie de tests, le programme est séparé en états logiques du logiciel avant d'être testé. Ces états logiques du logiciel sont par la suite testés séparément.

Les états logiques sont obtenus en groupant les énoncés du programme qui ont le même prédicat conditionnel global. Lorsque les états logiques du logiciel sont des espaces vectoriels, un nombre fini de tests suffisent à prouver que le programme est correct. Un exemple illustre l'approche proposée.

Mots clés: Etats logiques d'un logiciel, énoncés actifs, tests, testable, prédicat conditionnel global, comportement d'espace vectoriel, critère de sélection de tests, données de tests, certifier, preuve de l'exactitude.

Introduction et Orientation de la Recherche

Selon Myers (Myer79), les méthodes de tests permettent de trouver des erreurs dans un programme. Les données de tests sont choisies selon un critère de sélection. Les données de tests sont bien choisies s'ils découvrent des erreurs dans le logiciel.

Howden (Howd76) a démontré qu'il n'y a pas d'algorithme universel pour choisir des données de tests afin de prouver l'exactitude d'un programme arbitraire. Ceci est vrai, même si le programme est correct sur tout l'ensemble de test choisi. La remarque souvent citée de Dijkstra (Dijk72) que les tests ne peuvent découvrir l'absence d'erreurs, mais seulement leur présence est une autre façon de souligner ce problème.

Le choix d'une stratégie de test demeure une matière de jugement professionnel. Goodenough et Gerhart (Good77) discutent du critère idéal de sélection des tests sur une base théorique. Weyuker et Ostrand (Weyu80) critiquent l'application de la théorie de Goodenough et Gerhart et proposent leur propre critère de sélection des tests, pour éviter les problèmes qu'ils soulèvent.

Pour certifier qu'un programme est correct, c'est-à-dire qu'il ne comporte pas d'erreurs, des tests doivent être faits sur tous les chemins possibles du programme et sur tous les domaines d'entrées possibles de ce programme. Comme il y a une infinité de chemins dans un logiciel et que les domaines d'entrées sont souvent infinis, cette stratégie de test est une tâche impossible.

La théorie et la méthode de "Tests des Etats Logiques" est une nouvelle approche qui intègre la théorie des espaces vectoriels avec la théorie moderne des tests. Le programme est séparé en "Etats Logiques du Logiciel", avant d'être testé, en utilisant les prédicats conditionnels du programme. Le nombre total d'états logiques du logiciel dans un programme est un nombre fini et chacun des états logiques peut être testé indépendamment des autres. Lorsqu'un état logique du logiciel est un espace vectoriel, il est possible d'utiliser la théorie des espaces vectoriels. Un nombre fini de tests suffisent alors à prouver que le logiciel ne comporte aucune erreur. Le nombre total de tests à effectuer est fonction du nombre d'états logiques du logiciel et du nombre de tests par état logique.

1 Programme, logiciel et cycle de vie.

1.1 Programme et logiciel.

Dans Robillard (ROBI85), une définition d'un programme d'ordinateur est:

> "...la description, dans un langage formel, de la solution d'un problème exécutable par une machine."(1)

Une logiciel est définit comme étant:

> "Ensemble de programmes destinés à effectuer un traitement sur ordinateur."(2)

Le programme d'ordinateur est le document final de l'aboutissement du cycle de vie d'un logiciel. L'informatisation d'une tâche commence par l'énoncé de

1. (ROBI85) Robillard, P.N. Le logiciel: de sa conception à sa maintenance, p.14.
2. Ibid. , p.135.

la tâche à informatiser et du projet à réaliser et se termine par la maintenance du logiciel.(3)

> Le <<génie logiciel>> "...établit des normes et des critères basés sur une étude ou un modèle scientifique de conception de logiciels. Ces normes doivent viser à aider la conception de logiciels, à créer des produits fiables et efficaces, et à protéger le client contre l'achat de produits de mauvaise qualité, improvisés et inefficaces...", afin de ne pas, "...hypothéquer des générations futures par la maintenance de systèmes devenus indispensables à notre société."(4)

Le produit final est un programme en langage source qui peut être compilé, par exemple le COBOL, le FORTRAN, le C, qui peut être semi-compilé, comme les P-codes ou les I-codes de PASCAL ou de COBOL sur certains

---

3. Ibid. , p.15-22.

4. Ibid. , p.10.

micro-ordinateurs, ou qui peut être tout simplement interprété par l'ordinateur, comme le BASIC, et certains langages de base de données.

1.2 Cycle de vie d'un logiciel.

Le tableau 1.2-1, provenant de Peters (PETE81), illustre comment trois auteurs voient le cycle de vie d'un logiciel. Ce cycle est composé de différentes phases à partir de l'énoncé du projet jusqu'au programme final.

Cycle de vie selon trois auteurs. (5)

Freeman	Metger	Boehm
Analyse des besoins	Définition du système	Exigences du système
Spécifications		Exigences du logiciel
Conception d'architecture	Conception	Conception préliminaire
Conception détaillée		Conception détaillée
Implantation	Programmation	Codage et mise-au-point
	Tests du système	Tests et pré-opération
	Acceptation du système	
	Installation et opération	
Maintenance		Opération et Maintenance

Tableau 1.2-1. Modèles du cycle de vie selon trois auteurs.

Le cycle de vie d'un logiciel tel que vu par Robillard (ROBI85) est montré au tableau 1.2-2:

5. (PETE81) Lawrence J. Peters, Software design: methods & techniques , p.8.

Cycle de vie d'un logiciel. (6)

1. L'énoncé du projet
2. La définition
3. L'analyse
4. La spécification
5. La réalisation
6. La vérification (Les tests)
7. La maintenance

Tableau 1.2-2 Cycle de vie d'un logiciel de Robillard.

Le dernier modèle d'un cycle de vie est un bon exemple de ce qui se passe dans les projets réels. Il sera utilisé pour discuter du rôle de la vérification et de la validation dans le cycle de vie d'un logiciel.

Ce cycle de vie commence par l'énoncé de la tâche à informatiser. La définition du projet décrit de façon aussi précise que possible le travail humain et le

6. (ROBI85) Robillard, P.N. Op.cit. , p.13-23.

processus scientifique à informatiser. L'analyse du projet est constituée des actions et des décisions pour réaliser la définition du projet. La spécification décrit les activités nécessaires à la réalisation. La réalisation du projet est la traduction des spécifications en langage informatique exécutable par la machine. La vérification du logiciel assure le bon fonctionnement du logiciel à différents niveaux d'opérations.

La vérification d'un logiciel "demande souvent plus de la moitié de l'effort total dans l'élaboration d'un logiciel."(7)

La maintenance termine ce cycle de vie. Cette phase corrige les fautes, les défauts et répond à l'évolution des besoins de l'environnement.

---

7. (ROBI85) Robillard, P.N. Op.cit. , p.20.

2 Vérification, validation et tests.

2.1 Définition de la vérification et de la validation.

La publication du IEEE (IEEE83) définit la vérification et la validation comme suit:

Vérification : Processus pour déterminer si le produit d'une phase donnée du cycle de vie rencontre oui ou non les exigences établies durant la phase précédente. Par exemple, la phase analyse est vérifiée par rapport à la phase définition.

Validation : Le processus d'évaluer le logiciel ou programme à la fin du processus de développement du logiciel pour assurer la rencontre des exigences en logiciel. Par exemple, un programme de paie est validé par rapport aux exigences premières du département du personnel qui en a fait la demande.(8)

8.(IEEE83) IEEE Standard Glossary of Software Engineering Terminology , IEEE Std. 729-1983.

En d'autres mots, la vérification se préoccupe à savoir si la construction du produit se fait correctement. La validation elle, se demande si le produit final est correct. La validation voit si le programme final rencontre les besoins originaux de l'usager.(9)

Les besoins en validation sont déterminés au début du projet, en même temps que les spécifications.

---

9. (BOEH84) Boehm, B.W. "Verifying and validating software requirements and design specifications, " IEEE Software, janvier 1984, p. 75.

2.2 Critères de Vérification et de Validation.

Les études de Boehm (BOEH78), McCall, Richard et Walters (McCA77), ainsi que celle de Jones (JONE76) mentionnent les caractéristiques d'un logiciel testable.

D'après Boehm (BOEH84), il y a quatre critères qui permettent la vérification et la validation par rapport aux spécifications. Les spécifications doivent être complètes, consistantes, faisables et testables.(10) Les définitions de ces termes sont les suivantes:

Complète : Une spécification est complète lorsque chacune de ses parties est présente et développée. Un exemple de spécification incomplète est:

> "Le FORMAT des informations sur le personnel sera déterminé plus tard pendant le projet."

Consistante : Une spécification est consistante si elle ne se contredit pas, si elle n'entre pas en conflit

---

10. Ibid. p.76-80.

avec d'autres spécifications et avec les objectifs généraux du projet. Un exemple de spécification qui n'est pas consistante est:

1) Les entrées sont des entiers.

2) Les sorties sont des réels.

Faisabilité : Une spécification est faisable si les bénéfices du cycle de vie excèdent les coûts de sa réalisation. Un exemple de faisabilité est le choix de spécifications pour la maintenance d'un logiciel. Les coûts de la maintenance avec spécifications doivent être inférieurs aux coûts de la maintenance sans spécifications afin de contrebalancer les coûts des spécifications.

Testabilité : Une spécification est testable, s'il est possible d'identifier une technique de test pour s'assurer que le programme développé satisfait à la spécification. Un exemple de spécification non testable est:

"Le logiciel aura les interfaces appropriés avec les sous-systèmes".

Un exemple de spécification testable est:

"Le délai de réponse au terminal sera inférieur à 2 secondes".

2.3 Tests.

Selon Myers (MYER79), le processus des tests est:

> "Un processus d'exécution d'un programme avec l'intention d'y trouver des erreurs."(11)

D'après Beizer (BEIZ83):

"Les tests sont des procédures formelles. Les entrées sont préparées, les sorties sont prévues, les tests sont documentés, les commandes sont exécutées et les résultats sont observés. Ces étapes elles-mêmes sont sujettes à des erreurs... Les tests sont un processus continu dans lequel nous créons des modèles de l'environnement, du programme, de la nature humaine et

---

11. (MYER79) Myers, G.J. The Art of Software Testing , p.5.

des tests eux-mêmes."[12]

Les tests examinent et permettent de visualiser le comportement d'un programme en l'exécutant sur un ensemble ou un sous-ensemble de données. Dans le cas de sous-ensembles, il s'agit alors de tests par échantillonnage. L'objectif général des tests est de comparer les résultats obtenus et les résultats prévus en exécutant le logiciel dans un environnement et dans des circonstances bien contrôlés.

---

12. (BEIZ83) Beizer, B Software Testing Techniques , p.12.

2.4 Intégration des concepts de validation, de vérification et de tests.

La vérification d'une phase peut se faire par une multitude de méthodes dont l'une d'elle est le test du logiciel dans cette phase.

En résumé, une procédure de tests est une des méthodes possibles de vérification d'un logiciel. La vérification de toutes les phases du cycle de vie d'un logiciel permet de conclure à la validation du logiciel par rapport aux besoins originaux s'ils étaient exprimés clairement.

La figure 2.4-1 illustre l'intégration de la vérification aux différentes phases possibles du cycle de vie d'un logiciel. La validation peut se conclure par la vérification de chacune des phases du cycle de vie du logiciel. La figure 2.4-2 fait bien voir la relation et le lien entre la vérification, la validation et les tests.

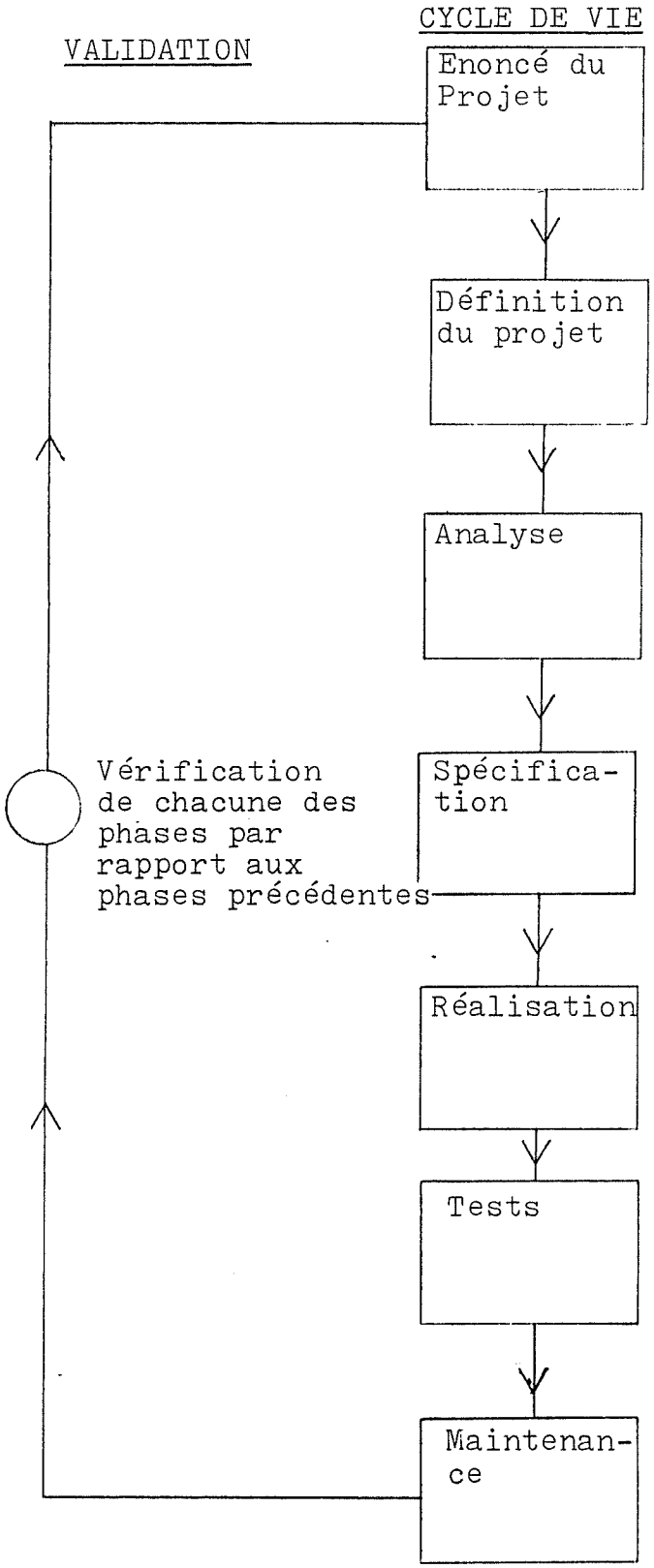


Figure 2.4-1 Exemple de vérification des phases d'un cycle de vie du logiciel.

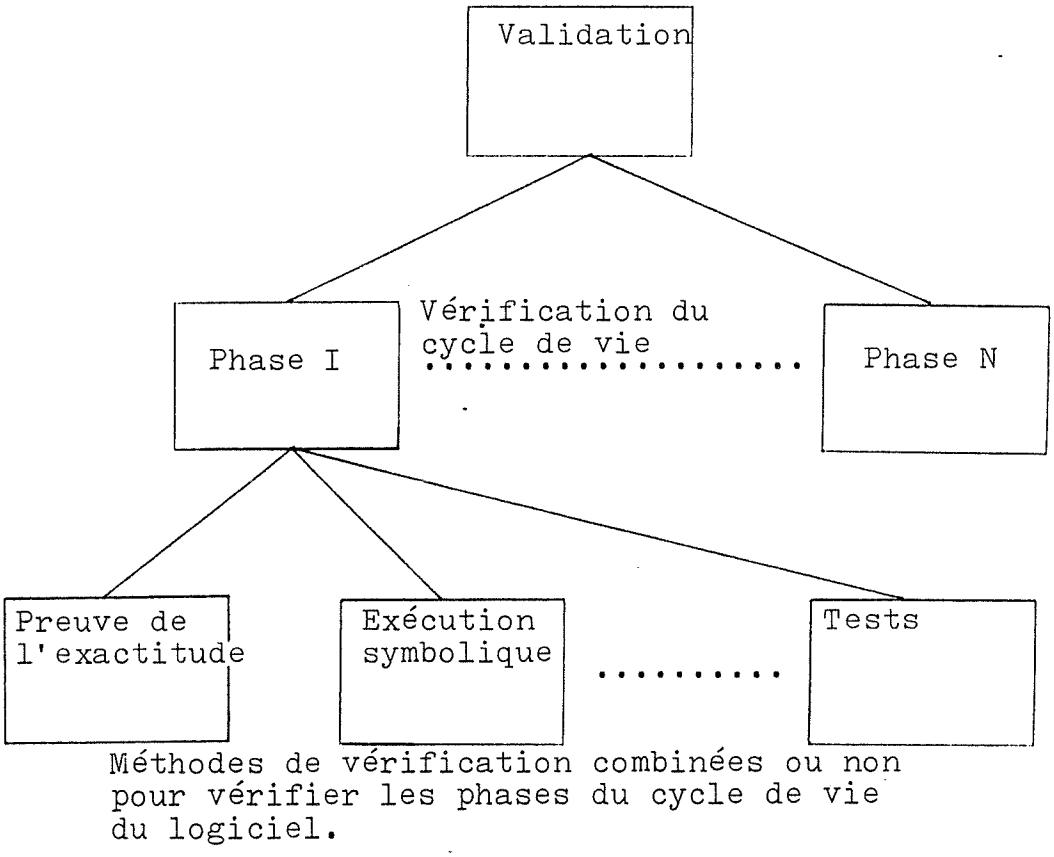


Figure 2.4-2 Validation, vérification et tests.

2.5 Forces motivantes de la validation, de la vérification et des tests.

Les programmes qui provoquent des conséquences graves lors de leurs utilisations justifient un plus grand déploiement de ressources et de budgets pour leur validation. Par exemple, les logiciels qui contrôlent une centrale nucléaire, les navettes spatiales ou qui manipulent le transfert électronique de fonds interbancaires demandent un plus grand effort de vérification et de validation.

INFOTECH (INFO79) a trouvé que plus une erreur avance dans le cycle de vie d'un logiciel sans être trouvée, plus son coût de correction augmente. Par conséquent si la minimisation des coûts de correction des erreurs est un objectif à atteindre, la vérification devrait se faire dès les premières phases du cycle de vie plutôt que dans une phase qui est située à la fin du cycle de vie.

La figure 2.5-1 montre que s'il n'y a pas de vérification dès les phases premières du cycle de vie d'un logiciel, le taux d'erreur augmente considérablement dans les phases finales. En prenant en considération le temps requis pour trouver et corriger une erreur, le temps du projet et les coûts risquent d'augmenter indûment. Par contre lorsqu'il y a vérification des premières phases du cycle de vie, le taux d'erreur est plus élevé au début où le temps requis pour trouver et corriger les erreurs est plus faible. Dans ce cas, le temps réel du projet et les coûts réels auront moins de variance par rapport aux budgets de temps et aux budgets monétaires prévus.

Boehm (BOEH77) a aussi souligné qu'une des erreurs les plus coûteuses est de reporter la détection et la correction des erreurs dans le logiciel jusqu'à la fin du projet. Le but d'intégrer la vérification au cycle de vie du logiciel est de trouver et de corriger les erreurs le plus rapidement possible. La figure 2.5-2 prise dans Boehm (BOEH81) et (BOEH84) présente les coûts pour changer ou corriger le logiciel en fonction de son

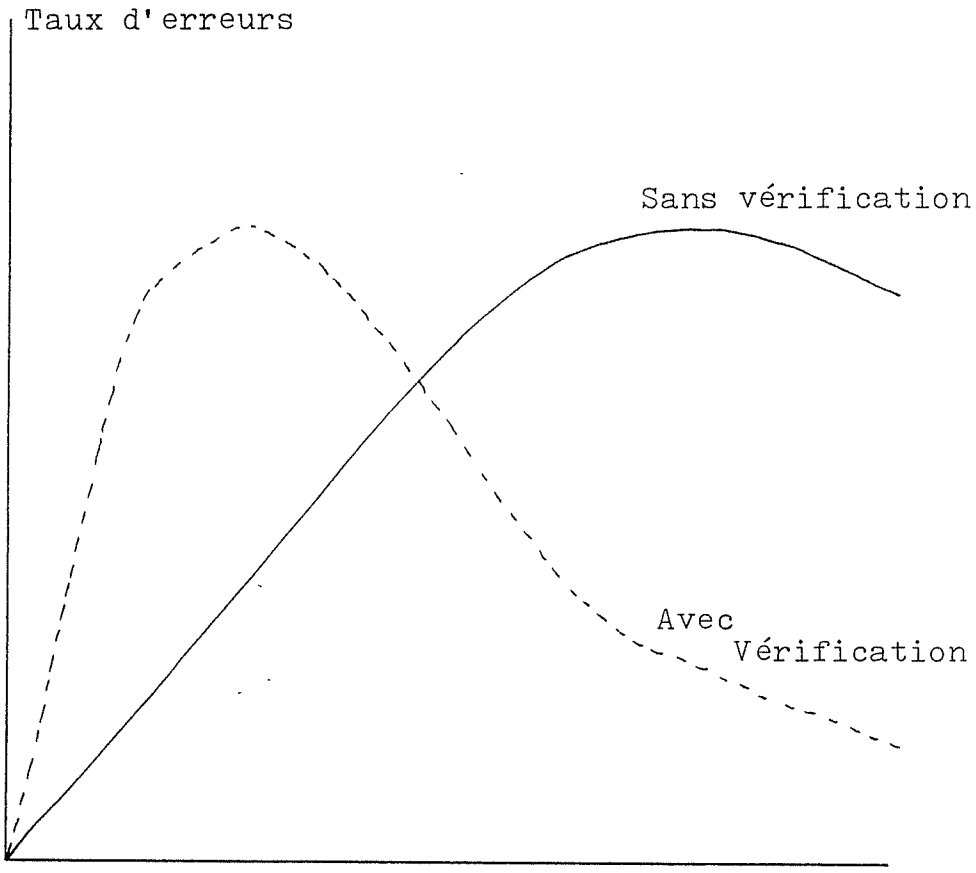


Figure 2.5-1 Taux d'erreurs avec et sans vérification au cours du cycle de vie du logiciel.

Source: Ramamoorthy, C.V. et F. Sui-Bien Ho "Testing large software with automated software evaluation systems", IEEE Trans. Software Eng., Mars 1975.

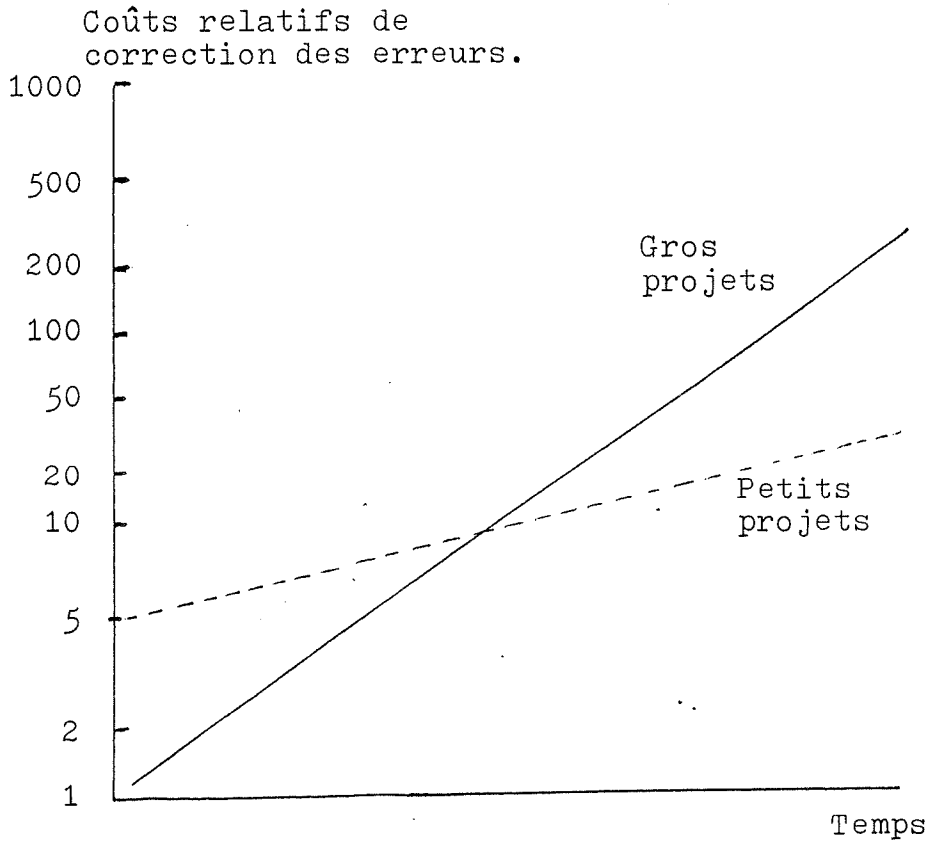


Figure 2.5-2 Coûts relatifs de correction des erreurs en fonction de l'avancement dans le temps du cycle de vie du logiciel.

Source: Boehm, B.W. Software Engineering Economics, Prentice-Hall, Englewood Cliffs, N.J. 1981.

degré d'avancement dans le cycle de vie du logiciel.

La correction d'une erreur dans un logiciel qui s'est rendue à l'usager coûte très cher.

1)Si elle n'est pas trouvée, par les dommages connus et inconnus qu'elle cause.

2)Si elle est trouvée, par la correction des phases précédentes du cycle de vie du logiciel et du programme lui-même. Ce processus est onéreux et peut engendrer d'autres erreurs.

Comme la vérification et la validation demandent beaucoup d'efforts et de temps, en pratique, le logiciel sera souvent livré aux usagers sans être complètement validé.

"Les méthodes actuelles ne permettant pas la validation et la vérification..."(13)

---

13. Robillard, P.N. op.cit. , p.21.

la maintenance aura alors comme rôle de corriger les erreurs trouvées par les usagers en améliorant la version du programme utilisée par les usagers.

2.6 Coûts et bénéfices de la validation.

Les coûts associés à une mauvaise validation et à des erreurs dans le logiciel sont:

1) Les coûts des erreurs lors de l'utilisation des logiciels. Ces coûts peuvent être apparents, par exemple le coût du personnel du département informatique pour retraiter le problème ou non-apparents, comme la perte d'une vente par une entreprise.

2) Les coûts de réparation de l'erreur à partir du retraçage de l'erreur jusqu'à la vérification de la correction de l'erreur pour éviter la génération d'autres erreurs.

3) Les coûts de délai de certaines phases du cycle de vie du logiciel, dus à la correction d'erreurs. Le temps de développement est plus long et les coûts qui y sont associés sont ainsi plus grands.

Si ces coûts sont éliminés, il en résulte des bénéfices pour l'usager. Ces bénéfices compensent les coûts des ressources humaines et matérielles du programme de validation.

Il y a aussi les bénéfices qualitatifs d'un plan de validation qui sont difficilement mesurables en terme de coûts. Les principaux bénéfices qualitatifs sont:

1) Une meilleure acceptation des logiciels par l'usager.

2) Un système de logiciels sans erreurs.

3) La possibilité de réutiliser un logiciel prouvé.

4) Une plus grande confiance dans le procédé de développement du logiciel.

3 Vérification des différentes phases du cycle de vie.

3.1 Enoncé du projet.

La phase de l'énoncé du projet demande que l'énoncé soit correct, complèt et consistant. Un énoncé de logiciel trop vague met en doute la validité du programme final. Dans ce cas, il sera difficile de conclure si le programme final adresse correctement l'énoncé original du projet.

3.2 Définition et analyse du projet.

La compétence des personnes qui définissent le projet est d'une grande importance pour définir clairement le projet. Autant le travail humain, qui est de la compétence du superviseur, que le processus scientifique, qui est la compétence du spécialiste dans la tâche à accomplir, doivent être clairement définis, sans ambiguités.

Lors de l'analyse du projet, une attention particulière est portée aux fonctions introduites par l'analyste, aux structures du système et aux sous-systèmes proposés, aux cheminements dans le logiciel et aux structures de contrôles. Des erreurs communes sont produites par des cas manquants, des logiques fautives, de mauvaises interfaces entre deux modules, des inconsistances dans les structures de données, des hypothèses d'entrées et de sorties erronées, et des interfaces inadéquates avec l'usager. Une technique de conception formelle facilite l'analyse en produisant un énoncé clair de la conception.

3.3 Spécification du projet.

Le projet global est séparé en des ensembles de sous-éléments qui sont plus facilement conceptualisés. Chacun des sous-éléments est définit par une méthode formelle en laissant à plus tard les détails non pertinents. Les caractéristiques requises des spécifications furent discutées au paragraphe 2.2.

3.4 Réalisation du projet.

Plusieurs outils et techniques existent lors de la phase de la réalisation du projet. La revision du programme et l'inspection sont des méthodes manuelles efficaces décrites dans Fagan (FAGA76). Des méthodes d'analyse statique étudient le cheminement des données et la construction du langage. Des outils d'analyse dynamique étudient le logiciel en opération par des techniques d'instrumentation qui indiquent l'état du logiciel en différents points donnés du logiciel.

3.5 Maintenance.

Une fois la conception du logiciel terminée, le logiciel est utilisé par l'usager et la maintenance du logiciel débute.

D'après Pressman (PRESS82), les coûts de la maintenance des logiciels ont augmentés considérablement depuis 20 ans. Le tableau 3.5-1, de Pressman[14], indique le pourcentage du coût de la maintenance par rapport au budget total alloué aux logiciels dans une organisation.

Année	Pourcentage de la Maintenance
1970	35-40%
1980	40-60%
1990	70-80%

Tableau 3.5-1. Pourcentage des coûts de la maintenance par rapport au budget total des logiciels.

Lors de son utilisation, un logiciel requière soit la correction des erreurs ou des changements à ses

14. Pressman, R.G. Software Engineering: A practitioner's approach , p. 326.

capacités originales. La modification d'une phase du cycle de vie d'un logiciel pour corriger une erreur change le logiciel. Après corrections, le logiciel corrigé diffère de sa version non-corrigée.

Après chacune des modifications effectuées, le logiciel doit être retesté. Ce sont les tests de régression. Habituellement, seulement les modules du logiciel affectées par les modifications sont retestés. La documentation est aussi mise à jour.

Les modules du logiciel à tester sont aussi bien à des niveaux hiérarchiques supérieurs ou inférieurs. Pour les fins de cette discussion, un module du logiciel est de niveau supérieur s'il utilise ou appelle d'autres modules du logiciel pour accomplir sa tâche.

Pour faire des tests de régression, les tests utilisés pendant le développement sont réutilisés ou modifiés en conséquence, d'où l'importance de bien documenter les tests originaux lors du développement. Si les données de tests, les résultats des tests, la documentation sur les

tests sont bien préservés et bien classés, le dédoublement des efforts est réduit.

4 Transformation de l'énoncé de projet.

Pour généraliser le processus décrit dans les sections précédentes, l'énoncé du projet original est transformé par les phases du cycle de vie d'un logiciel pour se terminer au programme. Le programme est le document final.

La figure 4-1 montre la transformation de l'énoncé initial en programme exécutable.

L'énoncé initial du projet est transformé lors de différentes phases jusqu'à ce que son contenu puisse être exécuté par l'ordinateur et que son exécution donne les résultats voulus par les usagers. L'énoncé initial est maintenant devenu un programme exécutable par l'ordinateur.

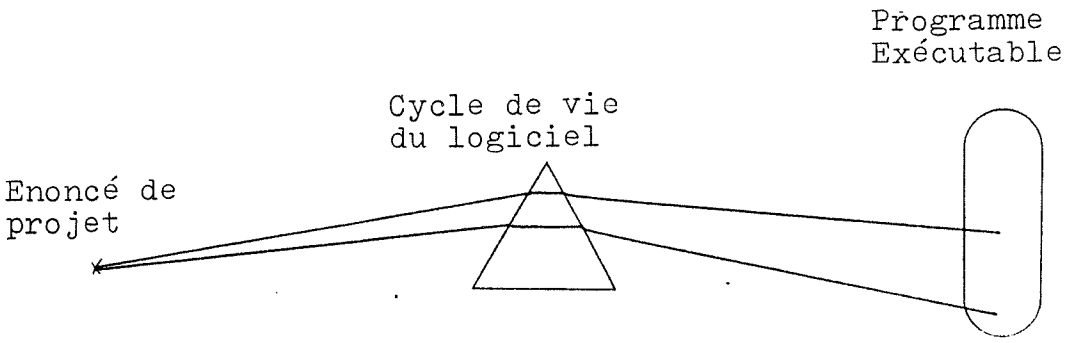


Figure 4-1 Enoncé de projet interprété par le cycle de vie du logiciel et résultant en un programme exécutable.

5 Erreurs.

5.1 Définition des Erreurs.

Les erreurs peuvent être définies comme étant la différence entre deux valeurs.(15)

Les erreurs dans les logiciels peuvent ainsi être classées en:

1) Erreurs de performance: Le défaut de produire les résultats spécifiés dans les limites désirées de temps et d'espaces.

2) Erreurs de logique: La production de résultats incorrectes indépendamment des limites désirées de temps et d'espaces.

Ce travail se penchera uniquement sur le deuxième classe d'erreurs.

---

15. Robillard, P.N. Op.cit. , p21.

5.2 Fautes et défauts des erreurs de logiques.

Une faute est une mauvaise réalisation qui cause une erreur. Un défaut est l'absence d'un élément nécessaire.(16) Il est utile de distinguer les différentes fautes et défauts des erreurs de logiques, en:

1) Faute et défaut de réalisation: La faute et le défaut de ne pas satisfaire une spécification par la réalisation.

2) Faute et défaut de spécification: La faute et le défaut de ne pas écrire une spécification qui représente correctement une analyse du projet.

3) Faute et défaut d'analyse: La faute et le défaut de ne pas reconnaître et comprendre par l'analyse une description de projet.

4) Faute et défaut de description du projet: La

Op.cit. p.21.

faute et le défaut de ne pas décrire correctement un énoncé de projet.

5) Faute et défaut de l'énoncé du projet: La faute et le défaut de ne pas affirmer correctement les besoins de traiter l'information par ordinateur et de ne pas désigner une personne responsable.

5.3 Fautes et défauts dans le programme source.

Il est utile de séparer les fautes et les défauts dans le programme source lui-même.

1) Faute d'une branche de contrôle: Une condition est mal exprimée et il y a sélection d'une action qui est accomplie ou omise sous de fausses conditions. Par exemple: Tester l'égalité de 3 nombres par (X+Y+Z)/3=X.

2) Défaut d'une branche de contrôle: Une condition particulière n'est pas exprimée. Par exemple: La division par 0 n'est pas contrôlée.

3) Faute de l'action inappropriée: Une action inappropriée ne donne pas le bon résultat ou la bonne condition. Par exemple: W*W au lieu de W+W.

4) Défaut de l'action inappropriée ou du chemin manquant: Une action qui devrait être dans le

programme n'y est pas et le traitement requis ne se fait pas.

Des fautes et des défauts de calculs sont causées par de mauvais énoncés d'assignations qui n'affectent pas la structure des contrôles à travers un programme.

Des fautes et des défauts de sous-domaines des données à l'entrées sont causés par de mauvais énoncés d'assignation qui affectent les structures de contrôles mais non les valeurs des variables de sorties.

Les erreurs mesurent les fautes et les défauts dans un logiciel. Les fautes et les défauts multiples ont des effets cumulatifs et combinatoires sur le comportement d'un logiciel.

Le nombre d'erreurs dans un programme mesure les corrections à apporter pour corriger les fautes et les défauts d'un programme. En effet, il faut discerner les deux dimensions des erreurs, soit les mesures des erreurs par les écarts dans les résultats et les causes

de ces erreurs, qui sont les fautes et les défauts dans le programme ou le logiciel.

6 Méthodes de vérification.

6.1 Introduction.

Les différentes méthodes de vérification qui sont discutées dans la littérature de recherche et qui sont utilisées en pratique seront revisées. Le but est de voir l'état de l'art actuel des méthodes de vérification et l'orientation de la recherche actuelle. Par la suite la théorie et la méthode de "Test des états logiques (TEL)" sera présentée comme une solution au problème de tests dans un environnement informatique bien défini.

6.2 Inspection de la logique du programme.

Une équipe est formée de membres comprenant le programmeur, un membre de l'équipe de conception, un responsable de la qualité et un membre de l'équipe de tests. Le concepteur fait une lecture étape par étape du produit et chaque étape est comparée à une liste de critères, qui peuvent être, les erreurs communes dans le passé, les normes de programmation, la consistance avec les spécifications,etc...

Des critères possibles sont discutés dans Myers (MYER79), Fagan (FAGA76) et Weinberg (WEIN71). Les examinateurs réfèrent toujours à leur liste de critères pour juger du programme sous examen. Le but n'est pas de trouver les fautes et les défauts des erreurs, mais tout simplement de souligner les erreurs qui ont franchies les étapes de la compilation et de l'analyse statique.

Les erreurs sont trouvées par le programmeur lui-même en lisant à haute voie ou par les membres du comité d'inspection.

6.3 Simulation manuelle de la logique du programme.

Dunn (DUNN83) mentionne la méthode de la simulation manuelle qui diffère de l'inspection de la logique du programme dans les points qui suivent. Une équipe est formée comprenant un modérateur, le concepteur et 3 à 6 autres membres qui ne sont pas impliqués dans le développement du projet. Une lecture du produit, des exigences, des spécifications et du code source, est faite dans une rencontre formelle ayant des règles précises d'évaluation.

Le directeur du comité choisit des données de tests et une simulation manuelle est faite à travers le système. Les résultats sont inscrits sur papier ou sur un tableau. Le but principal de cette méthode est d'encourager la discussion.

6.4 Preuve de l'exactitude.

La preuve de l'exactitude est une méthode statique d'analyse ou la logique mathématique est utilisée pour prouver un logiciel. Cette technique consiste à valider une hypothèse de consistance d'une sortie en rapport avec un programme donné et une hypothèse d'entrée. Les hypothèses qui viennent d'être mentionnées sont précisées dans les spécifications. Pour prouver complètement le programme, le vérificateur doit aussi prouver que le programme se termine.

Floyd (FLOY67) traite de la logique mathématique et de la programmation en gardant les deux séparées. Constable et O'Donnel (CONS78) intègrent la logique mathématique à la programmation. Ambler (AMBL78), Robinson (ROBI79) et Neuman (NEUM75) utilisent des méthodes de preuve de l'exactitude dans les différentes phases du cycle de vie d'un logiciel. King (KING76) et Apt (APT81) résument la littérature sur cette méthode de vérification.

La complexité de la preuve de l'exactitude croît grandement avec la grosseur du programme à prouver ce qui rend son application difficile.

6.5 Simulation.

La simulation est l'utilisation d'un modèle exécutable pour représenter le comportement d'un logiciel.(17)

Pour utiliser cette méthode, un modèle est construit à partir de la représentation formelle du logiciel en un moment précis du cycle de vie du logiciel. Le modèle doit aussi bien représenter le logiciel qu'il veut simuler, que l'environnement dans lequel le logiciel évolue. Le modèle comprend donc une représentation de l'équipement et des logiciels d'exploitation sur lequel le logiciel fonctionnera.

---

17. (ROBI85) Robillard, P.N. Op.cit. p.199.

6.6 Analyse statistique des erreurs.

Mills (MILL72) a développé une méthode où le vérificateur sème lui-même des erreurs dans le logiciel. Ces erreurs ont un comportement statistique semblable aux erreurs connues dans le logiciel. Les erreurs non-semées sont déduites par le ratio suivant:

Estimé E = I * S / K

où

E : nombre total estimé d'erreurs non-semées.

S : nombre total d'erreurs semées.

K : nombre d'erreurs semées découvertes.

I : nombre d'erreurs non-semées découvertes.

L'hypothèse du comportement statistique semblable des erreurs semées et non-semées fut critiquée par Schick et

Wilverton (SCHI78). Les dernières erreurs qui restent à la fin de la phase de réalisation sont subtiles et comportent des erreurs profondes de logique tel que mentionné par Demillo, Lipton et Sayward (DEMI78). Les niveaux de confiance de cette méthode sont examinés dans Tausworthe (TAUS77) et Duran et Wiorkowski (DURA81a).

6.7 Analyse par mutation.

Une autre méthode fut développée par DeMillo, Lipton et Sayward (DEMI78). La méthode consiste à semer le programme original avec des erreurs, et à faire des programmes mutants. Le programme et ses mutants sont testés par les données de tests. Il y a deux hypothèses de base pour cette méthode.

1) La première est l'hypothèse du programmeur compétent. Cette hypothèse assure que le programme du concepteur ne diffèrera pas substantiellement du programme voulu par les usagers.

2) Une deuxième hypothèse est celle de l'effet de couplage. Un test qui découvre une erreur simple, donnera des indices sur des erreurs plus subtiles. Ces deux hypothèses facilitent la constructions de programmes mutants. L'application de la méthode fut faite par Acree (ACRE80) sur des programmes de 1700 lignes.

6.8 Exécution symbolique.

Il s'agit d'une méthode qui définit symboliquement les données à l'entrée pour forcer le programme à prendre les chemins voulus. Au lieu d'utiliser des données de tests réelles, des valeurs symboliques comme a,b...z seront employées. Le résultat d'une exécution symbolique est une expression algébrique complexe qui peut être décomposée et étudiée comme une structure d'arbre, où chaque branche est un chemin à travers le programme. La méthode est discutée dans Clarke (CLAR77) et Howden (HOWD77).

7 Tests comme méthode de vérification.

7.1 Introduction.

En pratique, les tests sont difficiles à faire, prennent beaucoup de temps et sont habituellement incapables de vérifier complètement un programme. De plus, les tests sont faits à la fin du cycle de vie du logiciel lorsque presque toutes les phases sont terminées. Normalement, il ne reste plus que l'implantation dans un environnement réel si cela n'est pas déjà fait et la phase de maintenance.

7.2 Documentation.

Le IEEE (IEEE79) mentionne que la documentation des tests doit être adéquate. La publication numéro 38 du FIPS, le "National Bureau of Standards", recommande qu'une documentation de tests soit préparée pour tout projet à usages multiples ou à usagers multiples, dont le coût dépasse $5000. La même publication recommande la planification des tests et un rapport d'analyse sur les résultats des tests.

La planification des tests doit contenir les objectifs, les cédules et les exigences des tests. En plus, la planification doit contenir les spécifications, les descriptions, les procédures pour les tests, les critères de réduction des tests et les critères d'évaluation. Le rapport d'analyse des tests doit résumer et documenter les résultats des tests et les conclusions trouvées.

Le sommaire de l'analyse doit présenter et documenter les forces et les faiblesses du logiciel ainsi que des recommandations pour les améliorations du logiciel.

7.3 Outils pour les tests.

Pour bien faire les tests il est utile d'avoir les outils suivants:

1) Génération de données de tests: C'est une méthode systématique pour obtenir des données pouvant servir aux tests.

2) Couverture de tests: C'est une mesure qui dit si les tests sont plus ou moins complets. Les mesures de couverture les plus usuelles sont le nombre d'énoncés, le nombre de chemins, le nombre de branches conditionnelles et le nombre de données à l'entrée.

3) L'administration des résultats: C'est la méthode pour gérer les résultats. Il est utile de garder les données de tests et les résultats dans une base de données, car l'enregistrement manuel des tests devient vite fastidieux.

4) Génération des rapports: Les rapports sur les tests et leurs résultats doivent être faciles à sortir et à utiliser.

7.4 Hiérarchie des modules testés.

Deux approches pour tester les programmes entiers sont les tests ascendants et les tests descendants. Dans les tests ascendants, les modules de plus bas niveaux hiérarchiques sont testés en premier, puis les tests des modules qui les appellent sont ensuite faits et ainsi de suite. Il est nécessaire de simuler les programmes qui appellent les modules pour tester les modules de plus bas niveau.

Dans les tests descendants, les modules de plus hauts niveaux sont testés, puis les modules qu'ils appellent sont ensuite testés et ainsi de suite. Dans cette approche, il est nécessaire de simuler les modules ou sous-routines qui sont appelés par les modules testés.

7.5 Stratégie générale.

La stratégie générale des tests est de:

1) Sélectionner les méthodes de tests.

2) Choisir les critères d'évaluation des tests.

3) Décider d'un plan d'action pour les tests.

4) Choisir une équipe indépendante pour faire les tests.

5) Déterminer une cédule en fonction des différents objectifs des tests.

7.6 Etapes pour effectuer les tests.

Les 5 étapes pour faire un test sont:

1) Obtenir une valeur de l'ensemble de départ.

2) Déterminer le comportement voulu et anticipé.

3) Exécuter le programme.

4) Observer son comportement réel.

5) Comparer le comportement réel au comportement voulu.

7.7 Automatisation.

Certains systèmes automatisés pour faire des tests, comme le TPL/2.0 de Panzl (PANZ78), acceptent les données d'entrées, les données de sorties, les noms des modules à exécuter, les valeurs à être retournées par les modules et d'autres paramètres.

En plus d'initialiser les tests, les sorties actuelles sont comparées aux sorties prévues et des rapports concis des différences et des similitudes sont produits.

PRUFSTAND de Sneed et Kirchnoff (SNEE78) est un autre exemple de système de tests automatisés. PRUFSTAND est un sytème interactif qui comprend:

1) Un préprocesseur pour instrumenter le code.

2) Un convertisseur pour transformer les définitions de données dans le code source en

tableaux internes afin de générer les données de tests.

3) Un générateur de tests pour initialiser et mettre à jour l'environnement logiciel.

4) Des modules simulant les modules appelés par le logiciel.

5) Un moniteur pour tracer le cheminement du logiciel lors du test.

6) Un analyseur de résultats des tests.

7) Une bibliothèque de fichiers de tests.

8) Un générateur de rapports.

L'avantage d'un outil automatisé est de standardiser la forme des tests.

7.8 Principes fondamentaux des tests.

Il arrive dans chaque phase du cycle de vie de tester dans le sens d'évaluer les cas discernables et leurs effets. D'après Hansen (Hans73), les tests étant inévitables, il faut identifier les exigences des tests dès le début du cycle de vie du logiciel. Il faut aussi identifier les couplages faibles et critiques des modules d'un système.

Les programmes devraient être structurés de tel sorte que des tests logiques sur différentes abstractions du logiciel, dans des phases antérieures au codage, puissent réduire les tests sur le programme final. L'emphase est mise sur les tests des premières phases du cycle de vie.

Les spécifications devraient être suffisamment précises pour être testables, car elles apparaissent dès le début du cycle de vie du logiciel.

Le besoin d'obtenir de l'information pour vérifier, au moyen de tests un logiciel donné, affecte la conception de ce logiciel. La planification des tests requière de l'information qui n'est pas toujours disponible dans un cycle de vie habituel. Par exemple, la connaissance des valeurs pour les données de tests des variables en des points précis du logiciel forceront les personnes responsables de la conception du logiciel à déterminer ces valeurs, ce qui ne se ferait peut-être pas s'il n'y avait pas de tests.

Les objets à être testés sont:

1)Les modules codés.

2)Les spécifications.

3)Les structures de données.

4)Tout autre objet pour le développement correct et l'implantation des logiciels.

Pour bien tester un programme, il faut se représenter les fonctions qui décrivent les relations des éléments à l'entrée ou intrants dans un domaine d'entrée aux éléments de sorties ou extrants dans les domaines de sortie.

L'utilisation de tests dans la validation de logiciels présume l'existence d'un oracle de test[18] pour prouver l'exactitude des sorties de tests. Une oracle de test est une source d'information externe au programme qui est testé. Un oracle peut prendre plusieurs formes: une table, un autre programme, des spécifications, les connaissances des programmeurs.

Faire un test consiste à obtenir une valeur valide du domaine d'entrée fonctionel ou une valeur invalide à l'extérieur, si le test en est un de robustesse, de déterminer le comportement prévu, d'exécuter le programme et d'observer son comportement réel et finallement de comparer le comportement réel au comportement anticipé.

18. (HOWD85) Howden, W.E. "The Theory and Practice of Functional Testing," p.6.

Un test est conçu pour trouver une erreur d'un type particulier. Si le comportement réel est identique au comportement voulu par l'usager, il n'y a pas d'erreur du type particulier que le test est censé retracé.

Par contre, si le comportement réel du programme est différent du comportement prévu, une erreur est trouvée.

Les tests démontrent la présence de fonctions non voulues dans le logiciel en se basant ou non sur la structure logique du programme.

7.9 Tests exhaustifs.

La citation la plus souvent employée dans la littérature est celle de Dijkstra (Dijk72). Dijkstra nous dit que les tests pratiques montrent la présence d'erreurs, mais qu'ils ne peuvent jamais montré leur absence. Cette remarque est basée sur les 2 assertions suivantes:

1) Boehm (Boeh73) affirme que la seule façon de certifier qu'un programme n'a pas d'erreur est d'inspecter chaque chemin. La figure 7.9-1 montre par un "X" les chemins du logiciel qui doivent être testés pour prouver le logiciel.

2) Pool (Pool73) avance que tester un logiciel exhaustivement signifie tester tous les intrants. La figure 7.9-2 indique par la région hachurée les tests qui doivent être faits sur les intrants pour prouver le logiciel.

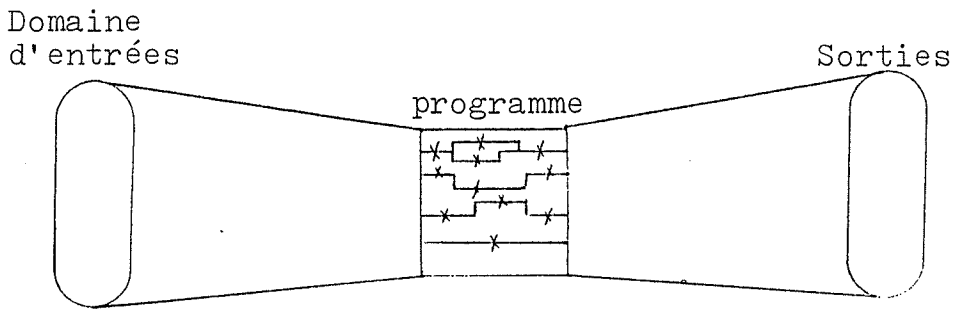


Figure 2.9-1 Tests de tous les chemins d'un programme.

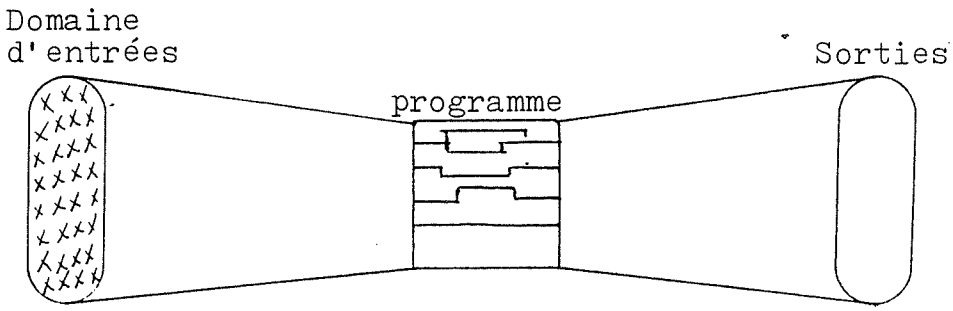


Figure 2.9-2 Tests de tout le domaine des entrées.

Des tests exhaustifs garantissent la validité d'un programme(Stuc74), c'est-à-dire que tous les éléments dans le domaine fonctionnel sont testés.

Des tests exhaustifs, définis en termes de chemins de programmes ou de domaine d'entrées du programme sont infaisables en pratique à cause des ressources considérables qu'ils demandent car les domaines d'entrées et les chemins possibles sont habituellement très grands.

Il se peut que les chemins d'un programme ne traitent pas de la bonne façon les domaines d'entrées différents. Par exemple, les deux chemins dans le programme Schémacode à la figure 7.9-3 comportent une erreur lorsque la condition d'égalité est respectée et que x,y et z sont différents. Par conséquent, le test de tous les chemins d'un logiciel ne garantie pas qu'il n'y a pas d'erreurs.

Les tests exhaustifs d'un domaine de programme n'est pas un procédé qui est garanti de se terminer.

UQAM
Pierre Vermeau
SCHEMACODE 1985 (PC-DOS EDUC) #96

Fichier	Programme principal	Langage cible	Date de creation	Date de derniere : utilisation	codification	Heure
TOUSCHEM	TOUSCHEM	C	1985-12-09	1985-12-09	1985-12-09	00:38:54

R O

```
0  1 /*-TOUSCHEM -UQAM-SCHEMACODE 1985 (PC-DOS EDUC)
   2 /*-----------------------------------------------
   3 /*-          . Programme qui demontre que le test de tous
   4 /*-les chemins ne prouve pas la validite.
   5 /*-----------------------------------------------
   6 /*main
   7 /*{
   8 /*int x,y,z;
   9 /*scanf ("Entrer x espace y espace puis z : %d %d %d",&x,&y,&z);
  10 /*******(x+y+z)/3==x
  11 /*      |print ("x,y et z sont egaux en valeur");
  12 /*******
  13 /*      |print ("x,y et z sont inegaux en valeur");
  14 /*
  15 /*exit(0);
  16 /*}
```

Figure 7.9-3

D'après Hoare dans (BUXT70) la validité d'un programme est prouvable par des tests si le domaine d'un programme peut être séparé en un nombre fini de classes d'équivalence de tel sorte qu'un test représentatif d'une classe permettra une conclusion sur la classe entière. Ainsi, l'équivalent des tests exhaustifs du domaine d'entrées peut être effectué.

Pour ce faire, il faut absolument des critères pour choisir des éléments représantatifs du domaine fonctionnel. Le sous-ensemble choisi est un ensemble de données de tests ou un ensemble de tests. ("test data set" ou "test set").

Goodenough et Gerhart (Good75) traite formellement des critères pour choisir les ensembles de tests.

Howden (Howd76) démontre qu'il n'y a pas d'algorythme pour trouver des critères de tests consistants, fiables, valides et complets qui s'appliqueraient à tous les logiciels.

8 Théorie des tests.

Dans la littérature de recherche actuelle, les théorêmes fondamentaux des tests nous viennent de Goodenough et Gerhard (GOOD75).

L'article de Goodenough et Gerhard (GOOD75) fait contraste avec la plupart des articles sur les tests qui compare un test complet avec un test exhaustif comme Pool (POOL73) et Stuck (STUC74).

Les buts d'une théorie de tests d'après Weyuker et Ostrand (WEYU80) sont de:

1) Produire une base pour les méthodologies de tests de programmes.

2) Etablir des façons de déterminer l'efficacité des tests dans la détection d'erreurs.

Idéalement, nous voudrions construire des tests pour

détecter toutes les erreurs dans un programme. Goodenough et Gerhart furent les premiers à décrire une base théorique pour produire des tests.

8.1 Théorie de Goodenough et Gerhart.

Goodenough et Gerhart (GOOD77) ont argumenté que les tests ayant des données sélectionnées sur la base d'une structure de programme ne sont pas adéquats ni concluants. Il est facile de construire des programmes simples et incorrects pour lesquels des tests couvrant tous les énoncés, toutes les branches ou tous les chemins donneraient des résultats corrects.

Les auteurs donnent une définition d'un test idéal en terme des entrées et des sorties spécifiées pour le programme. La notation utilisée est la suivante:

P: pour le programme

Il existe les domaines d'entrée et de sortie suivants:

D: domaine des entrées ("Input domain")

R: domaine des sorties ("Range")

Sur une entrée $d \in D$, le programme P produit la sortie $P(d) \in R$. Les spécifications de sorties pour P sont données par OUT(x,y) où:

$x \in D$

$y \in R$

P est correct pour l'entrée d, indiqué par OK(d) si P(d) existe et OUT(d,P(d)).

Un <u>test</u> T pour le programme P est simplement un sous-ensemble fini de D. Un <u>critère de sélection de test</u> C spécifie les conditions qui sont remplies par les données du test. Tout ensemble d'entrées qui satisfont les conditions du critère de sélection de tests est dit être un <u>test sélectionné</u> par le critère C.

Exemple d'un critère:.

Le domaine D est l'ensemble des entiers. Un critère de sélection hypothétique pour notre exemple est: "Toute donnée de test doit contenir un entier positif, un entier négatif, et un zero".

{3,0,-7} , {122,0,-11} et {1,0,-1} sont 3 tests sélectionnés par notre critère hypothétique.

Les définitions et le théorême de base sont présentés dans les paragraphes qui suivent.

Définitions.

1. Un test T est réussi si et seulement si:

$$(\forall t \in T)\ (OK(t))$$

2. C est fiable si et seulement si tous les tests choisis par C sont réussis ou si dans le cas

contraire aucun des tests choisis n'est réussi.

3. C est valide si et seulement si C sélectionne au moins un ensemble de test T qui n'est pas réussi pour T , lorsque le programme P est incorrect.

4. Un test idéal pour un programme P consiste en un ensemble de données de test:

$$T = \{ t \}$$

tel qu'il y ait une entrée d pour laquelle une sortie incorrecte est produite par le programme P, si et seulement si il y a une donnée de test t appartenant à T pour laquelle P est incorrect.

Théorême fondamental (GOOD77)

Si C est un critère de sélection fiable et valide, alors tout test sélectionné par C est un test idéal.

En d'autres mots, un critère qui est fiable et valide pour P est un critère idéal .

8.2 Difficultées avec la théorie de Goodenough et Gerhart.

Les problèmes suivants sont discutés dans Weyuker et Ostrand (WEYU80):

1. La fiabilité et la validité sont définies par rapport à tout le domaine d'entrée d'un programme.

2. Les erreurs présentes dans un programme ne sont pas connues à l'avance. Les erreurs peuvent être d'une variété considérable.

3. Les définitions de Goodenough et Gerhart sont relatives à un programme P. Un critère idéal pour P n'est pas nécessairement un critère idéal pour P', une version légèrement différente de P, par exemple une version légèrement améliorée par la maintenance. Ceci est indésirable, si la validité est déduite des tests. Il y a donc un problème pour

conserver la validité dans la phase de réalisation et de maintenance où les logiciels évoluent vers leur version finale et subissent des changements constants.

4. Il y a un manque de dépendance pour les propriétés de validité et de fiabilité.

Comme conclusion le choix des tests ne devrait pas être basé sur les programmes qui évoluent et changent constamment, mais plutôt sur les spécifications.

8.3 Impossibilité de trouver des critères de sélection pour des tests valides et fiables s'appliquant à un programme.

Toujours d'après Weyuker et Ostrand (WEYU80), la validité dépend de la nature des erreurs présentes dans un programme. Les deux possibilités sont:

1) Le programme P est correct.

Formellement

$$(\forall d \in D)(OK(d))$$

Par contre, P n'est pas connu comme étant correct et le programmeur doit utiliser d'autres moyens pour montrer la fiabilité et la validité du critère de sélection C. Ce procédé est équivalent à prouver l'exactitude du programme.

2) Le programmme P n'est pas correct.

Il n'y a pas de façon de voir si le critère est idéal sans connaître les erreurs et leurs natures à l'avance.

8.4 Exemple des concepts précédents.

Soit les spécifications et le programme suivant:

Spécifications de sorties: P(d) = d + d

Programme conçu: P = d * d

1) Un critère qui choisit le sous-ensemble {0,2} est fiable, mais non valide parce qu'il n'indique pas l'erreur dans le programme P.

2) Un critère qui choisit la sous-ensemble {0,1,2,3,4} expose l'erreur dans P. Il est valide, mais non fiable car {0,2} est fiable et {0,1} ne l'est pas.

3) Si le programme est changé à

P' = d + 2

d = 2 est la seule entrée correcte. Le critère qui choisit {0,2} devient valide, mais non fiable pour P'.

4) Si le programme est changé à

$$P'' = d + 5$$

le critère qui choisit {0,1,2,3,4} est simultanément valide et fiable.

8.5 Conclusions.

Il n'y a aucune façon, avec la méthode de Goodenough et Gerhart (GOOD77), de savoir si un critère est idéal sans connaître préalablement les erreurs du programme. L'implantation et la maintenance change constamment le programme. Il se peut que:

1) les erreurs soient localisées et corrigées.

2) quelques fois, de nouvelles erreurs soient introduites non-intentionnellement, par le processus de correction.

8.6 Exemple de corrections d'erreurs.

Spécifications: P(d) = (d + d) + 6

Programme soumis: P(d) = (d * d) + 3

OK(3) et OK(-1). Pour tout autre d, OK(d). Le critère C(T) <=> T = { t } et t = {0,1,2} est un critère valide et fiable. Supposons que l'erreur d'addition de la mauvaise constante est localisée. P devient

$$P'(d) = (d * d) + 6$$

Nous obtenons OK(0) et OK(2), mais pour tout autre d, C(T) n'est plus fiable. C'(T) <=> T = { t } où

$$t = \{-1,1,3\}$$

est valide, mais non fiable pour le programme original et ce même sous-ensemble de tests de données devient un critère idéal pour le nouveau programme.

Le problème des propriétés de Goodenough et Gerhart est le même que les test basés sur la structure des programmes, ce problème est la dépendance des tests sur le programme à tester.

Une solution est d'utiliser un critère de sélection qui ne dépende que des spécifications de sorties, ce qui est équivalent aux tests de boîte noire.

Howden (HOWD76) montra qu'il n'y a pas de procédure générale qui, étant donné un programme P et des spécifications de sorties, produira un ensemble de données de test non vide T D tel que si P est correct sur T alors P est correct sur tout D. La solution de Howden est de chercher des sous-classes de programmes pour lesquels des procédures de génération de tests ont du succès.

De la même façon, Weyuker et Ostrand (WEYU79) ont montré qu'il n'y a pas d'algorythme qui peut décider si un

énoncé , une branche ou un chemin de programme donné , ne sera jamais pris par le programme, ni si tous ces éléments seront pris par le programme.

D'après Weyuker et Ostrand (WEYU80), le domaine d'entrées du problème doit être partitionné intelligemment en sous-ensembles significatifs et ayant les mêmes caractéristiques.

9 Tests en pratique.

Malgré les problèmes soulevés par la théorie actuelle sur les tests des logiciels, les praticiens doivent quand même valider du mieux qu'ils peuvent les logiciels qui leur sont soumis.

9.1 Psychologie des tests.

Myers (MYER79) parle de tests comme le processus d'exécution d'un programme avec l'intention d'y trouver des erreurs.

D'après Myers, le processus de tests est un processus destructif. Un cas de test qui ne découvre pas d'erreur est un gaspillage. Un cas de test qui trouve une nouvelle erreur est un investissement valable.

Cette mentalité de tests va à l'encontre de l'esprit de

créativité qu'il est nécessaire d'avoir dans la conception d'un logiciel. D'où la nécessité d'avoir une équipe de tests qui est indépendante de l'équipe de conception. Un objectif impossible est de ne pas avoir d'erreur. Même si des erreurs sont présentes, un programme peut fonctionner quand même, dans la plupart des utilisations courantes.

Des définitions qui ne sont pas toujours bonne d'après Myers (MYER79) sont:

1)Le procédé de tests est un procédé pour démontrer qu'il n'y a pas d'erreur.

2)Le but des tests est de montrer qu'un programme remplit correctement les fonctions pour lesquelles il a été conçu.

3)Le but des tests est d'établir la confiance qu'un programme fait ce qu'il doit faire.

9.2 Choix des données de tests.

Le choix des données de tests est l'étape critique de toute méthodologie de tests. La sélection des données de tests comprend deux étapes, soit le choix des données à l'entrée et la détermination de leur sorties.

9.2.1 Choix des données selon les techniques de tests fonctionnelles.

Une première approche est de tester les fonctions d'un programme en décomposant le domaine d'entrées en sous-domaines ayant des caractéristiques semblables de sorte que des données représentatives sont choisies pour les tests. De plus, le vérificateur choisira les valeurs extrêmes comme dans Myers (MYER79) ou spéciales comme dans Howden (HOWD80a). Cette méthode de choix dépend beaucoup de l'habilité naturelle du vérificateur pour choisir des données de tests qui lui feront découvrir les erreurs cachées dans le logiciel.

Cette technique de choix s'appuie sur le concept de boîte noire.

9.2.2 Définition d'une boîte noire.

Une boîte noire est une partie de logiciel, habituellement un module qui fait une fonction, la partie boîte, mais qui cache complètement la manière dont la fonction est exécutée. La Figure 9.2.2-1 donne une exemple de boîte noire qui fait une fonction f sur des intrants A,B,..,Z pour donner des extrants a,b,..,z.

Pour être complètement spécifiée, la description de la boîte doit inclure l'intervalle des intrants qui sont corrects et un énoncé clair et précis des relations des extrants aux intrants, sur le domaine des intrants. Une fois que les interfaces (intrants et extrants) sont définis et que la fonction de la boîte est claire, l'usager a l'information voulue pour évaluer si la boîte est appropriée à son problème.

Les interfaces et les descriptions fonctionnelles de la boîte sont les énoncés lorsque la boîte est construite.

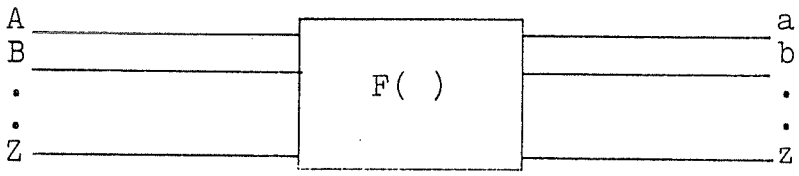


Figure 9.2.2-1 Boîte Noire.

Toute implantation de la boîte est acceptable si elle rencontre les énoncés des interfaces et des propriétés fonctionnelles qui sont définies dans les spécifications.

9.2.3 Test de boîte noire.

La personne responsable des tests regarde le programme comme une boîte noire. Les tests de boîte noire s'apparentent aux tests d'entrées et de sorties ou de domaines d'intrants. Le vérificateur ne se préoccupe pas du comportement ou de la structure du programme, il veut plutôt trouver des circonstances dans lesquelles le programme n'agira pas selon les spécifications.

Les tests exhaustifs de boîte noire sont aussi impossibles. Myers (MYER79) prends l'exemple des tests d'un compilateur COBOL. Pour bien tester le compilateur, il faudrait choisir les ensembles de tests suivants:

1)des cas de tests représentant tous les programmes COBOL corrects ou sans erreurs.

2)des cas de tests représentant tous les programmes COBOL incorrects ou avec erreurs.

3)En plus, si les logiciels COBOL utilisent une "mémoire", les actions présentes dépendent de ce qui s'est passé en mémoire auparavant. Ex: Systèmes d'exploitation, système de base de données, systèmes de réservations aériennes, etc...Dans ce cas, d'après Myers, il faut essayer toutes les séquences possibles de transactions.

9.2.4 Choix des données de tests selon les techniques de tests structurelles.

Dès que le logiciel est représenté dans un langage formel, il est possible d'analyser les structures du logiciel et de choisir les données de tests qui couvriront un programme suivant une métrique donnée. La métrique peut en être une de couverture ou de complexité et elle mesure la plénitude des tests.

La couverture représente les énoncés du logiciel, les chemins du logiciel, les segments comme dans Miller (MILL77), c'est-à-dire les chemins d'un prédicat à un autre, ou les branchements des structures de contrôles qui sont exécutés par les tests.

La métrique de complexité est rarement utilisée pour les tests. Une mesure de complexité bien connu est la métrique de McCabe (McCA76). McCabe propose une métrique de complexité cyclomatique. La complexité cyclomatique est calculée à partir du graphique des décisions dans le

logiciel. Une technique pour calculer cette métrique est de compter le nombre de régions fermées dans l'organigramme décisionnel du logiciel. La complexité actuelle est la complexité effectivement traversée par les données de tests. La complexité actuelle est toujours inférieure à la complexité cyclomatique. D'après Maitland (MAIT80), une stratégie de tests serait de rapprochée la complexité actuelle le plus prêt possible de la complexité cyclomatique.

Même avec l'utilisation d'une métrique de couverture ou de complexité, il n'y a aucune garantie que le logiciel testé est correct .

9.2.5 Test de boîte blanche.

Dans cette technique de la boîte blanche, la documentation et le programme source sont disponibles.

L'objectif d'une analyse de boîte blanche est d'inférer les données de tests appropriées à partir des structures de contrôles et des structures de données du logiciel. Pour trouver les erreurs en testant, il est nécessaire de comprendre les caractéristiques du langage de programmation afin d'identifier les structures du langage qui entraînent souvent des erreurs.

Les tests de boîte blanche ou de la logique du programme sont faits en examinant la structure interne d'un programme. Les données de tests sont trouvées en étudiant la logique du programme.

9.3 Comparaison de la preuve de l'exactitude et les tests.

D'après Berg (BERG82) les techniques mathématiques de validation sont des méthodes formelles en vérification pour argumenter qu'un programme donné est correct. Cette argumentation requière une définition rigoureuse de "correct". La définition de " correct " est l'exactitude du programme par rapport aux spécifications.

Berg dit que les stratégies de tests sont limitées de façon inhérente, parce que le nombre de tests requis pour tester un programme est très grand. Par conséquent, un sous-ensemble de tests est une condition nécessaire , mais non suffisante pour assurer qu'un programme soit correct.

Les spécifications sont plus abstraites que les besoins qu'elles représentent. La réalisation décrit le comportement du logiciel en terme de procédures.

Pour argumenter sur un logiciel, il faut un modèle mathématique formel et rigoureux, afin de représenter ses différents éléments. L'abstraction du modèle mathématique se concentre sur les aspects importants du logiciel et ne porte aucune attention sur les aspects jugés non importants. Par exemple, les périphériques ne sont pas compris dans ces modèles abstraits.

Il y a deux autres limitations des preuves d'exactitude de programme. Il a été montré par Clarke (CLAR77a) qu'il existe des constructions de programme dont la sémantique ne peut pas être exprimée par les règles d'inférence dans un système déductif. Ceci veut dire que les preuves de l'exactitude ne peuvent qu'être faites pour des sous-ensembles des langages en question.

Gode (GODE31) a démontré que tout système déductif admet des théorêmes qui sont vrais, mais dont la preuve ne peut être faite. Ceci implique que la conception de programmes basée sur des systèmes déductifs peut admettre des programmes qui sont corrects, mais dont la

preuve de l'exactitude ne peut être faite.

Les preuves de l'exactitude d'un logiciel ne sont pas complètement fiables parce qu'elles sont basées sur les hypothèses suivantes:

1) Il existe des axiomes qui définissent complètement l'environnement du programme. (Langage, système d'exploitation, et équipement comme les processeurs, etc...)

2) La consistance des processeurs avec les axiomes est prouvée.

3) Le programme est complètement et formellement implanté de tel sorte qu'une preuve peut être effectuée et vérifiée mécaniquement.

4) Les spécifications sont correctes dans le sens que si chaque programme dans le système est correct par rapport à ses spécifications, alors le système complet aura la performance désirée. Cette

hypothèse dilue les problèmes de couplage.

Gerhart (GERH76) donne une douzaine d'exemples où des programmes prouvés par la preuve de l'exactitude contenaient des erreurs soit dans les spécifications données pour les programmes, soit dans la construction des programmes ou dans les preuves elles-mêmes. Trois causes communes d'erreurs dans les programmes "prouvés" sont:

1) L'informalité dans l'utilisation de la preuve mathématique.

2) L'informalité dans la preuve de la fin du programme.

3) L'inattention aux autres méthodes de vérification comme la revision et les tests.

D'après Luckham et Suzuki (LUCK79a), des problèmes dans la preuve de l'exactitude surviennent aussi dans les structures de données complexes, spécialement dans les

variables pointeurs, qui ont le potentiel pour générer des voies d'accès multiples aux éléments de données, compliquant ainsi le procédé de la preuve.

Une preuve est limitée à des conclusions dans un environnement postulé.

L'avantage des tests, par rapport aux preuves de l'exactitude, est de produire de l'information précise sur le comportement réel d'un programme dans son environnement réel.

10 Trois caractéristiques des programmes.

10.1 Caractéristique I: Multiciplicité des chemins à l'intérieur d'un programme.

Un programme est indiqué à la figure 10.1-1. Les branchements conditionnels 1 et 2 peuvent prendre respectivement les chemins 1-2-3 ou 4 et les chemins 5-6-7 ou 8. La boucle répétitive se fait 10 fois.

Le nombre de chemins possibles n est:

n = (4x4)x(4x4).....(4x4)

<---------10 fois---------->

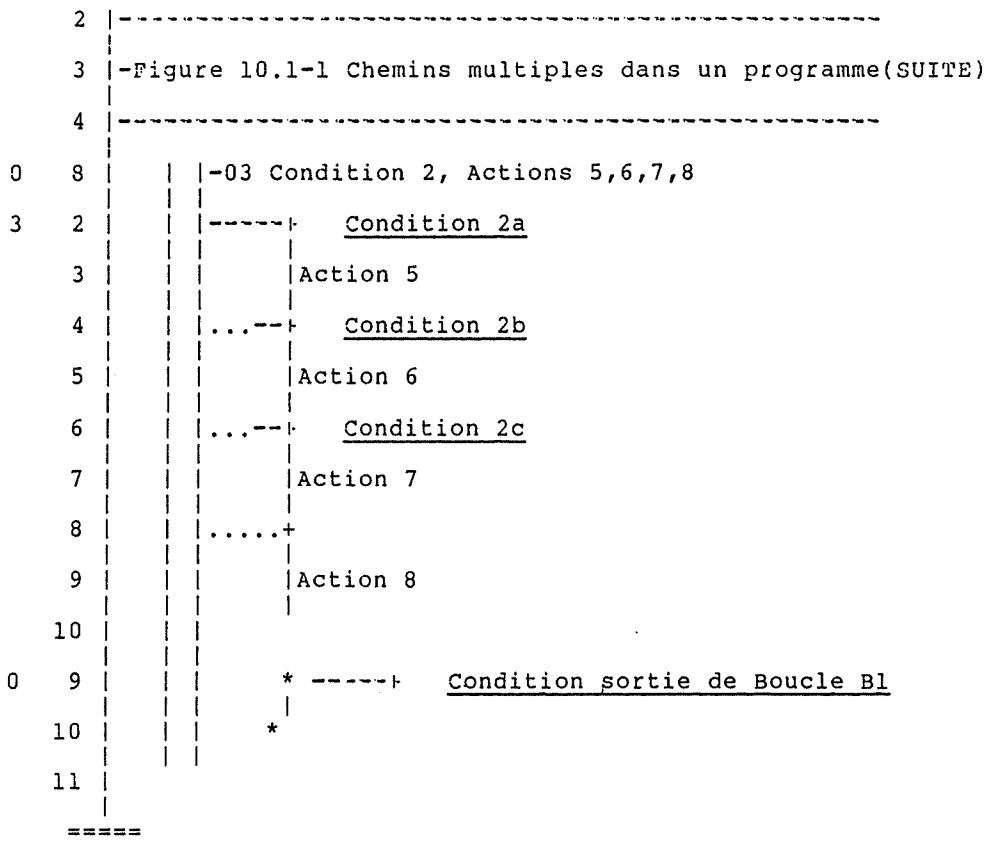
n = 16**10

Toute stratégie qui viserait à suivre les chemins possibles est impraticable sans modèle théorique pour

Pour education seulement
00 - POLY - 1 Pierre Veronneau
SCHEMACODE V1.3

Fichier	Programme principal	Langage cible	Date de creation	Date de derniere : utilisation	codification	Heure
CHEMINS	CHEMINS	C	1986-10-06	1986-10-06	1986-10-06	09:55:1

```
     R   0
     =====
 0   1 |-CHEMINS  -Pour education seulement-SCHEMACODE V1.3
       |
     2 |-------------------------------------------------
       |
     3 |-Figure 10.1-1 Chemins multiples dans un programme
       |
     4 |-------------------------------------------------
       |
     5 |-01 Début du programme
       |
 1   2 |actions de début de programme
       |
 0   6 |---+-+
       |   | |
     7 |   | |-02 Condition 1, Actions 1,2,3,4
       |   | |
 2   2 |   | |-----+   Condition 1a
       |   | |     |
     3 |   | |     |Action 1
       |   | |     |
     4 |   | |...--+   Condition 1b
       |   | |     |
     5 |   | |     |Action 2
       |   | |     |
     6 |   | |...--+   Condition 1c
       |   | |     |
     7 |   | |     |Action 3
       |   | |     |
     8 |   | |.....+
       |   | |     |
     9 |   | |     |Action 4
       |   | |     |
    10 |   | |
       |   | |
```


-Figure 10.1-1 Chemins multiples dans un programme(SUITE)
-03 Condition 2, Actions 5,6,7,8
Condition 2a
Action 5
Condition 2b
Action 6
Condition 2c
Action 7
Action 8
Condition sortie de Boucle B1

réduire le nombre de possibilités.

Généralement les logiciels sont beaucoup plus compliqués que l'exemple précédent et le nombre de chemins est à toute fin pratique infini.

10.2 Caractéristique II: Multiplicité des programmes possibles.

Un problème tout aussi complexe est la multiplicité des programmes possibles pour un énoncé de projet donné.

Comme exemple, un programme a 7 conditions indépendantes, chacune des conditions ayant deux branchements. Chaque branchement mène soit à une action ou à une autre des 7 conditions restantes. 8 actions possibles sont associées à ce logiciel. Un exemple de programme possible peut être représenté par la figure 10.2-1.

Le programme montré comme exemple n'est qu'un parmi 19x10**26 programmes possibles. Pourtant, le nombre de conditions (7) et d'actions (8) est petit.

Humby (HUMB73) obtient le nombre de programmes possibles par induction*(16).

Le tableau 10.2-1 indique la progression rapide du

16. (HUMB73) Humby, E. Programs from decision tables , p.34

Fichier	Programme principal	Langage cible	Date de creation	Date de derniere utilisation	: codification	Heure
POSSIBIL	POSSIBIL	C	1985-12-10	1985-12-10	1985-12-10	02:00:00

```
    R  0
    ▀▀▀▀▀
 0  1 ▌-POSSIBIL -UQAM-SCHEMACODE 1985 (PC-DOS EDUC)
    2 ▌-**********************************************
    3 ▌-PROGRAMME POSSIBILITES. UNE VERSION POSSIBLE PARMI
    4 ▌-19*10**26 PROGRAMMES POSSIBLES.
    5 ▌-**********************************************
    6 ▌(main()
    7 ▌{
    8 ▌-predicats conditionnels representes ici par c[1],...,c[7]
    9 ▌int c[7],termine;
   10 ▌------c[1]>0
   11 ▌      -01 condition numero 2.
 1  2 ▌      ------c[2]>0
    3 ▌           -03 condition numero 3.
 3  2 ▌           ------c[3]>0
    3 ▌                -05 condition numero 4.
 5  2 ▌                ------c[4]>0
    3 ▌                     -07 condition numero 5.
 7  2 ▌                     ------c[5]>0
    3 ▌                          -09 condition numero 6.
 9  2 ▌                          ------c[6]>0
    3 ▌                               [11]condition numero 7.          *** 11
    4 ▌                          ''''''
    5 ▌                               [12]action numero 6.             *** 12
    6 ▌
 7  4 ▌                     ''''''
    5 ▌                          -10 action numero 5.
10  2 ▌                          termine=0;
 7  6 ▌
 5  4 ▌                ''''''
    5 ▌                     -08 action numero 4.
 8  2 ▌                     termine=0;
 5  6 ▌
 3  4 ▌           ''''''
    5 ▌                -06 action numero 3.
 6  2 ▌                termine=0;
 3  6 ▌
 1  4 ▌      ''''''
    5 ▌           -04 action numero 2.
 4  2 ▌           termine=0;
 1  6 ▌
 0 12 ▌''''''
   13 ▌      -02 action numero 1
 2  2 ▌      termine=0;
 0 14 ▌
    ▀▀▀▀▀
```

Figure 10.2-1

Fichier	Programme principal	Langage cible	Date de creation	Date de derniere : utilisation	codification	Heure
POSSIBIL	POSSIBIL	C	1985-12-10	1985-12-10	1985-12-10	02:08:5

```
   R12
   ■■■■
12 1 ■-action numero 6.
   2 determine=0;
   ■■■■
   R 9
```

UQAM
Pierre Vermeau
SCHEMACODE 1985 (PC-DOS EDUC) #94

Fichier	Programme principal	Langage cible	Date de creation	Date de derniere : utilisation	codification	Heure
POSSIBIL	POSSIBIL	C	1985-12-10	1985-12-10	1985-12-10	02:08:20

```
    R11
    ▄▄▄▄▄
11  1 ▌-condition numero 7.
    2 ▌▄▄▄▄▄▄c[7]>0
    3 ▌      -13 action numero 8.
13  2 ▌      termine=0;
11  4 ▌▄▄▄▄▄▄
    5 ▌      -14 action numero 7.
14  2 ▌      termine=0;
11  6 ▌
    ▀▀▀▀▀
    R 9
```

nombre de programmes possibles avec le nombre de conditions indépendantes.

Tableau 10.2-1. Nombre de programmes possibles en fonction du nombre de conditions indépendantes.

Nombre de conditions indépendantes	Nombre de programmes possibles
2	2
3	12
4	576
5	1.658.880
6	16.513x10**9
7	19x10**26

Un programme trouvé pour un énoncé de projet donné est un programme parmi une multitude de programmes possibles.

10.3 Caractéristique III: Domaine d'entrée infini.

Le domaine des données à l'entrée d'un logiciel est souvent infini. Un exemple est l'ensemble des nombres réels acceptables pour un programme d'actualisation financier.

De même, une chaîne de caractères alphanumériques d'une longueur de 18 caractères fournit un très grand éventail de possibilités de données à l'entrée.

11. Théorie et Méthode de "Tests des Etats Logiques " ou TEL.

11.1 Introduction.

TEL est une méthode de test proposée où un programme est séparé en états logiques testables. Ces états sont obtenus en regroupant tous les énoncés d'actions ayant le même prédicat conditionnel global au niveau du logiciel.

Lorsque les états logiques sont des espaces vectoriels, il est alors possible de prouver l'exactitude du programme en utilisant un nombre fini de données de tests. Un exemple illustre la méthode proposée.

Le choix de la stratégie de test est demeuré une matière de jugement professionnel. Jusqu'à maintenant, il n'y a pas de critère de sélection des données de tests applicable en pratique qui peut prouver qu'un programme n'a pas d'erreur ou est correct. Pour certifier qu'un

programme est correct, des tests doivent être fait sur tous les chemins du programme et sur tous les domaines à l'entrée. Comme il y a un très grand nombre de chemins possibles et que les domaines à l'entrée sont très grands, voire même infini, cette tâche est impossible en pratique.

La méthode de test TEL proposée intègre la théorie des espaces vectoriels à la théorie moderne des tests. Le programme à être testé est séparé en "Etats Logiques du Logiciel", suivant les prédicats conditionnels dans le programme. Les états logiques du logiciel peuvent ensuite être classés en "Etats Logiques Testables".

Le nombre total d'états logiques testables est fini dans un programme donné et chacun des ces états logiques peut être testé indépendamment des autres. Quand un état logique testable est un espace vectoriel, il est possible, connaissant la théorie des espaces vectoriels, de prouver que l'état logique testable en question n'a pas d'erreur en utilisant un nombre fini de données de tests.

Un exemple illustre la méthode TEL proposée. Le nombre total de tests pour prouver l'exactitude d'un programme est fonction du nombre d'états logiques testables et du nombre de données de tests requises pour chacun de ces états.

11.2 Définition d'un "Etat Logique du Logiciel".

Si les énoncés de conditions et les prédicats conditionnels sont exclus des énoncés, les énoncés d'actions demeurent de la forme: assignation, appel de fonction, appel de module, etc...

Ces énoncés d'action actifs, pris dans leur ensemble, ont des variables d'entrées, des variables de sorties et des variables transitoires. Toutes ces variables peuvent être représentées par les composantes d'un n-tuple. Quand le programme est exécuté, les variables prennent leurs valeurs respectives et génèrent un ensemble de n-tuples ayant comme composants les valeurs des variables précédentes.

Par exemple, les deux énoncés qui suivent seront supposés actifs, où BRUT est la variable à l'entrée, PENSION est la variable transitoire interne au programme et IMPOT est la variable de sortie.

```
PENSION := BRUT * 0.03
IMPOT := (BRUT - 5000 - PENSION) * 0.25
```

Les deux énoncés peuvent être représentés par le n-tuple, qui est en l'occurence un triplet:

(BRUT, PENSION, IMPOT)

Quand le programme est exécuté avec différentes valeurs de BRUT à l'entrée, les n-tuples suivants seront générés:

```
(15000, 450, 2387.5)
(17000, 510, 2872.5)
   :     :      :
```

Le concept des n-tuples qui évoluent en réponse aux différentes valeur des variables à l'entrée est un concept dynamique. Ce concept fut discuté et définit par Zeil (ZEIL83) dans son modèle "d'environnement", un programme étant une séquence d'environnements, lorsque les différents chemins du programme sont parcourus.

Par définition, dans le modèle TEL, un prédicat conditionnel global est attaché à chaque énoncé d'action

qui est la condition Booléenne le rendant "actif".

Définition. : Un "Etat Logique du Logiciel" est l'ensemble de tous les énoncés d'action qui ont le même prédicat conditionnel global.

En ajoutant des conditions à l'exemple précédent, le programme suivant est obtenu:

```
if (EMPLOYE == CELIBATAIRE)
     PENSION := BRUT * 0.03
     IMPOT := (BRUT - 5000 - PENSION) * 0.25
else if (EMPLOYE == marie)
     PENSION := BRUT * 0.04
     IMPOT := (BRUT - 7000 - PENSION) * 0.25
```

Dans l'exemple, les prédicats conditionnels globaux sont trouvés, puis les énoncés d'action ayant les mêmes prédicats conditionnels globaux sont regroupés pour donner:

----------Etat Logique no. 1.----------

Prédicat conditionnel global.

EMPLOYE = CELIBATAIRE

Enoncés actions.

```
PENSION := BRUT * 0.03
IMPOT := (BRUT - 5000 - PENSION) * 0.25
```

----------Etat Logique no. 2.----------

Prédicat conditionnel global.

```
(not(EMPLOYE = CELIBATAIRE))
    and (EMPLOYE == marie)
```

Enoncés actions.

```
PENSION := BRUT * 0.04
IMPOT := (BRUT - 7000 - PENSION) * 0.25
```

Dans l'exemple, il y a deux états logiques du logiciel. Un programme peut être représenté par un nombre fini d'états logiques du logiciel. En un instant donné, un seul état logique est actif.

11.3 Etat Logique du Logiciel Testable.

Pendant son exécution, les n-tuples d'un programme se promènent à l'intérieur d'un état logique du logiciel ou ils sautent d'un état logique du logiciel à un autre état logique différent du premier. Le programme ne peut avoir aucun autre comportement à cause de la définition même des états logiques du logiciel qui comprend tous les énoncés d'actions du programme.

Si chacun des états logiques du logiciel est testé ou prouvé séparemment, alors le programme entier sera testé ou prouvé.

Dans l'exemple pré-cité, des tests sur les états logiques no. 1 et no. 2 sont équivalents à des tests sur tout le programme. Par conséquent, avec la définition du prédicat conditionnel global rattaché à chaque énoncé d'action, les "Etats Logiques du Logiciel Testable" sont définis.

Le problème de tester le programme entier est devenu un problème de tester chacun des états logiques du logiciel testable individuellement.

11.4 Comportement associé à un Espace Vectoriel.

Les n-tuples qui représentent les états logiques du logiciel génèrent un espace de dimension m, m étant plus petit ou égal à n. De la théorie des espaces vectoriels, si les n-tuples satisfont les axiomes des vecteurs, alors l'espace en question sera un espace vectoriel de dimension m sur les variables du n-tuple, où m est le nombre de variables indépendantes.

Dans l'exemple précédent, les deux états logiques du logiciel de 3-tuples ou triplets sont en réalité deux espaces vectoriels de dimension 1. La seule variable indépendante est la variable "BRUT".

11.5 Critère de Sélection des Données de Test.

Dans un espace vectoriel de dimension m, le critère de sélection des données de test est déduit de la théorie classique des espaces vectoriels.

> Critère: L'ensemble de m+1 n-tuples qui sont linéairement indépendants.

L'ensemble de m+1 données de tests qui sont des n-tuples linéairement indépendants est suffisant pour prouver que l'espace vectoriel est le bon espace vectoriel, c'est-à-dire qu'il est "correct". L'espace vectoriel programme est correct s'il est le même que l'espace vectoriel prévu. De façon équivalente, l'état logique du logiciel est le bon s'il est le même que l'état logique prévu.

Si l'espace vectoriel programme est correct, alors il n'y a aucune erreur dans le programme. Comme l'espace vectoriel représente fidèlement l'état logique du

logiciel alors l'état logique n'a pas d'erreur non plus.

Dans l'exemple, l'espace vectoriel est de dimension 1 pour les deux états logiques du logiciel. Des données de tests constituées par deux triplets ou 3-tuples linéairement indépendants, c'est-à-dire n'étant pas sur le même point de la ligne, sont suffisants pour prouver l'exactitude de chacun des états logiques du logiciel.

Pour illustrer, les données de tests suivants seront choisis dans l'exemple de cette section:

Résultats anticipés de l'oracle de tests.
Etat Logique no. 1.

BRUT=15000,IMPOT=2387.5
BRUT=17000,IMPOT=2872.5

Etat Logique no. 2.

BRUT=15000,IMPOT=1850
BRUT=17000,IMPOT=2330

En exécutant le programme, les n-tuples suivants sont générés:

Données de tests choisies pour le programme.

Etat Logique no. 1.

(15000, 450, 2387.5)
(17000, 510, 2872.5)

Etat Logique no. 2.

(15000, 600, 1850)
(17000, 680, 2330)

La conclusion qui est montrée par l'exemple utilisant la méthode TEL et les données de test choisies par le critère de sélection est que le programme est correct , ce qui prouve sont exactitude . Il ne peut y avoir d'erreur dans le programme.

11.6 Conclusion.

Il est possible de visualiser un programme en utilisant une représentation différente de celle d'une séquence d'énoncés ou de chemins à travers un programme. Le modèle de n-tuples dynamiques et le regroupement des énoncés d'actions en Etats Logiques du Logiciel est utilisé pour obtenir un nombre fini d'Etats Logiques du Logiciel Testable qui sont indépendants l'un de l'autre.

Si les Etats Logiques Testables sont des espaces vectoriels, il est alors possible de prouver qu'il n'y a pas d'erreur avec un nombre fini de tests pour chaque Etat Logique Testable.

La méthode TEL est indépendante de la méthodologie de design pour obtenir le logiciel final. La méthode est aussi indépendante du langage source utilisé.

La méthode TEL est indépendante des types d'erreurs

pouvant être présents dans le programme final. La méthode TEL affirme l'exactitude ou l'erreur , c'est-à-dire l'absence ou la présence d'erreurs en connaissant les résultats prévus des données de tests choisies à l'aide du Critère de Sélection des Données de Test.

La méthode TEL repose sur la théorie des espaces vectoriels. Son applicabilité dépend de la modélisation du programme source en Etats Logiques du Logiciel et dans le comportement vectoriel de ces états logiques.

Chacun des états logiques testables peut être testé indépendamment des autres états logiques du logiciel. La méthode TEL peut être combinée avec d'autres méthodes de vérification pour des Etats Logiques du Logiciel qui n'ont pas de comportement d'espace vectoriel.

La méthode TEL est actuellement appliquée à un logiciel complet de gestion de la paie et les résultats de recherche sur le potentiel d'applications de la méthode proposée seront divulgués sous peu.

Annexes.

Annexe I.

Scalaires et Approche axiomatique.

Approche axiomatique.

Dans la recherche présentée, l'approche axiomatique ou par postulats est utilisée. Lors de l'application de systèmes qui ont des caractéristiques communes, il est souvent avantageux de simuler le système réel par un système abstrait ayant des axiomes représentant les caractéristiques communes.

Tout théorème qui est déduit des axiomes ou des postulats, s'applique à tous les systèmes ayant comme modèle les dits axiomes ou postulats, alors que sans l'approche axiomatique, il faudrait reprouver les théorème pour chaque système réel.

Scalaires.

Un ensemble F de scalaires {a} est composé d'éléments appelés scalaires ou nombres, dénotés ici par a,b,c..., et deux opérations, l'addition et la multiplication. Les scalaires répondent aux axiomes suivant:

	Addition.	Multiplication.
1) Fermeture	a+b est dans F.	ab est dans F.
2) Associativité	(a+b)+c=a+(b+c)	(ab)c=a (bc)
3) Commutativité	a+b=b+a	ab=ba
4) Existance d'un scalaire 0.	a+0=0+a=a	
5) Existance d'un scalaire 1.		a*1=1*a=a
6) Existence d'un négatif.	a+(-a)=(-a)+a=0	
7) Existence du réciproque.		$a*a^{-1}=a^{-1}*a=1$
8) Distributivité à gauche.	a(b+c)=ab+ac	
9) Distributivité à droite.	(a+b)c=ac+bc	

Des exemples d'ensembles de scalaires sont l'ensemble des nombres réels, l'ensemble des nombres complexes, l'ensemble des nombres rationnels, l'ensemble des

fonctions rationnelles $f(x)=p(x)/q(x)$, l'ensemble des nombres entiers modulo 5,...

Annexe II

Espace vectoriel.

Espace vectoriel.

L'espace cartésien est un exemple d'un espace vectoriel. Un espace vectoriel V est constitué d'éléments appelés vecteurs, que nous dénoterons ici par A,B,C,... pour différencier de la notation des scalaires, ayant deux opérations, l'addition des vecteurs et la multiplication d'un vecteur par un scalaire. Les vecteurs satisfont aux axiomes suivant:

Axiomes pour l'addition des vecteurs.

1)	Fermeture	A+B=C, C étant un vecteur dans V.
2)	Associativité	(A+B)+C=A+(B+C)
3)	Vecteur 0	A+0=0+A=A pour tout A
4)	Vecteur négatif	A+D=D+A=0, D est écris -A.
5)	Commutativité	A+B=B+A

Axiomes pour la multiplication d'un vecteur par un scalaire.

1)	Fermeture	a*C est dans V
2)	Associativité	c(A+B)=cA+cB
3)	Associativité	(C+D)a=Ca+Da
4)	Associativité	(CD)a=C(Da)
5)	Unitaire	1 * A=A

Des exemples d'espaces vectoriels sont les espaces cartésiens de dimension 0,...,n, les polynômes, les solutions des équations homogènes linéaires,...

BIBLIOGRAPHIE.

ACRE80
Acree, A. "On Mutation, " Dissertation Ph.D., Dep. Information et Computer Science ,Georgie Institut de technologie, Atlanta, Juin, 1980.

ADRI82
Adrion, W.R., Branstad, M.A., et Cheniavsky, J.C. "Validation, Verification, and Testing of Computer Software, " ACM, Computing Surveys, Vol. 14, No.2, (Juin 1982).

ALFO77
Alford, M.W. "A requirement engineering methodology for real-time processing requirements," IEEE Trans. softw. eng. SE-2, 1(1977).

ALLE74
Allen, F.E. "Interprocedural data flow analysis, " dans Proc. IFIP congress 1974, North-Holland, Amsterdam, 1974.

ALLE76
Allen, F.E., and Cocke, J. "A program data flow procedure, " Commun. ACM 19, 3, Mars 1976.

AMBL78
Ambler, A.L., Good, D.I., Browne, J.C., Burger, F.W., Cohen, R.M., Hoch, C.G., et Wells, R.W. "Gypsy: A language for specification and implementation of verifiable programs, " dans Proc. Conf. Language Design for reliable software, D.B. Wortman. (Ed.), ACM, New York.

ANDR81
Andrews, D.M., et Benson, J.P. "An automated program testing methodology, and its implementation, " dans Proc. 5th Int. Conf. Software Engineering (San Diego, Calif., Mars 9-12), IEEE Computer Society Press, Silver Spring, Maryland, 1981.

APT81
APT, K.R. "Ten years of Hoarés logic: A survey-Part I, " Trans. Program. Lang. Syst. 3, 4 (Oct. 1981).

BEIZ83
Beizer, B Software Testing Techniques , Van Nostrand, 1983.

BEIZ84
Beizer, B Software System Testing and Quality Assurance , Van Nostrand, 1984.

BELL77
Bell, T.E., Bixler, D.C., et Dyer, M.E. "An extendable approach to computer-aided software requirements engineering, " IEEE Trans. Softw. Eng. SE-3, 1(1977).

BERG82
Berg, H.K., Boebert, W.E., Franta, W.R. et Moher, T.G. Formal Methods of Program Verification and Specification , Prentice Hall, Inc. Englewood Cliffs, N.J. 1982.

BOEH75
Boehm, B.W., McClean, R.K. et Urfrig, D.B. "Some experience with automated aids to the design of large-scale reliable software, " IEEE trans. on Softw. Eng., Mars 1975.

BOEH77
Boehm, B.W. "Seven basic principles of software engineering," dans Software Engineering Techniques , Infotech State of the Art Report, Infotech, Londres, 1977.

BOEH78
Boehm, B.W., Brown, J.R., Kaspar, H., Lipow, M., McLeod, G.J., et Merrit, M.J. Characteristics of software quality , North-Holland, New York, 1978.

BOYE75
Boyer, R.S., Elpas, B., et Levitt, K.N. "SELECT-A formal system for testing and debugging programs by symbolic execution, " dans Proc. 1975 Int. Conf. Reliable

Software (Los Angèles, Avril), 1975.

BRAN80
Branstad, M.A., Cherniavsky, J.C., et Adrion, W.R. "Validation, verification and testing for the in dividual programmer, " Computer 13, 12(Dec. 1980).

BROW73
Brown, J.R. et al. "Automated software quality as surance, " dans W. Hetzel (Ed.), Program Test Methods , Prentice-Hall, Englewood Cliffs, N.J. 1973.

BUCK79
Buckley, F. "A standard for software quality assurance plans, " Computer 12, 8(Août 1979).

BUDD78b
Budd, T.A., et Lipton, R.J. "Mutation analysis of decision table programs, " dans Proc. 1978 Conf. Information Science and Systems, Johns Hopkins Univ. de Baltimore, Md..

CAIN75
Caine, S.H., et Gordon, E.K. "PDL-Baltimore, A tool for software design, " dans Proc. National Computer Conf., vol. 44, AFIPS Press, Arlington, Va., 1975.

CARP75
Carpenter, L.C., et Tripp, L.L. "Software design validation tool, " dans Proc. 1975 Int. Conf. Reliable Software (Avril 1975).

CHAP79
Chapin, N. "A measure of software complexity, " dans Proc. AFIPS National Computer Conf., vol. 48, AFIPS Press, Arlington, Va., 1979.

HER79a
Cherniavsky, J.C. "On finding test data sets for loop free programs, " Inform. Process. lett. 8, 2 (1979).

CLAR77
Clarke, A. "A system to generate test data and symbolically execute programs, "IEEE Trans. Softw. Eng. SE-2, 3(Sept. 1977).

CLAR77a
Clarke, E.M.Jr. "Programming Language constructs for which it is impossible to obtain good Hoare like axiom systems, " Proc. 4th Symposium on Principles of Programming Languages, 1977.

CONS78
Constable, R.L., et O'Donnel, M.J. A Programming Logic , Winthrop, Cambridge, Mass., 1978.

DEMI78
Demillo, R.A., Lipton, R.J., et Sayward, F.G. "Hints on test data selection: Help for the practicing programmer, " computer, vol. 11, Avril 1978.

DEMI79
Demille, R.A., Lipton, R.J., et Perlis, A.J. "Social processes and the proofs of theorems and programs, " Commun. ACM 225 (Mai 1979).

DIJK72
Dijkstra, E.W. "Notes on Structured Programming," in O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare, Structured Programming , Academic Press, New York, 1972.

DIJK78
Dijkstra, E.W. "On a political pamphlet from the Middle Ages (regarding the POPL paper of R.A. DeMillo, R.J. Lipton and A.J. Perlis), " Softw. Eng. Notes 3, 2(Avril 1978).

DUNN83
Dunn, R. Software Defect Removal , McGraw-Hill, 1983

DURA81
Duran, J.W., et Wiorkowski, J.J. "Capture-recapture sampling for estimating software error con tent, " IEEE Trans. Softw. Eng. SE-7 (Jan. 1981).

ERSH76
Ershov, A.P. "Axiomatics for memory allocation, " Acta Informatica, Vol. 6, No. 1, 1976.

FAGA76
Fagan, M.E. "Design and code inspections to reduce errors in program development, " IBM SYST. J. 15, 3(1976).

FIPS76
FIPS. "Guidelines for documentation of Computer Programs and Automated Data Systems, " FIPS38, Federal Information Processing Standards Publications, U.S. Department of Commerce/National Bureau of Standards, Washington, D.C., 1976.

FLOY67
Floyd, R.W. "Assigning meaning to programs, " dans Proc. Symposia Applied Mathematics, vol.19, American Mathematics Society, Providence, R.I., 1967.

FOSD76
Fosdick, L.D., et Osterweil, L.J. "Data flow analysis in software reliability, " Comput. Surv. (ACM) 8, 3(Sept. 1976).

FUJI77
Fujii, M.S. "Independent verification of highly reliable programs, " Proceedings COMPSAC 77, IEEE.

GERH76
Gerhart, S.L.et Yelowitz, L., "Observation on Faillibility in applications of modern programming Methodologies, " IEEE Trans. on Software Engineering, Vol. 2, No. 3, 1976.

GERH78
Gerhart, S.L. "Program verification in the 1980s: Problems, perspectives, and opportunities, " Rep. ISI/RR-78-71, Information Sciences Institute, Marina del Rey, Calif., Août 1978.

GERH80
Gerhart, S.L., Musser, D.R., Thompson, D.H. Baker, D.A., Bates, R.L., Erickson, R.W., London, R.L., Taylor, D.G., et Wile, D.S. "An overview of AFFIRE; A specification and verification system, " dans Proc. IFIP Congress

1980, North-Holland, Amsterdam.

GLAS81
GLASS, R.L. "Persistent software errors, " IEEE Trans. Softw. Eng. SE-7, 2(Mars 1981).

GODE31
Gode, K, "Uber formal unentscheidbare Seitze der Principia Mathematica and verwandler Systeme I, " Monatsheft fur Mathematik and Physik , vol 38, 1931.

GOOD75
Goodenough, J.B., et Gerhart, S.L. "Toward a theory of test data selection, " IEEE Trans. Softw. Eng. SE-1, 2(Mars 1975).

GOOD77
Goodenough, J.B. et Gerhart, S.L. "Towards a theory of testing: Data selection criteria, " Current trends in Programming Methodology , vol. 2, R.T. Yeh, Ed. Englewood Cliffs, N.J. Prentice-Hall, 1977.

HALS77
Halstead, M.H. Elements of Software Science , Elsevier North-Holland, New York, 1977.

HAMI76
Hamilton, N., et Zeldin, S. "Higher order software. A methodology for defining sogtware, " IEEE Trans. Softw. Eng. SE-2, 1(1976).

HANS73
Hansen, B. "Testing a multiprogramming system, " Software practice and experience, vol. 3, (1973).

DANT76
Hantler, S.L., et King, J.C. "An introduction to proving the correctness of programs, " Comput. Surv. (ACM) 8, 3(Sept. 1976).

HECH72
Heicht, M., et Ullman, J. "Flow-graph reducibility, " Siam. J. Appl. Math. 1(1972).

HORO82
Horowitz, E. et Sahni, S. Fundamentals of Data Structures , Computer Science Press, 1982.

HOWD76
Howden, W.E. "Reliability of the path analysis testing strategy, " IEEE Trans. Softw. Eng. SE-2, 3 (1976).

HOWD76a
Howden, W.E. "Theoretical and empirical studies of program testing," IEEE Trans. Softw. Eng. SE-4, Juillet 1976.

HOWD77
Howden, W.E. "Symbolic testing and the DISSECT symbolic evaluation system, " IEEE Trans. Softw. Eng. SE-3,HH 4(77).

HOWD78
Howden, W.E. "A survey of dynamic analysis methods, " dans E. Miller et W.E. Howden (Eds.), Tutorial: Software Testing and Validation Techniques , IEEE Computer Soc., New York, 1978.

HOWD78a
Howden, W.E. et Eichorst, P. "Proving properties of programs from program traces, " Tutorial: Software testing and validation techniques , Miller, E. et Howden, W.E. Eds., IEEE 1978

HOWD78b
Howden, W.E. "A survey of static analysis methods, " dans E. Miller et W.E. Howden (Eds.), Tutorial: Software Testing and Validation Techniques , IEEE Computer Soc., New York, 1978.

HOWD78c
Howden, W.E., "An evaluation of the effectiveness of symbolic testing, " Software Practice and Experience , vol. 8, 1978.

HOWD80a
Howden, W.E. "Functional program testing, " IEEE Trans.

Softw. Eng. SE-6, 2(1980).

HOWD80b
Howden, W.E. "Applicability of software validation techniques to scientific programs, " Trans. Program. Lang. Syst. 2, 3(Juin 1980).

HOWD81a
Howden, W.E. "Completeness criteria for testing elementary program functions, " dans Proc. 5TH Conf. on Software Engineering (San Diego, Mars 9-12), IEEE Computer Society Press, Silver Spring, Md., 1981.

HOWD81b
Howden, W.E. "Errors in data processing programs and the refinement of current program test methodologies, " Final Rep., NBS Contract NB79BCA0069 National Bureau of Standards, Washington, D.C., Juillet 1981.

HUAN75
Huang, J.C. "An approach to program testing". ACM Computing Surveys, 7, Sept. 1975.

HUMB73
Humby, E. Programs from Decision Tables , MacDonald and co., London, 1973.

IEEE79
IEEE, Draft Test Documentation Standard, IEEE Computer Society Technical Committee on Software Engineering, Subcommittee on Software Standards, New York, 1979.

IEEE83
IEEE, IEEE Standard Glossary of Software Engineering Terminology , IEEE Std. 729-1983, IEEE-CS order no. 729, Los Alamitos, Calif., 1983.

INFO79
INFOTECH, Software Testing , Infotech State of the Art Report, Infotech, Londres, 1979.

JACK75
Jackson, M.A. Principles of Program Design , Academic Press, New York, 1975.

JONE76
Jones, C. "Program quality and programmer productivity, " IBM Tech. Rep., International Business Machines Corp., San Jose, Calif., 1976.

KEIR73
Keirstead, R.E. et Parker, D.B. "On the feasibility of formal certification, " in Program test methods , W. Hetzel, Ed. Englewood-Cliffs, N.J.: Prentice-Hall, 1973.

KING76
King, J.C. "Symbolic execution and program testing, " Commun. ACM 19,7 (Juillet 1976).

LAMB78
Lamb, S.S., Leck, V.G., Peters, L.J., et Smith, G.L. "SAMM: A modeling tool for requirements and design specification, " dans Proc. COMPSAC 78, IEEE Computer Society, New York, 1978.

LEDG78
Ledgard, H.F. et Chmura, L.J. Fortran with Style , Hayden book co., New Jersey, 1978.

LUCK79
Luckham, D., German, S., Von Henke, F., Karp, R., Milne, P., Oppen, D., Polak, W., et Schenlis, W. "Stanford Pascal Verifier user's manual, " AI Memo. CS-79-731, Computer Science Dep., Stanford University, Stanford, Calif., 1979.

LUCK79a
Luckham, D. et Suzuki, N. "Verification of Array, record and pointer operations in Pascal, " ACM TOPLAS, Vol. 1, No. 2, 1979.

LYON74
Lyon, G., et Stillman, R.B. "A Fortran Analyzer, " NBS Tech. Note 849, National Bureau of Standrds, Washington, D.C., 1974.

MAIT80
Maitland, R. "NODAL, " dans NBS SOFTWARE TOOLS DATABASE, R. Houghton and K. Oakley (Eds.), NBSIR, National Bureau of Standards, Washington, D.C., 1980.

McCA76
McCabe, T.J. "A complexity measure, " IEEE Trans. Softw. Eng. SE-2, 4(1976).

McCA77
McCall, J., Richard, P., et Walters, G., Factors in Software Quality , vols. 1-3, NTIS Rep. File Nos. AD-A049-014, 015, 055, 1977.

MILL70
Mills, H.D. "Top down programming in large systems, " dans Debugging Techniques in Large Systems , R.Rustin (Ed.), Prentice-Hall, Englewood Cliffs, N.J., 1970.

MILL72
Mills, H.D. "On statistical validation of computer programs, " IBM Rep. FSC72-6015, Federal Systems Division, IBM, Gaithersburg, Md., 1972.

MILL74
Miller, E.F. Jr., Paige, M.R., Benson, J.P. et Wisehart, W.R. "Structural techniques of program validation, " Digest of papers, COMPCON, 1974, IEEE, Fév. 1974.

MILL75
Miller, E.F. Jr. " RXVP-An automated verification system for FORTRAN, " dans Proc. Workshop 4, Computer Science and Statistics: 8th Ann. Symp. on the Interface (Los Angeles, Calif., Fev.), 1975.

MILL75a
Miller, E.F. Jr. et Melton, R.A. "Automated generation of test case datasets, " Proceedings, 1975 International Conference on Reliable Software, IEEE, 1975.

MILL77
Miller, E.R. Jr. "Program testing: Art meets theory," Computer 10, 7(1977).

MORA78
Moranda, P.B. "Limits to program testing with Random number inputs " Proceedings COMSAC '78, New Yore: IEEE, 1978.

MYER79
Myers, G.J. The Art of Software Testing , Wiley, New York, 1979.

NEUM75
Neumann, P.G., Robinson, L., Levitt, K., Boyer, R.S., et Saxema, A.R. "A provably secure operating system, " SRI Project 2581, SRI International, Menlo, Park, Calif., 1975.

OSTE76
Osterweil, L.J., et Fosdick, L.D. "DAVE-A validation, error detection, and documentation system for FORTRAN programs, " Softw. Pract. Exper. 6(1976).

OSTE80
Osterweil, L.J., "A strategy for effective integration of verification and testing techniques, " Tech. Rep. CU-CS-181-80, Computer Science Dep., Univ. of Colorado, Boulder, 1980.

PAIG77
Paige, M.R. "On partitioning program graphs, " IEEE Trans. Softw. Eng. SE-3, 6(1977), 87.

PANZ78
Panzl, D.J. "Automatic revision of formal test procedures, " dans Proc. 3rd Int. Conf. Software Engineering (Atlanta, Mai 10-12), ACM, New York, 1978.

PANZ78a
Panzl, D.J. "Automatic software test drivers, " Computer, Avril 1978, IEEE.

PARN77
Parnas, D.L. "The use of precise specifications in the development of software, " dans Information Processing 77, B. Gilchrist (Ed.), North-Holland, Amsterdam, 1977.

PETE81
Peters, L.J. Software design: Methods and Techniques , Yourdon Press, 1981.

POOL73
Poole, P.C. "Debugging and Testing, " Advance Course on Software Engineering , F.L. Bauer Ed., New York: Springer-Verlag, 1973.

PRAT77
Pratt, V.R. "Semantic considerations in Floyd-Hoare logic , " dans Proc 17th Annu. IEEE Symp. on the Foundations of Computer Science, IEEE Computer Society Press, Long Beach, Calif., 1976.

PRES82
Pressman, R.S. Software Engineering: A practitioner's approach , McGraw-Hill, 1982.

RADE75
Rader, R.J. Advanced Software Design Techniques , PBI, Petrocelli, New York, Princeton, 1975.

RAMA74
Ramamoorthy, C.V., et Ho, S.F. FORTRAN automated code evaluation system, ERL-M466, Electronics Research Lab., Univ. of California, Berkeley, 1974.

RAMA75
Ramamoorthy, C.V., et Ho, S.F. "Testing large soware evaluation systems, " IEEE Trans. Softw. Eng., Mars 1975.

ROBI79
Robinson, L. The HDM Handbook , vol.I-III, SRI Project 4828, SRI International, Menlo Park, Calif., 1979.

ROBI81
Robillard, P.N. et Plamondon, R. "Harness a computer to write better software, faster " Electronic design, Hayden Publishing Co. Inc., Juillet 1981.

ROBI81a
Robillard, P.N. et Plamondon, R. "An Interactive tool for descriptive, operational and structural documentation ", COMPCON '81, Washington, D.C., Sept. 15-17, 1981.

ROBI85
Robillard, P.N. Le Logiciel: de sa conception à sa maintenance , Gaëtan Morin Editeur, 1985

ROSS77
Ross, D.T., et Schoman, K.E.Jr. "Structured analysis for requirements definition, " IEEE Trans. Softw. Eng. SE-3, 1(1977).

ROUB76
Roubine, O., and Robinson, L. Special Reference Manual, Stanford Research Institute Tech. Rep. CSG-45, Menlo Park, Calif., 1976.

SCHI78
Schick, G.J., et Wolverton, R.W. "An analysis of competing software reliability models, " IEEE Trans. Softw. Eng. SE-4 (Mars, 1978).

SNEE78
Sneed, H., et Kirchoff, K. "Prufstand-A testbed for systematic software components, " dans Proc. Infotech State of The Art Conf. Software Testing, Infotech, Londres, 1978. SRS79

STUC77
Stucki, L.G. "New directions in automated tools for improving software quality, " dans R. Yeh (Ed.), Current Trends in Programming Methodology , vol. II-PROGRAM VALIDATION, Prentice-Hall, Englewood Cliffs, N.J., 1977.

TAUS77
Tausworthe, R.C. Standardized Development of Computer Software , Prentice-Hall, Englewood Cliffs, N.J., 1977.

TEIC77
Teichroew, D., et Hershey, E.A., III "PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing systems, " IEEE Trans. Softw. Eng. SE-3, 1(Jan. 1977).

THRA57
Thrall, R.M. et Tornheim, L. Vector Spaces and Matrices , John Wiley & Sons, Inc. 1957.

VOGE80
Voges, U. Lothar, G et von Mayrhauser, A.A. "SADAT-An automated testing tool," IEEE transactions on software engineering, Vol. SE-6, No. 3, Mai 1980.

WARN72
Warnier, J-D et Flanagan, B.M. Entraînement à la Programmation, Tome I, Construction des Programmes , Les Editions d'Organisation, Paris, 1972.

WARN72a
Warnier, J-D Entraînement à la Programmation, Tome II, Exploitation des Données , Les Editions d'Organisation, Paris, 1972.

WEIN71
Weinberg, G.M. The Psychology of Computer Programming , Van Nostrand-Reinhold, Princeton, N.J., 1971.

WEYU79
Weyuker, E.J. "The applicability of program schema results to programs, " Int. J. Comput. Inform. Sci. , vol. 8, Oct. 1979

WEYU80
Weyuker, E.J. et Ostrand, T.J. "Theories of program testing and the application of revealing subdomains, " IEEE trans. on Software Eng., vol. SE-7, No.3, Mai 1980.

WITH80
White, L.J., et Cohen, E.I. "A domain strategy for computer program testing, " Digest for the Workshop on

Software Testing and Test Documentation (Ft. Lauderdale, Fla.). Aussi dans IEEE Softw. Eng. SE-6 (Mai 1980).

YOUR79
Yourdon, E., et Constantine, L.L. Structured Design , Prentice-Hall, Englewood Cliffs, N.J., 1979.

ZEIL83
Zeil, S. J. "Testing for perturbations of program statements" IEEE Trans. Software Eng., Vol. SE-9, May 1983.

VER
ROE

TI

MI
DE
DE
LO