Etat de l'art et apport relatif des diffrentes techniques de test du logiciel.

L'objectif de ce rapport, et des fiches jointes, est de présenter un état de l'art sur les techniques de *test dynamique de logiciel* et leurs apports relatifs. Ces techniques visent à exécuter le produit logiciel sous test sur un sous-ensemble de ses entrées possibles. A l'exception de la fiche sur la preuve de logiciel, cette étude ne développe pas les techniques de test dites *statiques* qui reposent sur l'examen des sources du produit logiciel sans exécution de son code. Elle ne traite pas non plus des différents documents élaborés pour le (ou participant au) test. Cette étude se limite donc aux *techniques* de test de logiciel et n'aborde pas les aspects liés à l'organisation matérielle et humaine du test. En particulier, cette étude ne discute pas des questions d'indépendance des équipes de test et de développement.

Ce rapport descriptif ainsi que les fiches jointes reposent sur une bibliographie établie d'une part à partir des éléments déjà possédés par les auteurs, et d'autre part à partir d'une interrogation de la base de données INSPEC. De plus, des rapports de visite dans divers services informatiques aéronautiques sont fournis.

Dans ce rapport descriptif, les notions de base nécessaires à la compréhension des fiches jointes sont définies. Il contient en fait une synthèse des techniques détaillées dans les fiches.

Introduction

L'objet du test de logiciel est d'atteindre un niveau de confiance satisfaisant. En ce qui concerne le test dynamique, un des meileurs moyens pour atteindre ce niveau de confiance est de sélectionner des tests ayant le maximum de chances de détecter des défauts éventuels du produit logiciel sous test.

N.B. : atteindre un niveau de confiance satisfaisant ne signifie pas avoir établi formellement la correction du produit logiciel. En effet, si le test ne détecte aucune anomalie, rien ne prouve qu'un test plus élaboré n'en découvrira pas. Il faut en particulier éviter une perversion de la stratégie de test qui consisterait à choisir des tests uniquement pour illustrer le "bon fonctionnement" du produit logiciel. De plus, pour des parties cruciales de certains logiciels critiques, il peut être judicieux de suivre des méthodes de preuve en complément des méthodes de tests.

Il existe deux classes de méthodes de test de logiciel :

- celles qui reposent sur l'examen et l'inspection des sources d'un logiciel sans exécution de son code : les méthodes de *test statique (static testing) ;*

- celles qui reposent sur la vérification des résultats obtenus par exécution du logiciel sur un sous-ensemble fini des entrées possibles : les méthodes de *test dynamique (dynamic testing).*

L'*exécution symbolique (symbolic evaluation)* se trouve à la frontière de ces deux méthodes. Cette technique permet de soumettre des tests plus élaborés que le test dynamique, par exemple en soumettant des tests "génériques" avec variables. Ajoutons une classe de méthodes plus "futuristes" :

- celles qui utilisent des *preuves partielles* ; elles peuvent être vues comme, ou participer à, des méthodes de tests. Ceci est en particulier vrai dans la mesure où l'échec d'une preuve peut révéler des anomalies logicielles.

Il est important de noter que puisque le test n'est qu'un outil de validation partielle, il n'est pas question de dégager une préférence pour une méthode de test plutôt qu'une autre. Toutes ces méthodes de test sont complémentaires : chacune couvre des types d'anomalies spécifiques. De plus, certaines méthodes sont plus appropriées lors de certaines phases de test (test unitaire, test d'intégration, test de validation, voir section 2.2). Malgré de nombreuses recherches sur le test de logiciel, donnant parfois lieu à des formalismes très élaborés, l'activité de test reste fortement dépendante de l'expertise des équipes de conception et développement impliquées. On pourrait espérer pour cette raison que le test de logiciel soit un domaine pour lequel les systèmes experts peuvent apporter une aide importante dans les différentes phase du test de logiciel. En fait, il n'en est rien, comme le démontre [ZTV 86]. En effet, le type d'expertise requis pour le test de logiciel est encore trop complexe, et trop dépendant de la nature des produits logiciels développés, pour être structuré de manière compatible avec le type de logique utilisé dans les systèmes experts.

1 Le test "statique"

Les méthodes de test statique sont peu coûteuses et s'avèrent très efficaces ([Mye 78], [Ban 86]). Elles permettent en particulier de tester la conformité des sources par rapport à une norme ou un standard d'écriture des logiciels ou même de déceler certaines aberrations (variables ou procédures utilisées mais non définies, variables ou procédures (re)définies mais non utilisées ...). Elles peuvent mettre en œuvre des outils permettant d'analyser le flot des données, le graphe de contrôle, le domaine de variation des variables ou les conditions d'exécution de certains chemins d'un logiciel. Il est aussi possible d'effectuer des mesures de complexité permettant de guider l'effort de test sur certaines parties du produit logiciel.

La mise en œuvre de ces méthodes et outils de test statique est plus facile lorsque le langage de programmation est évolué. Cependant, ces méthodes ne permettent pas de vérifier *dynamiquement* le comportement d'un logiciel. De plus, lors d'une modification due à la maintenance ou à l'évolution du produit logiciel, il est très difficile, voire impossible, de réutiliser les résultats du test de la version précédente pour valider la nouvelle version.

Ces méthodes de test statique (à l'exception de la fiche sur la preuve de logiciel) ne sont pas abordées, bien qu'elles soient en général très efficaces, peu coûteuses et un préalable souvent indispensable au test dynamique.

2 Le test "dynamique"

2.1 Généralités

Les méthodes de test dynamique consistent en l'exécution du logiciel sous test sur un sous-ensemble fini de ses entrées possibles appelé *un jeu de tests (test data set).* Il se pose alors quatre problèmes :

- quels sont les critères permettant de choisir un bon jeu de tests : *le problème de la sélection ;*
- comment soumettre le jeu de tests sélectionné : *le problème de la mise en œuvre ;*
- comment décider du succès ou de l'échec d'un jeu de tests : *le problème de l'oracle ;*
- à partir de quel moment peut-on estimer que le produit logiciel n'a plus besoin d'être testé : *le problème de l'arrêt du test.*

Il existe plusieurs façons de sélectionner un jeu de tests.

- Lorsqu'on utilise les sources du logiciel pour guider cette sélection, on parle de *test structurel (glass-box testing).*
- Lorsqu'on utilise les spécifications (spécifications techniques des besoins [STB] ou spécifications détaillées) du produit logiciel, on parle de *test fonctionnel (black-box testing).*
- Lorsque cette sélection est faite par tirage aléatoire sur le domaine des entrées du produit logiciel, on parle de *test aléatoire (random testing)*, ou de *test statistique (statistical testing).*
- Enfin, une approche moins connue appelée *test boîte grise (grey-box testing)* a pour but d'exploiter les aspects complémentaires entre les approches précédentes.

Si l'on considère que le test de logiciel a pour but de détecter des "fautes", les besoins du test de logiciel et ceux du test de matériel peuvent sembler voisins. Malheureusement, si pour le test de matériel des modèles de fautes efficaces peuvent être relativement aisément construits, ce n'est pas toujours le cas pour le test dynamique de logiciel (c'est en tout cas insuffisant). C'est ce que démontre [How 89]. Remarquons toutefois qu'il n'en est pas de même pour le test statique de logiciel, où les modèles de fautes sont très efficaces. Dans le cadre du test dynamique, c'est la méthode fondée sur les graphes cause-effet qui présente le plus d'analogies avec les méthodes utilisées en test de matériel (voir la fiche sur le test fonctionnel).

2.2 Tests unitaires, d'intégration et de validation

On distingue classiquement trois phases successives de test dans le processus de vérification/validation d'un produit logiciel :

1. Les *tests unitaires* concernent la vérification d'un composant logiciel (ou module de programme). Ils prennent donc en compte évidemment le composant logiciel sous test et les spécifications détaillées (lorsqu'elles existent à ce niveau). La phase de tests unitaires doit être particulièrement soignée du fait qu'elle permet de détecter au plus tôt la plupart des erreurs. De plus, l'instrumentation, matérielle ou logicielle, nécessaire pour les tests unitaires est très lourde. En effet, il faut simuler les modules appelants (il faut développer alors des *"drivers"*, parfois appelés "lanceurs" en français) et les modules appelés (il faut développer alors des *"stubs"*, parfois appelés "muets" en français). Le test unitaire peut faire appel à des tests des procédure élémentaires du composant logiciel (avant intégration de ces procédures dans le composant).

2. Les *tests d'intégration* concernent la vérification du comportement relatif des composants logiciels entre eux. Ils prennent en compte à la fois le graphe d'appels et les spécifications (STB ou détaillées) de façon à activer les fonctionnalités et/ou éprouver les performances de la partie concernée du produit logiciel (partiellement) intégré. Ceci revient souvent à exercer les interfaces des modules en cours d'intégration. La politique d'intégration à mettre en place dépend de la méthode de conception suivie, de la nature des tests que l'on veut pouvoir soumettre dès la phase d'intégration, ainsi que des caractéristiques propres au produit logiciel sous test (temps réel, proximité du matériel, ...). Certaines méthodes de conception rendent difficile et très coûteux le test d'intégration, principalement lorsque cette conception ne permet pas de construire facilement un graphe d'appels. De plus, comme pour les tests unitaires, l'instrumentation matérielle ou logicielle est très lourde. En effet, il faut simuler les modules non encore intégrés via des "stubs" ou des "drivers".

   Il existe principalement trois méthodes d'intégration : la méthode ascendante qui ne requiert aucun "stub" (muet) mais beaucoup de "drivers" (lanceurs), la méthode descendante qui ne requiert aucun "driver" mais beaucoup de "stubs", et enfin, la méthode "sandwich" où l'intégration part à la fois des composants les plus externes et des composants les plus internes vers les composants intermédiaires (lorsqu'elle est applicable cette méthode est un bon compromis permettant de limiter le nombre de stubs et drivers).

   Lorsqu'aucune de ces méthodes n'est appliquée, l'intégration est de type "big bang" (tous les éléments logiciel sont intégrés simultanément), qui ne nécessite que très peu de stubs ou drivers, mais rend complexe l'analyse et la localisation des défauts.

3. Les *tests de validation* ou tests système concernent la vérification des fonctionnalités décrites dans la STB du produit logiciel. Ils sont généralement soumis lorsque le produit logiciel est intégré cependant certains d'entre eux peuvent être soumis dès la phase d'intégration.

N.B. : l'échec d'un test de validation est très lourd de conséquences et doit par conséquent être exceptionnel.

Toutefois, les frontières entre ces trois phases de test sont relativement floues. Par exemple, lors du test unitaire, une phase d'intégration préliminaires des procédures élémentaires en un composant logiciel peut être appliquée.

Domaines d'application des méthodes de sélection

- Le test structurel est abondamment utilisé pour le test unitaire et le test d'intégration. Il n'est pas utilisable en phase de validation.
- Le test fonctionnel est particulièrement bien adapté pour le test de validation (par rapport à la spécification technique des besoins). Il peut aussi compléter utilement les tests d'intégration, et parfois même les tests unitaires (par rapport aux spécifications détaillées).
- Selon les variantes, le test aléatoire peut être appliqué à chacune des phases du test (voir la fiche correspondante).

Le désavantage majeur des tests structurels et aléatoires est l'oracle : il est en effet difficile de prévoir les résultats attendus pour des tests qui ne sont pas directement dérivés d'une spécification (STB ou détaillée).

3 Les méthodes de sélection

3.1 Le test structurel (Glass-box)

Les méthodes de test structurel reposent sur le graphe de contrôle (control flow analysis), sur le flot des données (data flow analysis) ou sur le graphe d'appels du logiciel. Il est alors possible de sélectionner un jeu de tests selon certains critères de couverture (coverage criteria).

Pour le test unitaire, les critères les plus connus sont :

- toutes les instructions,
- toutes les portions linéaires de code suivies d'un saut ("LCSAJ" en anglais),
- tous les enchaînements possibles entre les blocs d'instructions,
- tous les chemins,
- toutes les définitions de variables,
- tous les couples définition-utilisation de variable,
- tous les chemins élémentaires entre la définition et l'utilisation d'une variable.

Le critère de couverture de tous les chemins d'un programme n'est pas toujours possible. Cette impossibilité est liée à l'existence de boucles ; il est nécéssaire alors de limiter le nombre de passages dans ces boucles : chemins d'ordre 2, 3, $\ldots n$.

Pour le test d'intégration, des critères similaires de couverture du graphe d'appels sont utilisés pour exercer les interfaces des modules. Le critère le plus utilisable dans ce cas est "tous les couples module appelant/module appelé" (qui correspond à la stratégie "tous les enchaînements possibles").

En règle générale, il est facile de mettre en œuvre ces critères de couverture. Le fait que le langage de programmation soit évolué facilite les choses. En effet, le graphe de contrôle et le flot de données sont alors plus petits et beaucoup plus simples à analyser. Cependant, il peut être nécessaire de faire d'importants développements logiciels et matériels pour pouvoir appliquer un jeu de tests structurel, surtout dans le cas de logiciels intégrés ou embarqués. D'autre part, ces critères structurels ne peuvent pas révéler des chemins manquants. Ces chemins correspondent à des cas oubliés ou confondus par rapport aux cas figurant dans la spécification détaillée (lorsqu'elle existe). De plus, il n'est pas judicieux de réutiliser un jeu de tests structurels d'une version à une autre d'un même produit logiciel. En effet, lors d'une modification due à la maintenance ou à l'évolution d'un produit logiciel, le graphe de contrôle et le flot des données peuvent être modifiés, et il n'y a plus aucune garantie de couverture. Lors d'une modification, le jeu de tests doit évoluer en fonction des modifications correspondantes du graphe de contrôle ou du flot de données.

3.2 Le test fonctionnel (Black-box)

Il est possible de sélectionner directement à partir des spécifications (STB ou détaillées), des tests révélant des cas manquants dans le produit logiciel. De plus, pour une modification du produit logiciel qui ne remet pas en cause sa spécification, il est possible de ré-appliquer le même jeu de tests fonctionnel. On peut ainsi effectuer des tests de non-régression, i.e. vérifier que les modifications n'ont pas introduit d'erreurs. Bien qu'il soit surtout utilisé en phase de validation, le test fonctionnel peut aussi être appliqué en test unitaire ou d'intégration.

Contrairement au test structurel, il existe très peu d'outils pour assister la sélection de jeux de tests fonctionnels. Ceci est essentiellement dû au fait que les spécifications de logiciels (STB ou détaillées) sont souvent des spécifications en langage naturel à partir desquelles il est impossible de sélectionner automatiquement des cas de tests. Depuis l'apparition des spécifications formelles (ou semi-formelles) il est possible de définir des outils permettant d'assister la sélection de jeux de tests fonctionnels (ces outils travaillent sur le texte des spécifications, alors que les outils de test structurel travaillent sur les sources des produits logiciel). Confère par exemple les expériences réalisées au LAAS et à la CGE à partir de réseaux de Petri ([BCCLMM 86]), ou bien celles réalisées chez IGL et VERILOG à partir de diagrammes SADT ([Rig 86], [Rub 89]). Dans ces deux cas, les critères de couverture employés reposaient sur les graphes sous-jacents à ces spécifications. D'autres travaux [GHM 81], [BCFG 86], [BGM 91], montrent que les spécifications algébriques sont bien adaptées à la production automatique de jeux de tests fonctionnels. Il devient alors possible d'exprimer le rapport entre le test et la preuve de

logiciel.

3.3 Le test aléatoire (Random testing)

Par définition, la sélection est effectuée en choisissant au hasard un jeu de tests parmi les entrées acceptables du logiciel. Les méthodes de sélection de jeux de tests aléatoires sont donc à priori très simples, sous réserve de disposer d'une caractérisation de l'ensemble des entrées acceptables du logiciel testé. Lorsque la loi de probabilité des entrées lors de l'utilisation effective du produit logiciel est connue, elle peut être utilisée pour guider la sélection. De plus, il est possible de définir un modèle de croissance de fiabilité reposant sur cette loi; ce dernier permet alors de fournir un critère d'arrêt du test. Que la loi de probabilité utilisée soit uniforme ou dépendante des utilisations futures, on parle indifféremment de méthodes de sélection de jeux de tests *aléatoires* ou *statistiques*.

Cependant, ces approches (aléatoires ou statistiques) rendent complexe le problème déjà cité de l'oracle (décision du succès ou de l'échec d'un jeu de tests). En effet, lorsqu'un test est choisi au hasard, il n'y a aucune raison pour que l'on puisse facilement prévoir le résultat correct qui lui est associé. Toutefois, le test aléatoire, comme le test fonctionnel, peut révéler des chemins manquants du logiciel que le test structurel ne saurait révéler.

4 Mise en œuvre et oracle

Bien que les moyens à mettre en œuvre pour soumettre les tests soient souvent très importants, ce problème est fort peu abordé dans la littérature. Il ne sera donc pas développé dans les fiches de test malgré son importance. Les techniques utilisées sont essentiellement empiriques, et il ne semble pas exister de méthodologie générale permettant de résoudre le problème de la soumission des tests. En effet, l'instrumentation est souvent spécifique aux applications envisagées et donc fort peu généralisable. Il est néanmoins crucial de souligner la part importante que nécessite l'instrumentation dans l'effort global du test. Il faut donc prévoir au plus tôt les coûts associés. Les lanceurs et muets (stubs et drivers) déjà mentionnés en sont des cas particuliers, ainsi que les bancs de test, de simulation etc. sans oublier les modifications temporaires de codes éventuellement utiles pour le test.

L'un des très rares articles portant sur ce sujet aborde le problème des drivers [Mel 86]. Les faits suivants semblent importants :

- Un driver de tests a son propre cycle de développement de logiciel qui est généralement fortement contraint par le cycle de développement du produit logiciel sous test.
- Un driver de test ne peut pas être totalement transparent. Par exemple il consomme des ressources du système sur lequel il s'exécute, ce qui peut nuire aux performances du produit sous test s'il est exécuté sur le même système.
- L'environnement au sein duquel le produit logiciel sous test est exécuté sous le contrôle du driver doit être défini avec précision et reproduit sans faille par le driver.

- Plus le driver et le produit sous test sont interdépendants, plus le risque d'apparition d'erreurs liées à la configuration est grand. Ceci influe sur le choix de la machine hôte du driver par rapport à celle du produit sous test.
- Un langage de description de cas de tests facilite à la fois la soumission des tests, l'archivage des tests, et l'oracle.

Le problème de l'oracle consiste à décider du succès ou de l'échec d'un jeu de tests. Lorsque la méthode de sélection est fonctionnelle, la spécification (STB ou détaillée) permet souvent de prévoir le résultat attendu des tests sélectionnés. Souvent, ces résultats attendus sont consignés dans une "matrice de test" ainsi que les moyens à mettre en œuvre pour comparer ces résultats avec ceux effectivement obtenus. Lorsque la méthode de sélection employée est structurelle ou aléatoire, il peut être très difficile de prévoir les résultats corrects que devraient retourner les tests sélectionnés. Cependant, s'il existe un prototype de référence, il peut être utilisé pour comparer ses résultats avec ceux fournis par le produit logiciel. Plus généralement, pour des composants logiciels très critiques, plusieurs composants peuvent être développés *de manière indépendante* à partir de la même spécification détaillée. Chacun des composants développés peuvent être testés et les résultats obtenus peuvent être comparés entre eux. Cette méthode est appelée test dos-à-dos (back-to-back testing en anglais).

N.B. : cette méthode ne résoud pas totalement le problème de l'oracle, puisqu'on ne sait pas a priori quels sont les composants logiciels qui fournissent les résultats corrects. De plus, la comparaison des résultats est difficile car les structures de données choisies lors du développement de chaque composant sont généralement différentes.

Dans le cadre du test dos-à-dos il est crucial d'éviter l'erreur de jugement consistant à retenir in fine le composant logiciel ayant donné lieu au plus grand nombre de modifications, suite à l'échec des tests. Il faut retenir au contraire le composant ayant fourni le moins d'échecs possibles ; en effet, si un composant a contenu de nombreuses erreurs, il est probable qu'il en contienne encore de nombreuses. En règle générale le problème crucial de l'oracle est malheureusement fort peu abordé dans la littérature. Il sera donc lui aussi insuffisamment développé dans les fiches de test, malgré son importance. Du point de vue des auteurs à l'heure actuelle, l'usage de spécifications formelles semble être le moyen le plus approprié pour dégager des oracles fiables [BGM 91].

5 L'arrêt du test

Généralement, l'arrêt du test est un arrêt sur objectif (formalisé ou non). Selon la méthode de sélection utilisée, il existe des critères adaptés.

5.1 Arrêt du test structurel

Dans le cas du test unitaire, celui-ci est directement déterminé par les critères qui ont servi à la sélection : couverture de toutes les instructions (ou un certain pourcentage), de

toutes les portions linéaires de code suivies d'un saut (ou d'un certain pourcentage), de tous les enchaînements, etc.

Dans le cas du test d'intégration l'arrêt peut être déterminé par des critères similaires de couverture du graphe d'appels du logiciel.

5.2 Arrêt du test fonctionnel

Quelle que soit la phase de test considérée (unitaire, intégration, validation) le critère évident est la couverture de toutes les fonctionnalités mentionnées dans les spécifications (qu'elles soient STB ou détaillées). Il doit de plus être affiné par une couverture satisfaisante des cas d'utilisation de ces fonctionnalités : tests nominaux, aux limites et éventuellement hors limite (robustesse). Ceci est facilité par des matrices de test et les documents d'accompagnement.

N.B. : lorsqu'on dispose de spécifications formelles, il est possible d'affiner cette couverture des fonctionnalités par une analyse "structurelle" (automatique) des spécifications.

5.3 Arrêt du test aléatoire

En général, le test aléatoire ne posséde pas de critère d'arrêt privilégié, comme pour toute les autres méthodes de test. Toutefois, le test aléatoire *opérationnel* est la seule méthode de sélection pour laquelle des modèles de croissance de fiabilité permettent de décider l'arrêt du test en fonction d'un critère *quantitatif* de fiabilité ([The 89]). Cependant, ces modèles sont paramétrés par les lois statistiques correspondant à l'utilisation effective du logiciel. Il est alors nécessaire de sélectionner les jeux de tests selon ces lois statistiques. Il est donc impossible d'utiliser ce type de critère d'arrêt dans le cas où la méthode de sélection est déterministe. De plus, même lorsque la méthode de sélection de jeux de tests est aléatoire, ce critère d'arrêt n'est pas fondé si les lois de probabilité considérées ne sont pas celles correspondant au profil opérationnel.

5.4 Utilisation du test par mutation (Mutation testing)

Comme son nom ne l'indique pas, le "test par mutation" est une méthode d'évaluation de la qualité des jeux de tests (quelle que soit la méthode de sélection), et non pas une méthode de test. Il s'agit de créer des mutants du produit logiciel en modifiant de façon automatique certaines instructions ou certaines conditions (typiquement, changement d'un $\leq$ en un $<$, changement du signe d'une expression arithmétique ...). Le critère d'arrêt du test est alors généralement la détection d'une anomalie pour tous les mutants (ou un certain pourcentage d'entre eux). Un des problèmes principaux est alors de pouvoir traiter tous les mutants ainsi engendrés : si le critère de mutation est de créer un mutant pour chaque instruction du programme le nombre de mutants devient rapidement impraticable. Il est d'autre part difficile de déterminer les mutants réellement erronés. Toutefois, cette méthode est relativement efficace et facile à mettre en œuvre ([CR 81]) et peut être appliquée quelle que soit la méthode de sélection employée.

N.B. : il ne faut pas confondre les injections de fautes du test par mutation avec celles

du test de tolérance aux fautes. En effet, le test par mutation consiste à modifier les *sources* du produit logiciel, tandis que pour du test de tolérance aux fautes il faut placer le produit logiciel sous test dans des contextes de fautes à tolérer.

6 Interaction du test et de la preuve

La preuve de logiciel consiste à démontrer qu'un logiciel satisfait certaines propriétés formelles (issues de ses spécifications, STB ou détaillées) en utilisant des règles d'inférences propres au langage de programmation utilisé (logiques de Hoare par exemple). Bien que les spécifications de départ ne soient pas nécessairement formelles, les propriétés que l'on prouve doivent par contre toujours être exprimées formellement.

N.B. : même lorsqu'une preuve réussit, cela ne dispense pas du test. En effet, toute preuve suppose un comportement idéal du langage de programmation considéré (au minimum : correction du compilateur, ou une méthode de programmation ...), et le test peut tout-à-fait invalider ce genre d'hypothèse (en particulier en intégration).
Il est important de bien comprendre que l'échec du test établit la non conformité du produit logiciel alors que l'échec d'une preuve n'implique pas que le produit logiciel ne soit pas conforme. Cependant dans certains cas, l'échec d'une preuve peut révéler des anomalies logicielles. La preuve de logiciel pourrait donc être utilisée comme méthode partielle de test.

De manière plus réaliste, diverses techniques de preuve peuvent *participer* au déroulement d'un test. Par exemple, des techniques de preuve peuvent être utilisées pour décider du succès ou de l'échec d'un test ([BGM 91]), ou encore, dans le cas du test structurel, la preuve permet de détecter des chemins infaisables, etc.

7 Usage des techniques de test dynamique

Afin de maximiser l'apport de chacune des techniques de test présentées ci-dessus, il semble qu'à l'heure actuelle l'enchaînement des diverses techniques de test dynamique préconisé dans [LCGP 88] reste approprié.

- En phase de test unitaire :
  - Une première étape de test aléatoire (généralement uniforme) permet de détecter au plus tôt les erreurs les plus flagrantes. A ce niveau du test, le problème de l'oracle peut être une entrave majeure. L'usage de spécifications formelles détaillées peut faciliter la définition d'un oracle. D'autre part, le test aléatoire opérationnel est ici rarement judicieux, par conséquent, l'arrêt de cette étape de test sera décidé par mutation.
  - Une étape de test fonctionnel par rapport aux spécifications détaillées peut ensuite être appliquée. Ceci peut là encore être automatisé, lorsque les spécifications détaillées sont formelles. L'arrêt du test peut être décidé par mutation, ou mieux, par rapport à des critères de couverture des spécifications.

Les tests sélectionnés lors de ces deux étapes (aléatoires et fonctionnels), pourront être réutilisés comme tests de non-régression.

  - Une étape de test structurel peut alors être mise en œuvre. Selon le niveau de confiance requis, les critères de sélection ou d'arrêt du test peuvent reposer sur l'un des critères de couverture déjà cités, que l'on considère le graphe de contrôle ou le flot de données.
  - Les trois étapes précédentes ayant en général donné une meilleure connaissance des "portions de code à risque" et des "données d'entrées sensibles" (i.e. celles ayant occasionné le plus d'échecs lors des tests précédents), il peut être judicieux de compléter par du "test aléatoire spécialisé" afin d'exercer plus finement ces portions de code et ces données.

- En phase de test d'intégration certains tests fonctionnels issus de la STB peuvent déjà être appliqués, et complétés par du test structurel fondé sur la couverture des couples module appelant/module appelé.
  N.B. : une technique récemment étudiée visant à enrichir le graphe d'appels avec les informations du graphe de contrôle [HS 91] permet d'utiliser des critères de couverture (et donc d'arrêt) plus performants.
  Si l'on connaît les lois de probabilité des données en usage opérationnel pour les modules intégrés, une étape de test aléatoire opérationnel peut être appliquée. Son arrêt est décidé selon un modèle de croissance de fiabilité.

- En phase de test de validation ou système, le test fonctionnel est à l'heure actuelle probablement le plus adapté, son arrêt repose sur une notion de couverture de la STB. Comme pour la phase d'intégration, si les lois de probabilités des données sont connues, il peut êtrev intéressant d'appliquer une étape de test aléatoire opérationnel.

8 Plan des fiches

Il y aura donc quatre fiches concernant :

- les méthodes de test structurel,
- les méthodes de test aléatoire ou de test statistique,
- les méthodes de test fonctionnel,
- le test par mutation,

auxquelles s'ajoutent deux fiches traitant de sujet moins souvent abordés dans la littérature :

- le test dit "boîte grise",
- les aspects de preuve (incluant les aspects "cleanroom").

9 Structure des fiches

La structure générale des fiches sera la suivante. Certaines des sections mentionnées ci-dessous peuvent éventuellement être sans objet, elles seront alors vides.

1. une section décrivant le principe général de la méthode,
2. pour chaque variante de la méthode une section décrivant ses particularités (même lorsque certaines variantes partagent des techniques communes),
3. une section décrivant les phases d'application privilégiées (tests unitaires, d'intégration ou de validation),
4. une section décrivant les difficultés d'oracle et de mise en œuvre associés,
5. une section décrivant les critères d'arrêt du test applicables,
6. une section éventuelle de conclusion et commentaires,
7. une section de bibliographie pour qui veut en savoir plus.

Références bibliographiques

NDR : Seuls les articles présentant un minimum d'intérêt sont répertoriés ci-dessous. Cette étude a porté sur un nombre plus important d'articles

- [AB 81] D. Andrews, J.P. Benson
  "An automated program testing methodology and its implementation"
  Proc. 5^{th} Int. Conf. Soft. Eng., pp. 254-261, 1981.

- [Bac 86] R.C. Backhouse
  "Program Construction and Verification"
  Prentice-Hall International Series in Computer Science, 1986, 280p. Traduction française parue, sans doute chez Masson.

- [Ban 86]
  Workshop on Software Testing, Banff Canada, IEEE Catalog Number 86TH0144-6, pp 132-141 (Juillet 1986)

- [Ban 88]
  Proceedings of the 2nd IEEE-ACM Worshop on Software Testing Analysis and Verification, Banff, 1988.

- [BCCLMM 86] B. Berthomieu, N. Choquet, C. Colin, B. Loyer, J.M. Martin, A. Mauboussin
  "Abstract data nets; combining Petri nets and abstract data types for high level specifications of distributed systems"
  Proc. 7th European Workshop on Application and Theory of Petri Nets, Oxford, pp. 25-48, (Juin 1986)

- [BCFG 86] Luc Bougé, Nicole Choquet, Laurent Fribourg, Marie-Claude Gaudel
  "Test sets generation from algebraic specifications using logic programming"
  Journal of Systems and Software Vol 6, n^{o}4, pp.343-360 (Novembre 1986).
  Aussi: rapport de Recherche LRI n^{o}240.

- [BDLS 80] T.A. Budd, R.A. Demillo, R.J. Lipton, F.G. Sayward
  "Theoretical and empirical studies on using program mutation to test the functional correctness of programs"
  Proc. ACM Symp. Principles of Prog. Lang., pp. 220-222, 1980.

- [BGM 91] G. Bernot, M.-C. Gaudel, B. Marre
  "Software testing based on formal specifications : a theory and a tool"
  Software Engineering Journal, novembre 1991.
  Aussi : rapport de recherche LRI n^{o}581 (Juin 1990).

- [BL 78] T.A. Budd, R.J. Lipton
  "Mutation analysis of decision table programs"
  Proc. Conf. Information Science and Systems, pp. 346-349, 1978.

- [BM 79] R. Boyer, J. Moore
"A Computational Logic"
Academic Press, 1979

- [BM 83] D.L. Bird, C.U. Munoz
"Automatic generation of random self-checking test cases"
IBM Syst. J. (USA) vol.22, n°3, pp.229-245, 1983.

- [Bri 89] E. Brinksma
"A theory for the derivation of tests"
Book. The formal description technique LOTOS, P.H.J. Van Eijk, C.A. Vissers & M. Diaz Eds., Elsevier Sci. Pub. B.V. (North-Holland), 1989.

- [BY 91] W. Bevier, W. Young
"The proof of Correctness of a Fault-tolerant Circuit Design"
2nd IFIP Working Conference on Dependable Computing for Critical Applications, Tucson, Feb. 1991.

- [Car 81] R. Cartwright
"Formal program testing"
Proc. 8^{th} Annual ACM Symp. Principles of Programming Languages, 1981.

- [CDM 86] P.A. Currit, M. Dyer, H.D. Mills
"Certifying the reliability of software"
IEEE Transactions of Software Engineering, Volume SE-12, n^o 1, pp. 3-11 (Janvier 1986)

- [CDM 89] P.A. Currit, M. Dyer, H.D. Mills
"Correction to Certifying the reliability of software"
IEEE Transactions of Software Engineering, Volume SE-15, n^o 3, p. 362 (Mars 1989).

- [CF 84] J.S. Collofello, A.F. Ferrara
"An automated Pascal multiple condition test coverage tool"
COMPSAC pp. 20-26 IEEE 1984.

- [Cha 79] J. Chan
"Program debugging methodology"
Phd thesis, Leicester polytechnic, 1979.

- [Cho 78] T.S. Chow
"Testing software design modeled by finite-states machines"
IEEE Trans. Soft. Eng., vol. SE-4, may 1978.

- [Chu 87] T. Chusho
"Test data selection on quality estimation based on the concept of essential branches for path testing"
IEEE Trans. Soft. Eng. Vol-SE-13, num.5, May 1987.

- [Coh 89] A. Cohn
  "The notion of Proof in Hardware Verification"
  Journal of Automated Reasoning, Vol.5, pp. 127-139, 1989.

- [Coo 76] D.W. Cooper
  "Adaptative testing"
  Proc. 2^{nd} Int. Conf. Soft. Eng., pp. 223-226, 1976.

- [Cow 88] P.D. Coward
  "A review of software testing"
  Inf. Soft. Techn., UK, Vol.30, num.3, April 1988.

- [CR 81] B. Chandrasekaran, S. Radicchi (eds)
  "Computer Program Testing"
  North-Holland 1981.

- [CR 84] L.A. Clarke, D.J. Richardson
  "Symbolic evaluation - an aid to testing and verification"
  Book Software Validation, H.L. Hausen Ed., Elsevier Science Pub. B.V. (North-Holland), pp. 141-166, 1984.

- [CR 87] L.A. Clarke, D.J. Richardson
  "Evaluation and integration of software testing techniques"
  Proc. CIPS Edmonton'87: Intelligence Integration, Canada, Nov. 1987.

- [DBC 91] B. DiVito, R. Butler, J. Caldwell
  "High Level Design Proof of a Reliable Computing Platform."
  2nd IFIP Working Conference on Dependable Computing for Critical Applications, Tucson, Feb. 1991.

- [DGMOK 88] R.A. Demillo, D.S. Guindi, W.M. McCracken, A.J. Offutt, K.N. King
  "An extended overview of the Mothra software testing environment"
  Proc. 2^{nd} ACM SIGSOFT workshop on software testing verification and analysis, Banff, Canada, pp. 142-151, july 1988.

- [Del 91] F. Delamotte
  "Spécifications et preuves formelles, expériences aux USA: présentation par M-C. Gaudel"
  Mémo CNES H/PS/PA/91/103, Juin 1991.

- [Den 91] R. Denney
  "Test-case generation from prolog-based specifications"
  IEEE Software journal, special issue on Testing, Tracking design, Atomic Delegation, vol. 8, num. 2 (march 1991).

- [DN 81] J.W. Duran, S.C. Ntafos
  "A report on random testing"
  5th International Conference on Software Engineering, San Diego, pp 179-183 (March 1981).

- [DN 84] J.W. Duran, S.C. Ntafos
"An evaluation of random testing"
IEEE Transactions of Software EngineeringVolume SE-10, n^o 4, pp. 438-444 (Juillet 1984).

- [Dye 87] M. Dyer
"A Formal Approach to Software Error Removal"
Journal of Systems and Software 7, 1987, pp. 109-114.

- [ESW 90] W.K.Ehrlich, J.P.Stampfel, J.R. Wu
"Application of software reliability modeling to product quality and test process"
12th ACM-IEEE International Conference on Software Engineering, Nice, pp. 108-116 (Avril 1990).

- [Gal 84] J.M. Galvin
"Mutation analysis : a user's view"
Proc. 7th annual MICRO-DELCON'84, the Delaware Bay Comp. Conf., IEEE Comp. Soc. Press Silver Spring, pp. 30-40, march 1984

- [Gau 91] M-C. Gaudel
"Advantages and Limits of Formal Approaches for Ultra-High Dependability"
2nd PDCS Worshop, Newcastle, Mai 1991.

- [GH 90] G. Guiho, C. Hennebert
"SACEM Software Validation"
12th IEEE-ACM International Conference on Software Engineering, Mars 1990, pp. 186-191.

- [GHM 81] J. Gannon, R. Hamlet, P. McMullin
"Data-Abstraction Implementation, Specification, and Testing"
ACM-TOPLAS, vol. 3, n^o 3, pp. 211-223 (Juillet 1981).

- [GJM 91] C. Ghezzi, M. Jazayeri, D. Mandrioli
"Fundamentals of Software Engineering"
Prentice-Hall Internalional Editions, 1991, 571p.

- [HS 91] M.J. Harrold, M.L. Soffa
"Selecting and using data for integration testing"
IEEE Software journal, special issue on Testing, Tracking design, Atomic Delegation, vol. 8, num. 2 (march 1991).

- [How 81] W.E. Howden
"Errors, design properties and functional program tests"
Computer program testing, Chandrasekaran B. and Radichi S. eds., North-Holland 1981.

- [How 82] W.E. Howden
"Weak mutation testing and completeness of test sets"
IEEE Trans. Soft. Eng., Vol. SE-8, No. 4, pp. 371-379, july 1982.

- [How 89] W.E. Howden
  "A comparison of software and hardware testing"
  Proc. IFIP 89, 11^{th} world comp. congress, San-Francisco, USA, North-Holland, G.X. Ritter Eds, august 1989.

- [Inc 87] D.C. Ince
  "The automatic generation of test data"
  The Comput. Journal, vol. 30, num. 1, pp. 63-69 (1987).

- [JC 88] P. Jalote, M.G. Caballero
  "Automated test case generation for data abstraction"
  Proc COMPSAC 88, The twelfth Int. Comp. Soft. & Applications Conf., Chicago Il. USA, pp. 205-210, IEE Comp. Soc. Press. Washington DC, USA, (oct 88).

- [JLM 91] A. Jassim, B. Littlewood, P. Mellor
  "Random testing compared with structural testing"
  à paraître

- [Las 87] J.W. Laski
  "On the comparative analysis of some data flow testing strategies"
  Dep. Eng. Comp. Sci. Oakland Univ., Rochester, MI Technical Report 87-05, May 1987.

- [LCGP 88] B. Courtois, M.-C. Gaudel, J.-C. Laprie, D. Powell
  "Sureté de Fonctionnement des Systèmes Informatiques - Matériel et Logiciel -"
  Rapport final de contrat CNES, publié en monographie Dunod, Avril 1989, 215 pages.

- [LK 84] J.W. Laski, B. Korel
  "A data flow oriented program testing strategy"
  IEEE Trans. Soft. Eng., Vol.SE-9, num 3, may 1983.

- [MDL 87] H.D. Mills, M. Dyer, R.C. Linger
  "Cleanroom Software Engineering"
  IEEE Software, Septembre 1987, pp. 19-25.

- [Mel 86] J. S. Melvaine
  "Development of test drivers for dynamic testing of software systems"
  First australian software engineering conference, Camberra, Instn. Eng. Australia. Barton, ACT,(May 1986).

- [Mil 84] E.F. Miller
  "Quality management technology: practical applications"
  Proc. symposium Darmstadt, Germany, september 1983, software validation, H.L. Hausen ed., Elsevier Science Pub. B.V. (North-Holland) 1984.

- [MIO 87] J. D. Musa, A. Iannino, K. Okumoto
  "Software Reliability : Measurement, Prediction, Application"
  McGraw-Hill Series in Software Engineering and Technology (1987).

- [MMS 1990] L.E. Moser, P.M. Melliar-Smith
"Formal Verification of Safety-critical Systems"
Software Practice and Experience, Vol. 20, n^o8, 1990, pp. 799-821.

- [Mul 88] M. Mullerburg
"On fundamentals of program testing"
Premier séminaire EOQC sur la qualité des logiciels, Bruxelles, 25-27 avril 1988.

- [Mye 78] G.J. Myers
"A Controlled Experiment in Program Testing and Code Walkthroughs / Inspections"
CACM, pp. 760-768 (Septembre 1978).

- [Mye 79] G.J. Myers
"Errors, design properties and functional program tests"
Computer program testing, Chandrasekaran B. and Radichi S. eds., North-Holland 1981.

- [Mye 79] G.J. Myers
"The art of software testing"
John Whily & Sons, London, 1979

- [MYV 90] N. Malevris, D. Yates, A. Veevers
"Predictive metric for likely faisability of program path"
Inf. Soft. Techn., UK, Vol.32, Num.2, March 1990.

- [Nta 88] S.C. Ntafos
"A comparison of some structural testing strategies"
IEEE Trans. Soft. Eng., Vol.SE-14, num 6, jun 1988.

- [Ost 85] T.J. Ostrand
"The use of formal specifications in program testing"
Third Int. Workshop on Software Specification and Design, London, UK, IEEE Comp. Soc. Press, Washington, DC, USA, pp. 253-255 (august 1985)

- [OW 86] T.J. Ostrand, F.J. Weyuker
"Design for a Tool to Manage Specification-Based Testing"
Proc. Workshop on Software Testing, Verification and Analysis, Banff, Canada, IEEE Comp. Soc. num. 723, July 1986.

- [Pro 82] R.L. Probert
"Grey-Box (Design-Based) Testing techniques "
Proc. 15th Hawaii Int. Conf. on System Sciences, 1982, Honolulu, HI, USA, vol.1, (jan. 1982).

- [RC 81] D.J. Richardson, L.A. Clarke
"A partition analysis method to increase program reliability"
Proc. 5^{th} IEEE Int. Conf. Soft. Eng., 1981.

- [Rig 86] G. Rigal
"Generating Acceptance Tests from SADT/SPECIF"
IGL technical report (Août 1986).

- [Rub 89] J.P. Rubaux
"Rapport final d'étude du poste de generation de test ASA"
RATP Paris, Direction des Equipements Electriques, TT-SEL/89-183/JPR (1989).

- [RVH 1989] J. Rushby, F. Von Henke
"Formal Verification of a Fault Tolerant Clock Synchronization Algorithm"
NASA Contractor Report 4239. Juin 1989. 217 pages.

- [RW 85] S. Rapps, E.J. Weyuker
"Selecting software test data using data flow informations"
IEEE Trans. Soft. Eng., Vol.SE-11, num 4, April 1985.

- [SBB 87] R.W. Selby, V.R. Basili, F.T. Baker
"Cleanroom Software Development: an empirical evaluation"
IEEE Transactions on Software Engineering, vol. SE-13, Sept. 1987, pp1027-1037.

- [SG 88] P. Saunier, E. Girard
"Des Modèles de Fiabilité du Logiciel? Pourquoi faire?"
Revue Génie Logiciel, n^o 10, EC2-Editeur, pp 50-62 (Janvier 1988).

- [Sil 88] M. Sillon
"Une gamme d'outils pour le test du logiciel"
Génie Logiciel et Systèmes Experts n^{o}12, pp.54-57, (Juin 1988).

- [The 89] P. Thévenod
"Software validation by means of statistical testing: retrospect and future direction"
IFIP Working Conf. on dependable computing for critical applications, Santa Barbara (Août 1989).

- [The 90] P. Thévenod-Fosse
"On the efficiency of statistical testing wrt software structural test criteria"
IFIP Working Conference on Approving Software Products (ASP-90), Garmisch-Partenkirchen, pp. 29-42 (Septembre 1990).

- [TWC 90] P. Thévenod-Fosse, H. Waeselynk , Y. Crouzet
"An experimental study on software structural testing : deterministic versus random input generation"
Rapport LAAS n^o 90.387 (Novembre 1990).

- [TW 91] P. Thévenod-Fosse, H. Waeselynk
"An investigation of software statistical testing"
Rapport LAAS n^o 91.003 (Janvier 1991).

- [Van 78] D. Van Tassel
"Program style, design, efficiency, debugging and testing"
Book, Prentice Hall 1978

- [WHH 80] M.R. Woodward, D. Hedley, M.A. Hennell
"Experience with path analysis and testing of programs"
IEEE Trans. Soft. Eng., Vol. 6, num. 6, pp. 278-285, 1980.

- [WH 88] M.R. Woodward, K. Halewood
"From weak to strong, dead or alive ? analysis of some mutation testing issues"
Proc. 2^{nd} ACM SIGSOFT workshop on software testing verification and analysis, Banff, Canada, pp. 152-158, july 1988.

- [Wil 88] C. Wild
"The use of generic constraint logic programming for software testing and analysis"
Proc. 2^{nd} ACM-IEEE Workshop on Software Testing, Verification and Analysis, Banff, Canada, july 1988.

- [WO 80] F.J. Weyuker, T.J. Ostrand
"Theories of program testing and the application of revealing subdomains"
IEEE Trans. Soft. Eng., vol. SE-6, num. 3, pp. 236-246, 1980.

- [ZTV 86] I. Zualkernan, W. Tsay, D. Volovik
"Expert systems and software engineering: ready for marriage ?"
IEEE Expert, Vol. 1, No. 4, pp. 24-31, Winter 1986.

- [Zei 84] S.J. Zeil
"Perturbation testing for computation errors"
Proc. 7^{th} Int. Conf. Soft. Eng., IEEE pp. 257-265, march 1984.

- [Zei 89] S.J. Zeil
"Perturbation techniques for dectecting domain errors"
IEEE Trans. Soft. Eng., Vol. 15, No 6, june 1989.

10 Test structurel

10.1 Principe général

Les techniques dites de "test structurel" sont caractérisées par le fait que le jeu de tests est sélectionné à partir d'une description de la structure du produit logiciel sous test et non pas de ses spécifications (STB ou spécifications détaillées). Le test structurel peut reposer sur l'exécution de seulement quelques chemins dans le produit logiciel ou peut reposer sur un certain niveau de couverture (tel que l'exécution de toutes les instructions). Bien évidemment, la meilleure stratégie serait de couvrir tous les chemins possibles du produit logiciel sous test. Un chemin est une séquence d'instructions consécutives menant du point d'entrée à un point de sortie. La conjonction des conditions apparaissant dans un chemin définit un sous-domaine du domaine des entrées possibles. Afin de couvrir un tel chemin, il suffit de sélectionner une valeur satisfaisant la conjonction de ses conditions. Toutefois, deux obstacles majeurs existent à l'exécution de tous les chemins.

Le premier obstacle est le nombre impraticable de chemins possibles. Ce nombre est déterminé par le nombre de conditions et de boucles apparaissant dans le/les source/s du produit logiciel. Toutes les combinaisons possibles de ces conditions conduisent rapidement à un nombre de chemins inenvisageable. Ce phénomène est souvent cité comme le problème de l'explosion combinatoire du test. Les boucles participent fortement à cette explosion combinatoire (jusqu'à produire un nombre infini de chemins). Ceci est particulièrement dramatique si le nombre d'itérations n'est pas fixé au départ mais déterminé par les variables d'entrée.

Le second obstacle est le nombre de chemins infaisables. Un chemin infaisable est un chemin qui ne peut pas être exécuté à cause de contradictions entre certains prédicats apparaissant dans des instructions conditionnelles. La proportion de ces chemins infaisables est parfois très importante (jusqu'à 98% des chemins pour le programme du triangle qui décide si un triangle est équilatéral, isocèle, rectangle ...).

En conséquence, il est généralement impossible de couvrir tous les chemins et il faut donc se restreindre à un sous-ensemble "bien choisi" de ces chemins. Il existe principalement deux classes de méthodes de choix structurel. Ces méthodes reposent soit sur le graphe de contrôle soit sur le flot de données.

10.2 Variantes

La notion de chemin largement utilisée dans la section précédente repose sur la représentation du produit logiciel par un graphe. Selon le type de couverture souhaité, le *graphe de contrôle* ou le graphe associé au *flot de données* sont considérés. En phase d'intégration, il est possible d'utiliser des notions de couverture du *graphe d'appel* reposant sur des critères similaires à ceux employés pour le graphe de contrôle.

10.2.1 Sélection fondée sur le graphe de contrôle

Chaque bloc d'instructions élémentaire constitue un nœud du graphe. Lorsqu'un bloc d'instruction élémentaire est terminé par une instruction conditionnelle, cela donne nais-

sance à deux branches dans le graphe : la branche où la condition est satisfaite et la branche où la condition n'est pas satisfaite. Les branches ainsi crées vont vers les nœuds associés aux blocs d'instructions correspondants. Dans le cas d'une condition de boucle, le graphe de contrôle contient donc un cycle entre les nœuds correspondants.

Diverses notions de couverture minimale permettant d'assurer un certain degré de fiabilité ont été étudiées assez largement ces dernières années. Les plus pratiquées sont généralement :

- couverture de toutes les instructions ;
- couverture de toutes les branches (ou seulement de branches dites essentielles, cf. [Chu 87]) ;
- couverture de toutes les portions linéaires de code suivies d'un saut [WHH 80] (PLCS ou LCSAJ en anglais : linear code sequence and jumps) ; une PLCS est une séquence d'instructions consécutives qui se termine par un transfert de contrôle en dehors de cette séquence ;
- couverture de tous les i-chemins (i.e. tous les chemins en limitant toutefois à i le nombre de passage dans les boucles) ;
- couverture de tous les chemins (généralement infaisable) ou d'un certain pourcentage d'entre eux.

Les critères ci-dessus sont classés par ordre croissant de qualité à l'exception près des critères PLCS et i-chemins qui sont incomparables pour $i = 2$. Il existe d'autres critères intermédiaires [Ban 86] dont le classement devient alors plus complexe.

Les instructions d'affectations des chemins sélectionnés sont utilisés pour dériver ce qui est calculé par ces chemins. Ces approches peuvent utiliser des idées issues du domaine de l'exécution symbolique. On associe alors une expression algébrique à chaque variable de sortie en terme des variables d'entrée pour chaque chemin. Ceci permet d'associer à chaque chemin le sous-domaine des valeurs d'entrée qui le suivront. Si ce domaine est vide, il s'agit d'un chemin infaisable (la faisabilité d'un chemin est malheureusement un problème indécidable). Sinon, on sélectionne, selon les stratégies, des données nominales, ou aux bornes du sous-domaine correspondant.

De tels jeux de tests peuvent être engendrés automatiquement à partir d'une description syntaxique des sources et du domaine des entrées du produit logiciel. Cette description est toujours engendrée par les compilateurs (mais malheureusement pas toujours accessible) et elle peut être utilisée pour la sélection de jeux de tests. Les prédicats obtenus pour chaque chemin peuvent être insatisfaisables du fait de l'existence de contradictions : le chemin est alors infaisable. Sinon, toute solution de ces prédicats fournit un cas de test.

10.2.2 Sélection fondée sur le flot de données

Pour cette variante, les chemins sont sélectionnés pour couvrir des utilisations et/ou définitions particulières des variables. A la base, le graphe de contrôle est encore utilisé

mais cette fois pour en extraire un ensemble de sous-chemins particuliers (éventuellement de longueur nulle). En effet, l'analyse de flot de donnée ne considère que les définitions et utilisations de variables, ainsi que les sous-chemins allant d'une définition de variable à son utilisation. On comprend alors qu'il soit nécessaire de considérer aussi des chemins de longueur nulle, par exemple pour des instructions de la forme `x := x+1`. Ceci conduit donc à une version modifiée du graphe de contrôle, appelé le graphe du flot de données.

Il existe de nombreux critères de couverture pour cette variante. Les plus connus sont (par ordre croissant de qualité) :

- toutes le P-utilisations, c.a.d. toutes les utilisations d'une variable dans une condition ("P" comme prédicat) ;
- toutes les définitions de variable (incluant les redéfinitions comme `x := x+1`) ;
- toutes les C-utilisations, c.a.d. toutes les utilisations de variable dans un calcul, ou quelques P-utilisations s'il n'y a pas de C-utilisation :
- toutes les P-utilisations, ou quelques C-utilisations s'il n'y a pas de P-utilisations (critère incomparable avec le précédent) ;
- toutes les utilisations (sans redéfinition ni boucle) :
- tous les D-U-chemins, c.a.d. tous les sous-chemins élémentaires d'une définition à une utilisation (sans redéfinition ni boucle) [RW 85] ;
- quelques autres critères trop complexes pour être présentés ici (le vocabulaire nécéssaire ne pouvant être introduit dans ce type de fiche, leurs principes sont présentés dans les références citées) tels que "Reach-coverage", "Context-coverage" ou "Ordered-context-coverage" [LK 83] et [Las 87], "Required-2-tuples", "Required-$(k-1)$-tuples", "Required-k-tuples" [Nta 88] ;
- et enfin tous les chemins.

Les données associées aux chemins ainsi sélectionnés sont alors choisies selon les mêmes techniques que pour les critères fondés sur le graphe de contrôle.

10.2.3 Techniques de perturbations : un complément utile

Pour obtenir une couverture plus fine des sous-domaines propres au produit logiciel sous test, il est possible d'utiliser des techniques de perturbation. Ces techniques sont fondées sur une mesure d'efficacité des jeux de tests sélectionnés. Ces mesures sont utilisées pour engendrer des tests complémentaires.

La base de ces techniques est l'usage d'assertions insérées par le développeur dans les sources du produit logiciel. Une assertion est une information sur les valeurs raisonnables des variables. Le but est de maximiser le nombre de violations de ces assertions. Un premier jeu de test, obtenu par les variantes précédentes, est pris en considération. Il est exécuté et les violations d'assertions sont enregistrées. Chaque test est alors considéré

individuellement et on identifie le paramètre d'entrée qui contribue le moins à la violation des assertions. Des procédures d'optimisation sont alors utilisées pour trouver la meilleure valeur de ce paramètre afin de maximiser les violations d'assertions. On dit alors que le test a subi une perturbation. Le jeu de tests ainsi perturbé est de nouveau exécuté et cette méthode de perturbation lui est réappliquée. On continue ainsi jusqu'à ce que le nombre de violations d'assertions ne puisse plus être augmenté de manière significative. Dans [Coo 76] et [AB 81], des systèmes permettant d'automatiser cette technique sont décrits.

Ces approches peuvent être utilisées plus spécifiquement pour améliorer la couverture des prédicats de branchement ainsi que celle des expressions arithmétiques (voir respectivement [Zei 89] et [Zei 84]).

10.3 Phases d'application privilégiées

Les critères de sélection structurels fondés sur le graphe de contrôle ou le flot de données ne sont en général utilisables qu'en phase de test unitaire. Un langage de programmation évolué simplifie bien sûr les graphes considérés, et donc la sélection de jeux de tests correspondante. Lorsque ces graphes sont de taille assez petite, il est envisageable d'utiliser ces critères en phase d'intégration. On peut par exemple citer les travaux de [HS 91] qui démontre l'utilisation de critères fondés sur le flot de données pour le test d'interfaces en phase d'intégration. Généralement, la phase d'intégration ne considère que le graphe d'appel et ceci afin de couvrir les interconnexions entre modules. Ce sujet ne semble malheureusement pas abordé dans la littérature (hormis [HS 91]). Du point de vue des auteurs, les techniques de test structurel pour le test unitaire devraient pour la pluspart pouvoir être adaptées au test d'intégration, le principal problème étant d'étendre le graphe d'appels en utilisant les informations du graphe de contrôle ou du flot de données de chaque module considéré.

10.4 Difficultés d'oracle et de mise en œuvre

Une difficulté majeure reste l'oracle car tous les critères présentés ne concernent que la sélection d'un sous-domaine des entrées possibles et ne fournissent pas une caractérisation des sorties attendues. Il est donc nécessaire d'analyser chaque test sélectionné en fonction des spécifications pour déterminer la sortie attendue.

Pour toutes les approches présentées, la sélection des chemins permet seulement de caractériser des domaines sur lesquels doit opérer la sélection. Ceci ne fournit pas directement des jeux de tests. Toufefois, il est possible d'engendrer automatiquement des jeux de tests à partir d'une description syntaxique de ces domaines exprimée par exemple sous la forme de grammaires BNF [Inc 87] (comme pour la sélection de jeux de tests fonctionnels).

Un autre problème majeur concernant la mise en œuvre de ces critères est l'existence de chemins infaisables, ce problème étant en général indécidable. Il est nécessaire alors d'analyser indépendamment chacun des chemins sélectionnés. Une proposition récente consiste à calculer une métrique probabiliste de faisabilité des chemins ; cette approche est toutefois encore fort simpliste puisque le seul travail existant sur ce sujet, [MYV 90],

revient simplement à considérer que plus un chemin fait intervenir de conditions, moins il a de chances d'être faisable ! Enfin, lors de la soumission proprement dite des jeux de tests, les problèmes habituels d'instrumentation, lanceurs, muets, etc. des tests unitaires et d'intégration restent entiers.

10.5 Critères applicables d'arrêt du test

Un avantage important de ces méthodes de test structurel est qu'elles fournissent de manière immédiate des critères d'arrêt du test, puisqu'il suffit d'atteindre le niveau de couverture des chemins préalablement choisi par le critère de sélection. En sus, le niveau de confiance du critère choisi peut bien sûr être évalué via du "test par mutation" (voir la fiche correspondante).

10.6 Conclusion et commentaire

Les critères de sélection présentés précédemment conduisent à un grand nombre de tests, mais cet aspect est compensé par le fait qu'ils soient généralement automatisables. Par contre, les tests ainsi sélectionnés ne sont guère réutilisables d'une version à une autre d'un même produit logiciel. En effet, lors d'une modification due à la maintenance ou à l'évolution d'un produit logiciel, le graphe de contrôle et le flot des données peuvent être modifiés, et il n'y a alors plus aucune garantie de couverture. Lors d'une modification, le jeu de tests doit évoluer en fonction des modifications correspondantes du graphe de contrôle ou du flot de données.

N.B. : même lorsqu'un programme a été testé avec une couverture maximale (tous les chemins), il peut néanmoins être incorrect. Ceci peut par exemple être dû à une omission dans le produit logiciel de l'une des fonctions définies dans les spécifications. Il est donc nécessaire de compléter cette approche par du test fonctionnel.

Les critères structurels ne permettent pas toujours de couvrir de façon satisfaisante le domaine des conditions intervenant dans les chemins sélectionnés. En effet, lorsqu'une condition fait intervenir ne serait ce qu'un "or", aucun des critères proposé n'impose de couvrir toute sa table de vérité. Cependant il existe des propositions pour pallier à ce problème dans [Mye 79] et [CF 84].

En conclusion, les méthodes de test structurel sont des méthodes de test cruciales qui, dans presque tous les cas, doivent faire partie de l'effort global de test.

10.7 Bibliographie

- [AB 81] D. Andrews, J.P. Benson
  "An automated program testing methodology and its implementation"
  Proc. 5^{th} Int. Conf. Soft. Eng., pp. 254-261, 1981.

- [Ban 86] Banff 1986
  Proc. Workshop on Soft. Testing, Verification and Analysis, Banff, Canada, IEEE Comp. Soc. num. 723, July 1986

- [CF 84] J.S. Collofello, A.F. Ferrara
"An automated Pascal multiple condition test coverage tool"
COMPSAC pp. 20-26 IEEE 1984.

- [Chu 87] T. Chusho
"Test data selection on quality estimation based on the concept of essential branches for path testing"
IEEE Trans. Soft. Eng. Vol-SE-13, num.5, May 1987.

- [Coo 76] D.W. Cooper
"Adaptative testing"
Proc. 2^{nd} Int. Conf. Soft. Eng., pp. 223-226, 1976.

- [Cow 88] P.D. Coward
"A review of software testing"
Inf. Soft. Techn., UK, Vol.30, num.3, April 1988.

- [CR 87] L.A. Clarke, D.J. Richardson
"Evaluation and integration of software testing techniques"
Proc. CIPS Edmonton'87: Intelligence Integration, Canada, Nov. 1987.

- [HS 91] M.J. Harrold, M.L. Soffa
"Selecting and using data for integration testing"
IEEE Software journal, special issue on Testing, Tracking design, Atomic Delegation, vol. 8, num. 2 (march 1991).

- [Inc 87] D.C. Ince
"The automatic generation of test data"
The Comput. Journal, vol. 30, num. 1, pp. 63-69 (1987).

- [Las 87] J.W. Laski
"On the comparative analysis of some data flow testing strategies"
Dep. Eng. Comp. Sci. Oakland Univ., Rochester, MI Technical Report 87-05, May 1987.

- [LK 84] J.W. Laski, B. Korel
"A data flow oriented program testing strategy"
IEEE Trans. Soft. Eng., Vol.SE-9, num 3, may 1983.

- [Mil 84] E.F. Miller
"Quality management technology: practical applications"
Proc. symposium Darmstadt, Germany, september 1983, software validation, H.L. Hausen ed., Elsevier Science Pub. B.V. (North-Holland) 1984.

- [Mul 88] M. Mullerburg
"On fundamentals of program testing"
Premier séminaire EOQC sur la qualité des logiciels, Bruxelles, 25-27 avril 1988.

- [Mye 79] G.J. Myers
  "The art of software testing"
  John Whily & Sons, London, 1979

- [MYV 90] N. Malevris, D. Yates, A. Veevers
  "Predictive metric for likely faisability of program path"
  Inf. Soft. Techn., UK, Vol.32, Num.2, March 1990.

- [Nta 88] S.C. Ntafos
  "A comparison of some structural testing strategies"
  IEEE Trans. Soft. Eng., Vol.SE-14, num 6, jun 1988.

- [RW 85] S. Rapps, E.J. Weyuker
  "Selecting software test data using data flow informations"
  IEEE Trans. Soft. Eng., Vol.SE-11, num 4, April 1985.

- [WHH 80] M.R. Woodward, D. Hedley, M.A. Hennell
  "Experience with path analysis and testing of programs"
  IEEE Trans. Soft. Eng., Vol. 6, num. 6, pp. 278-285, 1980.

- [Zei 84] S.J. Zeil
  "Perturbation testing for computation errors"
  Proc. 7^{th} Int. Conf. Soft. Eng., IEEE pp. 257-265, march 1984.

- [Zei 89] S.J. Zeil
  "Perturbation techniques for dectecting domain errors"
  IEEE Trans. Soft. Eng., Vol. 15, No 6, june 1989.

11 Test Aléatoire et Test Statistique

11.1 Principe général

La caractéristique de ces méthodes est que les jeux de test sont obtenus par tirage selon une loi de probabilité qui peut être :

- uniforme sur le domaine d'entrée du module/produit logiciel : on parle selon les auteurs de test aléatoire (random testing) [DN 84] ou statistique (statistical testing) [The 89] ;
- similaire à la distribution des données d'entrée lors de l'exploitation future : on parle alors aussi de test aléatoire [CDM 86, JLM 90] ;
- dépendante de la structure du programme pour assurer une couverture probabiliste de celle-ci selon certains critères (cf. les fiches sur le test structurel : couverture du graphe de contrôle, couverture du flot des données) : on parle alors de test statistique structurel (statistical structural testing) [The 90].

Par la suite, les termes suivants seront utilisés pour distinguer ces trois cas :

- test aléatoire uniforme
- test aléatoire opérationnel
- test statistique structurel

11.2 Variantes

11.2.1 Test aléatoire uniforme

Le test aléatoire uniforme a été étudié par Duran et Ntafos [DN 81, DN 84] qui ont mis en évidence des résultats expérimentaux surprenants : sur certains logiciels, ce type de test a un pouvoir de révélation des fautes excellent, et très mauvais sur d'autres.

Duran et Ntafos rapportent que des jeux de tests aléatoires de tailles modérées (de 20 à 120 tests), sur des programmes de petite taille (de 50 à 100 lignes) assurent en moyenne une couverture de 97% des instructions, de 93% des enchaînements et de 57% des chemins avec 0, 1 ou 2 passages par boucle. Les chemins non exécutés correspondent souvent aux traitements d'exceptions. Ce type de test doit donc être complété par des tests des valeurs aux/hors limites ou exceptionnelles.

Ces résultats ont été récemment corroborés par l'équipe de Thévenod-Fosse au LAAS [TWC 90]. Il n'y pas d'explication connue des auteurs de ce rapport à ces phénomènes, aussi bien pour les très bons résultats que pour les très mauvais. Une explication possible pourrait être : certains programmes sont d'une structure facilement testable en ce sens que tous les critères de test structurel coïncident ou presque. Dans ce cas, il semble qu'un tirage aléatoire peut assurer une bonne couverture selon tous ces critères, à condition de le compléter par des valeurs exceptionnelles. C'est ce que Duran et Ntafos ont constaté

sur certains programmes, et ce que l'équipe de Thévenod-Fosse a constaté sur deux programmes. Sur des programmes plus complexes, le test aléatoire uniforme ne donne pas de bons résultats [TWC 90, TW 91].

Les résultats du LAAS semblent montrer que le test statistique structurel (voir section 11.2.3) donne plus régulièrement de meilleurs résultats que le test aléatoire uniforme. Il faut cependant considérer tous ces résultats avec prudence, aussi bien ceux de Duran et Ntafos que ceux du LAAS, car le nombre de programmes étudiés est faible.

11.2.2 Test aléatoire opérationnel

Ce test a pour but essentiel d'évaluer la fiabilité en utilisant les modèles de croissance de fiabilité (par exemple celui de Musa [MIO 88] ou d'autres [SG 88]) et pourrait permettre de décider de l'arrêt du test quand le niveau de fiabilité estimé est suffisant.

L'utilisation de ce type de test a été faite dans le cadre du "Cleanroom Software Engineering" au moment de l'intégration ([CDM 86], [CDM 89]) strictement pour évaluer la fiabilité du logiciel obtenu : il n'y a pas de recherche ni de correction des fautes. Plus récemment, une équipe proche de Musa [ESW 90] et l'équipe de Littlewood [JLM 91] ont rapporté des expériences plus classiques, où l'estimation de la fiabilité est utilisée pour arrêter le processus de test et de correction. Il s'agissait du test de validation d'un gestionnaire de réseau. La méthode a permis d'arrêter le test lorsque l'estimation de la fiabilité, qui s'est avérée ultérieurement bonne, a atteint un niveau suffisant.

Dans [JLM 91] l'expérience a un caractère plus académique, puisqu'il s'agit d'un programme de taille moyenne, écrit pour l'expérience, sur lequel l'efficacité de plusieurs méthodes de test a été comparée. Une de ces méthodes consiste à faire uniquement du test aléatoire opérationnel : sur cet unique exemple, les résultats sont en faveur du test aléatoire opérationnel.

11.2.3 Test statistique structurel

Cette approche est récente. Elle est développée et experimentée au LAAS [The 89], [The 90], [TWC 90], [TW 91]. Elle combine des stratégies de choix des jeux de test structurelles et statistiques.

Un critère structurel C (par exemple : tous les enchaînements) est dit "couvert avec une probabilité qN" si chaque élément de C (par exemple chaque enchaînement) a une probabilité au moins qN d'être activé en N exécutions avec des données aléatoires.

Une étude comparative de l'efficacité de ce type de test par rapport au test structurel classique et au test aléatoire uniforme est rapportée dans [TWC 90]. Cette méthode a donné de meilleurs résultats que les deux autres pour ces expériences. Elle est à compléter par des test des valeurs aux limites, comme toutes les méthodes statistiques.

11.3 Phases d'application privilégiées

Les phases d'application privilégiées du test aléatoire ou statistique dépendent de la variante considérée :

- Pour le test aléatoire uniforme, les expériences concernent toutes du test unitaire. Il n'y a a priori pas de raison à cette limitation : la méthode est applicable pour le test d'intégration et le test de validation.

- Pour le test aléatoire opérationnel, il s'agit en général d'un test global (intégration ou validation) car il est souvent trop difficile de trouver la distribution des données d'entrées de chaque composant logiciel d'un système.

  Son avantage est de mettre plus facilement en évidence des fautes dont l'incidence est amplifiée par le profil opérationnel et qui peuvent se révéler plus difficilement par d'autres méthodes de test. Son utilisation se recommande donc, au moins, à la fin du test d'intégration. L'investissement se justifie du fait qu'on obtient une évaluation de la fiabilité du programme avant de le mettre en exploitation : c'est la seule méthode connue actuellement.

- Pour le test statistique structurel, les expériences ont été menées uniquement pour du test unitaire. D'autres sont envisagées pour le test d'intégration. Cependant, de par l'utilisation de critères structurels, cette méthode de test aléatoire n'est pas utilisable en validation.

11.4 Difficultés d'oracle et de mise en œuvre

Quelle que soit la variante de test aléatoire considérée, la décision du succès ou de l'échec d'un test (oracle) est difficile. En effet, lorsqu'un test est choisi au hasard, il n'y a aucune raison pour que l'on puisse facilement prévoir le résultat correct qui lui est associé.

Concernant les difficultés de mise en œuvre :

- Pour le test aléatoire uniforme, la sélection des jeux de test est facile à mettre en oeuvre : des outils de génération selon une loi de probabilité uniforme existent (par exemple, via une grammaire [BM 83]).

- Le test aléatoire opérationnel est difficile à mettre en oeuvre : il faut connaître la loi de probabilité des entrées en exploitation et se doter des outils de tirage aléatoire correspondants.

- Pour le test statistique structurel, dans [TW 91] les auteurs affirment que la sélection de jeux de test est plus facile que dans le cas structurel classique, mais rien n'argumente cette affirmation.

11.5 Critères applicables d'arrêt du test

Le test aléatoire ou statistique est la seule méthode permettant de mettre en œuvre des modèles de croissance de fiabilité qui permettent de décider l'arrêt du test quand le niveau

de fiabilité estimé est suffisant. Toutefois, la plupart de ces modèles exigent un ensemble de rapports de défaillance de taille importante pour donner de bonnes estimations. Ils sont donc impraticables en fin de phase de validation quand très peu de défaillances (et même aucune) se produisent. C'est donc un problème majeur pour la variante dite aléatoire opérationnelle qui s'applique essentiellement en validation.

De plus, le test statistique structurel permet d'évaluer la qualité d'un jeu de tests par rapport à des critères de test structurel.

11.6 Conclusion et commentaire

Sans objet.

11.7 Bibliographie

Test aléatoire uniforme

- [BM 83] D.L. Bird, C.U. Munoz
  "Automatic generation of random self-checking test cases"
  IBM Syst. J. (USA) vol.22, n°3, pp.229-245, 1983.

- [DN 81] J.W. Duran, S.C. Ntafos
  "A report on random testing"
  5th International Conference on Software Engineering, San Diego, pp 179-183 (March 1981).

- [DN 84] J.W. Duran, S.C. Ntafos
  "An evaluation of random testing"
  IEEE Transactions of Software EngineeringVolume SE-10, n° 4, pp. 438-444 (Juillet 1984).

- [Sil 88] M. Sillon
  "Une gamme d'outils pour le test du logiciel"
  Génie Logiciel et Systèmes Experts n°12, pp.54-57, (Juin 1988).

Test aléatoire opérationnel et modèles de croissance de fiabilité

- [CDM 86] P.A. Currit, M. Dyer, H.D. Mills
  "Certifying the reliability of software"
  IEEE Transactions of Software Engineering, Volume SE-12, n° 1, pp. 3-11 (Janvier 1986)

- [CDM 89] P.A. Currit, M. Dyer, H.D. Mills
  "Correction to Certifying the reliability of software"
  IEEE Transactions of Software Engineering, Volume SE-15, n° 3, p. 362 (Mars 1989).

- [ESW 90] W.K.Ehrlich, J.P.Stampfel, J.R. Wu
"Application of software reliability modeling to product quality and test process"
12th ACM-IEEE International Conference on Software Engineering, Nice, pp. 108-116 (Avril 1990).

- [JLM 91] A. Jassim, B. Littlewood, P. Mellor
"Random testing compared with structural testing"
à paraître

- [MIO 87] J. D. Musa, A. Iannino, K. Okumoto
"Software Reliability : Measurement, Prediction, Application"
McGraw-Hill Series in Software Engineering and Technology (1987).

- [SG 88] P. Saunier, E. Girard
"Des Modèles de Fiabilité du Logiciel? Pourquoi faire?"
Revue Génie Logiciel, n^o 10, EC2-Editeur, pp 50-62 (Janvier 1988).

Test aléatoire structurel

- [The 89] P. Thévenod-Fosse
"Software validation by means of statistical testing : retrospect and future direction"
1st IFIP Working Conference on Dependable Computing for Critical Applications, Santa Barbara, pp. 15-22 (Août 1989).

- [The 90] P. Thévenod-Fosse
"On the efficiency of statistical testing wrt software structural test criteria"
IFIP Working Conference on Approving Software Products (ASP-90), Garmisch-Partenkirchen, pp. 29-42 (Septembre 1990).

- [TWC 90] P. Thévenod-Fosse, H. Waeselynk , Y. Crouzet
"An experimental study on software structural testing : deterministic versus random input generation, Rapport LAAS n^o 90.387 (Novembre 1990)."

- [TW 91] P. Thévenod-Fosse, H. Waeselynk
"An investigation of software statistical testing"
Rapport LAAS n^o 91.003 (Janvier 1991).

12 Test Fonctionnel

12.1 Principe général

Le but du test fonctionnel est de sélectionner des tests à partir d'une spécification (souvent STB mais parfois aussi à partir des spécifications détaillées) et de vérifier si le produit logiciel fournit des résultats corrects vis à vis de cette spécification. Il peut être employé aussi bien pour tester un nouveau produit logiciel que pour tester une nouvelle version d'un produit logiciel. Dans ce dernier cas il s'agit de test de non-régression. Des méthodes similaires peuvent aussi être employées en phase de conception préliminaire pour vérifier les spécifications détaillées par rapport à la STB.

Partir des spécifications impose tout d'abord d'identifier les fonctionnalités requises. Par conséquent, il est nécessaire de structurer les spécifications de façon adéquate. Ensuite, le test fonctionnel permet de vérifier si leurs fonctionnalités sont effectivement disponibles. En phase de sélection, le produit logiciel sous test (i.e. le "comment") n'est pas pris en compte mais seulement ses spécifications (i.e. le "quoi"). Pour cette raison, le test fonctionnel est aussi appelé du test "boîte noire" ("black-box testing" en anglais).

On observe actuellement nombre d'avancées vers la formalisation systématique de l'expression des fonctionnalités requises. Ceci permet d'envisager, dans un avenir éventuellement proche, une approche beaucoup plus systématique du test fonctionnel où des règles de dérivation de jeux de tests pourront être définies à partir d'une documentation de conception plus formelle ([WO 80], [OW 86]). La pratique actuelle du test fonctionnel est en fait fondée sur la compréhension qu'a le testeur des fonctionnalités requises par la spécification. Cette spécification est souvent rédigée en langage naturel et peut donc être ambiguë. A partir de cette compréhension (souvent facilitée par l'expertise des équipes concernées) des jeux de tests sont définis, ainsi que les résultats attendus (ou des ensembles de valeurs plausibles).

La pratique la plus rencontrée du test fonctionnel considère des spécifications en langage naturel. Dans le cas où le test fonctionnel est appliqué méthodiquement, une matrice de test est dérivée de la STB. Cette matrice contient pour chaque fonctionnalité de la STB des cas de test nominaux, aux limites et hors limites. Elle décrit aussi les développements nécessaires à la soumission et à l'interprétation des tests.

Une composante importante du test fonctionnel est l'oracle. Rappelons qu'un oracle est une procédure capable de décider du succès ou de l'échec d'un test. Souvent, on ne dispose que d'heuristiques relativement imprécises, comme pour toutes les autres techniques de test.

12.2 Variantes

12.2.1 Partition des domaines

Cette approche consiste en une classification des exigences de la spécification en sous-domaines des entrées possibles. Cette classification peut être fondée sur les fonctions requises ou sur les données d'entrées.

Bien qu'une partition des domaines puisse être effectuée de manière empirique à partir d'une spécification informelle, cette approche prend toute sa puissance lorsqu'elle est appliquée à des spécifications formelles. Il devient alors possible d'utiliser des notions d'équivalence, fournies par le formalisme, pour guider la partition du domaine en classes d'équivalence. Dans [WO 80], un calcul des prédicats simplifié est utilisé. Il permet de décomposer la relation entre les entrées et les résultats d'un calcul. Ceci permet de définir plusieurs domaines d'entrées disjoints et de spécifier les valeurs possibles de sortie correspondantes. Ceci n'est pas sans rappeler les techniques de décomposition des domaines issues des chemins en test structurel (voir fiche correspondante). Néanmoins, la plupart du temps, la décomposition obtenue diffère de celle obtenue en test structurel. Une idée intéressante serait de considérer une décomposition plus fine obtenue par combinaison des domaines issues des techniques fonctionnelles et structurelles. Une telle idée a été développée dans [CR 84] où l'on utilise simultanément la partition issue de la spécification et celle issue du produit logiciel pour sélection des jeux de test. Pour obtenir ces partitions, [CR 84] exploite des techniques d'évaluation symbolique. On rejoint largement ici les idées suggérées dans la fiche sur le test boîte grise.

12.2.2 Graphes cause-effet

La puissance des graphes cause-effet résulte de leur capacités à explorer les combinaisons d'entrées possibles. On utilise pour cela un graphe combinatoire logique assez comparable à la spécification d'un circuit. Ce graphe n'utilise que des opérateurs booléens logiques comme "and", "or" et "not". La construction d'un tel graphe suit généralement les étapes suivantes : [Mye 79]

- découper la spécifications en pièces élémentaire représentant une transaction individuelle ; cette étape est nécessaire car un graphe cause-effet pour un système entier serait beaucoup trop grand pour être utilsable ;
- identifier les causes et les effets : une cause est un stimuli d'entrée, par exemple une commande sur un terminal ; un effet est une réponse en sortie ;
- construire un graphe reliant les causes et les effets de manière compatible avec la sémantique de la spécification (le graphe cause-effet) ;
- annoter le graphe pour exhiber les effets impossibles ainsi que les combinaisons impossibles de causes ;
- convertir le graphe en une table de décision avec entrées limitées (par les annotations) ; les conditions représentent les causes, les actions représentent les effets et les cases de la table représentent les cas de test (les cases se trouvant à l'intersection d'une cause et d'un effet impossible à partir de cette cause sont vides : ce sont là les limitations pré-citées).

Au vu de la description précédente, on peut penser que la construction du graphe est une étape inutile et que l'on pourrait construire directement la table. En fait, pour des systèmes importants il n'est pas envisageable de construire cette table directement.

12.2.3 Spécifications formelles ou semi-formelles

C'est seulement avec les spécifications formelles que des solutions systématiques pour le test fonctionnel sont devenues possibles. De nombreuses recherches relativement récentes sur ce sujet ont été effectuées. Des expériences ont été menées par exemple avec des réseaux de Petri au LAAS en liaison avec CIT-Alcatel, ou avec des diagrammes SADT chez IGL, etc. Dans tous les cas, ces travaux développent des critères de couverture liés aux spécifications. Toutefois, du fait de la grande variété de langages de spécifications formelles, chacun des travaux utilise son propre formalisme. Il est donc très difficile de les comparer. Nous nous contenterons donc de quelques commentaires introductifs.

Une des premières tentatives fondées sur des spécifications relativement simples est décrite dans [WO 80]. On utilise alors des spécifications exprimées dans un simple calcul des prédicats. Ces spécifications peuvent fournir une partition du domaine des entrées de manière relativement simple (cf. section 12.2.1). Il n'est pas clair que cette approche puisse être généralisée, en particulier à des formules avec quantificateurs.

Une autre approche un peu plus ancienne est décrite dans [Cho 78]. Elle utilise des spécifications réduites à des machines à états finis et suppose de plus que le produit logiciel sous test se comporte comme une machine à états finis. On constate alors que des méthodes de test très complètes (allant jusqu'à démontrer la correction) peuvent être obtenues avec un nombre de tests beaucoup plus réduit que l'explosion combinatoire inhérente aux machines à états finis pourrait le laisser croire. Ces machines peuvent clairement être appliquées à certains types de logiciels tels que les protocoles ou les analyseurs lexicaux, mais cette méthode reste néanmoins assez limitée et ne peut pas fournir une base pour des méthodes plus générales. Le problème majeur est en effet que cette méthode n'est applicable qu'à des produits logiciels se comportant comme des machines à états finis.

Deux approches plus récentes ([Car 81], [RC 81]), suivent les travaux de [WO 80] pour dériver une partition du domaine des entrée à partir d'une spécification. Tous deux proposent de nouveaux langages de spécification, qui sont en fait des langages de programmation applicatifs de haut niveau. [RC 81] effectue ensuite simplement une sélection de jeux de tests reposant sur des critères de couverture structurels classiques des spécifications. [Car 81] propose d'utiliser des méthodes d'évaluation symbolique sur les spécifications afin de partitionner le domaine des entrées aux points de branchements dans l'évaluation.

Les meilleurs résultats sont en fait obtenus à partir de spécifications écrites dans un style déclaratif ([Bri 89], [Wil 88], [Den 91]) (et non pas applicatif), en particulier les spécifications algébriques ([GMH 81], [Ost 85], [JC 88], [BGM 91]).

Les travaux décrits dans [Wil 88] et [Den 91] utilisent des spécifications qui sont en fait des programmes logiques. Ils montrent comment utiliser la procédure de résolution d'un langage de programmation logique pour sélectionner des cas de test. Une telle procédure fournit des valeurs satisfaisant une certaine contrainte, tout en prenant en compte la spécification donnée par un programme logique. Ceci permet de sélectionner automatiquement des cas de test à partir de la donnée, d'une part de la spécification logique, et

d'autre part des sous-domaines que l'on veut considérer. Malheureusement aucune indication sur la façon d'extraire les sous-domaines intéressants n'est fournie.

Dans [Bri 89], des spécifications de protocoles décrites en LOTOS sont utilisées pour dériver, à l'aide de stratégies spécifiques, des interactions typiques permettant de tester le protocole spécifié. La même critique que pour les approches fondées sur des machines à états finis s'applique : ces méthodes ne sont pas généralisables à d'autres types de produits logiciels.

Enfin dans les approches fondées sur les spécifications algébriques, on utilise les noms de fonctions déclarés dans la spécification des types abstraits définis. Les propriétés de ces fonctions sont définies par des axiomes déclaratifs (souvent dans le cadre de la logique du premier ordre). Pour certaines approches, seuls les noms d'opération, ainsi que le profil de leurs arguments, sont exploités pour construire une description des tests à soumettre sous forme de termes (i.e. de compositions successives licites de ces fonctions). C'est le cas de l'approche décrite dans [JC 88]. Une limitation majeure de telles approches est que les propriétés des fonctions testées ne sont pas utilisées pour sélectionner les jeux de tests, mais seulement pour aider à la décision de succès/échec (oracle) des tests soumis (cf. [GMH 81]). Il en résulte une couverture assez pauvre des différents cas d'application des fonctionnalités car on n'obtient en fait qu'une couverture syntaxique des fonctionnalités requises par les spécifications.
Une meilleure approche consiste à utiliser également les axiomes afin de caractériser des sous-domaines judicieux du domaine des entrées des fonctions spécifiées. Cette caractérisation peut faire appel à des techniques proches de l'évaluation symbolique. Dans [BGM 91] on utilise des mécanismes issus de la programmation logique et de la démonstration automatique pour sélectionner automatiquement des jeux de tests à partir d'une spécification algébrique. De plus, cette approche prend en compte le problème de l'oracle dès la phase de sélection. En effet, des critères d'observabilité (à propos du produit logiciel) sont utilisés de façon à ne sélectionner que des cas de tests dont la décision succès/échec est immédiate. De telles approches devraient permettre à terme de concevoir des techniques fonctionnelles de sélection de jeux de test (i.e. à partir de spécification) qui soient aussi élaborées que celles, mieux connues, du test structurel (i.e. à partir des sources du produit logiciel).

12.2.4 Hyper-text

Les travaux de [WO 80] et [OW 86] montrent comment l'utilisation de spécifications en langage naturel structurées sous forme d'une documentation de conception systématique (forme d'hyper-text) peut permettre d'engendrer automatiquement des scénarii de tests fonctionnels. Les règles utilisées ne prennent pas en compte des notions de classes de fautes probables. Une avancée prochaine de ces travaux est une méthode formelle de documentation qui inclue une description de fautes associées avec chaque partie de la conception ainsi que les caractéristiques de conception elles-mêmes. Dans [How 81], l'auteur soutient qu'il ne suffit pas d'identifier des classes de fautes pour chaque partie de la conception. Il faut aussi isoler les propriétés particulières de chaque fonction, chaque propriété possédant elle aussi certaines classes de fautes associées. Notons qu'il existe de très nombreuses classes

(et classifications !) de fautes que l'on peut trouver dans [Van 78] et [Cha 79].

12.3 Phases d'application privilégiées

Le test fonctionnel est bien sûr plus particulièrement utilisé lors de la phase de validation, et éventuellement en phase d'intégration lorsque les modules à intégrer remplissent pleinement certaines des fonctionnalités requises. Certains tests fonctionnels peuvent parfois être utilisés dès la phase de test unitaire (par rapport aux spécifications détaillées).

12.4 Difficultés d'oracle et de mise en œuvre

Le problème de l'oracle reste difficile dans le cadre du test fonctionnel. Même lorsque l'usage de spécifications permet de prévoir le résultat attendu d'un test, il n'en demeure pas moins qu'il n'existe pas de mécanisme général pour décider si un résultat obtenu est égal (abstraitement) au résultat attendu. Dans le cas de test de protocoles ce problème peut être abordé de façon satisfaisante dans la mesure où l'observation du produit logiciel est très bien définie. Pour d'autres types de produits logiciels, il peut être très difficile de définir exactement la façon d'observer les résultats obtenus (signaux, images, etc).

Hormis pour les approches fondées sur des spécifications formelles, la sélection de jeux de tests reste un problème non trivial. En effet, certaines des approches présentées permettent seulement de caractériser des domaines sur lesquels doit opérer la sélection. Ceci ne fournit pas directement des jeux de tests. Toufefois, il est possible alors d'engendrer automatiquement des jeux de tests à partir d'une description syntaxique de ces domaines exprimée par exemple sous la forme de grammaires BNF (comme pour la sélection de jeux de tests structurels) [Inc 87].

Enfin, lors de la soumission proprement dite des jeux de tests, les problèmes habituels d'instrumentation, lanceurs, muets, etc. en phase de test unitaire et d'intégration restent entiers.

12.5 Critères applicables d'arrêt du test

Lorsque la méthode de sélection de jeux de tests fonctionnels considérée reposent sur des notions de couverture de spécifications, ces notions de couverture constituent elle-mêmes des critères d'arrêt du test. Que l'on dispose ou non d'une notion de couverture, il est possible d'évaluer le niveau de confiance des tests soumis (et donc la notion de couverture choisie lorsqu'elle existe) via du "test par mutation" (voir la fiche "arrêt du test").

12.6 Conclusion et commentaire

Le champs d'application du test fonctionnel n'est pas nécessairement restreint au seul test dynamique du produit logiciel ; il peut aussi être utilisé avec fruits pour "tester" une spécification détaillée par rapport à une spécification de niveau supérieur (par exemple la STB). Cette démarche est en fait souvent implicitement employée dans les méthodologies de conception. Une pratique plus systématique des spécifications formelles (ou semi-formelles) devrait en fait permettre de mieux appréhender la conception de logiciel, en

autorisant par exemple la définition rapide de prototypes ou maquettes, une assistance automatique ou semi-automatique à la sélection de jeux de tests fonctionnels aussi bien pour la conception que pour le développement.

Un obstacle majeur à l'heure actuelle pour une telle approche est le manque de standard de spécifications formelles. Il en résulte que les résultats obtenus jusqu'à maintenant, et décrits ci-dessus, sont quelque peu hétérogènes, difficilement comparables, et qu'il convient de choisir soigneusement le type de spécifications selon le problème traité. Pour des applications industrielles, un mauvais choix pourrait être catastrophique.

N.B. : avoir testé que le produit logiciel satisfait correctement toutes les fonctions spécifiées n'implique pas qu'il satisfasse à tous les critères requis pour sa qualification. En effet, des critères tels que le temps de réponse, les standards de programmation, etc. ne peuvent être vérifiés par du test fonctionnel. De plus, certaines fonctionnalités annexes non requises par les spécifications (mais programmées pour réaliser les fonctionnalités requises) ne sont pas couvertes de façon systématique par le test fonctionnel et il est donc nécessaire de compléter cette approche par d'autres approches du test dynamique (et même du test statique).

12.7 Bibliographie

- [BGM 91] G. Bernot, M.C. Gaudel, B. Marre
  "Software testing based on formal specifications: a theory and a tool"
  Software Eng. Journal, (nov. 1991).

- [Bri 89] E. Brinksma
  "A theory for the derivation of tests"
  Book. The formal description technique LOTOS, P.H.J. Van Eijk, C.A. Vissers & M. Diaz Eds., Elsevier Sci. Pub. B.V. (North-Holland), 1989.

- [Car 81] R. Cartwright
  "Formal program testing"
  Proc. 8^{th} Annual ACM Symp. Principles of Programming Languages, 1981.

- [Cha 79] J. Chan
  "Program debugging methodology"
  Phd thesis, Leicester polytechnic, 1979.

- [Cho 78] T.S. Chow
  "Testing software design modeled by finite-states machines"
  IEEE Trans. Soft. Eng., vol. SE-4, may 1978.

- [CR 84] L.A. Clarke, D.J. Richardson
  "Symbolic evaluation - an aid to testing and verification"
  Book Software Validation, H.L. Hausen Ed., Elsevier Science Pub. B.V. (North-Holland), pp. 141-166, 1984.

- [Den 91] R. Denney
  "Test-case generation from prolog-based specifications"

IEEE Software journal, special issue on Testing, Tracking design, Atomic Delegation, vol. 8, num. 2 (march 1991).

- [GMH 81] J. Gannon, P. Mac Mullin, R. Hamlet
"Data-abstraction, implementation, specification and testing"
ACM-TOPLAS 3, 3, pp. 211-223 (july 1981).

- [How 81] W.E. Howden
"Errors, design properties and functional program tests"
Computer program testing, Chandrasekaran B. and Radichi S. eds., North-Holland 1981.

- [Inc 87] D.C. Ince
"The automatic generation of test data"
The Comput. Journal, vol. 30, num. 1, pp. 63-69 (1987).

- [JC 88] P. Jalote, M.G. Caballero
"Automated test case generation for data abstraction"
Proc COMPSAC 88, The twelfth Int. Comp. Soft. & Applications Conf., Chicago Il. USA, pp. 205-210, IEE Comp. Soc. Press. Washington DC, USA, (oct 88).

- [Mye 79] G.J. Myers
"Errors, design properties and functional program tests"
Computer program testing, Chandrasekaran B. and Radichi S. eds., North-Holland 1981.

- [Mye 79] G.J. Myers
"The art of software testing"
John Whily & Sons, London, 1979.

- [Ost 85] T.J. Ostrand
"The use of formal specifications in program testing"
Third Int. Workshop on Software Specification and Design, London, UK, IEEE Comp. Soc. Press, Washington, DC, USA, pp. 253-255 (august 1985)

- [OW 86] T.J. Ostrand, F.J. Weyuker
"Design for a Tool to Manage Specification-Based Testing"
Proc. Workshop on Software Testing, Verification and Analysis, Banff, Canada, IEEE Comp. Soc. num. 723, July 1986.

- [RC 81] D.J. Richardson, L.A. Clarke
"A partition analysis method to increase program reliability"
Proc. 5^{th} IEEE Int. Conf. Soft. Eng., 1981.

- [Van 78] D. Van Tassel
"Program style, design, efficiency, debugging and testing"
Book, Prentice Hall 1978

- [Wil 88] C. Wild
"The use of generic constraint logic programming for software testing and analysis"
Proc. 2^{nd} ACM-IEEE Workshop on Software Testing, Verification and Analysis, Banff, Canada, july 1988.

- [WO 80] F.J. Weyuker, T.J. Ostrand
"Theories of program testing and the application of revealing subdomains"
IEEE Trans. Soft. Eng., vol. SE-6, num. 3, pp. 236-246, 1980.

13 Test Boîte Grise

13.1 Principe général

Le test "boîte grise", en anglais "Grey-Box Testing", est un vocable introduit par R. L. Probert [Pro 82]. Il ne s'agit pas à proprement parler d'une nouvelle méthode de test, mais simplement d'une "idée" de l'auteur qui considère que le découpage habituel *white-box* v.s. *black-box* testing est trop restrictif, et ne reflète pas de possibles interactions ou mélanges entre ces approches. Nous ne fournissons donc cette fiche que parce que le terme "grey-box" peut être occasionnellement rencontré dans la littérature sur le test.

Contrairement à ce que peut laisser croire cette terminologie, le test boîte grise n'est pas une méthode de test intermédiaire entre le test "boîte blanche" et le test "boîte noire", mais plutôt une union des deux.

Plus généralement, l'auteur considère que toute classification des méthodes de test conduit à des restrictions non souhaitables des techniques utilisées. Par exemple, comme déjà mentionné dans le rapport descriptif, il faut considérer toutes les méthodes de test comme complémentaires, et il n'est pas question de privilégier l'une ou l'autre de ces méthodes. En bref, toutes les classifications habituelles telles que statique/dynamique, fonctionnel/structurel/aléatoire, unitaire/intégration/validation sont remises en cause.

Cette approche n'est pas nécessairement entièrement destructive : elle argumente par exemple que le découpage entre test unitaires, d'intégration et de validation ne reflète pas le fait que certains tests sélectionnés à partir d'une STB peuvent être appliqués dès la phase de test unitaire, et que par conséquent, penser en ces termes peut nuire a une bonne souplesse du processus de test ; la structure rigide d'un tel découpage ne permettant pas de détecter au plus tôt certains défauts.

13.2 Variantes

Diverses instances de cette idée peuvent être citées. Il a déjà été mentionné que certains tests unitaires peuvent être sélectionnés à partir de la STB.

Une autre instance, en quelque sorte en "sens contraire", de cette interdépendance entre les diverses phases de test est l'usage des connaissances acquises lors des tests unitaires sur les graphes d'appels entre les interfaces des modules pour sélectionner des tests d'intégration.

Un autre exemple d'application de ce principe est la variante dite de test "structurel aléatoire" décrite dans la fiche sur le test aléatoire, où des connaissances probabilistes sur les chemins obtenus par le test structurel sont utilisées pour guider le test aléatoire.

De plus, des prototypes issus des STB peuvent être utilisés pour faire du test "dos à dos" lors des phases de test d'intégration ou de validation.

Bien d'autres instances peuvent être imaginées, telles que l'usage d'informations issues du test statique pour sélectionner des tests dynamiques structurels, etc.

13.3 Phases d'application privilégiées

Toutes les phases de test sont donc bien sûr concernées par le test boîte grise.

13.4 Difficultés d'oracle et de mise en œuvre

Ces difficultés sont propres à chaque instance considérée, mais le mélange de plusieurs techniques peut éventuellement compenser les défauts propres à chacune des techniques si elles étaient appliquées isolément.

13.5 Critères applicables d'arrêt du test

Même remarque que pour la section précédente.

13.6 Conclusion et commentaire

Le test "boîte grise" pourrait peut-être être plus simplement appelé "test de logiciel" en général...

L'avantage d'une telle approche est de ne pas confiner artificiellement le testeur dans une technique particulière, avec les problèmes qui lui sont typiques, alors que des techniques issues d'autres méthodes de test pourraient les résoudre, au moins partiellement. Par contre, une telle approche risque de complexifier l'activité de certification, déjà fort difficile à dominer, et contredire ainsi le proverbe "diviser pour régner".

13.7 Bibliographie

- [Pro 82] R.L. Probert
  "Grey-Box (Design-Based) Testing techniques "
  Proc. 15th Hawaii Int. Conf. on System Sciences, 1982, Honolulu, HI, USA, vol.1, (jan. 1982).

14 Test par mutation

14.1 Principe général

Le but du test par mutation n'est pas directement d'engendrer des jeux de tests, ni de démontrer la correction du produit logiciel ; il s'agit en fait d'évaluer la qualité d'un jeu de tests existant. Les autres formes de test utilisent les jeux de test pour tester un produit logiciel ; le test par mutation utilise le produit logiciel pour tester un jeu de tests...

Le test par mutation crée de nombreux composants logiciels presque identiques afin d'évaluer dans quelle mesure le composant sous test a été correctement testé. Un seul changement du composant original est effectué par "mutant". Chacun des mutants, ainsi que le composant original, sont exécutés avec le *même* jeu de tests. Les sorties du composant original sont alors successivement comparées avec celles de chacuns des mutants. Si les sorties diffèrent pour un mutant particulier, alors il présente un intérêt assez faible puisque le jeu de tests a découvert une différence avec le composant original. Ce mutant est alors dit "mort" et ignoré par la suite. Un mutant produisant les mêmes sorties que le composant original est intéressant car la modification n'a pas été détectée. Celui-ci est alors dit "vivant".

Une fois que les sorties de tous les mutants ont été examinées, il est aisé de calculer la proportion entre les mutants morts et vivants. Une forte proportion de mutants vivants indique un jeu de tests trop pauvre, un jeu de test complémentaire doit être ajouté. Le processus d'évaluation est alors répété jusqu'à obtenir une faible proportion de mutants vivants, indiquant que le composant a été correctement testé.

Le test par mutation repose principalement sur l'hypothèse que si le jeu de tests est capable de découvrir les modifications élémentaires des mutants, alors il est capable de découvrir des fautes plus graves dans le composant original. Par conséquent, si un jeu de tests ne découvre aucune faute dans le composant original, mais est capable de tuer une forte proportion de mutants, alors le composant sous test peut être raisonnablement supposé correct. L'hypothèse précitée n'est crédible que si l'hypothèse, plus connue, dite de "compétence du programmeur" est admise. En effet, le test par mutation présuppose que le composant sous test est relativement proche de la correction (donc, que les fautes restantes sont relativement mineures, comparables à celles introduites par mutation).

14.2 Variantes

Les variantes du test par mutation reposent essentiellement sur les critères appliqués pour engendrer l'ensemble des mutants. Le principe décrit précédemment concerne des mutations élémentaires applicables sur chaque instruction. Ceci conduit à un nombre de mutants souvent trop élevé. Pour cette raison il est parfois appelé "mutation forte" [BDLS 80], [Gal 84], [DGMOK 88]. Il existe des techniques de mutation moins coûteuses, en particulier la mutation dite faible de [How 82], où le composant logiciel est lui même découpé en plusieurs parties sur lesquelles les mutations sont appliquées indépendamment. Ceci permet d'exécuter plusieurs mutations à la fois (une sur chaque partie). Un problème important de cette approche (hormis la nécessité de découper le composant logiciel) est

que certaines mutations peuvent en "réparer" d'autres, laissant ainsi plus de mutants vivants qu'en considérant individuellement les parties mutées.

14.3 Phases d'application privilégiées

A cause de l'explosion combinatoire du nombre des mutants engendrés, il est très rarement envisageable de faire du test par mutations sur des produits logiciels entiers. La méthode n'est en fait applicable qu'à des composants logiciels. Le test par mutation est donc essentiellement utilisé pour évaluer les jeux de tests issus du test unitaire.

14.4 Difficultés d'oracle et de mise en œuvre

Une difficulté majeure du test par mutation est que certains mutants peuvent être équivalents au composant original [DGMOK 88], [WH 88]. En effet, bien que le mutant soit textuellement différent de l'original, il peut produire exactement les mêmes sorties. Le test par mutation va alors abusivement considérer ce mutant comme vivant bien qu'il ne puisse exister aucun test permettant de le tuer. La difficulté repose sur le fait qu'il est généralement impossible de décider l'équivalence de deux composants et que par conséquent ceci ne peut pas être pris en compte dans le calcul de la proportion entre mutants morts et vivants.

Une autre difficulté tout aussi importante est que selon les opérateurs de mutation considérés (c.a.d. le choix des instructions à modifier et la nature de ces modifications), le test par mutation peut produire un nombre très élevé de mutants (jusqu'à un facteur quadratique du nombre d'instructions en cas de mutation de variables). Ceci pose un problème de consommation de ressources aussi bien humaines que matérielles.

14.5 Critères applicables d'arrêt du test

Ce processus d'évaluation de jeux de tests est stoppé lorsque la proportion de mutants morts est jugée suffisante.

14.6 Conclusion et commentaire

La mutation permet de décider l'arrêt du test par un critère quantitatif de qualité d'un jeu de tests (à savoir la proportion entre mutants morts et vivants). Bien qu'elle puisse être coûteuse si l'on engendre trop de mutants, elle présente l'avantage d'être très simple dans son principe et c'est la seule approche applicable à toutes les méthodes de test.

14.7 Bibliographie

- [BDLS 80] T.A. Budd, R.A. Demillo, R.J. Lipton, F.G. Sayward
  "Theoretical and empirical studies on using program mutation to test the functional correctness of programs"
  Proc. ACM Symp. Principles of Prog. Lang., pp. 220-222, 1980.

- [BL 78] T.A. Budd, R.J. Lipton
"Mutation analysis of decision table programs"
Proc. Conf. Information Science and Systems, pp. 346-349, 1978.

- [DGMOK 88] R.A. Demillo, D.S. Guindi, W.M. McCracken, A.J. Offutt, K.N. King
"An extended overview of the Mothra software testing environment"
Proc. 2^{nd} ACM SIGSOFT workshop on software testing verification and analysis, Banff, Canada, pp. 142-151, july 1988.

- [Gal 84] J.M. Galvin
"Mutation analysis : a user's view"
Proc. 7^{th} annual MICRO-DELCON'84, the Delaware Bay Comp. Conf., IEEE Comp. Soc. Press Silver Spring, pp. 30-40, march 1984

- [How 82] W.E. Howden
"Weak mutation testing and completeness of test sets"
IEEE Trans. Soft. Eng., Vol. SE-8, No. 4, pp. 371-379, july 1982.

- [WH 88] M.R. Woodward, K. Halewood
"From weak to strong, dead or alive ? analysis of some mutation testing issues"
Proc. 2^{nd} ACM SIGSOFT workshop on software testing verification and analysis, Banff, Canada, pp. 152-158, july 1988.

15 Les Preuves de Développement de Logiciel

15.1 Principe général

Les preuves sont utilisées depuis toujours pour démontrer la validité de propriétés en mathématique et dans tous les domaines scientifiques. En ce qui concerne le développement de logiciel, elles peuvent être utilisées à toutes les phases de ce développement, à diverses fins. Par exemple, il est possible de prouver :

- une certaine propriété d'une spécification,
- qu'une étape de conception est correcte: une spécification détaillée satisfait une spécification moins détaillée,
- qu'un programme est correct par rapport à une spécification détaillée (preuve de programme),
- qu'un programme termine toujours sous certaines conditions,
- etc.

La forme des preuves peut être *classique*, du type de ce qu'on trouve dans des textes scientifiques habituels, ou *formelle*, c'est-à-dire que la preuve est décomposée en pas élémentaires, appelés *inférences*, qui correspondent à des régles d'inférences d'un *système formel*. Une preuve classique est une présentation abrégée d'une preuve formelle. Pour plus de précisions sur les notions d'inférences et de systèmes formels, se reporter à l'annexe I du rapport final de contrat CNES [LCGP 88] sur la sûreté de fonctionnement des systèmes informatiques.

Les preuves classiques sont validées par consensus: ces preuves sont lues et acceptées par un auditoire compétent; les preuves formelles peuvent être vérifiées par un système informatique (proof checker).

Une preuve peut être établie à la main, c'est-à-dire en la rédigeant sur papier sans assistance d'un système informatique. Dans ce cas, il est fréquent de se limiter à des preuves classiques car les preuves formelles sont très fastidieuses à établir. Il est aussi possible d'utiliser un système (proof assistant) qui prend en charge les parties triviales de la démonstration. Dans ce cas les preuves sont forcément formelles, le système étant l'implémentation d'un système formel: il vérifie que toute étape de la preuve correspond à une régle d'inférence.

Les preuves font partie des méthodes de test statique: il s'agit d'un examen du texte d'une spécification ou d'un programme, sans exécution du système correspondant. La plupart des systèmes d'assistance à la preuve de programmes font partie, ou sont couplés avec, des systèmes d'analyse statique de programmes.

Dans la prochaine section, cette fiche présente brièvement différentes techniques de preuves applicables lors du développement de logiciel. Quelques outils d'assistance et de vérification de preuves sont décrits dans la troisième section. La quatrième section précise à quelles phases du développement sont applicables ces méthodes et ces outils. Enfin, plusieurs exemples d'utilisation de preuves dans des projets industriels sont décrits.

15.2 Variantes

Le cadre habituel des preuves est le *calcul des prédicats du premier ordre*: ce cadre permet de raisonner sur des formules qui comportent des variables. Des exemples de telles formules sont:

$$(i < j \wedge j < k) \Rightarrow i < k$$
$$\exists x,\ x \neq 0$$
$$\forall x,\ x + 1 > 0$$
$$\forall p, d, \exists q, r,\ (p = q \times d + r\ \wedge\ 0 \leq r < d)$$

Pour pouvoir raisonner sur des développements de logiciel, ce calcul a du être enrichi de diverses manières: par l'introduction de types de données (types abstraits algébriques, ...), par l'introduction de nouveaux symboles permettant d'exprimer des contraintes temporelles (logiques temporelles), par le passage à un ordre supérieur (possibilité d'avoir des variables qui désignent des fonctions ou des prédicats: voir ci-dessous la description du système HOL), par la possibilité d'associer des propriétés à des morceaux de programmes (assertions de Floyd, logique de Hoare, ...). Le choix du cadre formel d'une preuve dépend du type de logiciel considéré et des propriétés à démontrer: il faut pouvoir exprimer ces propriétés; cela revient à dire qu'il faut choisir un *langage de spécification* formelle adéquat, et si l'on veut faire de la preuve de programme, un *langage de programmation* pour lequel un système de preuve a été établi [LCGP 88].

15.2.1 Réécritures, récurrences, généralisations

Comme cela a été dit plus haut, les langages de spécification utilisés pour exprimer les propriétés d'un programme sont tous plus ou moins basés sur le calcul des prédicats du premier ordre: certains en sont des restrictions (par exemple les clauses de Horn, ce qui permet d'automatiser certains raisonnements en utilisant des systèmes à la Prolog), d'autres des extensions (voir plus haut). Le but de cette section est de présenter sommairement les différentes techniques de preuves utilisables dans ce cadre.

- La réécriture consiste à remplacer, dans une formule, une sous-formule par une autre qui a été démontrée égale. Par exemple, étant donné l'axiome $x + 0 = x$, la formule $(a + 0) * 2 = 4$ peut être réécrite en $a * 2 = 4$. Cette technique est fondamentale pour la simplification des formules à prouver. Son automatisation est maintenant maîtrisée: la plupart des systèmes de preuve récents effectuent automatiquement ce type de simplifications[1].

  Il est bien connu en logique que la réécriture n'est pas toujours suffisante pour prouver certaines propriétés: il est souvent nécéssaire de faire des preuves par récurrence.

- Les preuves par récurrence sont une généralisation de la récurrence classique sur les entiers naturels où, pour prouver que $\forall n, P(n)$, on prouve d'abord $P(0)$, puis que $P(m) \Rightarrow P(m + 1)$: ce raisonnement repose sur le fait que tous les entiers

[1] Ce sont, pour une grande part, les progrès récents et spectaculaires dans ce domaine (à rapprocher des avancées dans le domaine du calcul formel) qui rendent maintenant praticables les preuves de logiciels.

naturels sont engendrés par 0 et $\_ + 1$. Des schémas de preuve par récurrence peuvent être définis pour d'autres types de données, lorsque les opérations permettant d'engendrer ces types sont connues (cf. l'annexe 2 de [LCGP 88]). Les preuves par récurrence ne peuvent être que *partiellement automatisées*: dans le cas où une formule comporte plusieurs variables, le choix de la variable sur laquelle porte la récurrence doit être fait manuellement (des recherches sont bien avancées pour assister ce choix); de plus, dans certains cas les algorithmes utilisés pour la preuve ne terminent pas. La plupart des systèmes de preuve permettent de vérifier ou d'automatiser certaines étapes de cette technique de preuves, mais ils demandent une intervention de l'utilisateur.

- Certains cas où la preuve par récurrence ne termine pas peuvent être résolus en inventant une *généralisation* de la propriété à prouver, dont la preuve termine. Une généralisation d'une propriété P est une propriété dont P est une conséquence (en mathématique classique, cela correspond à l'invention d'un lemme intermédiaire lors d'une démonstration). A l'exception du système de Boyer-Moore, qui est décrit ci-dessous, les systèmes de preuve actuels n'assistent pas cette technique de preuve: l'invention reste à la charge de l'utilisateur.

Ce sont ces trois techniques de preuve qui interviennent, par fréquence décroissante, dans les deux premières activités de preuve mentionnées dans l'introduction, à savoir: la preuve d'une certaine propriété d'une spécification; la démonstration qu'une étape de conception est correcte.

15.2.2 Preuves de programme

Lorsqu'il s'agit de raisonner sur des programmes et sur leurs spécifications, il faut pouvoir construire des formules où programmes et spécifications cohabitent, ainsi que des axiomes et des régles d'inférence permettant de mener des preuves sur ces formules.

Le système formel le plus connu est *la logique de Hoare*. Cette logique permet de raisonner sur des formules de la forme:

$$P\ \{Inst\}\ Q$$

où $Inst$ est une phrase d'un langage de programmation, et où P et Q sont des formules d'un langage de spécification, appelées les assertions. La formule $P\ \{Inst\}\ Q$ s'interprète: "si l'assertion P est vraie avant l'exécution des instructions et si cette exécution termine, alors l'assertion Q est vraie à la fin de l'exécution." P est appelée la précondition, Q la postcondition.
Cette logique est présentée dans l'annexe I du rapport final de contrat CNES [LCGP 88] sur la sûreté de fonctionnement des systèmes informatiques: elle n'est donc pas décrite à nouveau ici. Elle est enseignée dans la plupart des cursus en informatique et il existe de nombreux livres qui la décrivent. En particulier, [Bac 86] est sans doute un des meilleurs livres d'introduction à la preuve de logiciel, et le chapitre 6.4 de [GJM 91] pour son excellente discussion des problèmes posés par les tableaux.
Une théorie voisine, celle des plus faibles préconditions de Dijkstra, est souvent employée et enseignée conjointement: elle présente l'avantage d'aider à l'établissement de

la précondition minimum pour un programme et une postcondition, elle permet donc de simplifier les preuves.
Ces deux méthodes sont très voisines dans leur principe: ls formule initiale est composée d'assertions globales et du programme entier. Ces logiques permettent de réduire la formule de départ à des formules composées de parties du programme plus simples et d'assertions plus compliquées, jusqu'à obtenir des formules où n'interviennent plus que des instructions élémentaires. Ces formules sont alors prouvées.
Un des problèmes techniques qui a le plus freiné la mise en pratique de ces méthodes est la croissance de la taille des assertions lors de la réduction des formules. Ce problème est maintenant résolu grâce à l'existence de bons systèmes automatiques de simplification de ces assertions (cf. la section précédente) et par l'existence de langages de programmation modulaires (ADA, Modula) et des techniques de conception adaptées, ce qui permet de limiter la taille du programme de départ. De plus ([LCGP 88]): les boucles sont prouvées par récurrence ce qui nécessite l'invention d'invariants de boucles (ce sont des hypothèses de récurrence) ce qui est équivalent à une généralisation et n'est donc pas automatisé; toute synonymie dans les noms de variables (usage incontrôlé de pointeurs, COMMON, passage de paramètres par référence en Pascal, dans une moindre mesure utilisation de tableaux) complique les preuves.

En plus des théories de Hoare et de Dijkstra, il faut mentionner une méthode de preuve proposée par Harlan Mills qui est moins connue, mais qui est à la base du "Cleanroon Software Development" présenté section 15.5.1: cette méthode associe à chaque morceau de programme une expression qui dénote une fonction des variables initiales vers les variables résultats. A chaque construction du langage de programmation est associée une règle de composition de fonctions. Cette méthode nécessite également d'établir des invariants de boucle et d'éviter les synonymies.

15.3 Quelques outils

Il existe de très nombreux systèmes d'assistance à la preuve de développement de logiciel dans des laboratoires de recherche. Ceux-ci ne sont pas mentionnés dans ce rapport, seuls certains systèmes utilisés en contexte industriel sont cités.

N.B. : les principaux systèmes sont américains. En effet, de nombreux travaux ont été financés aux USA (DARPA, DoD, ONR, ...) pour résoudre des problèmes de sécurité, dans le sens confidentialité ou intégrité des données. D'autres systèmes sont anglais et résultent directement ou indirectement du projet Alvey. La liste ci-dessous montre clairement que la réalisation de tels systèmes demande un effort important et soutenu.

- Le système dit de *Boyer-Moore* est un des ancêtres des démonstrateurs de théorèmes. Il a été utilisé dans le projet SIFT (cf. section 15.5) dans les années 80. Il est basé sur un système formel particulier, la logique de Boyer-Moore [BM 79]: il faut donc exprimer la propriété à démontrer dans cette logique. C'est un vrai démonstrateur de théorèmes, par opposition à d'autres systèmes qui sont plutôt des vérificateurs de preuves. Ce système comporte plusieurs heuristiques puissantes qui lui permettent de trouver des preuves non triviales, par exemple des preuves par généralisation.

En contre-partie, il est d'un usage délicat. Une société d'Austin, Texas, la Computational Logic Inc., commercialise des preuves par ce système: l'expression de la propriété à prouver et les preuves sont faites par des ingénieurs de C.L.I. qui connaissent bien le système. Il s'agit de preuves de conception de matériel, par exemple [BY 91], ou de logiciel.

- *Gypsy* est un système qui a été développé à l'Université du Texas à Austin depuis les années 80 et dont les preuves sont également commercialisées par la C.L.I. Gypsy permet de faire de la preuve de programmes: le langage de programmation est Pascal, avec les restrictions classiques pour la preuve (cf. section 1.2); le langage de spécification ressemble à un langage de spécification algébrique. La logique est la logique de Hoare. Depuis plusieurs années, il est difficile d'obtenir des informations sur ce système qui est sans doute utilisé à des fins militaires aux USA.

- *EVES* est le résultat d'un effort important du gouvernement canadien qui depuis environ cinq ans finance le développement et l'expérimentation d'un système de preuve de programmes. Ce système a repris certains aspects de Gypsy, mais aussi de plusieurs prototypes universitaires. Le langage de programmation est spécifique, mais proche de Modula. Le langage de spécification pourrait devenir Z (des décisions sont en cours). Il n'y a pas eu vraiment d'expériences industrielles avec ce système, mais le gouvernement canadien donne une bonne publicité à ce projet et affiche une volonté d'en exploiter les résultats.
  Une comparaison de EVES et Gypsy par les auteurs de EVES est faite dans [Ban 88].

- *MALPAS* est un ensemble d'analyseurs statiques de programmes qui acceptent plusieurs langages de programmation: le langage d'origine est traduit en un langage intermédiaire, IL, sur lequel sont faites les analyses. Il existe des traducteurs vers IL depuis Fortran, ADA et de nombreux langages. Un des analyseurs de MALPAS, l'analyseur sémantique, construit et simplifie l'expression correspondant au résultat d'un programme, ainsi que la précondition correspondante, par évaluation symbolique (voir par exemple le chapitre 4 de [GJM 91] sur le test statique de programme). Un autre analyseur, appelé l'analyseur de conformité, compare cette expression et cette précondition à la spécification du programme, ce qui revient à une preuve du programme par rapport à cette spécification.
  MALPAS est commercialisé par la société anglaise "Rex Thomson and Partners". Il est en cours d'expérimentation en France à la SAGEM et au CEA (au moins). Un produit similaire, SPADE, est commercialisé en Angleterre.

- *HOL* (High Order Logic) a été développé principalement à l'université de Cambridge et est largement diffusé depuis 1988. Ce système a été utilisé essentiellement pour des preuves de conception de matériel. La preuve la plus spectaculaire est celle du microprocesseur tolérant aux fautes Viper pour le RSRE (Royal Signals and Radars Establishment du ministère de la défense britannique) [Coh 89]. Un des aspects les plus attrayants de ce système est la possibilité de définir des tactiques de démonstration adaptées à certains problèmes et à certains langages de spécification,

des schémas de récurrence et de bâtir des extensions de la logique de départ appelées des "theorys". Depuis deux ans des expériences de preuves de logiciels ont été faites avec HOL. En particulier une théorie correspondant à la logique de Hoare est disponible. HOL est en cours d'évaluation et d'expérimentation au LRI dans le cadre d'un contrat avec l'INRETS.

Cette liste est loin d'être complète: il faudrait aussi mentionner le système EHDM du SRI [RVH 89] qui a été utilisé pour des preuves de la NASA (voir la section 15.5.3), et le système B d'Abrial qui a été utilisé en contexte industriel (chez BP en Angleterre?), mais sur lequel il y a peu de choses publiées.

15.4 Phases d'application privilégiées

Comme c'est apparu dans les sections précédentes, les preuves sont utilisables pour la vérification des étapes de conception (passage de spécifications à des spécifications plus détaillées) et au niveau de la vérification d'un module de programme par rapport à ses spécifications détaillées, donc au niveau du test unitaire (mais il ne s'agit plus de test).

Une autre utilisation des preuves est la validation des spécifications globales (STB) par la démonstration de propriétés qui doivent en être des conséquences, ou par la réfutation de propriétés décrivant des comportements qui ne doivent pas se produire [Gau 91].

15.5 Quelques exemples de mise en oeuvre

Il y a de plus en plus d'exemples d'utilisation de preuves dans le développement de systèmes industriels. Une des premières expériences a été le projet SIFT, commandité par la NASA dans les années 80, et dont les conclusions ont été récemment publiées dans [MMS 90]. Il s'agissait d'une preuve de conception, qui a été sévèrement critiquée, mais qui a sans aucun doute fait progresser l'état de l'art.
Depuis, IBM a mis en place une méthode de développement appelée le Cleanroom Software Engineering qui est incrémentale, basée sur la preuve de chaque modules.
D'autres exemples plus récents et très convaincants de projets comportant des preuves ont été publiés, par exemple: la vérification formelle d'un algorithme tolérant aux fautes de synchronisation d'horloges dans un système distribué [RVH 89]; la preuve de correction des programmes (21000 lignes de Modula, partiellement embarquées) du système SACEM qui contrôle la survitesse sur la ligne A du RER [GH 90]; la vérification de la conception d'un système d'exploitation tolérant aux fautes pour un système de contrôle de vol [DBC 91].
On peut remarquer que les travaux rapportés dans [RVH 89] et [DBC 91] ont été financés par la NASA. Toutes ces preuves s'avèrent être longues[2], fastidieuses, et il y a un consensus général sur le besoin de systèmes pour vérifier et assister ces activités. De tels systèmes ont été décrits plus haut (Boyer Moore, EHDM, Gypsy, HOL, ...).

[2]Durant une table ronde à la 2nd IFIP Working Conference on Dependable Computing for Critical Applications (Tucson, Février 1991), les chiffres suivants sont mentionnés pour la preuve du noyau du système d'exploitation ASOS avec Gypsy : 19383 instructions ADA , 7604 lignes de spécification formelle, 882 049 lignes de transcription de la preuve.

Cette énumération est loin d'être complète: en France des travaux sont menés et appliqués chez Merlin-Gérin autour du système SAGA; en Angleterre les preuves sont pratiquées et il est question de les exiger pour certains logiciels critiques développés avec des financements publiques. La suite de cette section présente plus en détail le Cleanroom Software Engineering d'IBM, le projet SACEM, et les travaux financés récemment par la NASA.

15.5.1 Le "Cleanroom Software Engineering" d'IBM

Cette méthode de développement de logiciel a été proposée par Mills, Dyer et Linger de l'IBM Federal Systems Division. Son nom est inspiré des techniques de fabrication des circuits intégrés, où l'on prévient les défauts en travaillant dans une atmosphère stérile. Cette approche est donc basée sur la prévention des erreurs, par opposition aux méthodes classiques qui sont orientées vers la correction des erreurs. Chaque module est spécifié formellement et prouvé correct (manuellement) par rapport à cette spécification. Il n'y a pas de test unitaire: la preuve de programme remplace le test unitaire structurel et fonctionnel.
Il y a par contre du test d'intégration, qui est du type test aléatoire opérationnel (cf. la terminologie définie dans la fiche sur le test aléatoire; les auteurs du cleanroom parlent de test statistique). Le rôle de ce test est d'établir la fiabilité du logiciel avant de le livrer. Ce qui précède est traduit du livre de Carlo Ghezzi. En principe il n'y a pas de recherche des erreurs et de corrections à partir de ces tests.

La méthode a été développée et expérimentée depuis 1982. Plusieurs articles ont été publiés en 1986-87: [CDM 86], [MDL 87], [SBB 87], [Dye 87]. Il ne semble pas y avoir de publications plus récentes. Trois projets menés avec cette méthode sont décrits dans [MDL 87]: un compilateur de Cobol structuré (40000 lignes de code); un programme de vol pour un hélicoptère développé dans le cadre d'un contrat Air Force (35000 lignes de code); et un système de planification de transport spatial pour la NASA (45000 lignes de code).
Les résultats publiés sont très encourageants: il apparaît que la preuve de correction manuelle ne prend pas plus de temps que le test unitaire et les mises au point classiques, même avec des développeurs qui utilisent la méthode pour la première fois [SBB 87]; plus de 90% des défauts sont découverts avant la première exécution; les programmes obtenus sont plus simples et de meilleure qualité.

15.5.2 Le système SACEM [GH 90]

SACEM est un système informatique partiellement embarqué qui contrôle la vitesse des trains sur la ligne A du RER à Paris: périodiquement chaque train reçoit du sol ou de ses capteurs un ensemble d'informations à partir desquelles il calcule la vitesse maximum instantanée compatible avec les exigences de sécurité; de plus, une vitesse optimale est transmise au conducteur du train.
Le logiciel est exécuté sur un monoprocesseur codé (un processeur autotestable); il comporte 21000 lignes de Modula 2, réparties en 14000 lignes embarquées, structurées en 170 procédures, et 7000 lignes au sol, structurées en 70 procédures. Chaque procédures comporte environ 70 lignes source, jamais plus de 100.

Les exigences de sécurité sont très élévées. Pour les atteindre, le groupement industriel chargé du développement (dirigé par GEC-Alsthom) et la RATP ont mis en place un processus de développement et de validation comportant d'une part toutes les "bonnes" méthodes connues et utilisées habituellement, en les renforçant (inspections, contrôle qualité, tests globaux intensifs par des équipes indépendantes ...), d'autre part l'établissement de spécifications formelles et la preuve des procédures identifiées comme critiques (132, dont 111 embarquées et 21 au sol).
Un outil a été développé et utilisé pour décomposer, en suivant les règles de Hoare, les formules à prouver. Les preuves élémentaires ont été faites à la main. La principale difficulté rencontrée dans l'activité de preuve a été l'établissement des propriétés à prouver, précisément les préconditions et postconditions de chaque procédure. En cours de projet une équipe de "re-expression formelle" a été mise en place pour vérifier toutes ces assertions après que des erreurs y aient été découvertes. Le système SACEM est en service depuis Septembre 1989.

15.5.3 Les expériences de la NASA

La NASA a lancé depuis quelques années un effort de recherche majeur sur les méthodes de validation et de vérification des systèmes de contrôle de vol digitaux. Cet effort est mené soit directement au Langley Resarch Center [DBC 91], soit via des contrats par exemple avec le SRI [RVH 89], ou C.L.I. [BY 91].
Le but est d'éliminer les fautes de conception et de vérifier formellement la tolérance aux autres types de fautes (erreurs de spécifications, défaillances physiques). Les études actuelles portent sur la preuve de la conception d'une "plate-forme d'exécution" tolérantes aux fautes, c'est-à-dire d'un ensemble formé de processeurs redondants et d'un système d'exploitation distribué, dont le comportement reste correct si 50% des processeurs fonctionnent normalement.
La conception est guidée par les preuves à réaliser et la facilité de mise en oeuvre des futurs tests. L'article [DBC 91], et le rapport interne de la NASA dont il est un résumé, décrivent en détail la preuve de la première étape de conception, c'est-à-dire le passage d'un modèle de fonctionnement abstrait, monoprocesseur, à un modèle avec redondance matérielle, tolérance de 50% de processeurs défaillants et élimination des erreurs temporaires du système après un temps borné. Les modèles sont décrits en calcul des prédicats du premier ordre (cf. section 1.1) et les preuves sont manuelles. Il est prévu de transcrire la preuve et de la vérifier sous le système EHDM du SRI.
Le modèle avec redondance est considéré comme synchrone. L'étape de conception suivante sera de passer à un modèle asynchrone et de décrire et vérifier les algorithmes de synchronisation des processeurs redondants: une preuve de ce type est décrite dans [RVH 89]; c'est une preuve formelle qui a été menée et vérifiée avec le système EHDM du SRI. Il est intéressant de citer les auteurs de cette preuve:

> " we did not find theorem proving to be a bottleneck (discovering the theorems to prove was a bottleneck)".

Une présentation abrégée et simplifiée de ces deux preuves figure dans le mémo CNES [Del 91].

15.6 Arrêt de la preuve

Bien évidemment, l'arrêt de la preuve est décidé lorsque toutes les propriétés requises ont pu être prouvées.

15.7 Conclusion et commentaires

Le mythe des preuves de logiciels impraticables car: inapplicables aux grands programmes, aux systèmes distribués ou temps-rééls, trop difficiles pour les ingénieurs de développement, etc, a vécu. Ces arguments ne sont plus valables et les preuves doivent être envisagées dans les méthodes de vérification et validation des systèmes informatiques critiques [Gau 91].
Elles ne doivent pas être envisagées seules, comme la solution miracle, mais en complément d'une politique exigeante de qualité du développement du logiciel: c'est le cas dans toutes les expériences mentionnées ci-dessus: validation poussée des spécifications de départ et des propriétés à prouver, inspections, tests de validation, etc.

Références bibliographiques relatives aux preuves

- [Bac 86] R.C. Backhouse
  "Program Construction and Verification"
  Prentice-Hall International Series in Computer Science, 1986, 280p. Traduction française parue, sans doute chez Masson.

- [Ban 88]
  Proceedings of the 2nd IEEE-ACM Worshop on Software Testing Analysis and Verification, Banff, 1988.

- [BM 79] R. Boyer, J. Moore
  "A Computational Logic"
  Academic Press, 1979

- [BY 91] W. Bevier, W. Young
  "The proof of Correctness of a Fault-tolerant Circuit Design"
  2nd IFIP Working Conference on Dependable Computing for Critical Applications, Tucson, Feb. 1991.

- [CDM 86] P.A. Currit, M. Dyer, H.D. Mills
  "Certifying the reliability of software"
  IEEE Transactions of Software Engineering, Volume SE-12, n^o1, janvier 1986 , pp. 3-11.

- [Coh 89] A. Cohn
  "The notion of Proof in Hardware Verification"
  Journal of Automated Reasoning, Vol.5, pp. 127-139, 1989.

- [DBC 91] B. DiVito, R. Butler, J. Caldwell
  "High Level Design Proof of a Reliable Computing Platform."

2nd IFIP Working Conference on Dependable Computing for Critical Applications, Tucson, Feb. 1991.

- [Del 91] F. Delamotte
"Spécifications et preuves formelles, expériences aux USA: présentation par M-C. Gaudel"
Mémo CNES H/PS/PA/91/103, Juin 1991.

- [Dye 87] M. Dyer
"A Formal Approach to Software Error Removal"
Journal of Systems and Software 7, 1987, pp. 109-114.

- [Gau 91] M-C. Gaudel
"Advantages and Limits of Formal Approaches for Ultra-High Dependability"
2nd PDCS Worshop, Newcastle, Mai 1991.

- [GH 90] G. Guiho, C. Hennebert
"SACEM Software Validation"
12th IEEE-ACM International Conference on Software Engineering, Mars 1990, pp. 186-191.

- [GJM 91] C. Ghezzi, M. Jazayeri, D. Mandrioli
"Fundamentals of Software Engineering"
Prentice-Hall Internalional Editions, 1991, 571p.

- [LCGP 88] B. Courtois, M.-C. Gaudel, J.-C. Laprie, D. Powell
"Sureté de Fonctionnement des Systèmes Informatiques - Matériel et Logiciel -"
Rapport final de contrat CNES, publié en monographie Dunod, Avril 1989, 215 pages.

- [MDL 87] H.D. Mills, M. Dyer, R.C. Linger
"Cleanroom Software Engineering"
IEEE Software, Septembre 1987, pp. 19-25.

- [MMS 1990] L.E. Moser, P.M. Melliar-Smith
"Formal Verification of Safety-critical Systems"
Software Practice and Experience, Vol. 20, $n^o 8$, 1990, pp. 799-821.

- [RVH 1989] J. Rushby, F. Von Henke
"Formal Verification of a Fault Tolerant Clock Synchronization Algorithm"
NASA Contractor Report 4239. Juin 1989. 217 pages.

- [SBB 87] R.W. Selby, V.R. Basili, F.T. Baker
"Cleanroom Software Development: an empirical evaluation"
IEEE Transactions on Software Engineering, vol. SE-13, Sept. 1987, pp1027-1037.