UNIVERSITÉ DE MONTRÉAL

Conception de logiciels de communication testables

Par

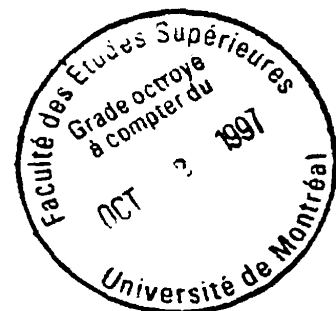
Kamel Karoui

Département d'Informatique et de Recherche Opérationnelle
Faculté des Arts et des Sciences

Thèse présentée à la Faculté des Études Supérieures
en vue de l'obtention du grade de
Philosophiæ Doctor (Ph. D.)
en Informatique

Juin, 1997

© Karoui Kamel, 1997



National Library of Canada
Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services
Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-32650-0

Canada

Université de Montréal
Faculté des Études Supérieures

Cette thèse intitulée:

Conception de logiciels de communication testables

présentée par:

Kamel Karoui

a été évaluée par un jury composé des personnes suivantes:

Professeur G. v. Bochmann	Président du Jury
Professeur R. Dssouli	Directeure de recherche
Professeur O. Cherkaoui	Co-Directeur de recherche
Professeur O. Rafiq	Examinateur externe
Professeur R. Keller	Membre du jury
Professeur R. DeSantis	Représentant du Doyen

Thèse acceptée le: 4 septembre 1997

Sommaire

Les protocoles associés aux systèmes répartis et aux réseaux de communication sont devenus très complexes. Le coût de l'activité de tests des protocoles est devenu de plus en plus important par rapport aux autres activités du cycle de développement. La considération de la testabilité dans le cycle de développement du protocole est devenue impérative si on veut réduire les coûts des tests. La prise en compte de la testabilité lors de l'étape de conception est appelée *DFT* (*Design For Testability*). La *DFT* est un champ de recherche à deux dimensions: la dimension de la conception et la dimension des tests. Elle consiste à intervenir lors de l'étape de conception pour améliorer la qualité d'un logiciel et générer un produit dont les activités de génération et d'application des tests est facile. Ceci a comme conséquence une réduction du coût des tests.

Au cours de ce travail de recherche, nous décrivons comment la testabilité peut être intégrée dans le cycle de développement du protocole. En tant que nouveau aattribut, la testabilité agit au niveau de toutes les étapes du cycle et en particulier lors de la conception. Nous expliquons pourquoi les nouvelles propriétés de testabilité apportées à la conception sont propagées dans l'implantation.

Plusieurs facteurs peuvent influencer les tests et en particulier le modèle de spécification utilisé. Dans cette thèse, nous avons étudié la testabilité des protocoles de communication sur trois modèles de conception différents: les relations, les automates à états finis et les automates à états finis étendus. L'utilisation d'un modèle ou d'un autre dépend du niveau d'abstraction voulu. À un niveau d'abstraction élevé, le modèle des relations fournit une bonne plateforme pour l'étude de la testabilité. Vu son niveau

d'abstraction, ce modèle peut être même utilisé au niveau de l'étape d'analyse des besoins. Ce modèle se fonde sur une théorie riche et souple. À un niveau d'abstraction moins élevé, nous avons utilisé les automates à états finis comme modèle de base pour la spécification et l'étude de la testabilité des protocoles de communication. Ce modèle a été largement exploité pour la spécification des protocoles de communication et la génération automatique des séquences de test. Ce modèle n'est associé qu'aux aspects de contrôle de la spécification. Pour prendre en compte l'aspect de données, nous avons utilisé le modèle des automates à états finis étendus. Ce modèle offre une représentation de la spécification à un niveau d'abstraction encore moins élevé ce qui permet de considérer les aspects de données.

Pour les deux premiers modèles (les relations et les automates à états finis) nous avons cherché les facteurs qui influencent les tests et nous en avons proposé une évaluation individuelle. Nous avons utilisé ces facteurs pour prédire la testabilité et pour guider le processus de raffinement de la spécification. Ce processus est itératif, il a pour objectif d'améliorer la testabilité du produit final. Ce processus se termine lorsque le concepteur juge que le niveau de testabilité est adéquat ou lorsqu'aucun changement n'est plus encore possible. Ce processus est constitué des étapes suivantes: évaluation des facteurs de testabilité de la spécification initiale, choix et application d'une transformation de la spécification, nouvelle évaluation de la testabilité et décision (fin du processus ou continuer le raffinement). Parmis l'ensemble des transformations possibles, le concepteur en choisira une qui peut améliorer une faible valeur associée à un facteur. Pour pouvoir juger de la faiblesse des valeurs associées aux facteurs, nous avons normalisé toutes nos métriques entre 0 et 1, où 1 est la meilleur valeur et 0 la plus faible. Partant d'une spécification de départ où l'un des facteurs a une valeur faible (proche de 0), une transformation est dite bonne si elle améliore cette valeur pour la rapprocher de 1.

L'évaluation de la testabilité, telle que nous l'avons proposée, est composée de l'évaluation individuelle de plusieurs facteurs décrivant chacun un aspect de la testabilité. L'avantage d'une telle approche par rapport à l'évaluation unique est qu'elle donne une idée détaillée sur les faiblesses (par rapport aux tests) de la spécification. Ceci permet une intervention plus précise pour une éventuelle correction du facteur en question. En plus, ces facteurs peuvent être utilisé pour une classification plus détaillée des spécifications.

Pour le modèle des automates à états finis étendus, les mesures précédantes restent encore applicables. Par contre, ces mesures ne nous informent pas sur l'aspect des données. Pour cela, nous avons proposé une transformation qui tient compte de l'aspect des données. Elle permet de compléter une spécification en vue de raccourcir la longueur de sa suite de test. Cette transformation bien qu'elle tient compte des aspects de données d'une spécification, elle peut être appliquées aux automates à états finis. Suite à l'application de la transformation, nous avons proposé une mesure pour évaluer le gain de testabilité.

Tous les facteurs et les transformations que nous avons proposé ont pour objectif de faciliter le processus de détection de fautes. Pour compléter ce travail, nous avons essayer de voir l'influence de certains de ces facteurs sur le processus de localisation de fautes. Ceci nous a conduit à proposer de nouveaux facteurs et de nouvelles évaluation.

Table des matières

1 Introduction **1**

1.1. Motivation 1

1.2. Contribution 2

1.3. Organisation de la thèse 4

2 Notions de bases de la conception et des tests **7**

2.1. Introduction 7

2.2. Les critères de qualité d'un logiciel 9

2.2.1. Critères de qualité externe 9

2.2.2. Critères de qualité interne 11

2.3. Spécificité des protocoles de communication 12

2.4. La conception de logiciel testable 14

2.5. La conception du logiciel 14

2.5.1. Processus de conception 14

2.5.2. Les différentes phases de la conception 15

2.5.3. Principes de base de la conception 16

2.5.4. Méthodes de conception 17

2.5.5. Les techniques formelles de description 18

2.5.6. Qualité d'une conception 19

2.6. Les tests 19

2.6.1. Introduction 19

2.6.2. Définition des tests et du débogage 21

2.6.3. Définition des fautes, des erreurs et des défaillances 21

2.6.4. Les types de test 22

2.6.5. Les principales philosophies de test....24
2.6.6. Les modèles de fautes....29
2.6.7. Le test des protocoles de communication....30
2.6.8. Facteurs influençant le coût des tests....34
2.6.9 - Couverture des tests....35

3 La testabilité: État de l'art 37

3.1. Introduction....37
3.2. Définitions de la testabilité....38
3.2.1. Définitions....38
3.2.2. Définitions relatives aux protocoles de communication....39
3.2.3. Discussion....40
3.3. Les mesures de testabilité....41
3.3.1. Métriques basées sur le domaine des entrées-sorties....41
3.3.2. Métriques basées sur la complexité du flux de contrôle....44
3.3.3. Métriques basées sur la complexité du flux de données....49
3.3.4. Métriques probabilistes....49
3.3.5. Mesures de testabilité des protocoles de communication....51
3.3.6. Discussion....51
3.4. Facteurs qui influencent la testabilité....52
3.4.1. Facteurs énoncés par Ould....53
3.4.2. Facteurs énoncés par Martin et McClure....53
3.4.3. Facteurs énoncés par Perry....53
3.4.4. Facteurs énoncés par McCall et al.....54
3.4.5. Facteurs énoncés par Boehm et al.....55
3.4.6. Facteurs énoncés par Boehm....56
3.4.7. L'indéterminisme et la coordination par Vuong et al....56
3.4.8. Protocoles auto-stabilisé....59
3.4.9. Facteurs influençant le test en orienté objet....60
3.4.10. Contrôlabilité et Observabilité....63
3.4.11. Méthodes formelles....68
3.4.12. Influence de l'environnement Drira....71
3.4.13. Facteurs qui influencent la testabilité des protocoles de communication....71
3.4.14. Discussion....72

3.5. Moyens utilisés pour améliorer la testabilité 73

4 Prise en compte de la testabilité dans le cycle de développement du protocole 76

4.1. Introduction 76

4.2. Prise en compte de la testabilité dans le cycle de développement du logiciel. 76

4.3. Considération précoce de la testabilité et influence sur l'implantation 79

4.4. Approche adoptée 81

5 Étude de la testabilité basée sur le modèle relationnel 82

5.1. Introduction 82

5.2. Définitions de base 83

5.3. Le modèle proposé 84

5.4. Testabilité d'un module 87

5.4.1. Définissabilité de R 88

5.4.2. Informabilité de R 89

5.4.3. Déterminisme de R 90

5.4.4. Convergence de R 91

5.4.5. Mesure Unique de la testabilité 92

5.5. Classification des modules d'après leur testabilité 93

5.6. Analyse de la testabilité 94

5.6.1. Transformation permettant d'éviter l'indéterminisme 96

5.6.2. Transformation permettant d'améliorer la convergence 99

5.6.3. Transformation permettant d'améliorer la définissabilité 100

5.6.4. Transformation permettant d'améliorer l'informabilité 101

5.6.5. Composition de transformations 102

5.6.6. Application aux spécifications de protocoles 103

5.7. Application au protocole X.25 103

5.7.1. Transformations préliminaires 104

5.7.2. Spécification du service X.25 104

5.7.3. Interprétation des résultats 105

5.8. Testabilité d'une composition de modules 107

5.9. Utilisation de la Testabilité comme critère de composition/décomposition... 108

5.9.1. Testabilité de la composition de modules en série 110

5.9.2. Testabilité de la composition de parallèle de modules 112

5.9.3. Influence des facteurs de testabilité sur la comp. de modules...... 114
5.10. Conclusion 114

6 Étude de la testabilité basée sur le modèle des automates à états finis..116
6.1. Introduction.. 116
6.2. Modèle des machines à états finis avec entrées-sorties 117
6.3. Les tests de conformité pour les machines à états finis........................ 118
6.3.1. Le modèle des fautes.. 118
6.3.2. Machine déterministe complètement spécifiée.............................. 119
6.3.3. Machine déterministe partiellement spécifiée............................... 120
6.3.4. Machines indéterministes.. 122
6.3.5. Méthodes de dérivation de tests pour les FSMs 124
6.3.6. Description des méthodes 124
6.4. Classification des Spécifications 127
6.5. Mesure de testabilité proposée... 128
6.5.1. Nouvelle mesure de testabilité 129
6.6. Transformations testables d'une spécification................................... 134
6.6.1. Transformations de complétude 135
6.6.2. Transformation permettant de déterminiser une specification 140
6.6.3. Transformation permettant de minimiser la spécification 140
6.6.4. Combinaison de transformations (K-trans) 142
6.7. Conclusion... 143

7 Étude de la testabilité basée sur le modèle des automates à états finis étendus... 145
7.1. Introduction.. 145
7.2. Définitions de bases 146
7.2.1. Le modèle des EFSMs.. 147
7.2.2. Le langage SDL... 149
7.2.3. Les tests de flux de données 152
7.3. Amélioration de la testabilité en complétant la spécification................... 155
7.3.1. Approches pour maintenir le comportement normal du protocole. 156
7.3.2. Avantages de ces approches... 158
7.3.3. Technique utilisée pour comléter un EFSM 158
7.4. Application... 163

7.5. Conclusion......166

8 Influence des facteurs de testabilité sur le diagnostic......168

8.1. Introduction......168

8.2. Le diagnostic......169

8.3. Influence des facteurs de test sur le diagnostic......170

8.3.1. Le degré de contrôlabilité......170

8.3.2. Le degré de Fuzziness......172

8.3.3. Le degré de distingabilité des états......173

8.3.4. Le degré d'abstraction......174

8.3.5. Facteurs dépendants des fautes......174

8.3.6. Discussion......175

8.4. La conception de systèmes faciles à diagnostiquer......175

8.4.1. Amélioration de la contrôlabilité......176

8.4.2. Amélioration de la distinction des états......177

8.5. Conclusion......179

9 Conclusion......180

Bibliographie......184

A Application de l'analyse de la testabilité sur le protocole X.25......196

B Algorithme des transformations de testabilité des FSMs......208

Liste des figures

2.1. Activités du cycle de développement de logiciel................................ 8
2.2. Architecture de test locale.. 31
2.3. Architecture de test distribuée... 32
2.4. Architecture de test coordonnée.. 32
2.5. Architecture de test distante... 33
3.1. Exemple de FSM complètement spécifié déterministe........................... 44
3.2. Un exemple de programme avec le graphe de contrôle associé................ 45
3.3. Exemple de graphe de flux D1((D1... 47
3.4. Graphes de flux primaires D0 IF-THEN, D1 IF-THEN-ELSE,.............. 47
3.5. Le contexte de test... 50
3.6. Modèle de Perry décrivant les relations entre les critères de qualité........... 54
3.7. Modèle de McCall de la qualité de logiciel....................................... 54
3.8. Modèle hiérarchique de Boehm.. 55
3.9. Exemple de processus concurrents.. 57
3.10. Exemple de spécification ayant un comportement indéterministe............ 58
3.11. Testabilité de la représentation.. 61
4.1. Processus de raffinement de la testabilité.. 77
4.2. Cycle de vie étendu... 78
5.1. Modèle du fournisseur de service.. 85
5.2. Séquences de primitives du service de transport d'OSI......................... 85
5.3. Relation associée à un module... 86
5.4. Représentation relationnelle du service au niveau de l'utilisateur A........... 87
5.5. Exemple de relation non complètement définie................................... 89
5.6. Service associé au processus d'établissement de la connexion................. 91

5.7. Relation associée au service de la Figure 5.6........ 91
5.8. Exemple de relations convergente........ 92
5.9. Modèle associé aux processus séquentiels........ 97
5.10. Le service d'INRES........ 97
5.11.a. La relation décomposée d'INRES........ 97
5.11.b. Considération de nouvelles variables d'état........ 98
5.11.c. La composition de relations après avoir rendu quelques variables observables........ 99
5.12. Considération de nouvelles variables pour résoudre la convergence........ 100
5.13. Extension de la Relation RA de la Figure 5.5........ 101
5.14. Passage d'une classe testabilité à une autre........ 103
5.15. La relation associée à l'état 13........ 105
5.16. Résultats de l'analyse de la testabilité de X.25........ 106
5.17. Exemple de décomposition de système........ 108
5.18. Illustration du masquage d'une relation composée........ 110
5.19. R = Ri || Rk........ 112
6.1. Relation entre la longueur d'une suite de test et sa couverture........ 126
6.2. Classification des FSMs........ 128
6.3. application de S-trans........ 136
6.4.a. Application de H-trans sur la spécification de la Figure 6.4.a........ 139
6.4.b. L'entité de protocole du répondeur d'INRES........ 139
6.5. Application de C-trans sur la spécification de la Figure 6.4.a........ 142
6.6. Application de K-trans sur la spécification de la Figure 6.4.a........ 143
7.1. Procédure de transformation d'une spécification SDL en un EFSM........ 152
7.2. Critères de test du flux de données........ 153
7.3. Méthode d'assignation des transitions non-spécifiées en SDL........ 158
7.4. Exemple d'une spécification décrite sous forme d'un EFSM........ 164
7.5. Assignation d'une transition I-undefined........ 165
8.1. Interactions entre les prédictions et les observations........ 169
8.2. La spécification S........ 172
8.3. L'implantation Im1........ 172
8.4. Une spécification et deux implantations fautives non distingables........ 174
8.5. Une spécification transformée........ 177
8.6. Une implantation de la spécification de la Figure 8.5........ 177

8.7. Amélioration de la distingabilité des états 179

Liste des tables

Table 3.1. Classification des définitions de la testabilité............................ 40
Table 3.2. Sommaire des structures en C et leur complexité respectives. 46
Table 3.3. Valeur de la métrique pour les graphes de flux primaires............... 48
Table 3.4. Valeur de la métrique pour des graphes de flux composés. 48
Table 3.5. Valeur de la métrique pour des graphes de flux composés. 48
Table 3.6. Résultats d'un test... 51
Table 3.7. Axiomes de 1 à 4. 62
Table 3.8. Axiomes de 5 à 8. 62
Table 3.9. Axiomes de 9 à 11... 63
Table 5.1. Classement de quelques types de relations................................ 94
Table 5.2. Influence de la composition sur la testabilité..............................114
Table 5.3. Table de composition de modules...117
Table 6.1. Relation de conformité pour des machines déterministes...............121
Table 6.2. Relation de conformité pour des machines indéterministes.............123
Table 6.3. bornes supérieures pour le test..123
Table 6.4. Résultats de l'application des transformations. 143
Table 7.1. Table des séquences de test avant et après l'assignation de la trans ...166

Remerciements

Mes remerciements vont tout d'abord à tous les membres du groupe de Téléinformatique du Département d'informatique et de recherche opérationnelle de l'Université de Montréal qui grâce à leur esprit de collaboration ont contribué directement ou indirectement à la réalisation de ce travail. J'exprime toute ma reconnaissance à tous les membres de ce groupe sans exception: les professeurs G. v. Bochmann et R. Dssouli, le personnel de support technique O. Belal et D. Ouimet, notre secrétaire administrative L. Lévesque, tous les étudiants, tous les professeurs invités et tous les chercheurs.

J'exprime aussi ma profonde gratitude au Professeur Rachida Dssouli de l'Université de Montréal pour avoir dirigé avec patience, confiance et enthousiasme cette thèse. Je remercie mon Co-Directeur le Professeur Omar Cherkaoui de l'Université du Québec à Montréal pour son assistance et ses conseils pertinents.

Je remercie les Professeurs: G. v. Bochmann, O. Rafiq et R. Keller pour l'honneur qu'il me font en faisant partie du jury de ma thèse.

De même, je remercie tous les membres de ma famille et en particulier mes parents Noureddine et Radhia qui m'ont soutenu tout au cours de mes études et à qui je dédie cette thèse.

Un grand remerciement du fond du coeur à celle qui n'a cessé de me procurer durant toute la période de la Thèse son soutien moral inconditionnel, ma femme Thouraya.

À mes merveilleux parents
Noureddine et Radhia.

Tu as droit à l'action, mais seulement à l'action, et jamais à ses fruits ; que les fruits de tes actions ne soient point ton mobile ; et pourtant ne permets en toi aucun attachement à l'inaction.

La Bhagavad-Gita

Chapitre 1

Introduction

1.1. Motivation

La pratique de test constitue une opération fondamentale et rassurante dans la vie de tous les jours et notamment dans le domaine de l'industrie. Essayer un produit avant de le consommer ne remet guère en cause la compétence de son concepteur et de son producteur, au contraire ceci permet de rassurer le consommateur (client) de la bonne qualité du produit.

Dans le domaine de l'informatique et notamment dans la télécommunication, le coût des tests est en continuelle augmentation. Plusieurs techniques sont utilisées pour faciliter les tests et réduire leur coût. La direction de recherche qui consiste à appliquer ces techniques pour identifier, mesurer et modifier certaines caractéristiques d'un protocole pour faciliter les tests est appelée testabilité du protocole. Ces techniques de testabilité peuvent être appliquées à différents niveaux du cycle de développement du protocole et en particulier à l'étape de conception. En agissant sur quelques propriétés particulières de la conception, on peut améliorer substantiellement le coût des tests. L'application des techniques de testabilité lors de l'étape de conception est appelée *Conception de Logiciels Testables* ou *DFT* (*Design For Testability*).

Comme les tests sont excessivement coûteux, l'activité de *Conception de Logiciels Testables* devrait occuper une place importante dans le cycle de développement des protocoles. En effet, on ne doit pas attendre la phase d'implantation pour prédire et réagir à d'éventuels problèmes de tests [Voas 95]. La phase d'implantation est assez complexe, tout changement lors de cette phase doit être justifié au niveau de la conception. Les étapes d'implantation et de test sont indépendantes; elles sont réalisées par des personnes

différentes ayant des vues différentes de la qualité du protocole. Muni d'une grande expérience de programmation, l'implanteur vise à optimiser le code en le compactant et en augmentant sa performance. Cette tendance d'optimisation diminue la lisibilité du code et la visibilité des actions via les interfaces. Une implantation ainsi générée n'est pas toujours facile à tester. D'où l'importance qu'on doit accorder aux problèmes de test dès les premières étapes du cycle de développement du logiciel et plus précisément lors de l'étape de conception.

La *Conception de Logiciels Testables* se fait suivant deux directions complémentaires. La première consiste à identifier et estimer les facteurs qui peuvent influencer le coût des tests. La deuxième utilise les résultats de la première pour générer une conception qui facilite les tests. Concernant la première direction, plusieurs chercheurs ont identifié un ensemble de facteurs pouvant influencer les tests. Nous citerons par exemple les méthodes formelles, le style de spécification, le niveau d'abstraction, la concurrence, l'indéterminisme, l'architecture ou la structure du système et les méthodes de test. L'ensemble des facteurs évoqués dans la littérature n'est pas complet; d'autres propriétés plus raffinées et plus formelles peuvent y être rajoutées. Quant à la deuxième direction, les recherches, menées dans ce sens, sont peu nombreuses. La plupart d'entre elles ont été conduites au niveau de la conception du matériel [Bard 81], [Prad 86], etc. Plusieurs chercheurs ont essayé d'adapter les méthodes du matériel au domaine du logiciel. Il y a un consensus sur certaines propriétés de testabilité que la conception doit avoir pour améliorer le coût des tests comme l'observabilité et la contrôlabilité. La première permet de déterminer l'état interne de la spécification, alors que la deuxième permet d'atteindre toutes les parties à tester de la spécification.

1.2. Contribution

Dans le cadre de cette thèse, nous proposerons d'étendre le cycle de développement du protocole en rajoutant l'étude de la testabilité comme une activité intégrante du cycle. Cette activité est effectuée au niveau de chaque étape du cycle de développement. Nous nous intéresserons particulièrement à l'étape de conception. Nous analyserons la *Conception de Logiciels Testables* dans les deux directions de recherche décrites précédemment, à savoir l'identification des facteurs qui influencent la testabilité et leur utilisation pour la conception des protocoles testables.

Nous étudierons la *Conception de Logiciels Testables* sur trois modèles que nous utiliserons à des niveaux d'abstraction différents: le modèle des relations, le modèle des automates à états finis (*FSMs*) et les automates à états finis étendus (*EFSMs*). Les méthodes de spécification ainsi que le niveau d'abstraction constituent deux facteurs qui influencent la testabilité. Le niveau d'abstraction dépend des limites imposées par le formalisme de description utilisée ainsi que du niveau de détail adopté par le concepteur. Pour le modèle des automates à états finis par exemple, la prise en compte des paramètres et variables dans la spécification provoquent un problème d'explosion d'états. Les modèles basés sur les automates étendus permettent de contourner ce problème.

Les méthodes de test existantes appartiennent à deux principales catégories: les tests de flux de contrôle et les tests de flux de données. La couverture de fautes caractérise l'aptitude d'un test à détecter des fautes dans l'implantation [Boch 91d] [Brin 93] [Petr 94b]. Cette approche est issue du test du matériel et a été adaptée aux systèmes réactifs tels que les protocoles de communication. Pour chaque faute recensée dans le modèle, on regarde s'il existe un test qui la détecte. On définit alors la couverture de fautes d'un ensemble de tests comme le nombre de fautes détectées sur le nombre total de fautes possibles. Au cours de ce travail nous proposerons un ensemble de transformations de spécifications permettant de faciliter les tests sans altérer la couverture de fautes.

Pour les modèles des automates à états finis et des relations, nous proposerons une approche d'analyse, d'évaluation et d'amélioration de la testabilité. Cette approche est basée sur un ensemble de propriétés que nous appellerons facteurs de testabilité. Ces derniers sont extraits à partir de certaines propriétés de la spécification. Pour chacun de ces facteurs, nous associerons une métrique dont les valeurs varient entre 0 et 1, où 0 est la pire valeur et 1 la meilleure valeur associée au facteur. L'ensemble de ces métriques est rassemblé dans un vecteur unique qui permet d'identifier les causes d'une mauvaise testabilité. Ce vecteur est important puisqu'il constitue une généralisation des métriques de testabilité existantes. Il peu aider le concepteur à détecter et à corriger certaines propriétés menant à une mauvaise testabilité. Dans cette même direction, nous proposerons un ensemble de techniques permettant de raffiner et de transformer une spécification pour améliorer la valeur d'un ou de plusieurs facteurs de testabilité.

L'approche d'analyse, d'évaluation et d'amélioration de la testabilité relative aux automates à états finis et aux relations ne prend pas en considération les aspects de flux de

données. Pour y remédier, nous proposerons une technique d'amélioration de la testabilité basée sur le modèle des automates à états finis étendus. Cette technique consiste à rajouter à la spécification un comportement qui facilite les tests sans altérer les fonctions de base du protocole. Le comportement rajouté sera transparent aux utilisateurs du protocole et ne sera accessible que pour des fins de test. Pour évaluer le gain de testabilité obtenu, nous proposerons une mesure basée sur la comparaison de la longueur des suites de test de la spécification initiale et de la spécification étendue.

Toutes les propriétés de la spécification que nous analyserons sont associées au problème de détection de fautes d'un protocole. Pour généraliser ce travail, nous étudierons l'influence de certaines de ces propriétés sur le problème de localisation de fautes. Nous montrerons que certaines de ces propriétés ne sont pas adaptées à la localisation. Pour cela, nous proposerons de les changer et de les évaluer autrement pour qu'elles soient plus adaptées au problème de localisation de fautes.

1.3. Organisation de la thèse

Après l'introduction, nous exposerons dans le second chapitre les deux principales facettes de la *Conception de Logiciels Testables*: la conception et les tests. Ces dernières seront définies et mises dans leur contexte d'utilisation.

L'état de l'art de la testabilité constitue le thème du troisième chapitre. Il s'intéressera particulièrement aux différentes définitions, aux mesures et aux facteurs qui peuvent influencer la testabilité. Il s'intéressera également aux méthodes utilisées pour améliorer la testabilité. Nous ferons suivre l'exposition de chacun de ces points par une brève discussion.

Dans le quatrième chapitre, nous considérerons la testabilité comme une activité intégrante du cycle de développement du protocole. Nous expliquerons comment le rajout de l'activité de testabilité améliore certaines activités du cycle (génération et application des tests) sans en altérer les autres activités. Nous montrerons comment on peut propager des propriétés de la testabilité de la spécification vers l'implémentation. À la fin de ce chapitre, nous introduirons les chapitres qui suivent.

Certains auteurs [Voas 91 et Free 91] ont montré qu'à partir des entrées-sorties d'un système on pouvait prévoir sa testabilité. Ceci reste valable pour tous les types de

systèmes réactifs et en particulier les protocoles de communication. Dans le cinquième chapitre, nous adopterons cette approche et nous proposerons d'étudier la testabilité d'un protocole à partir de sa spécification du service. Dans une architecture en couches, la spécification du service décrit les interactions d'un protocole d'une couche *n* avec sa couche utilisatrice. Les interactions sont souvent représentées par des diagrammes qui associent les primitives d'entrées aux primitives de sorties du protocole. Ces représentations sont assez similaires au modèle mathématique de relations. Ces dernières, associent certains éléments d'un espace d'entrée (ou domaine) à d'autres éléments de sortie appelés images de la relation. Partant de cette analogie, nous utiliserons l'algèbre relationnelle [Tars 41 et Mili 90] pour la spécification du service d'un protocole. Ce modèle est choisi pour sa simplicité et la puissance de la théorie qui lui est associée. À partir de la spécification du service (représentée sous forme relationnelle), nous dégagerons quatre propriétés de testabilité (qui influencent le coût des tests) que nous utiliserons pour classer le service du protocole. Pour chacune de ces propriétés, nous donnerons une évaluation individuelle. L'avantage de cette approche est qu'elle nous donnera une vue détaillée de la testabilité à un stade précoce du développement. Ceci nous permettra de détecter et éventuellement corriger les propriétés qui sont à l'origine d'une mauvaise testabilité. Pour atteindre cet objectif, nous proposerons des transformations qui agissent sur la (ou les) propriété(s) considéré(es) responsable(s) d'une mauvaise testabilité.

Le modèle des automates est l'un des modèles qui a été largement utilisé dans la spécification et la génération des suites de test d'un protocole de communication. Pour ce modèle la longueur de la suite de test est considérée comme une évaluation possible de la testabilité. Plus la suite est courte, moins on a à fournir d'efforts pour tester le protocole. Dans le sixième chapitre, nous exploiterons la relation entre suite de test et testabilité des protocoles modélisés sous forme d'automates. Nous proposerons des métriques individuelles de quatre éléments qui rentrent en jeux dans le calcul de la longueur de la suite du test. Ces éléments sont appelés les facteurs de testabilité. Nous classerons les spécifications suivant trois propriétés qui sont: le déterminisme, la minimalité et la complétude. Nous utiliserons cette classification ainsi que les facteurs de testabilité pour proposer un ensemble de transformations permettant d'améliorer la classe d'appartenance initiale des spécifications.

Les deux modèles utilisés dans le cinquième et sixième chapitres permettent de décrire le comportement des protocoles à un niveau d'abstraction élevé. Dans le septième chapitre, nous utiliserons le modèle des automates étendus afin d'étudier les problèmes des tests de données. Nous considérerons que la longueur d'une suite de test est proportionnelle à la difficulté de tester un protocole. Nous proposerons une méthode de transformation de la spécification qui permet la réduction de cette longueur lorsque possible.

Nous ferons dans le huitième chapitre le lien entre les aspects de détection et de localisation des fautes. Le processus de détection permet de savoir si un système contient des fautes; tandis que le processus de localisation permet de trouver leurs emplacements éventuels. Ainsi, certains facteurs de testabilité relatifs à la détection seront adaptés aux exigences de la localisation de fautes.

Nous terminerons ce travail par une conclusion générale où nous ferons une synthèse de notre travail et de ce qui pourrait être fait comme suite à ce travail.

Chapitre 2

Notions de bases de la conception et des tests

2.1 - Introduction

Le processus de développement de systèmes informatiques est composé de trois phases et d'un ensemble d'activités (voir figure 2.1 [Boch 90]). Ces phases sont: l'analyse des besoins, la conception et l'implantation [Pres 92]. Elles font partie de tous les cycles de développement de systèmes indépendamment de la nature, du domaine, de la taille et de la complexité du système à développer.

- La phase de l'analyse des besoins vise à définir le problème à résoudre et le développement de la spécification fonctionnelle. Cette dernière est produite à partir d'une description informelle du système représentée en langage naturel et de certains diagrammes. Cette phase est validée par rapport aux besoins de l'utilisateur en vue d'établir sa conformité et de vérifier sa consistance.

- La phase de conception vise à produire une spécification détaillée. Elle opère sur la spécification fonctionnelle. Le concepteur en fait une traduction sous une forme de représentation formelle (parmi plusieurs possibles). Par la suite, il raffine cette représentation jusqu'à en obtenir une qui décrit avec suffisamment de détail les structures de données, l'architecture du système, l'interface des différents modules du système et leur description algorithmique. Finalement, le concepteur génère les tests et spécifie la façon dont ils pourront être appliqués. La spécification détaillée est aussi validée par rapport à la spécification fonctionnelle. Plusieurs définitions de la conception ont été

proposées. Parmi ces définitions nous pouvons citer celle de Pressman [Pres 92] et celle de Wasserman [Wass 77]:

- *C'est le processus consistant à utiliser divers principes et diverses techniques pour définir un dispositif, un processus ou un système avec suffisamment de détails pour permettre son implantation physique* [Pres 92].
- *C'est l'utilisation de principes scientifiques, d'informations techniques et de l'imagination pour la définition d'une structure mécanique, d'une machine ou d'un système qui réalise des fonctions spécifiées à l'avance avec un maximum d'économie et d'efficacité* [Wass 77].

- L'implantation: cette étape consiste à transformer la spécification détaillée sous forme de langage de programmation exécutable. Cette transformation tient compte des paramètres de l'environnement du système: machine, interaction avec les systèmes adjacents, performance, etc. La validation de l'implantation est réalisée en se basant principalement sur les activités de tests. Ces dernières consistent à détecter les défectuosités de fonctionnement, de logique et d'implantation du système.

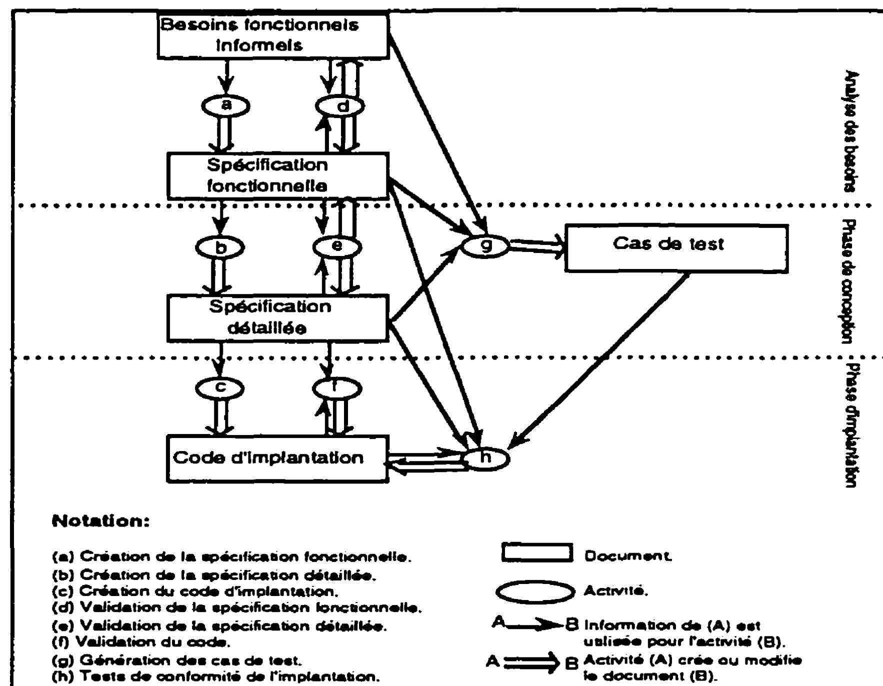


FIGURE 2.1. Activités du cycle de développement de logiciel.

Le cycle de vie de logiciel est un modèle de développement dans lequel sont placées des méthodes d'analyse, de conception, de codage, de tests et de maintenance. Les différentes phases du cycle de développement sont généralement séquentielles. Chaque phase exploite le produit délivré par la phase précédante et en délivre un nouveau produit qui à son tours sera exploité par la phase suivante. Le cycle de développement peut être considéré comme un processus de raffinement successif permettant de passer graduellement d'un certain niveau d'abstraction à un autre moins élevé. La formalisation et l'automatisation de ce processus permet de propager le comportement et les propriétés de qualité prise en compte à chaque phase du cycle.

2.2. Les critères de qualité d'un logiciel

Le rêve de toute personne voulant développer un logiciel est de produire un logiciel de qualité à 100%. La qualité d'un logiciel est encore une notion floue, car elle est généralement confondue avec la notion d'exactitude de logiciel ou de logiciel sans erreurs. Il est donc plus simple de la décomposer en plusieurs propriétés moins abstraites et qui peuvent être estimées ou évaluées. Plusieurs propriétés permettent d'affirmer qu'un logiciel est de qualité. Ces propriétés sont appelées les critères de qualité du logiciel. Parmi les critères de qualité on peut citer par exemple: la facilité d'utilisation, la rapidité, la lisibilité, la modularité, la testabilité, etc. Les critères de qualité que nous venons de citer se décomposent en deux classes distinctes [Meye 88]. La première est la classe des critères de qualité externes comme par exemple la facilité d'utilisation, la rapidité, etc. La deuxième est la classe des critères de qualité internes comme la lisibilité, la modularité, etc. La satisfaction d'un des critères de qualité externe dépend parfois de la satisfaction de l'un ou de plusieurs critères de qualité interne.

2.2.1. Critères de qualité externe

Dans ce paragraphe nous allons présenter quelques critères de qualité de logiciel externe tels qu'ils sont définies dans [McCa 77], [Meye 88] et [Pres 92]:

- l'exactitude: c'est la capacité du logiciel d'accomplir exactement ses fonctionnalités, comme elles ont été définies dans la description des besoins et la spécification. En plus, le logiciel ne doit pas contenir d'erreurs de logique (blocage, boucles infinies, etc.),

- la robustesse: c'est la capacité du logiciel de fonctionner même dans des conditions anormales,

- facilité d'extension (*extendabilité*): c'est la facilité avec laquelle le logiciel peut s'adapter à des changements dans les spécifications,

- la réutilisabilité: c'est la capacité du logiciel d'être entièrement réutilisable pour de nouvelles applications,

- la compatibilité: c'est la facilité de combiner le logiciel avec d'autres produits,

- l'efficacité: c'est l'aptitude du logiciel à utiliser efficacement (d'une façon optimale) les ressources matérielles comme les processeurs, les mémoires, les moyens de communication, etc.,

- la portabilité: c'est la facilité avec laquelle le logiciel peut être transféré dans divers environnements matériels et logiciel,

- l'intégrité: c'est la capacité du logiciel de protéger ses diverses composantes contre les accès et les modifications non-autorisées,

- la facilité d'utilisation: c'est la facilité d'apprendre comment utiliser le logiciel, de préparer les données d'entrée, d'interpréter les résultats et de réagir dans les cas d'erreurs d'utilisation,

- la facilité de vérification: c'est la facilité de préparer les procédures d'acceptation durant la phase de validation et d'opération, particulièrement la facilité de préparer les données de test, les procédures pour la détection des défaillances et de traces,

- la facilité de maintenance: un programme est facile à maintenir si toute modification des fonctionnalités du programme est facile à opérer. Un programme modulaire est généralement facile à maintenir ; on peut facilement localiser les modules où il faut effectuer les changements. En plus, les changements n'affectent que certains modules spécifiques,

- la fiabilité: la fiabilité est définie comme étant la capacité d'un système à accomplir le comportement défini par l'utilisateur et spécifiés par le concepteur. Un programme fiable est nécessairement correcte, complet et consistant,

- la flexibilité: c'est l'effort nécessaire pour modifier un programme opérationnel,
- l'interopérabilité: c'est l'effort nécessaire pour lier un système à un autre.

2.2.2. Critères de qualité interne

Les critères de qualité externes (présentés ci-dessus) ne peuvent être réalisés que si certains critères de qualité internes sont satisfaits. Ces derniers caractérisent un produit spécifique du cycle de développement par exemple, la spécification, l'implantation, etc.

Dans ce qui suit, nous présentons une liste (non exhaustive) des critères de qualité internes tels que définies dans [Troy 81], [Somm 92] et [Pres 92]

- le couplage: c'est la mesure de la force d'interconnexion des différents modules du système. Le degré de couplage d'un système dépend de la complexité et du type d'interconnexion des modules du système. Le meilleur cas est lorsque les modules sont indépendants, ils communiquent ensemble en s'échangeant des données,
- la cohésion: c'est la mesure du degré de liaison des éléments à l'intérieur d'un module. Idéalement un module ne doit décrire qu'une seule fonctionnalité,
- la modularité: c'est la mesure associée à la décomposition du système en sous-fonctions et modules. L'objectif de la modularité est de décomposer le système en parties qui sont fonctionnellement cohésives et qui sont modifiables indépendamment les unes des autres,
- la complexité: la complexité d'un programme est introduite en partie par la difficulté de codage de l'algorithme et la taille du programme associé. La complexité est fonction du nombre de chemins d'exécution possibles du programme et la difficulté de les exécuter avec des entrées aléatoires. Elle dépend de la taille, des éléments de contrôle, de leur nombre et de leur domaine,
- la taille: c'est la mesure qui permet de quantifier le nombre de composantes granulaires du produit. Ces composantes peuvent être les instructions, les procédures, les fonctions, etc.,
- la facilité de compréhension: la facilité de compréhension est définie comme étant la facilité avec laquelle on peut comprendre les fonctions du programme et comment ce

dernier peut accomplir ses fonctions. Un programme est compréhensible si c'est facile à un lecteur de déterminer les objectifs, les hypothèses, les contraintes, les entrées, les sorties, les composantes, les relations avec d'autres programmes et l'état du programme. La compréhension dépend de deux facteurs: l'aptitude du lecteur ou du programmeur et la forme du programme. L'aptitude du programmeur dépend de son expérience du langage de programmation et du domaine d'application du programme. Quant à la forme du programme, on peut la caractériser par les propriétés suivantes: modularité, style consistant, programmation simple, utilisation des noms de variables et de procédures ayant un sens et une structure simple,

- la structuration: un programme est dit bien structuré s'il respecte la philosophie de la décomposition modulaire et si la construction des modules et leurs interactions respectent certaines propriétés. Un programme est dit bien structuré si: il est structuré en une hiérarchie de modules, l'échange entre les modules est simple et minimal, la construction des modules est restreinte aux constructions standards de base (concaténation, sélection, répétition), chaque variable du programme est utilisée dans un seul module ou fonction et son champ d'action est limité, le traitement des cas d'erreurs ne perturbe pas le flux de contrôle sauf dans le cas d'erreur non récupérable et enfin le programme est bien documenté.

2.3. Spécificité des protocoles de communication

Dans un système de communication, la coopération de plusieurs processus par des échanges de messages est régie par un ensemble de conventions et de règles qu'on appelle protocole. Un protocole définit donc les relations entre les comportements des différents processus de l'activité d'un système, via leur interface observable. L'information sur l'état d'un processus peut être déduite en observant ses entrées et ses sorties (simples ou séquences). Les systèmes distribués utilisent beaucoup les protocoles. Physiquement les entités composantes du système distribué peuvent être distantes, l'échange de messages est donc l'unique moyen de coordonner les activités des processus.

Le protocole est donc un logiciel dédié à la communication entre les processus. Son cycle de vie est similaire à celui de tout logiciel (figure 2.1). L'ingénierie de protocole accorde une importance particulière à certaines activités du cycle de vie dont la spécification et les tests. Au cours de l'activité de spécification, des méthodes formelles sont utilisées pour produire une spécification abstraite du comportement du protocole.

Cette spécification est utilisée comme base pour d'autres activités comme la validation, l'implantation et la génération des tests.

Plusieurs raisons encouragent l'utilisation des méthodes formelles pour la spécification du comportement d'un protocole. La première raison est due aux différentes interprétations que les implanteurs peuvent donner à une même spécification. Ceci est due à l'ambiguïté des langages naturels et des méthodes informelles. La deuxième raison est la possibilité d'automatisation des activités de vérification et de validation. D'autres raisons peuvent renforcer le besoin d'utiliser les méthodes formelles telle que la génération semi-automatique du code (implantation).

Comme nous venons de le préciser, le protocole est un logiciel particulier dédié à la gestion de la communication entre des processus s'exécutant sur une ou plusieurs machines distribuées. Les critères de qualité classiques (Section 2.2) restent donc valides pour les protocoles. En plus, vue l'importance particulière accordée aux tests de conformité, certains critères spécifiques peuvent être ajoutés. Ces critères sont associés à la facilité des tests des protocoles (testabilité): les suites de test sont courtes et faciles à générer, les suites de test ne sont pas redondantes, les sorties sont faciles à interpréter, les fautes sont faciles à localiser, etc. D'une façon un peu plus formelle, nous pouvons citer deux critères de qualité très utilisés dans l'ingénierie des protocoles, ces critères sont l'observabilité et la contrôlabilité [Dsso 86], ils sont définis comme suit:

- la contrôlabilité est la facilité avec laquelle on peut acheminer des données (en particulier les tests) d'une des entrées principales du module implantant le protocole vers les différentes parties du module,

- l'observabilité est la facilité d'acheminer les résultats (de l'application) des données d'entrée vers une des sorties principales du module implantant le protocole.

Dans ce qui suit, nous allons étudier l'un des critères de qualité des logiciels appelé testabilité. Nous verrons comment le prendre en compte lors de l'étape de conception (*DFT: Design For Testability*). L'étude de la testabilité que nous effectuerons portera sur tous les types de logiciel et au besoin, nous donnerons les spécificités des protocoles de communication.

2.4. La conception de logiciel testable

La conception de systèmes testables décrit les techniques de conception utilisées pour optimiser le coût du test du produit résultant [Prad 86].

La conception de systèmes testables ou la *DFT* (*Design For Testability*) est donc un domaine à deux dimensions: la dimension conception et la dimension tests. Ces deux dimensions ne sont pas complètement compatibles, les forces de l'un peuvent être des faiblesses de l'autre. L'exemple le plus frappant est celui du couplage, d'après Miller et al. [Mill 93] une conception modulaire ayant un faible degré de couplage (échange minimale d'information entre les modules) est considérée comme idéale lors de la conception. Par contre, une implémentation issue d'une telle conception est difficile à tester vue que les tests nécessitent un échange maximal d'information entre modules. Pour concevoir un logiciel testable, il faut donc prendre en considération les deux dimensions de ce problème soit la conception et les tests. Ceci nécessite l'optimisation des critères de qualité de chaque dimension en favorisant certains critères s'il y a matière à incompatibilité.

Au cours des prochaines sections, nous donnerons les définitions et les méthodes de base des phases de conception et de tests.

2.5. La conception du logiciel

Durant les dernières années, un intérêt particulier a été porté aux méthodologies de conception et leur lien avec la qualité du logiciel. Dans cette section, nous résumerons les approches et techniques de base de la conception.

2.5.1. Processus de conception

L'étape de conception comporte un certain nombre d'activités. Ces activités sont essentielles à l'analyse de systèmes complexes [Somm 92]. Les principales activités de la conception sont:

- conception architecturale: c'est l'activité consistant à identifier les principales composantes du système (sous-systèmes), à les documenter et à définir les liens entre elles. Le résultat de cette activité est l'architecture du système,

• spécification: Pour chaque sous-système une description abstraite des fonctionnalités du produit sont produites,

• conception des interfaces: spécification et documentation des interfaces entre sous-systèmes. Ces interfaces doivent être non ambigus de façon à permettre l'utilisation des sous-systèmes sans connaissance de leurs détails internes,

• conception des composantes: pour chaque sous-système, on réparti les services sur ses différentes composantes,

• conception des structures de données: lors de cette étape, les structures de données de l'implantation sont spécifiées en détail,

• conception des algorithmes: pour chaque composante du système on spécifie les algorithmes qui fournissent les services définis lors de la conception des composantes.

2.5.2. Les différentes phases de la conception

La conception peut être décomposée en trois phases: la conception de haut niveau, la conception de niveau intermédiaire et la conception détaillée [Sinc 84]. Toutes les activités définies dans la section précédante sont effectuées au cours de ces phases.

• Conception de haut niveau: au cours de cette phase, le choix entre les différentes alternatives de conception est établi. L'architecture du système est identifiée. Elle est décrite à un haut niveau d'abstraction. Les décisions de nature à toucher à la structure du système sont prises et les composantes du système sont alors identifiées. Le document de spécification est analysé pour déterminer si la structure proposée est compatible avec les prérequis. Lors de cette étape, différents modèles de représentations peuvent être utilisés (exemple le diagramme de flot de données). Ces représentations définissent l'architecture du système. Au cours de cette étape, les entrées-sorties du système sont identifiées ; les principaux chemins de flot de données sont alors établis. Finalement, les structures de données globales sont identifiées.

• Conception de niveau intermédiaire: c'est un processus itératif qui raffine la structure obtenue lors de l'étape précédante. Chaque unité du système est alors

décomposée en sous-unités. Les interactions entre les différentes parties du système sont établies. Ces interactions sont définies en terme de contrôle et de données. Ceci implique une spécification des interfaces.

• Conception détaillée: cette étape est aussi appelée conception de bas niveau. En effet, c'est ici que les algorithmes détaillés et la définition des structures de données internes aux unités de programme sont développés. Cette étape doit faciliter l'étape d'implantation en délivrant au programmeur un produit facile à traduire en langage de programmation.

2.5.3. Principes de base de la conception

Toutes les méthodes et les techniques de conception se basent sur les principes suivants [Pres 92]:

• données et fonctions: les méthodes de conception se basent sur le domaine des données ou celui des fonctions pour partitionner le système en sous-systèmes. Ils servent donc comme guide pour la composition-décomposition,

• abstraction: c'est le fait d'extraire les propriétés essentielles d'un problème en omettant les détails moins importants. La conception de logiciel utilise ce principe à plusieurs niveaux en fonction du degré de détail voulu,

• raffinements successifs: c'est un principe qui est de nature itératif. Il permet de décomposer un système itérativement et par niveau de détail croissant. Le raffinement successif utilise les principes d'abstraction de données et de fonctions,

• primitives: ce sont des unités de programme standards qui, par composition, permettent de construire un système en entier,

• modularité: la modularité n'est pas simplement la décomposition d'un système en sous-systèmes (ou modules plus petits). Elle doit produire des modules largement indépendants. L'architecture obtenue doit être simple. Elle doit faciliter les changements, la localisation des erreurs et la traduction en langage de programmation,

• indépendance fonctionnelle: l'architecture du système doit être composée d'un ensemble de modules indépendants et représentant chacun une seule fonction ou sous-fonction.

2.5.4. Méthodes de conception

La conception est autant un art qu'une science. La conception exige une certaine compétence technique pour maîtriser et faire le meilleur usage des algorithmes existants et des notions formelles qui permettent de les représenter. Le processus de conception est aussi un art où l'humain utilise son intelligence, son savoir faire et son intuition pour aboutir à un bon produit. Quoi qu'il en soit, le concepteur a à sa disposition des lignes directrices, des techniques et des méthodes qu'il doit appliquer et combiner avec adresse pour obtenir de bons résultats. Dans ce qui suit, nous présentons les plus importantes méthodes de conception existantes [Pres 92].

• Conception orientée flot de données: c'est une méthode qui utilise les caractéristiques du flot d'information d'un système pour dériver sa structure. Le système initial est représenté par un diagramme de flot de données qui sera progressivement transformé en structure de programme. Cette transformation est faite grâce à deux techniques de raffinement et de factorisation: l'analyse transformationnelle et l'analyse transactionnelle. La première technique est appliquée au flot d'information dont les données d'entrée sont progressivement acheminées vers un centre de transformation constitué d'un ou de plusieurs processus. Après les traitements, les résultats sont acheminés vers des sorties. La deuxième technique est appliquée au flot d'information dont les données d'entrée sont acheminées vers un processus qui les aiguillonne vers un chemin parmi plusieurs possibles.

• Conception orientée objet: la méthodologie de développement orientée objet se décompose en quatre phases:

phase 1: elle est informelle ; elle consiste dans l'identification du problème ainsi que de ses éléments importants.
phase 2: elle consiste dans la modélisation du problème par un ensemble d'entités et des relations qui les lient.

phase 3: on précise les opérations offertes par chaque objet qui a été défini dans la phase précédante.
phase 4: on définit le comportement des objets.

• Conception orientée structures de données: Cette méthode agit sur le domaine de l'information; elle utilise la structure de l'information pour guider la décomposition.

• Conception d'interfaces utilisateur: l'interface utilisateur est un facteur de base pour évaluer la qualité d'un logiciel. Une mauvaise interface (difficile à utiliser) peut mener l'utilisateur à commettre des erreurs. Une bonne conception d'interfaces doit prendre en considération les capacités humaines telles que la perception, la compétence, le profil ainsi que d'autres critères.

• Conception temps réel: un système temps réel est un système qui doit répondre à des événements en temps réel. La qualité d'un tel système ne dépend pas uniquement des traitements qu'il est capable d'effectuer, mais aussi du temps nécessaire pour les effectuer. La conception temps réel peut inclure tous les aspects de la conception conventionnelle, en plus, elle introduit les nouveaux critères suivants: représentation des interruptions et des changements de contexte, concurrence (multitâche, multiprocesseur), synchronisation et communication entre les processus, large variation dans le domaine des données et du taux de communication, représentation des contraintes temporelles.

2.5.5. Les techniques formelles de description

La phase de l'analyse des besoins délivre à la phase de conception une spécification informelle. Dans le cadre des protocoles de communication, la majeure partie des normes développées par l'*ISO*, l'*ITU*, l'*IEEE*, etc., ont été rédigées en langage naturel et complétées par des tables et des schémas. Pour éviter l'ambiguïté dans l'interprétation de la spécification informelle, la phase de conception utilise souvent des techniques formelles pour la spécification du protocole. Plusieurs techniques formelles ont été utilisées [Boch 80b]: automates, réseaux de Pétri, les langages de programmation, etc. Cependant, trois langages ont été normalisés: *ESTELLE* [Budk 86], *LOTOS* [Bolo 87] et *SDL* [Beli 89].

2.5.6. Qualité d'une conception

Tout ce qu'on a vu dans les sections précédantes a un lien (direct ou indirect) avec la modularité. Une conception modulaire réduit la complexité, facilite les changements et accélère le processus d'implantation grâce au parallélisme qu'on peut utiliser lors du développement. Une structure modulaire est obtenue en utilisant différentes méthodes de composition/décomposition [Bera 91c]. Ces dernières jouent un rôle clé dans les méthodes de conception. Ce rôle est double, elles permettent d'une part d'analyser des systèmes complexes et d'autres parts elles sont d'excellentes techniques pour la construction des systèmes. Un système bien décomposé (respectivement composé) est plus facile à construire et est plus stable lors des opérations de maintenance. Le produit obtenu par la conception doit être modulaire. Les modules doivent être les plus indépendants possibles. Ils doivent communiquer à travers une interface bien définie ne laissant aucune place à l'ambiguïté. Une mauvaise modularité introduit une complexité additionnelle au système à développer. Cette complexité est due aux raisons suivantes:

- Un module se compose d'un ensemble de fonctions en interaction. Ceci rend impossible la distinction entre ces fonctions (qui fait quoi et comment),
- Une fonction qui n'a pas été identifiée lors de la conception est distribuée sur différents modules ce qui altère leur indépendance fonctionnelle et leur logique,
- Les modules interagissent et opèrent sur des données communes de façon qu'à un certain moment on ne peut plus savoir qui les utilise.

Deux propriétés de base sont utilisées pour évaluer une conception [Somm 92]: la cohésion et le couplage (voir Section 2.2).

2.6. Les tests

2.6.1. Introduction

Le cycle de développement de systèmes se compose d'un certain nombre d'activités dont une bonne partie est manuelle. La partie manuelle entraîne inévitablement un certain risque d'erreurs. Pour cela, les chercheurs en génie logiciel ont proposé d'inclure à la fin de chaque phase du cycle de développement une étape de vérification. Cette dernière permet de valider les résultats de la phase en cours avant de passer à la phase suivante.

L'étape de spécification est l'une des plus importantes du cycle de développement. Les erreurs introduites au niveau de la spécification se répercutent automatiquement sur le produit final. Depuis quelques années les spécifications sont de plus en plus décrites par des langages formels et normalisés appelés *Techniques Description Formelle TDFs* (*FDT*: *Formal Description Techniques*): par exemple, *Estelle* [Budk 86], *LOTOS* [Bolo 87], *SDL* [Beli 89], Z [Spiv 92], VDM [Jone 86]. Les propriétés d'une spécification formelle peuvent être vérifiées par simulation ou à l'aide de techniques de vérification partielles ou exhaustives, comme l'analyse d'accessibilité [Boch 80b, Goud 85, etc]. Au besoin, si une spécification n'a pas toutes les propriétés voulues, elle sera corrigée et réécrite. Le processus de vérification de la spécification se poursuit jusqu'à ce que la spécification ait toutes les propriétés voulues. La vérification d'une spécification formelle est un processus coûteux, complexe et souvent non décidable.

Les *TDF*s ont été introduites non seulement pour réduire l'ambiguïté engendrée par les langages naturels, mais aussi pour automatiser le plus possible chacune des étapes du cycle de développement. Cependant, même à partir d'une spécification formelle, la production d'une implantation demeure une tâche délicate. En effet, une spécification est une description abstraite, elle décrit ce que le système est supposé accomplir, sans fixer comment il doit le faire. Cette description abstraite peut être réalisée de plusieurs manières différentes, en utilisant différents langages de programmation et différentes architectures. Le processus de raffinement d'une spécification pour l'obtention d'une spécification moins abstraite puis d'une implantation est complexe et est sujet à différents types d'erreurs.

L'ultime recours pour s'assurer qu'un produit fait bien ce que l'on attend de lui est de le tester. L'effort de test représente une part significative du coût de développement d'un système. L'activité de formalisation du test fait l'objet de travaux de recherche depuis une bonne trentaine d'années. Cette activité utilise principalement les techniques de test de logiciel ainsi que celles du modèle de machines à états finis. Pour tester un système, il faut être capable de définir ce que nous attendons de lui: c'est ce que nous appelons sa spécification. Il nous faut ensuite définir une procédure permettant de s'assurer que le système sous test, appelé aussi implantation, est conforme à sa spécification.

L'étape de test représente la dernière activité du cycle ; elle suit immédiatement celle d'implantation. Elle permet de vérifier l'implantation avant de délivrer le produit. L'étape

de test est donc l'étape qui garantit une certaine qualité d'une implantation. Elle consiste à valider le comportement de l'implantation par rapport à des données d'entrées spécifiques (cas de test) sélectionnées à l'avance. L'objectif de cette étape est alors la stimulation des fautes, l'observation de leurs manifestations (erreurs), leurs localisations et éventuellement leurs corrections. Dans la littérature, les deux dernières activités (localisation et correction des erreurs) peuvent être regroupées dans une étape indépendante appelée "débogage" [Myer 79].

2.6.2. Définition des tests et du débogage

Plusieurs recherches sur les tests ont été réalisées. On y trouve une multitude de définitions plus ou moins différentes. Ci-dessous, nous en présentons quelques unes.

- *Le processus de test consiste en une collection d'activités qui visent à démontrer la conformité d'un programme par rapport à sa spécification. Ces activités se basent sur une sélection systématique des cas de test et l'exécution des segments et des chemins du programme* [Wass 77]

- *Les tests représentent l'essai d'un programme dans son milieu naturel. Ils nécessitent une sélection de données de test à soumettre au programme. Les résultats des traitements de ces données sont alors analysés et confirmés. Si on découvre un résultat erroné, l'étape de débogage commence* [Clar 76].

- *L'objectif du processus de test est limité à la détection d'éventuelles erreurs d'un programme. Tous les efforts de localisation et de correction d'erreurs sont classés comme des tâches de débogage. Ces tâches dépassent l'objectif des tests. Le processus de test est donc une activité de détection d'erreurs, tandis que le débogage est une activité plus difficile consistant en la localisation et la correction des erreurs détectées* [Whit 78].

2.6.3. Définition des fautes, des erreurs et des défaillances

Les notions de fautes, d'erreurs et de défaillances du matériel sont fortement liés. Leur définition a été adaptée du matériel [Arse 80], [Prad 86], [Lapr 91], etc.:

- *La faute représente une condition anormale dont la manifestation est une erreur. Elle consiste en un état logique du système différent de celui attendu.*

- *L'erreur est la conséquence directe ou indirecte d'une faute, elle en est la manifestation.*
- *La défaillance est l'effet d'une erreur sur le service, elle survient lorsque le système a un comportement erroné.*

Les tests visent à stimuler les fautes, observer leur manifestation, localiser les erreurs et éventuellement les corriger. Les tests minimisent la probabilité que des utilisateurs potentiels du produit puissent y observer une défaillance. Les fautes peuvent être classées selon leur durée, leur origine, leur nature et leur étendue. Du point de vue durée, une faute peut être transitoire, intermittente ou permanente. Une faute commise devient une erreur latente aussi longtemps qu'elle n'a pas été activée. La nature de la faute dépend de son comportement. L'étendue de la faute peut être locale en affectant un seul module, ou globale en affectant plusieurs modules interconnectés. L'origine de la faute est soit humaine soit physique. Les fautes peuvent être simples ou multiples. Les fautes multiples peuvent rendre le processus de test et de diagnostic plus complexes.

2.6.4. Les types de test

Il existe plusieurs sortes de test de logiciel. Chaque type vise à vérifier un aspect particulier du comportement du logiciel. Ces aspects sont plus ou moins indépendants et peuvent être traités individuellement. Parmi ces aspects, nous pouvons citer, la conformité du logiciel à la spécification de référence, l'absence d'erreurs de logiques, les fonctions manquantes, la bonne interaction avec l'extérieur, etc.

Dans ce qui suit, nous présentons les principaux types de tests de logiciel et nous introduisons les tests de matériel.

Tests unitaires

Ces tests visent à vérifier individuellement chaque module [Myer 79]. Ils permettent de tester l'interface et l'intérieur du module soit: les structures de données locales, les chemins, les instructions, les conditions, les boucles, les entrées sorties et les traitements en cas d'erreurs. Ces tests sont effectués par l'implanteur même, puisqu'ils nécessitent la disponibilité du code et les détails internes du logiciel au complet.

Tests d'intégration

Après que les modules aient été testés individuellement, on vérifie leurs comportement ensembles, c'est-à-dire on les intègre ensemble et on teste si leur interaction est conforme à la spécification de référence. L'intégration des modules se fait de plusieurs façons: incrémentale ascendante (*bottom-up*), ou descendante (), ou hybride ascendante et descendants (*sandwich*), ou intégration du tout à la fois. Ces tests sont moins exigeants que les tests unitaires et ne nécessitent que la connaissance de la structure modulaire et des interfaces de modules du logiciel.

Tests de conformité ou de validation

Ce genre de tests occupe une place très importante dans le domaine de l'ingénierie du protocole [Raient 87]. Il consiste à vérifier la conformité de l'implantation par rapport à la spécification de référence. Ces tests se basent sur le comportement extérieur de l'implantation. Ils consistent à appliquer un ensemble d'entrées spécifiques (cas de test) au système à tester et à vérifier la conformité de ses sorties. Généralement, un tel test ne peut pas être exhaustif vu le nombre important de cas à traiter. Les entrées sont alors sélectionnées de façon à obtenir une couverture maximale de parties du système (fonctions, transitions, états, etc.) et à détecter le maximum de fautes.

Tests de l'usager

Ce sont les tests effectués au niveau de l'usager en vue de vérifier si le produit final répond bien à ses besoins. Généralement, ce test consiste à essayer le système sur des situations réelles et propres à l'environnement de l'usager. Le résultat de ce test est l'acceptation, l'acceptation conditionnelle ou le refus du produit.

Tests d'interopérabilité

Ces tests vérifient si le système développé interagit correctement avec d'autres systèmes extérieurs dont il est en relation. Ces tests nécessitent l'observation des échanges entre les systèmes. Ceci se fait par l'intermédiaire de modules spéciaux placés entre le système à tester et les autres systèmes dont il est en relation. Ces modules sont appelés les arbitres.

Tests de performance

Ces tests ont pour but de vérifier tout ce qui se rapporte à la performance du système comme le débit, le temps de réponse du système, etc. La performance est vérifiée en soumettant le système à différentes conditions d'utilisation. Ces conditions sont parfois extrêmes. Ce genre de tests est très important pour les systèmes à temps réel.

Tests de robustesse

Ces tests s'intéressent au degré de résistance de l'implantation à des événements externes ou à des erreurs non prévues par la spécification. C'est-à-dire les capacités du système à fonctionner même dans des situations non prévues par la spécification de référence.

Tests de matériel

Comme dans le logiciel l'étape de test est celle qui garantit la qualité d'un circuit. En effet, avant de commercialiser un circuit, il faut s'assurer de l'exactitude de son fonctionnement. Cette vérification se fait au cours de l'étape de test. Cette étape est constituée de trois principales phases: la génération, l'application et la vérification du test [Prad 86]. La génération de test consiste à utiliser des techniques algorithmiques ou aléatoires pour générer des entrées de test. Ces dernières sont appelées *vecteurs de test*. La deuxième phase consiste à appliquer les vecteurs de test au circuit à vérifier. Finalement, au cours de la dernière phase, on compare les réponses du circuit au vecteur de test aux sorties produites par une unité de référence.

2.6.5. Les principales philosophies de test

Dépendamment de ce qu'on veut tester dans un système (la logique, la structure, les fonctions ...), les tests varieront et seront orientés parfois au code, parfois à l'architecture et parfois aux entrées-sorties. Les tests de conformité par exemple ne s'intéressent qu'aux entrées-sorties du système, ces tests sont dits tests *boîte noire*. D'un autre côté, les tests d'intégration s'intéressent plutôt à l'architecture et à la structure modulaire du système, ces tests sont dits tests *boîte grise*. Finalement, les tests unitaires se basent sur le code de

chaque module pour mener à bien leur vérification, ce genre de tests sont appelés les tests *boîte blanche* [Myer 76].

D'autre part, lors de l'intégration des différentes parties du logiciel, on utilise certaines techniques de tests basées sur les approches d'intégration incrémentales. Ces dernières peuvent être soit ascendantes (*bottom-up*) soit descendantes (*top-down*) ou en *sandwich* dépendamment de l'architecture du système.

- **Tests boîte blanche**

Ils sont aussi appelés tests structurels. Ces tests sont effectués sur des produits dont la structure interne est accessible. Ils s'intéressent principalement aux structures de contrôle et aux détails procéduraux. Ils permettent de vérifier si les aspects intérieurs de l'implantation ne contiennent pas d'erreurs de logique. Ils vérifient si toutes les instructions de l'implantation sont exécutables (contrôlabilité). Ils permettent aussi de vérifier les chemins de base de l'implantation en vue de détecter les erreurs de logique tels que les blocages, les boucles infinies, etc. Plusieurs techniques de tests peuvent être utilisés dans cette catégorie, nous citons par exemple: les tests de chemins de base, les tests de conditions, les tests de flot de données, les tests de boucles, etc.

Les tests de chemins de base:

Cette technique permet de tester les chemins de base du logiciel. Il existe différentes méthodes pour déterminer les chemins de base d'un logiciel. Nous citons par exemple la complexité cyclomatique [McCa 89]. Cette dernière permet de définir les chemins de base d'un logiciel et de borner le nombre de cas de tests nécessaire à la vérification de toutes les instructions du programme (les exécuter au moins une fois).

Les tests de condition:

Ces tests permettent de vérifier toutes les instructions conditionnelles du programme. Ils vérifient l'accessibilité des branchements conditionnels et permettent de détecter des erreurs sur les conditions et les instructions se rapportant aux conditions.

Les tests de boucle:

Ces tests permettent de valider toutes les boucles du programme. Ils permettent de détecter les erreurs d'initialisation, d'indexage et de conditions d'arrêt des boucles du programme.

Les tests de flot de données:

Ces tests vérifient le flot de données du programme [Ural 87a] et [Weyu 93], etc. Les chemins de base du système sont considérés comme une succession de transformation sur la valeur des variables du logiciel. Cette suite de transformations (ou chemin) est alors vérifiée pour voir si elle aboutit au bon résultat.

Les tests de régression:

Ce genre de test est nécessaire pour vérifier l'impact des changements apportés à un système. En apportant des changements à un système, on peut introduire de nouvelles fautes. Les tests de régression vérifient si le programme étendu est encore conforme à sa spécification de référence. Un sous-ensemble des tests déjà existant peut être réutilisé pour effectuer les tests de régression. D'autres tests seront rajoutés en adaptant d'autres tests déjà existant ou en créant d'autres.

- **Tests boîte noire**

Ils sont aussi appelés tests fonctionnels. L'objectif de ces tests est la vérification de la conformité des fonctionnalités de l'implantation par rapport à une spécification de référence. Les détails internes des logiciels ne sont pas inspectés. Ils ne s'intéressent qu'au comportement externe du système à tester. Ces tests permettent de détecter une large gamme d'erreurs comme: fonctions incorrectes ou manquantes, erreurs d'interfaces, erreurs de structures de données ou d'accès aux bases de données externes, erreurs de performance, erreurs d'initialisation ou de terminaison.

Il existe plusieurs techniques de sélection des cas de test. L'idée de base est d'échantillonner les données en classes plus ou moins indépendantes qui serviront de bases pour le choix des cas de test les plus significatifs (qui couvrent le plus de classes). Généralement, la sélection est faite à partir de la spécification du système décrite soit en

langage naturel, en langage formel (ESTELLE, LOTOS ou SDL), ou sous un formalisme mathématique tel que les automates à états finis (*FSM*). Comme technique de sélection de cas de test, nous allons présenter quelques unes.

Partitionnement en classes d'équivalence:

Cette technique consiste à partitionner le domaine des données d'entrées en classes d'équivalences à partir desquelles on extrait les cas de test. L'idéal est que chaque cas de test puisse couvrir une classe d'erreurs. Le partitionnement en classes est guidé par des heuristiques basées sur les classes d'erreurs à couvrir.

Analyse des valeurs limites:

Généralement l'implanteur accorde peu d'importance aux valeurs limites des domaines d'entrées. Pour ces raisons, un grand nombre d'erreurs s'y accumulent. L'analyse de ces valeurs nous permet donc de détecter un certain nombre d'erreurs. Cette technique peut être complémentaire à la technique de partitionnement en classes d'équivalences.

Extraction de cas de test basée sur les automates à états finis:

Les automates à états finis sont bien adaptés à la spécification de logiciels et particulièrement les protocoles de communication. Les *FSM*s modélisent les modules du système sous forme d'un graphe étiqueté, où les noeuds représentent l'état du système (valeur des variables, contenu des files d'attentes, etc.) et les arcs représentent les transitions ou les événements internes ou externes qui, lorsqu'ils sont traités, peuvent changer l'état du système. Chaque module est représenté par un *FSM* et le comportement global du système peut être obtenu sous certaines contraintes par composition des *FSMs*. Cette méthode de spécification est une méthode mathématique, elle est non ambiguë et est facilement automatisable. Il existe plusieurs techniques d'extraction de cas de tests à partir des *FSM* représentants les spécifications de systèmes. L'avantage de telles techniques est la possibilité de les automatiser. Les cas de test ainsi générés sont appliqués à l'implantation en vue de tester toutes les transitions.

Génération de vecteur de test pour le matériel:

Ceci consiste à trouver une combinaison de valeurs "*vecteur de test*" qu'on appliquera aux entrées primaires du circuit pour stimuler les pannes et propager leur effet vers une ou plusieurs sorties primaires. En utilisant les modèles de pannes, on réussit à limiter les valeurs d'entrées (vecteurs de test) dépendamment des pannes qu'on cherche.

- **Tests boîte grise**

Cette approche est utilisée lorsque la structure modulaire du logiciel est accessible, c'est-à-dire lorsqu'on peut observer les interactions entre les modules qui composent le logiciel. L'observation des interactions se fait sur des points spécifiques du logiciel appelés points d'observation. Cette approche n'a donc pas besoin des détails internes des modules, mais exige l'observabilité de la structure du logiciel via les point d'observation. La visibilité de la structure interne du logiciel peut beaucoup aider lors de l'étape de débogage.

- **Tests ascendants**

C'est une stratégie classique de test. Elle consiste à tester les modules ensuite les sous-systèmes et en dernier le système en entier. La première phase consiste à tester les modules dans l'objectif de découvrir des erreurs de logique, de fonctionnalité ou de structure. Les modules sont testés individuellement dans un environnement qui simulera le sous-système qui les englobe. Par la suite on passe au tests des sous-systèmes en vérifiant les interactions entre les différents modules du sous-système. Ils s'intéressent principalement aux interfaces des modules et aux échanges entre ces modules. Le test des sous-systèmes est accompli par un processus répétitif qui permet d'intégrer les modules de bases aux sous-systèmes, les sous-systèmes d'un certain niveau aux sous-systèmes d'un niveau plus haut et ainsi de suite jusqu'à tester tout le système. Ces tests sont donc hiérarchiques et doivent se faire du bas vers le haut. Les techniques de test ascendants utilisent plusieurs types de test comme: les tests unitaire pour tester les modules et les tests d'intégration pour tester les sous-systèmes.

• **Tests descendants**

Cette approche est moins naturelle que la précédante. Elle commence par tester le programme principal avec ses sous-routines immédiates. Après cela, on passe aux tests de niveaux plus bas. Ces derniers se basent sur ce qui a été déjà validé pour tester les modules d'un niveaux plus bas. Ces techniques ont besoin d'utiliser des modules factices qui seront placés à un niveau plus bas que les modules à tester et qui ont pour rôle de récolter les sorties des modules à tester. Cette stratégie suppose que la structure du système est hiérarchique. Dans plusieurs cas, elle est impossible à effectuer, car les données d'entrées qui servent à tester un module proviennent des modules appartenant à des niveaux supérieurs. Le problème est de trouver les entrées adéquates des modules de niveaux supérieurs qui produisent les entrées de test pour les modules en question.

2.6.6. Les modèles de fautes

Les tests ont pour principal objectif la détection d'erreurs d'une implantation. Une erreur est la conséquence directe ou indirecte d'une faute (défectuosité dans le système). Pour tester le système, on doit donc tenir compte des différentes fautes possibles. Ces dernières peuvent être nombreuses et complexes. De plus, elles peuvent engendrer la même manifestation (erreur). Ceci nous mène à classer les fautes d'après les erreurs qu'elles induisent. Elles seront alors réunies au sein de modèles différents appelés modèles de fautes [Boch 91d]. Ces derniers permettent de décrire l'effet des fautes à un niveau d'abstraction élevé. Ceci aura pour avantage de réduire le nombre de possibilités à traiter lors de la génération de cas de tests.

• **Les modèles de fautes de logiciel**

Les fautes communes à tous les logiciels peuvent être classées en deux catégories principales:

- les fautes de traitement: fautes de séquencement dans le programme, fautes d'opérations arithmétiques et de manipulation de données, fautes dans l'appel de fonctions ...
- les fautes de données: fautes de type ou de représentation du format de données, fautes d'initialisation ou du domaine d'une donnée, référence à une variable qui n'est pas la bonne, fautes de référence de variables indéfinis, fautes de définition de variables non utilisées.

Malheureusement, l'ensemble des fautes n'est pas complètement connu et les modèles proposés ne couvrent qu'une partie de cet ensemble.

- **Les modèles de fautes du matériel**

Ils sont aussi appelés modèles de pannes et jouent le même rôle que les modèles de fautes pour le logiciel. Ils permettent de représenter à un niveau d'abstraction élevé les anomalies physiques d'un circuit [Prad 86]. Ils classent les anomalies physiques d'après la panne qu'ils induisent. Les pannes communes aux circuits appartiennent principalement aux classes suivantes: pannes de type "*collé-à*" (*Stuck-at*), pannes de type "*collé-ouvert*" (*Stuck-Open*), courts-circuit (*Short-Circuit*), pannes dynamiques, pannes de haut niveau.

2.6.7. Le test des protocoles de communication

Rafiq [Rafi 91] a présenté les tests des protocoles de communication comme étant des tests qui malgré certaines particularités, soulèvent des problèmes similaires aux tests du logiciel classique. D'une manière générale, on distingue deux problèmes particuliers: l'architecture du test et la construction des tests. Une implantation de protocole est conçue pour fonctionner dans un système en communiquant avec une implantation de protocole située dans un autre système, grâce à un service de communication. Pour tester une telle implantation, il est nécessaire de disposer d'un environnement matériel et/ou logiciel appelé architecture du test et qui doit lui permettre de fonctionner de manière adéquate afin de pouvoir subir les tests. Ensuite, il faut construire les échanges de messages à échanger dans cette architecture (les cas de test) pour vérifier le fonctionnement de l'implantation sous test par rapport à sa définition.

- **Architecture de test**

Pour pouvoir tester les fonctionnalités d'un système (tests de conformité), on a besoin de lui appliquer des séquences d'entrées et d'analyser les séquences de sorties. L'accès direct aux interfaces des modules à tester est parfois difficile. Les entrées-sorties du logiciel sont indirectement contrôlables à travers d'autres modules. En plus, l'application des entrées et l'analyse des sorties doivent être synchronisées. Ces aspects des tests peuvent être résolues en utilisant des systèmes dédiés aux tests appelés testeurs. L'architecture du testeur varie et dépend de l'accessibilité des points d'entrées-sorties du logiciel. L'*ISO* a proposé quelques architectures standards pour le test de protocoles de

communication qu'elle a appelé architectures de tests [Dsso 86a], [ISO9646] et [Rayn 87]. Une architecture de test est composée d'un environnement de test constitué de deux modules principaux appelés *testeur supérieur* (*TS*) et *testeur inférieur* (*TI*) et d'un mécanisme de coordination entre le *TS* et le *TI*. En plus, il y a le système à tester qu'on appelle *Implémentation Sous Test* (*IST*). Le *TS* et le *TI* ont pour objectif de soumettre le logiciel aux cas de test et d'analyser ses réponses. L'accès direct ou indirect à l'*IST* se fait via des points spécifiques appartenant à l'interface du système appelés points de contrôle et d'observation. Ces derniers sont utilisés par des modules spéciaux appelés observateurs pour observer et analyser le comportement des entités en interaction. Dans ce qui suit, nous présentons les principales architectures de test.

Architecture locale

Dans cette architecture, les points d'accès aux entrées-sorties de l'*IST* sont directement accessibles. L'*IST* est alors directement stimulée par le *TS* et le *TI*. Le *TS*, le *TI* et l'*IST* sont localisés sur le même site et sont synchronisés grâce à une procédure de coordination qui permet l'échange d'information entre le *TS* et le *TI*.

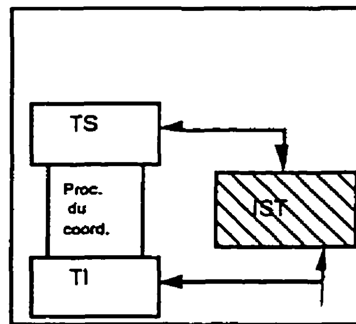


Figure 2.2. Architecture de test locale

Architecture distribuée

Les tests de conformité ne sont pas effectués par ceux qui ont développé le logiciel. Ils peuvent être réalisés par des institutions nationales ou internationales qui ne sont pas nécessairement localisées sur le même site que l'*IST*. Cette architecture est utilisée pour des tests "*in house*". Le *TS* se trouve sur le même site que l'*IST*, par contre le *TI* se trouve sur un site différent et communique avec l'*IST* via le réseau. Une procédure de coordination synchronise le *TS* et le *TI*.

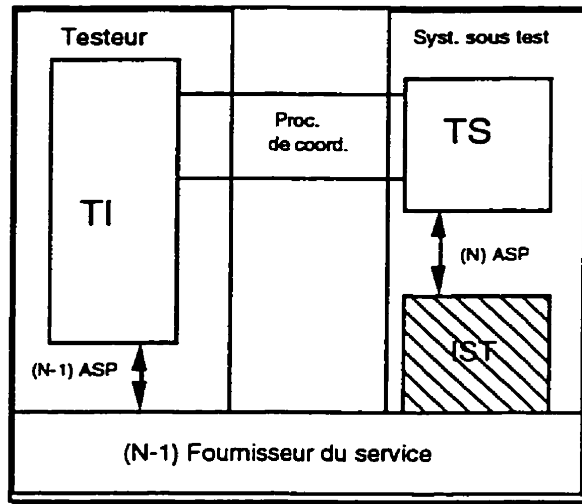


Figure 2.3. Architecture de test distribuée.

Architecture de test coordonné

Cette architecture est presque similaire à l'architecture distribuée, sauf que les procédures de coordination sont implantés avec l'*IST* comme un protocole de gestion de test (*TM-PDU*). Le *TS* n'utilise donc pas les points d'accès de l'*IST* pour communiquer avec le *TI*.

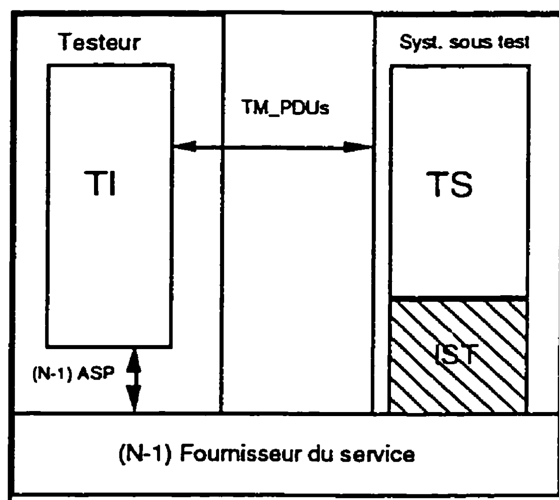


Figure 2.4. Architecture de test coordonnée.

Architecture distante

Cette architecture offre le moins de possibilité de contrôle des tests et de détection d'erreurs. Cette architecture s'applique lorsqu'on ne peut pas accéder aux interfaces de l'*IST*. L'*IST* est indirectement observable par le bas et non observable par le haut. Cette architecture ne peut pas utiliser un *TS* ce qui réduit son pouvoir de contrôle et d'observation.

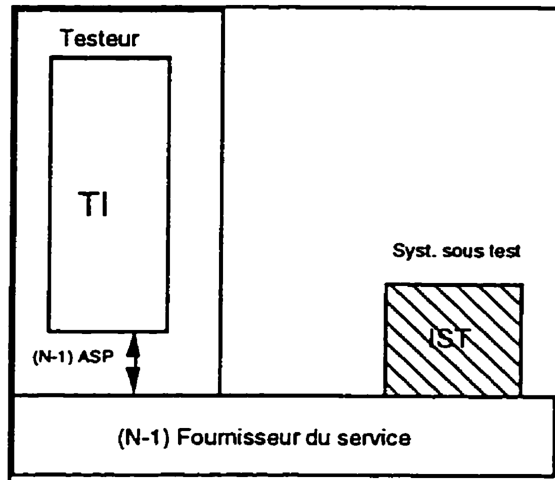


Figure 2.5. Architecture de test distante.

Pouvoir de détection de fautes imposé par l'architecture de test

Chaque architecture de test offre un certain nombre de points d'observation (*PO*) et de contrôle (*PCO*). Dssouli et Bochmann [Dsso 86 et Boch 89] décrit le pouvoir de détection d'une architecture comme dépendant de la proximité des points d'observation de l'implantation sous test. Plus ils sont proches, plus la détection des erreurs est meilleures. En général, l'emplacement des *PO*s et des *PCO*s est prédéterminé d'avance, il dépend de l'accessibilité des interfaces de l'implantation à tester. L'architecture de test a donc un impact important sur la possibilité d'observer et de détecter les erreurs.

- **La construction des tests des protocoles de communication**

En général, tous les aspects du logiciel classique restent valables pour le test des protocoles de communication. Cependant, ce dernier possède des caractéristiques propres comme par exemple l'aspect interactif inhérent à la communication avec des contraintes

temporelles sous-jacentes (synchronisation) et le fait que le protocole nécessite une entité de protocole homologue pour pouvoir fonctionner.

Pour tester le bon fonctionnement d'une implantation de protocole de communication, on a besoin d'y appliquer un ensemble de séquences d'entrées appelé *suite de tests* et de vérifier l'exactitude des sorties associées et leur synchronisation. Chaque séquence d'entrée est appelée *cas de test ;* elle sert à vérifier une ou plusieurs fonctions particulières du protocole. Un cas de test est généralement composé de trois principales parties (sous-séquences) appelées le préambule, le corps du test et le postambule. Le préambule est une séquence d'entrée permettant de positionner le protocole dans un état permettant de tester la fonctionnalité visée par le cas de test. Le corps du test permet de tester la fonctionnalité en question. Le postambule vise à réinitialiser le système en vue d'appliquer les prochains cas de test.

Il existe plusieurs méthodes de génération des suites de test. Ces méthodes dépendent fortement du formalisme utilisé pour la production de la spécification détaillée [Gone 70, Nait 81, Chow 78, Sabn 88, etc.].

2.6.8. Facteurs influençant le coût des tests

L'observation et la détection des erreurs d'un logiciel dépendent de plusieurs facteurs. À titre d'exemple, nous pouvons citer les facteurs suivants:

- Le modèle de fautes: Il sert de base pour la sélection des cas de test. Donc, l'utilisation d'un modèle ou d'un autre influe sur le type et le nombre de fautes à détecter.
- L'architecture de test: elle constitue la plate-forme d'observation des erreurs. Plus l'*IST* offre des points d'observation accessibles directement au *TS* et *TI*, plus le test est facile.
- Le choix et l'application des cas de test: ce facteur a beaucoup d'importance pour la stimulation des fautes et la propagation de leur effet vers des points observables. Il dépend des deux précédant facteurs.
- L'indéterminisme: c'est un autre facteur qui peut influencer les tests. En effet, l'indéterminisme engendre une certaine ambiguïté lors de l'analyse des sorties des cas de test. Il existe plusieurs sortes d'indéterminisme, les principales sont l'indéterminisme dû au manque d'observabilité et l'indéterminisme introduit

volontairement dans la spécification (ex. la concurrence). L'indéterminisme peut se trouver aussi bien dans la spécification que dans l'implantation.

- Les connaissances extra-spécification: en plus de la spécification, on peut utiliser d'autres informations qui peuvent améliorer l'observation et la localisation des erreurs et donc le coût des tests.

2.6.9. Couverture des tests

Les plus importantes classes de tests sont les tests fonctionnels (boîte noire) et les tests de structures (boîte blanche). La seule façon à ce jour de prouver qu'un système est correct est de le tester exhaustivement (fonction et structure). Or, les tests fonctionnels exhaustifs sont impraticables car ils impliquent le traitement d'un très grand nombre de cas. Le même problème se pose pour les tests de structures car la vérification de tous les chemins d'un programme entraîne une explosion combinatoire. Étant donné ces problèmes, certaines recherches se sont intéressées à trouver des méthodes de tests pour la validation des aspects les plus importants du logiciel et pour la détection d'un maximum d'erreurs, au lieu de chercher à affirmer l'exactitude du système. Pour cela, il faut se fixer des critères de test à atteindre comme par exemple le taux de fautes couvertes, le taux de chemins vérifiés, etc. De cette façon, l'efficacité d'une méthode de test peut être déterminée par des mesures de couverture appropriées qui évaluent les résultats de l'application des tests par rapport aux objectifs fixés. Il existe plusieurs façons d'appréhender la couverture de test, chacune d'elles apporte un éclairage différent sur le système. Deux classes regroupent ces approches: les méthodes fondées sur la structure de la spécification et les méthodes basées sur les fautes.

Les techniques fondées sur des éléments de la structure de la spécification sont issues des méthodes de test des programmes séquentiels [Papp 85]. Plusieurs auteurs les ont adaptées au cas des protocoles et ont ainsi défini des critères de couverture de spécification propres aux systèmes réactifs, comme par exemple le pourcentage de transitions exécutées ou le nombre d'états couverts. Les travaux de [Vuon 91] expriment la couverture par la qualité de l'exploration de la spécification par le test. La méthode consiste à munir l'ensemble des traces d'une distance et de prouver que l'espace métrique ainsi obtenu est compact. Ceci veut dire, qu'à partir d'un ensemble de traces on ne prend qu'un représentant. Ceci permet d'extraire de tout recouvrement de l'ensemble des traces un recouvrement fini. Si l'on considère un recouvrement par des boules de rayon R et que

l'on suppose que pour tester les éléments d'une boule il suffit de tester un seul élément de cette dernière alors la couverture peut être exprimée par (1-*R*). Si *R*=0, ceci signifie que toute les traces ont été testées et donc la couverture est totale.

La couverture de fautes caractérise l'aptitude d'un test à détecter des fautes dans l'implantation [Boch 91d], [Brin 93] et [Petr 94b]. Cette approche est issue du test du matériel ou l'on connaît mieux le processus de développement et de fabrication que pour le logiciel. En particulier, il est possible de dresser la liste des types de fautes qui ont pu être commises; c'est ce qu'on appelle le modèle de fautes. Ces techniques ont été adaptées aux systèmes réactifs tel que les protocoles: pour chaque faute recensée dans le modèle, on regarde s'il existe un test qui la détecte. On définit alors la couverture de fautes d'un ensemble de tests comme le nombre de fautes détectées sur le nombre total de fautes. Concrètement, il existe plusieurs façons pour calculer effectivement la couverture de fautes. La plus connue est certainement le calcul par mutation [Howd 82], où on construit réellement une implantation avec chaque faute du modèle et sur laquelle on fait passer les tests. Cette technique bien que relativement fiable est très lourde à mettre en place puisqu'il faut construire chaque mutant et exécuter les tests [Dubu 92]. Les travaux de [Boch 94h] et [Yao 94a] proposent une méthode par dénombrement pour s'affranchir des inconvénients précédents. Des travaux existants sur l'identification des systèmes séquentiels ont conduit [ElMa 93] et [Zhu 94] à proposer une notion de couverture basée sur cette technique. Le principe de la méthode repose sur le fait qu'un bon test doit être discriminant. En cherchant à identifier toutes les machines qui peuvent passer ces tests, on obtient justement la liste des implantations qui sont indiscernables par le test. On évalue alors la couverture au nombre d'implantation erronées qui ont été identifiées à partir des tests, puisqu'elles contiennent précisément les fautes qui ne sont pas couvertes par le test.

Pour les tests de flux de données on ne peut pas vérifier exhaustivement l'exactitude des changements de valeur des variables. Ceci car la cardinalité du domaine des variables est parfois trop importante (parfois infini). Certains auteurs [Ural 87b], [Sari 93], [Weyu 93], [Huan 95] et [Rama 95], ont proposé de tester quelques propriétés spécifiques associées au flux de données du logiciel. Ces propriétés sont basées sur un ensemble de règles appelées *critères de test*. Chaque critère de test est caractérisé par une couverture particulière des aspects de flux de donnée.

Chapitre 3

La testabilité: État de l'art

3.1. Introduction

Le but principal de la testabilité des logiciels est d'améliorer la qualité du logiciel tout en réduisant les coûts des tests. L'étude de la testabilité aide à faciliter la stimulation de fautes dont la manifestation est difficile à observer et permet la détection d'erreurs qui sont difficiles à localiser. La conception de logiciels testables (*DFT: Design For Testability*) est l'un des champs d'action de la testabilité. Lors de la phase de conception, les activités de *DFT* visent à identifier les parties de la conception qui sont difficiles à tester. Ceci permettra au concepteur d'y remédier moyennant l'utilisation de certaines techniques appropriées.

La testabilité est un critère de qualité de logiciel qui peut être influencé directement ou indirectement par divers facteurs associés aux différentes activités du cycle de développement de logiciel. Dans ce qui suit, nous allons faire une revue de la littérature associée à la testabilité et à la conception de logiciels testables. Au cours de la section 3.2, nous dresserons une liste des différentes définitions exprimant divers points de vue du sujet. Nous analyserons ces définitions et dégagerons une définition regroupant les divers points de vue. Dans la section 3.3, nous présenterons une liste de métriques de la testabilité. Nous classerons ces métriques en quatre types: les métriques basées sur le domaine des entrées/sorties, les métriques basées sur la complexité du flux de contrôle, les métriques basées sur le flux de données et en fin les métriques probabilistes. Nous consacrerons la section 3.4 à l'analyse de l'effet des différents facteurs qui influencent la testabilité. Nous présenterons les principaux facteurs et nous les classerons par auteur et par type. Nous terminerons ce chapitre par un classement des méthodes qui peuvent être utilisées pour améliorer la testabilité.

3.2. Définitions de la testabilité

Plusieurs définitions de la testabilité ont été déjà proposées. Elles s'accordent à dire que la testabilité s'occupe de l'étude des facteurs qui influencent la complexité des tests. Dans ce qui suit, nous donnons un ensemble de définitions de la testabilité que nous jugeons intéressantes pour la suite du travail.

3.2.1. Définitions

Définition 1: La testabilité est l'aptitude de générer, évaluer et appliquer les tests de façon à satisfaire les objectifs prédéfinis des tests (par exemple la couverture de fautes, l'isolation de fautes ...) qui sont sujets à des contraintes de coût (temps et argent) [Benn 94].

Définition 2: La testabilité est la capacité du logiciel de révéler ces fautes lors de l'étape de test [Voas 93].

Définition 3: La testabilité est la facilité avec la quelle on peut démontrer l'exactitude d'un programme [Clur 86].

Définition 4: Un logiciel est testable s'il inclut des facilités pour l'application des méthodes de test, pour la détection et l'isolation des fautes existantes et pour la correction rapide de ces fautes [Dsso 91a].

Définition 5: Une composante logicielle est testable si elle inclut les propriétés suivantes [Free 91]: les suites de test sont courtes et faciles à générer, les suites de test ne sont pas redondantes, les sorties sont faciles à interpréter, les fautes sont faciles à localiser, les entrées ne sont pas inconsistantes, les sorties ne sont pas inconsistantes.

Définition 6: La testabilité d'un programme P est la probabilité d'échec des tests de P si P contient des fautes [Voas 93].

Définition 7: Un programme est testable, si la présence de fautes entraîne automatiquement sa défaillance [Mill 93].

Définition 8: Conception de protocole testable: La conception de protocoles dont l'implémentation est facile à tester [Sale 92].

Définition 9: Une spécification est qualifiée de testable si elle décrit d'une façon non ambiguë les propriétés suivantes: les options permises de l'implémentation, les besoins de conformité, les interfaces normalisées, l'exécution des tests permet de vérifier la conformité du produit avec un certain degré de confidence [ETSI 94].

Définition 10: Une spécification est dite orientée testabilité (*testability-directed specification*) si sa description utilise une méthode formelle qui permet de faciliter les tests subséquents de l'implémentation [Yu 91].

Définition 11: La testabilité d'une spécification est la facilité de trouver une technique applicable et économique permettant de déterminer si le logiciel développé est conforme à sa spécification [Thay 90].

Définition 12: La testabilité est une caractéristique de la conception qui influence divers coûts associés aux tests. Les techniques de conception testable (*Design For Testability*) sont les efforts de conception employés pour assurer la testabilité d'un système [Abra 90].

Définition 13: La testabilité est:

(1) la facilité d'établir des critères de test d'un système ou d'une composante de système et la capacité des tests à déterminer si les critères sont satisfaits,

(2) la facilité avec la quelle on peut exprimer les objectifs de test et la capacité des tests à déterminer si de tels objectifs sont satisfaits [IEEE 90].

3.2.2. Définitions relatives aux protocoles de communication

Dans le chapitre 2 nous avons précisé que les tests des protocoles de communication possèdent certaines particularités, mais en général, ils soulèvent des problèmes similaires aux tests du logiciel classique. Les définitions de la testabilité des protocoles de communication sont donc assez rares. Généralement, les différentes définitions que nous venons de présenter ne spécifient pas le genre de logiciel auquel elle

s'adresse. Malgré cela elles restent directement applicables aux protocoles de communication.

3.2.3. Discussion

Toutes les définitions précédentes sont liées à la phase du cycle de développement à laquelle l'auteur s'est intéressé. Les phases concernées couvrent pratiquement tout le cycle de vie de logiciel. Ces définitions peuvent donc être classées d'après les phases et les activités du cycle de développement auxquelles elles sont associées (voir table 3.1). L'analyse de ces définitions nous montre que la prise en compte de la testabilité peut se faire dans toutes les étapes du cycle de développement de logiciel. Pour cela, nous proposons (voir chapitre 4) d'étendre le cycle de développement de logiciel en rajoutant l'étude de la testabilité comme une activité qui devra être effectuée au niveau de chaque étape du cycle.

	Conception	Implantation	Tests	Maintenance
Définition 1	x	x	x	x
Définition 2		x	x	x
Définition 3		x	x	
Définition 4		x	x	x
Définition 5	x	x	x	x
Définition 6		x	x	
Définition 7		x	x	
Définition 8	x	x	x	
Définition 9	x	x	x	
Définition 10	x	x	x	
Définition 11	x	x	x	
Définition 12	x	x	x	
Définition 13	x	x	x	

Table 3.1. Classification des définitions de la testabilité.

Les définitions 1 et 5 ([Free 91] et [Benn 94]) couvrent le plus grand nombre de phases du cycle de développement. Nous nous baserons sur ces deux définitions et particulièrement la définition 5 pour proposer une définition encore plus générale.

Définition proposée:

Une composante logiciel est testable si elle inclut les propriétés suivantes:

- la spécification a certaines propriétés qui permettent de générer facilement une suite de test courte,
- les cas de test ne sont pas redondants,
- l'application des cas de test produit des sorties faciles à interpréter,
- l'application des cas de test entraînent automatiquement une défaillance, si l'implémentation contient des fautes,
- lors d'une défaillance, la localisation des fautes est facile,
- l'application des cas de test permet de satisfaire des critères de test prédéfinis.

3.3. Les mesures de testabilité

Nous avons déjà vu que plusieurs propriétés peuvent caractériser la qualité d'un logiciel (voir chapitre 2). Ces propriétés peuvent être quantifiés en leur associant des métriques. Ces dernières nous permettent de classifier le logiciel selon ces propriétés de qualité.

Au cours des dernières années, plusieurs métriques de testabilité ont été proposées. Ces métriques peuvent être classées en quatre types: les métriques fondées sur le domaine des entrées/sorties, les métriques fondées sur la complexité du flux de contrôle, les métriques fondées sur le flux de données et les métriques probabilistes. Dans cette section, nous allons présenter pour chaque type, les principales métriques qui y sont proposées.

3.3.1. Métriques basées sur le domaine des entrées-sorties

Il existe principalement deux métriques de ce type, celle de Freedman [Free 91] et celle de Voas [Voas 93]. Toutes les deux se fondent sur l'analyse des domaines d'entrées-sorties pour quantifier la testabilité d'une spécification ou d'un programme. Elles ne nécessitent pratiquement pas d'information sur les détails internes des entités à tester.

• Métrique de Freedman

Freedman [Free 91] définit la testabilité sur le domaine en décrivant la sémantique de l'exécution d'une unité de programme (expression ou commande). Il exprime la testabilité sur le domaine d'une expression ou d'une commande en fonction de son observabilité et de sa contrôlabilité. Une expression ou une commande est dite observable si elle associe des sorties différentes pour chaque entrée différente. Par contre, elle est dite contrôlable si pour chaque sortie il existe au moins une entrée qui force le programme à donner cette valeur de sortie. L'auteur définit la testabilité sur le domaine comme étant la facilité de modifier un programme pour qu'il devienne observable et contrôlable. Il montre comment modifier un programme pour qu'il devienne observable et contrôlable. Ces modifications sont appelées extensions observables et contrôlables. Elles consistent à étendre le domaine des entrées et (ou) des sorties du programme. Ces extensions peuvent être mesurées en terme de la cardinalité du domaine des entrées ou des sorties ajoutées. La testabilité sur le domaine peut alors être exprimée en fonction de ces extensions. Cette métrique est relative au code.

Exemple

Pour illustrer cette métrique, nous présentons dans ce qui suit la description d'un module M_o:

```
Module Mo (In : in INTEGER, Ou : out INTEGER)
begin
      VAR X: INTEGER
      X := In * G
      Ou := X mod 7
end
```

M_o n'est pas observable car la valeur de *Ou* dépend de la variable globale *G* qui ne fait pas partie des entrées de M_o. En effet M_o(4, *Ou*) rend le résultat *Ou*=0 si *G*=0 et *Ou*=1 si *G*=2. Une extension observable du module M_O consiste à lui passer la valeur de G comme paramètre :

```
Module Mo (In : in INTEGER, G : in INTEGER, Ou : out INTEGER)
begin
      VAR X: INTEGER
      X := In * G
      Ou := X mod 7
end
```

D'un autre côté M_O n'est pas non plus contrôlable puisque le domaine des sorties spécifiées est: *Ou : out INTEGER*, tandisqu'en réalité on ne peut avoir que des valeurs entières de 0 à 7. Une extension contrôlable du module M_0 consiste à réduire le domaine des sorties de *Ou* de la façon suivante :

```
Module Modulo (In : in INTEGER, G : in INTEGER, Ou : out {0, ...., 7})
begin
        VAR X: INTEGER
        X := In * G
        Ou := X mod 7
end
```

- **Métrique de Voas**

Voas [Voas 91] introduit une nouvelle mesure relative au code appelée *DRR* (*Domaine Range Ratio*). L'auteur définit le *DRR* comme étant le rapport entre la cardinalité du domaine des entrées et celui des sorties. Voas présente le lien entre cette mesure et la testabilité du logiciel. Il l'illustre par des exemples de fonctions mathématiques simples (*div, mod, trunc*). Ces dernières, délivrent moins de sorties que d'entrées. Ce phénomène s'appelle l'état d'écroulement interne (*Internal state collapse*) et est directement responsable d'une faible testabilité. En effet, une fonction a une testabilité faible si elle reçoit des paramètres erronés en entrée et délivre des résultats exacts. Par contre une fonction est fortement testable si à chaque entrée correspond une sortie. De cette façon, on peut dire que plus le *DRR* est élevé plus il y a possibilité de masquage d'erreurs. Par la suite, l'auteur définit trois types de *DRR*:

∇ *FDFR* (*Fixed Domain Fixed Range*): lorsque la cardinalié des domaines d'entrées et des sorties est finie,

∇ *VDVR* (*Variable Domain Variable Range*): lorsque la cardinalié des domaines d'entrées et des sorties est infinie,

∇ *VDFR* (*Variable Domain Fixed Range*) lorsque la cardinalié du domaine d'entrées est infinie et celle du domaine des sorties est finie.

L'auteur considère le *FDFR* comme le meilleur *DRR* pour la testabilité. Cette métrique est relative au code.

Exemple: À titre d'exemple le module M_O suivant a un *DRR* (∞, 7), il fait donc partie de la classe *VDFR* :

```
Module Mo (In : in INTEGER, G : in INTEGER,Ou : out {0, ..., 7})
begin
        VAR X: INTEGER
        X := In * G
        Ou := X mod 7
end
```

3.3.2. Métriques basées sur la complexité du flux de contrôle

- **Métriques basées sur la longueur des suites de test**

Ce type de métriques se base sur la longueur des suites de tests nécessaires au test complet d'une implantation. Une suite de test est dite complète si elle peut détecter toutes implantations non conformes à une spécification de référence. Dans cette direction, Petrenko [Petr 93a] a proposé une métrique qui se base sur le modèle des automates à états finis (*FSMs* ou *Finite States Machine*) et sur les méthodes de test qui y sont associées. La longueur d'une suite de test peut être bornée en se basant sur la théorie des automates [Vasi 73]. Pour un *FSM* complètement spécifié et déterministe, la longueur d'une suite de test complète ($L(TS)$) est bornée par: $L(TS) \leq mn^2\ p^{m-n+1}$ où m est le nombre maximal d'états de l'implémentation, n le nombre d'états de la spécification et p le nombre d'entrées de la spécification. La testabilité peut être alors évaluée comme étant inversement proportionnelle à la longueur de la suite de tests complète. Petrenko et al. [Petr 93a] proposent une hypothèse stipulant que le nombre de sorties o de la spécification est généralement inférieure à la borne supérieure de la longueur de la suite de tests. Leur argument a été vérifié sur le protocole *INRES*. Ceci les conduit à la mesure de testabilité suivante: $T = o/\ mn^2\ p^{m-n+1}$.

Exemple: À titre d'exemple, si on considère la spécification de référence l'automate de la figure 3.1, si le nombre d'états des implémentations issues de cette spécification ne dépasse pas celui de la spécification, alors la testabilité est égale à: T=3/250.

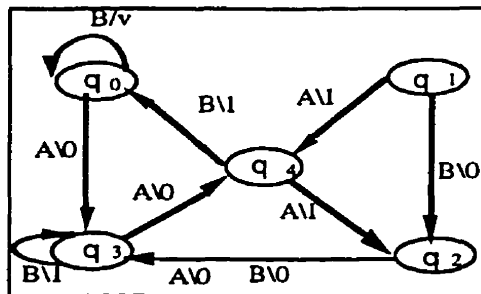


Figure 3.1: Exemple de *FSM* complètement spécifié déterministe.

- **Métriques basées sur le nombre de chemins de la spécification ou du code**

Une spécification est constituée d'un ensemble de chemins dont l'exécution de chacun décrit une des fonctionnalités du système. Une suite de test est un ensemble de séquences d'entrées/sorties extraites de la spécification. Ces séquences représentent des chemins distincts de la spécification. En se basant sur cette approche, la testabilité peut être définie comme étant inversement proportionnelle au nombre de chemins à tester pour affirmer la conformité de l'implémentation par rapport à la spécification de référence. Dans ce même sens, trois principaux travaux peuvent être cités:

Mesure basées sur la complexité cyclomatique de McCabe

Une implémentation ou une spécification peut être représentée sous forme d'un graphe appelé graphe de contrôle. Chaque noeud de ce graphe représente un ensemble d'instructions de bases qui ne contiennent pas d'instructions de branchement. Les arcs de ce graphe représentent le flux de contrôle du système à représenter. Dans la Figure 3.2, nous avons représenté un exemple de graphe de contrôle associé au programme de la partie gauche de la figure. Les étiquettes associées aux noeuds sont constituées des numéros des lignes du programme considéré.

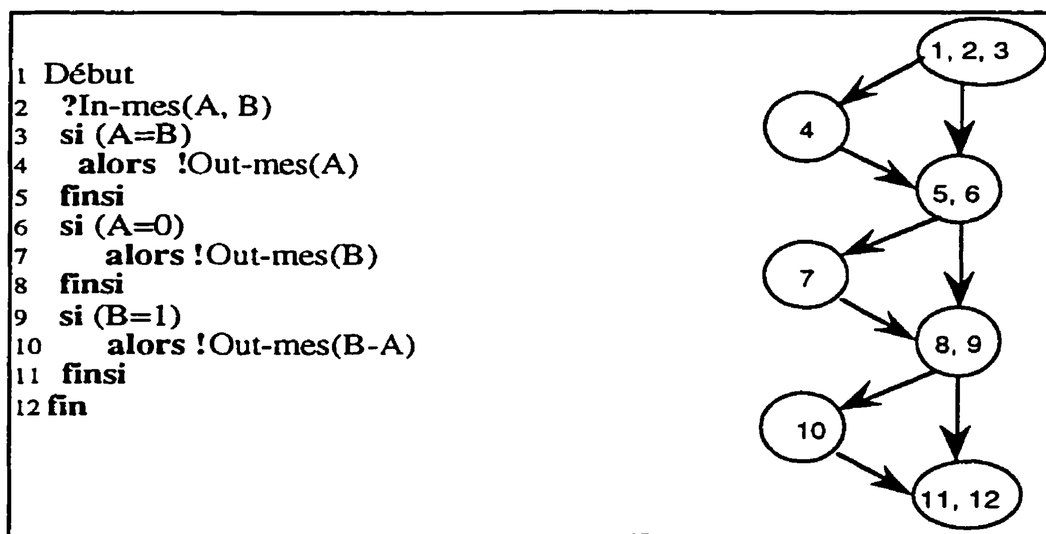


Figure 3.2. Un exemple de programme avec le graphe de contrôle associé.

À partir du graphe de contrôle, on peut tirer le nombre maximal de chemins à tester et l'utiliser comme métrique de testabilité. Comme tout programme contenant des boucles a un nombre de chemins d'exécutions potentiellement infini ; McCabe [McCa 89] propose de calculer un nombre restreint de chemins constitué des circuits linéairements indépendants du graphe de contrôle. Pour cela, il utilise le nombre cyclomatique $V(G) = e-n+2p$ du graphe de contrôle, où e est le nombre d'arcs, n le nombre de noeuds et p le nombre de composantes connexes du graphe, c'est-à-dire, le nombre de sous-graphes qui ne sont pas accessibles à partir d'autres noeuds du graphe. Plus ce nombre est petit, moins il y a de cas de tests et donc meilleure est la testabilité. L'application de cette formule sur l'exemple de la figure 3.2 donne un nombre de chemins $V(G)=4$.

Mesure du nombre de chemins avec visite unique des boucles

La mesure précédante est très restrictive ; elle élimine un nombre important de chemins, pourtant possible à tester. Pour palier à cette lacune, Nejmeh [Nejm 88] propose une mesure appelée *NP* (*NPATH*) qui comptabilise les chemins où chaque itération est exécutée au plus une fois. Dans [Nejm 88], l'auteur propose un algorithme automatisable pour le calcul de *NP* défini pour les programmes en langage C (voir table 3.2). Si on applique cette mesure sur l'exemple de la figure 3.2, on obtient *NP*=8. On peut constater que *NP* étend la première méthode en offrant une plus grande couverture des chemins.

Structure	NP
if: if(<exp>)<inst_if>	NP(inst_if)+NP(exp)+1
if-else: if(<exp>)<inst_if>else<inst_else>	NP(inst_if)+NP(inst_else)+NP(exp)+1
while: while(<exp>)<inst_while>	NP(inst_while)+NP(exp)+1
do: do(<exp>)<inst_do>	NP(inst_do)+NP(exp)+1
for: for(<exp1>;<exp2>;<exp3>)<inst_for>	NP(inst_for)+NP(exp1)+NP(exp2)+NP(exp3)+1
switch: switch(<exp>){<case1><case2>...<casen><default>}	NP(exp)+NP(default)+ΣNP(casei)
goto label	1
break	1
exp	nombre de && et || dans exp
continue	1
return	1
instructions séquentiels	1
appel de fonction	1
...	

Table 3.2. Sommaire des structures en *C* et leur complexité respective.

Mesures basées sur les constructions de base du programme

La métrique de Müllerberg [Bach 90] est un autre exemple des mesures qui comptabilisent le nombre de chemins de la spécification. Cette mesure est basée sur la

décomposition du graphe de flux d'un programme (voir Exemple figure 3.3) en sous-graphes appelés les graphes de flux primaires (voir figure 3.4). L'auteur propose deux opérateurs qui permettent de reconstruire le graphe global. Ces opérateurs sont le remplacement et le séquencement. Dans du code Pascal par exemple, le séquencement consiste à joindre deux segments de code en les séparant par un point virgule ';'. En contre partie, le remplacement consiste à changer une instruction simple par un block d'instructions délimités par ***Begin*** et ***End***. L'auteur adopte la notation $F_1;F_2$ pour décrire le séquencement des deux graphe de flux F_1 et F_2 et la notation $F(F_1, ..., F_n)$ pour décrire le remplacement des noeuds de F par les graphes de flux $F_1, ..., F_n$. À partir de ces opérateurs et des graphes de flux primaires, un graphe de flux peut être réécrit sous forme d'une équation unique.

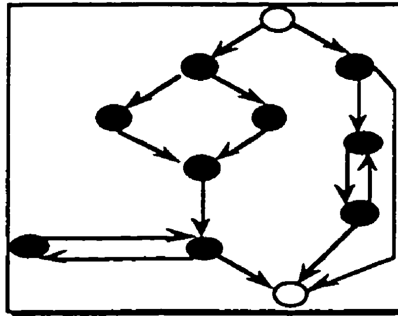


Figure 3.3. Exemple de graphe de flux: $D_1((D_1;D_2), D_0(D_3)))$.

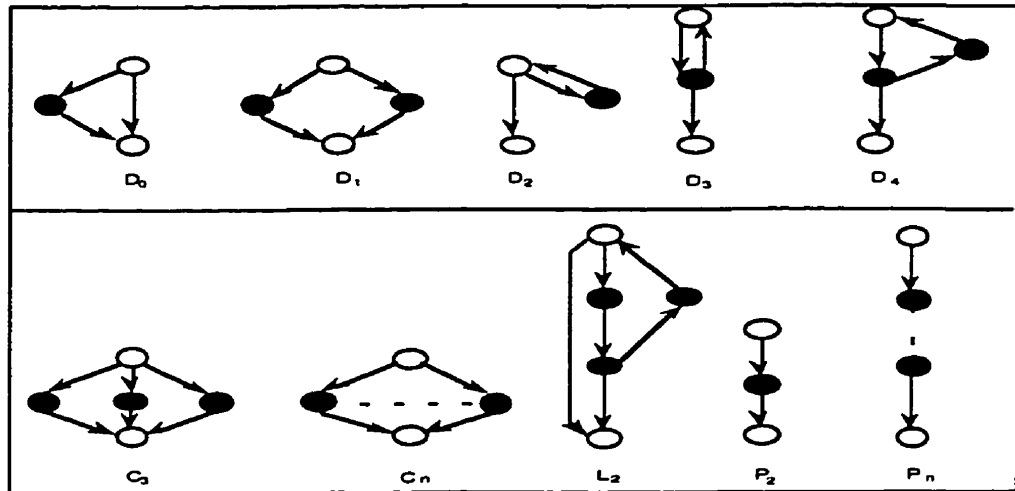


Figure 3.4. Graphes de flux primaires: D_0: IF-THEN, D_1: IF-THEN-ELSE, D_2:WHILE-DO, D_3: REPEAT-UNTIL, D_4: EXIT FROM MIDDLE, C_3: CASE-OF-3, C_n: CASE-OF-n, L_2: EXIT-LOOP, P_2 et P_n SEQUENCE.

Pour chaque critère de test, l'auteur propose une mesure μ qui associe aux graphes de flux primaires un nombre fixe de chemins à tester (voir table 3.3). Par la suite, l'auteur généralise la mesure pour les graphes composés (voir tables 4 et 5). Pour illustrer sa méthode, l'auteur l'applique sur le graphe de flux de la figure 3.3 ($G=D_1((D_1;D_2), D_0(D_3)))$). L'application donne comme résultats: $\mu(G)= 7$ pour les chemins où les boucles sont visités au plus une fois et $\mu(G)= 4$ pour le test de toutes les branches du programme.

Critère de test	$\mu(P_2)$	$\mu(D_0)$	$\mu(D_1)$	$\mu(C_n)$	$\mu(D_2)$	$\mu(D_3)$	$\mu(D_4)$	$\mu(L_2)$
Tous les chemins	1	2	2	n	-	-	-	-
NPATH	1	2	2	n	2	2	2	4
Les chemins simples	1	2	2	n	2	1	1	3
Les chemins indépendants	1	2	2	n	2	2	2	3
Toutes les branches	1	2	2	n	1	1	1	2
Toutes les instructions	1	2	2	n	1	1	1	1

Table 3.3. Valeur de la métrique pour les graphes de flux primaires.

Critère de test	$\mu(F_1; ...; F_n)$	$\mu(D_1(F_1, F_2))$	$\mu(C_n(F_1, ..., F_n))$	$\mu(D_0(F))$
Tous les chemins	$\prod_{i=1}^{n} \mu(F_i)$	$\mu(F_1)+\mu(F_2)$	$\sum_{i=1}^{n} \mu(F_i)$	$\mu(F)+1$
NPATH	"	"	"	"
Les chemins simples	"	"	"	"
Les chemins indépendants	$\sum_{i=1}^{n} \mu(F_i)-n+1$	"	"	"
Toutes les branches	$\max(\mu(F_i))$	"	"	"
Toutes les instructions	"	"	"	$\mu(F)$

Table 3.4. Valeur de la métrique pour des graphes de flux composés.

Critère de test	$\mu(D_2\&F)$	$\mu(D_2\&F)$	$\mu(D_1(F_1, F_2))$	$\mu(L_2(F_1, F_2))$
Tous les chemins	-	-	-	-
NPATH	$\mu(F)+1$	$\mu(F)+\mu(F)^2$	$\mu(F_1)+\mu(F_1)^2\mu(F_2)^2$	$\mu(F_1)+\mu(F_1)\mu(F_2)+1+\mu(F_1)^2\mu(F_2)$
Les chemins simples	"	$\mu(F)$	$\mu(F_1)$	$1+\mu(F_1)+\mu(F_1)\mu(F_2)$
Les chemins indépendants	"	$\mu(F)+1$	$\mu(F_1)+\mu(F_2)$	$1+\mu(F_1)+\mu(F_2)$
Toutes les branches	1	1	1	2
Toutes les instructions	1	1	1	1

Table 3.5. Valeur de la métrique pour des graphes de flux composés.

3.3.3. Métriques basées sur la complexité du flux de données

Il existe plusieurs métriques de complexité du flux de données dont par exemple les métriques de cohésion et de couplage. À titre d'exemple nous pouvons citer le travail de Weyuker [Weyu 88]. Dans ce dernier, un programme est alors décomposé en un ensemble de blocs séquentiels qui ne contiennent pas d'instruction de rupture de séquences. À partir de l'ensemble des variables d'un bloc B_i, l'auteur extrait le sous-ensemble des variables référencées dans B_i mais qui y sont définies dans d'autres blocs B_j avec $j \neq i$. La complexité d'un bloc est calculée comme étant $DF_i = \sum_{j=1}^{j=|v|} DEF(v_j)$, où $DEF(v_j)$ représente le nombre de définitions de la variable v_j qui atteignent le bloc.

3.3.4. Métriques probabilistes

Ce genre de métrique se base sur une définition probabiliste de la testabilité (voir Définition 6, section 2.2). Dans cette section, nous ferons un résumé des principales métriques de ce type récemment apparues.

- PIE (*Propagation-Infection-Execution*)

Voas [Voas 92] présente une technique dynamique appelée *PIE* (*Propagation-Infection-Execution*) pour estimer statistiquement les caractéristiques d'un programme. Cette métrique se base sur une série d'entrées d'un programme appartenant à une distribution d'entrée *D* pour calculer les probabilités suivantes: la probabilité $\hat{\varepsilon}_{lPD}$ qu'un emplacement *l* du programme soit exécuté (*Exécution*), la probabilité $\hat{\lambda}_{mly}{}^{lPD}$ qu'une faute située à cet emplacement affecte les données qui suivent (*Infection*) et la probabilité $\hat{\psi}_{ailPD}$ que cette faute soit propagée vers une sortie du programme (*Propagation*). L'auteur propose d'utiliser ces mesures ainsi qu'une quatrième mesure θ_l appelée l'analyse de sensibilité (*sensitivity Analysis*) pour prédire la probabilité minimum qu'une faute dans un emplacement *l* sera révélée par les tests. Cette mesure est composée des trois mesures précédentes (*Propagation-Infection-Execution*) et est définie comme suit:

$$\theta_l = (\hat{\varepsilon}_{lPD})_{min} . \sigma(\min_{mly} \left[\left(\hat{\lambda}_{mly}{}^{lPD} \right)_{min} \right], \min_{a} \left[\left(\hat{\psi}_{ailPD} \right)_{min} \right]) \text{ où}$$

$\sigma(a, b) = \begin{cases} a-(1-b) & si\ a-(1-b)>0 \\ 0 & sinon \end{cases}$ et $(.)_{min}$ est la borne inférieure de l'intervalle de confidence d'une estimation.

L'analyse de sensibilité θ_l à son tour peut être utilisée pour déterminer le nombre de cas de tests T nécessaires pour être convaincu avec une confidence acceptable qu'une localité l est correcte: $T=\frac{ln(1-c)}{ln(1-\theta_l)}$ où c est la confidence de la probabilité d'échec de l'emplacement l est inférieure à θ_l.

- **Probabilité d'échec si le programme contient une faute**

Ce genre de mesures se base sur une des définitions classiques de la testabilité (Définition 6, section 2.2). On peut trouver une telle mesure dans les travaux de Hamlet [Haml 93b]. Elle consiste à calculer la probabilité d'échec des tests si le programme contient des erreurs. Elle est définie comme suit:

$$Testab_{HV}=P(échec|\ prob.\ distrib.\ entrées, faute)\ .$$

- **Estimation de la dépendance et testabilité**

Bertolino et al. [Bert 96] décrivent le processus de test d'un programme comme étant l'observation du comportement d'un programme soumis à certaines séquences d'entrées. Ceci nécessite un environnement contrôlé dans lequel on analyse le comportement du programme. Les résultats du test dépendent de plusieurs paramètres dont par exemple la machine sur laquelle est exécuté le programme, etc. Les tests nécessitent un agent appelé oracle pour l'analyse des résultats et la prise de décision concernant l'exactitude du programme (voir figure 3.5). Cet agent peut ne pas juger correctement les résultats des tests, dans ce cas, on peut se trouver face aux différentes situations décrites dans la table 3.6. Pour cela les auteurs définissent la couverture d'un oracle comme:

$$Coverage=P(rejet|\ prob.\ distribution\ entrées, erreur\ dans\ les\ variables\ observées)\ .$$

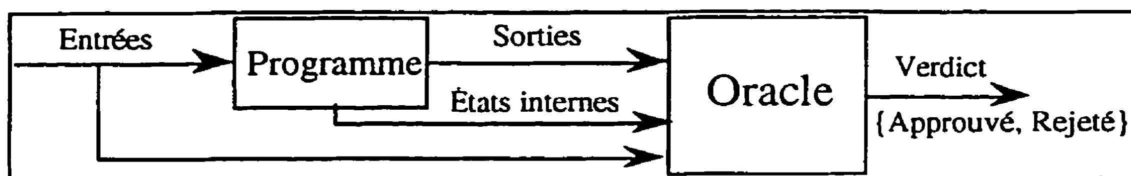


Figure 3.5. Le contexte de test.

États du Programme	Sorties du Programme	Verdict de l'oracle	Notes
Correcte	Correcte	Approuvé	
Correcte	Correcte	Rejeté	Mauvais oracle, ce comportement est facile à corriger, on assumera une probabilité nulle pour ce cas.
Correcte	Incorrecte	Approuvé	Impossible.
Correcte	Incorrecte	Rejeté	Impossible
Incorrecte	Correcte	Approuvé	Oracle avec couverture imparfaite des états.
Incorrecte	Correcte	Rejeté	Oracle avec bonne couverture des états.
Incorrecte	Incorrecte	Approuvé	Oracle avec couverture imparfaite des états ou mauvais oracle.
Incorrecte	Incorrecte	Rejeté	

Table 3.6. Résultats d'un test.

Les auteurs étendent alors la mesure de testabilité classique [Haml 93b] (voir section 2.2): $Testab_{HV}=P(\text{échec} \mid \text{prob. distrib. entrées}, \text{faute})$ par:

$$Testab_{ABS}=Testab_{HV}\frac{Coverage}{P(\text{échec} \mid \text{erreur})}.$$

3.3.5. Mesures de testabilité des protocoles de communication

Parmi la liste de mesures présentées, seule celle qui a été présentée dans [Petr 93a] a été initialement développée pour les protocoles de communication. Cette mesure prend en considération l'interaction qui existe entre plusieurs entités de protocole. Cela n'empêche pas que les autres mesures que nous avons présentées peuvent être, soit directement applicables, soit adaptables aux cas de protocoles de communication.

3.3.6. Discussion

Les métriques que nous venons de citer bien qu'elles expriment des points de vue différents, elles ne sont pas contradictoires. Les métriques basées sur le domaine des

entrées-sorties [Free 91], [Voas 93] ne s'occupent que de l'analyse de l'information au niveau des interfaces. L'influence de la structure interne des modules sur la testabilité est mesurée par les métriques basées sur la complexité du flux de contrôle, [Nejm 88], [McCa 89], [Bach 90] et [Petr 93a]. L'influence des données sur les tests est quant à elle évaluée par des métriques de complexité de flux de données [Weyu 88]. Enfin, les métriques probabilistes [Voas 92], [Haml 93b] et [Bert 96] apportent un plus aux autres méthodes. Elles sont de deux types: le premier type de métriques permet d'estimer d'une manière probabiliste les parties de la spécification ou du code difficiles à tester [Voas 92]. Le deuxième type, permet de relier d'une manière probabiliste l'existence de fautes à l'apparition de défaillances dans le code[Haml 93b] et [Bert 96], etc.

À part la métrique associée au code proposée par Voas [Voas 92], les autres mesures ne nous informent pas sur les causes d'une mauvaise testabilité. L'applicabilité de certaines métriques est restreinte à une classe de spécifications ayant des propriétés spécifiques comme le déterminisme et la complétude (voir [Petr 93a] section 3.3.2). Partant de ces remarques, nous proposerons dans les prochains chapitres, de nouvelles métriques associées aux spécifications de protocoles de communication (voir chapitre 5 et 6). Ces dernières seront plus détaillées et permettent d'analyser les causes d'une mauvaise testabilité. Grâce à ces métriques nous pouvons aussi couvrir l'étude de la testabilité d'une classe de spécification de protocoles plus large que celle couverte par la métrique de Petrenko et al. [Petr 93a]. Nous nous baserons sur ces métriques pour proposer des transformations dont l'objectif est d'améliorer la testabilité d'une spécification de protocoles de communication.

3.4. Facteurs qui influencent la testabilité

Une suite de test est composée d'un ensemble de cas de tests dont chacun visant à vérifier la conformité d'une ou de plusieurs entités de base de l'implémentation (transitions, états, instructions, branches, chemins, etc.). Certaines spécifications sont décrites dans un style et possèdent certaines propriétés (indéterminisme, distinction des états, accès aux états, etc.) qui ne favorisent pas la dérivation ou l'application des tests. Si la suite de test est difficile à extraire, ou si elle est complexe (longue, dont les résultats sont difficiles à interpréter, les séquences de sorties sont indéterministes, etc.), la spécification peut être qualifiée de non testable [Petr 93a]. Dans ce cas, il est des fois nécessaire de raffiner la spécification en vue d'améliorer sa testabilité.

Dans ce qui suit, nous allons présenter quelques travaux, et proposer une liste de facteurs qui peuvent influencer la testabilité d'une spécification ou d'une implantation.

3.4.1. Facteurs énoncés par Ould

D'après Ould [86] une spécification testable doit avoir les propriétés suivantes: elle conserve des facteurs de calcul initiaux et intermédiaires, comporte des routines de récupération pour les facteurs hautement à risques, possède des mécanismes de trace, de réinitialisation, d'enregistrement de l'avancement et de compte rendu, conception simple non compactée et non condensée, conception bien documentée, les actions sont visibles via les interfaces, elle possède des mécanismes de recouvrement qui sont déclenchés lors de l'apparition d'événement imprévu, utilise le moins possible de routines et de zones de stockage communes, les modules sont fonctionnellement indépendants.

3.4.2. Facteurs énoncés par Martin et McClure

Dans leur livre [Mart 83], Martin et McClure décrivent la testabilité comme une propriété de qualité de logiciel. La testabilité dépend de quatre autres propriétés qui sont la facilité de compréhension (*understandability*), la structuration, la fiabilité et la complexité (voir chapitre 2).

3.4.3. Facteurs énoncés par Perry

Perry [Perr 87] décrit la testabilité comme étant l'un des onze critères de qualité du logiciel. Ces critères sont l'exactitude (Correctness), la fiabilité, l'efficacité, l'intégrité, la facilité d'utilisation (usability), la facilité de maintenance (maintenability), la testabilité, la flexibilité, la portabilité, la réutilisabilité et l'interopérabilité. Ces critères ne sont pas complètement indépendants. D'après Perry, la testabilité est directement en relation avec l'exactitude, la fiabilité, l'utilisabilité et la facilité de maintenance (voir chapitre 2).

La figure 3.6 résume les relations entre ces critères de qualité.

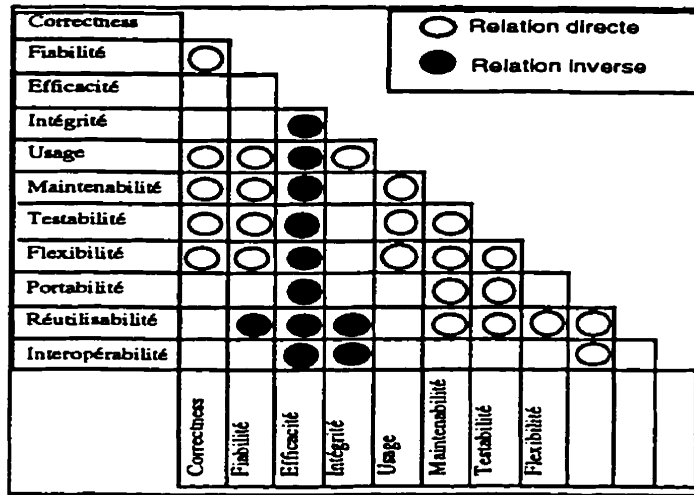


Figure 3.6. Modèle de Perry décrivant les relations entre les critères de qualité.

3.4.4. Facteurs énoncés par McCall et al.

McCall [McCa 77] a proposé un modèle hiérarchique de qualité de logiciel. Ce modèle est basé sur onze critères de qualité dont la testabilité, la facilité de maintenance, la flexibilité, la portabilité, la réutilisabilité, l'interopérabilité, l'exactitude, la fiabilité, l'efficacité, l'intégrité et la facilité d'utilisation (voir chapitre 2). Ces critères expriment aussi bien l'opinion de l'utilisateur que celle de celui qui développe. Ce modèle vise trois périodes d'utilisation du produit: la période d'utilisation normale ou d'opération, la période de révision durant laquelle les erreurs sont corrigées et le système est adapté et la période de transition durant laquelle le système est réadapté pour un nouvel environnement d'utilisation (voir figure 3.7).

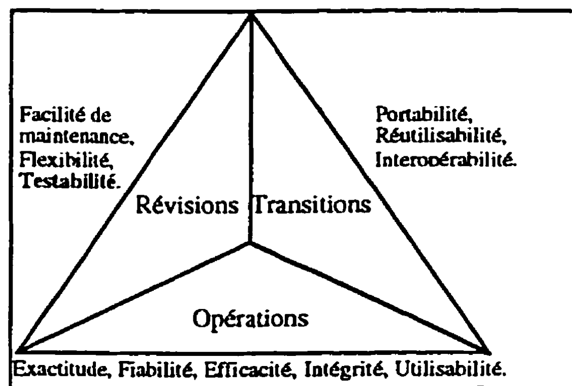


Figure 3.7. Modèle de McCall de la qualité de logiciel.

3.4.5. Facteurs énoncés par Boehm et al.

Le modèle proposé par Boehm et al. [Boeh 78] est hiérarchique. Il se base sur un ensemble de caractéristiques du logiciel bien définies et distinctes. La testabilité est l'une de ces caractéristiques, elle fait partie du deuxième niveau de ce modèle. La testabilité à son tour est raffinée en cinq propriétés qui sont la facilité de comptabilisation (*accountability*), l'accessibilité des composantes (*accessibility*), la facilité de communication et d'interprétation des entrées/sorties, la structuration (structuredness) et l'auto-documentation (*self descriptiveness*) (voir figure 3.8).

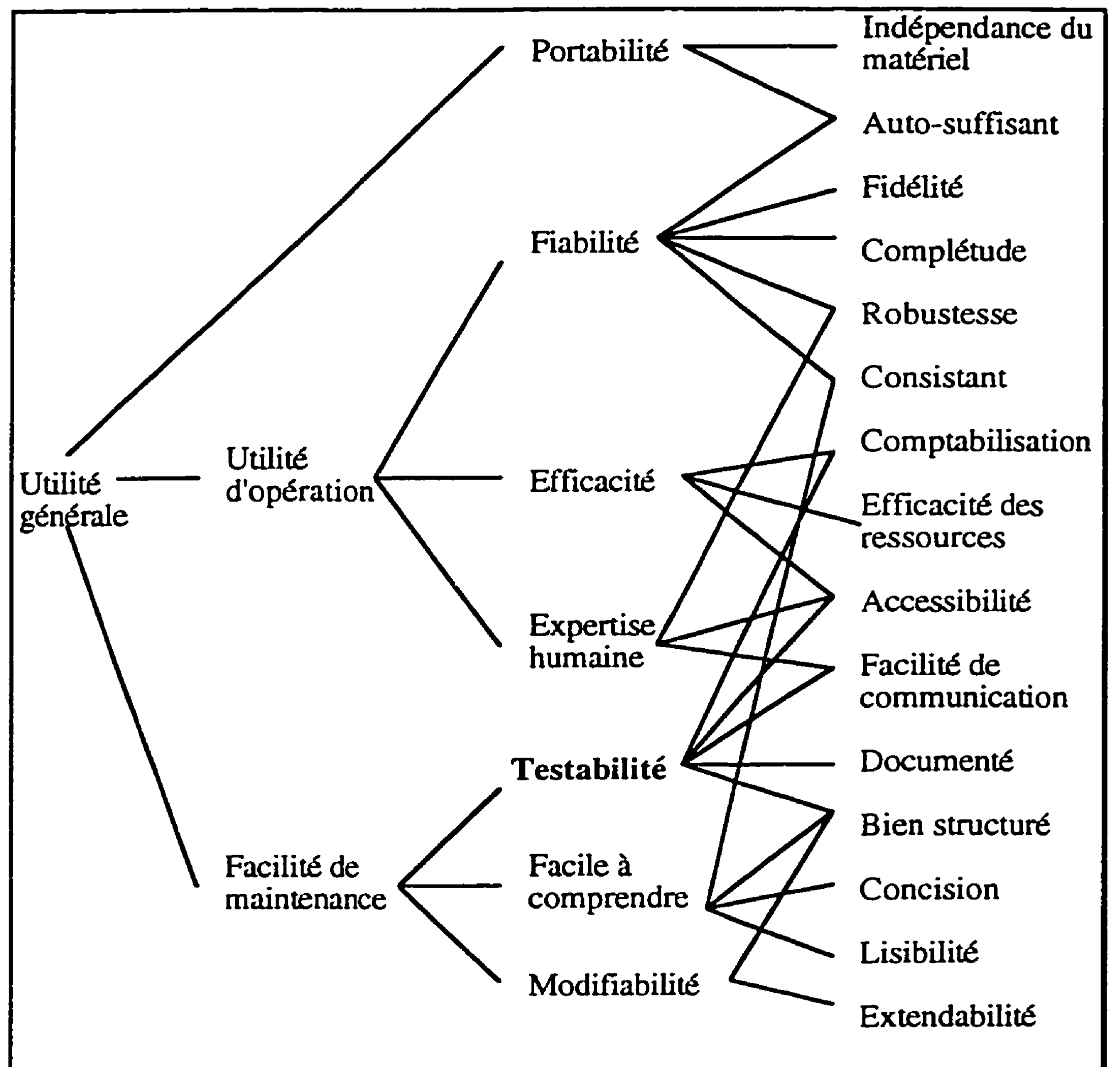


Figure 3.8. Modèle hiérarchique de Boehm.

3.4.6. Facteurs énoncés par Boehm

Boehm [Boeh 81] a classé la testabilité comme étant un des cinq critères dont l'objectif est la précision de la spécification. Les autres critères étant la complétude, la résistance aux fautes, la consistance et la faisabilité. La réalisation de ces critères implique que les aspects de fonctionnalité, de performance et d'interface du logiciel, ont été examiné minutieusement ne donnant lieu à aucune ambiguïté. Ces propriétés sont considérées dans le processus de développement de logiciel.

L'auteur définit une spécification testable comme étant une spécification à partir de la quelle il est possible d'affirmer la conformité du logiciel grâce à l'utilisation d'une technique économique.

3.4.7. L'indéterminisme et la coordination par Vuong Chanson et Loureiro

Dans leurs travaux [Vuon 93] les auteurs ont décrit avec détail l'influence de l'indéterminisme et de la coordination sur les tests de protocole.

- **L'indéterminisme**

Un comportement indéterministe signifie que deux exécutions avec les mêmes entrées d'un même système peuvent produire plus qu'une séquence unique valide de sorties. Un tel comportement constitue un problème important des tests de logiciels de communication. Il rend difficile l'accès à certaines parties à tester du système. Une même séquence d'entrées peut mener le système à plusieurs états différents. Ceci devient problématique si on veut stimuler des parties probablement erronées du système. En plus dans de tels systèmes (indéterministes), même si on découvre une faute et on la corrige, il est difficile de vérifier si elle a été bien corrigée puisqu'il n'est pas évident de reprendre une même exécution.

Il existe trois principaux types d'indéterminisme dans les logiciels de communication:

<u>L'indéterminisme dû à la concurrence entre processus</u>

Un système de communication distribué est constitué d'un ensemble de processus concurrents communiquants entre eux en échangeant des messages. Dans certains

modèles où la communication se fait par la technique de *rendez-vous*, cette communication constitue une synchronisation des processus. Vue que l'exécution de chaque processus est séquentielle, l'ordre d'exécution de l'ensemble des processus est difficile à déterminer. Ceci car le système est distribué et il est difficile de se synchroniser sur une horloge globale. Ceci peut engendrer des situations où l'ordre des messages de sorties entraîne des situations indéterministes. L'exemple de la figure 3.9, représente un exemple de système où ce genre de situations peut arriver. Le système considéré est composé de deux processus concurrents *F*1 et *F*2 recevant leurs entrées respectivement à partir des files q_1 et q_2, et c_{12} qui est une file interne reliant les sorties de *F*1 à *F*2. Supposons que nous envoyons au système une séquence d'entrées $a^n.b$. Si le système n'est pas synchronisé sur une horloge globale, il peut produire les séquences de sortie suivantes: aucune sortie ou f^n.

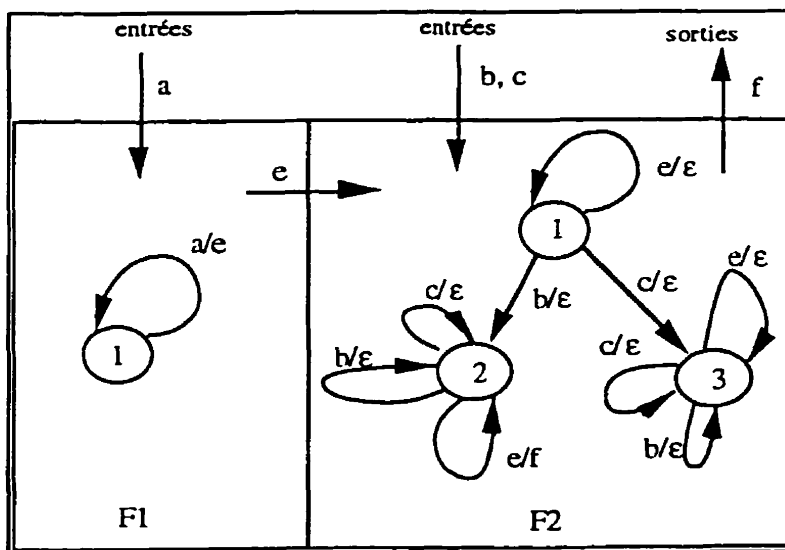


Figure 3.9. Exemple de processus concurrents.

<u>L'indéterminisme introduit dans la spécification</u>

À l'inverse du type d'indéterminisme précédent ce genre d'indéterminisme se trouve dans la spécification d'une entité de protocole. Il est dû en partie au niveau d'abstraction utilisé pour décrire la spécification. Cet indéterminisme peut être éliminé lors de l'implémentation en prenant en considération plus d'informations. La figure 3.10 représente un exemple d'une spécification où il y a ce genre d'indéterminisme. En effet, à

partir de l'état 2, pour l'entrée *a*, l'entité de protocole répond soit par *e* et reste dans l'état 2, soit par *d* et transite à l'état 1.

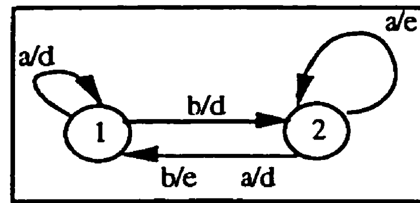


Figure 3.10. Exemple de spécification ayant un comportement indéterministe.

L'indéterminisme dû au manque d'observabilité

Le comportement d'un module déterministe peut être vu comme indéterministe si l'interface du module est indirectement accessible. Plusieurs exemples peuvent illustrer un tel comportement, nous citerons par exemple les architectures en couches des modèles *ISO* ou *TCP-IP*, etc.

- **Coordination**

La coordination dans un système distribué a deux différentes composantes: la communication et la synchronisation. Les deux composantes ensembles déterminent la meilleur manière de coordonner les processus en vue d'éviter certains problèmes de communication. La communication décrit la façon dont les processus d'un même système communiquent entre eux en échangeant des messages. La synchronisation décrit l'ordre d'exécution des processus pour accomplir le comportement du système.

Communication

Les processus appartenant à un système distribué ne peuvent pas utiliser des zones mémoire partagées pour échanger de l'information. Pour ces processus, il existe trois formes de communication: échange de messages, appels de procédures à distances (*RPC: Remote Procedure Call*) et les transactions.

Synchronisation

L'ordre d'échange de messages entre les processus d'un système distribué est important, il peut affecter le comportement normal du système. Une mauvaise synchronisation peut entraîner un blocage du système. Le problème des philosophes est

un exemple qui peut illustrer une situation où l'ordre est primordial. En plus, une mauvaise synchronisation peut engendrer des situations où l'ordre des messages de sorties entraîne des situations indéterministes (voir figure 3.9). Ceci complique le processus de tests en rendant difficile l'accès à certaines parties à tester du système.

L'analyse d'accessibilité est l'une des techniques les plus utilisées pour analyser les problèmes de synchronisation. Cette technique est très efficace, mais elle est limitée par le problème d'explosion d'états.

La synchronisation peut engendrer des situations où l'ordre des messages de sorties entraîne des situations indéterministes. L'exemple de la figure 3.9, représente un exemple de système où ce genre de situations peut arriver. Le système considéré est composé de deux processus concurrents *F*1 et *F*2 recevant leurs entrées respectivement à partir des files q_1 et q_2, et c_{12} qui est une file interne reliant les sorties de *F*1 à *F*2. Supposons que nous envoyons au système une séquence d'entrées $a^n.b$. Si le système n'est pas synchronisé sur une horloge globale, il peut produire les séquences de sortie suivantes: aucune sortie ou f^n.

3.4.8. Protocoles auto-stabilisés

Un protocole est dit auto-stabilisé si, indépendamment de son état initial, il converge automatiquement vers un état légal (*safe, legal or legitimate*) au bout d'un certain nombre d'étapes. Ce concept a été introduit par Dijkstra [Dijk 74], il garantit qu'à partir d'un état illégal du à une perturbation du fonctionnement normal (*unsafe, illegal or illegitimate*), le système peut y remédier automatiquement. Cette propriété est importante dans le domaine de tolérance aux fautes. Elle permet d'éviter une certaine classe d'erreurs appelées les erreurs perte de coordination. Ces erreurs sont les suivantes:

- Initialisation inconsistante: les processus d'un même système peuvent commencer dans des états qui sont inconsistants.
- Erreurs de transmission: dues au service offert par les couches inférieures, un message peut être corrompu, perdu, délivré en retard ou en désordre. Dans de pareils cas l'état de l'émetteur n'est plus consistant avec celui du receveur.

- Erreurs de reconfiguration: un système peut être reconfiguré dynamiquement. Ceci donne lieu à une nouvelle configuration qui peut entraîner des inconsistances entre les différents processus.

- Changement de modes: un système distribué permet l'utilisation de différents modes dépendament de plusieurs facteurs comme par exemple l'utilisation des liens, qualité de service, etc.. Tous les processus impliqués dans une communication doivent adopter un même mode, sinon l'état du système est inconsistant.

- Défaillance logicielle: si une partie du code d'un processus est temporairement hors service, son état interne devient inconsistant avec les autres processus.

- Défaillance matérielle: ceci est le même facteur que le précédant mais appliqué aux moyens matériels utilisés.

Vu que ces propriétés peuvent influencer les tests, Loureiro [Lour 96] a proposé certains principes pour la conception de protocoles auto-stabilisés. L'auteur a proposé des principes de conception permettant de détecter des propriétés instables du système, c'est-à-dire des propriétés qui peuvent changer tout au long de l'utilisation du protocole. Ceci permet d'améliorer les tests en trouvant les places où on doit les prendre en considération.

3.4.9. Facteurs influençant le test en orienté objet

À cause des différences entre les modèles, les méthodes de test classiques ne sont pas directement applicables au modèle orienté objet. Le polymorphisme, l'héritage et l'encapsulation sont des propriétés spécifiques des langages orientés objet. Certains types d'erreurs de l'orienté objet n'existent pas dans les langages conventionnels. Afin de garantir la conformité d'un système orienté objet, le test unitaire de chaque composante en isolation est nécessaire mais non suffisant. L'un des problèmes de test de programmes orientés objet est dû au fait que les méthodes dans une classe peuvent être invoquées dans n'importe quel ordre, ce qui engendre le problème d'ordonnancement des méthodes pour pouvoir détecter les erreurs.

Dans ce qui suit, nous allons citer quelques travaux portant sur les facteurs qui peuvent influencer les tests en orienté objet.

- **Facteurs énoncés par Binder**

D'après Binder [Bind 94], le coût et le niveau de difficulté du test en orienté objet peuvent être réduits en agissant sur un ensemble de principes de bases de la conception et en planifiant les tests. D'une façon générale, la testabilité d'un logiciel est le résultat de six facteurs qui sont les suivants: les caractéristiques de la représentation (analyse et conception), les caractéristiques de l'implémentation, les capacités des tests incorporés (*Built-in*), la suite de test et le processus du développement.

Pour la représentation, l'auteur la classe en quatre différents axes (voir figure 3.11): les besoins de l'utilisateur, la spécification, la trace et la séparation des intérêts. Pour chacun de ces axes, l'auteur donne un ensemble de caractéristiques qui influencent la testabilité (voir figure 3.11). Les besoins de l'utilisateur doivent être faisables, quantifiables, objectifs, et ayants les propriétés de la norme IEEE/ANSI 830 [ANSI 84]. La spécification de son coté, doit être complète et ayant les propriétés du standard IEEE/ANSI 1016 [ANSI 87]. La trace est une notion logique qui peut prévenir un large éventail de problèmes. Elle consiste à enregistrer les antécédents de chaque travail effectué. Dans le processus de test, il est important de faire l'association entre les modules implantés, la spécification et les besoins de l'utilisateur. Sans cela, il est impossible de conduire adéquatement les tests. La séparation des intérêts est un principe de base du génie logiciel. Chaque composante doit agir le plus indépendamment possible des autres composantes.

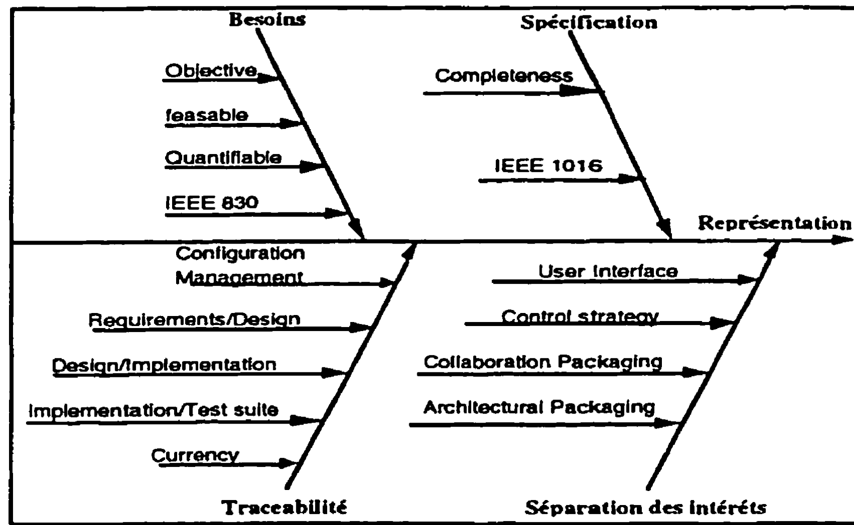


Figure 3.11. Testabilité de la représentation.

- **Facteurs énoncés par Perry et Kaiser**

Dans leur article [Perr 90], Perry et Kaiser décrivent l'application des axiomes présentés par Weyuker [Weyu 88] (voir table 3.7, 8 et 9) dans le domaine de l'orienté objet.

À travers des exemples, les auteurs illustrent les faiblesses de l'affirmation stipulant qu'un programme qui utilise du code hérité nécessite moins d'effort de test. Les axiomes 1 à 7 sont intuitivement évidents et/ou n'ont par d'effet particulier sur la programmation orienté objet. Par contre les axiomes 8, 9, 10 et 11 sont ceux qui affectent la programmation orienté objet. Ils ont conclu que *"L'héritage est l'un des points forts de la programmation orientée-objet. Mais c'est précisément à cause de l'héritage que l'on rencontre des problèmes dans la phase des tests.*

Axiome 1: *Applicabilité*
Pour chaque programme, il existe un ensemble adéquat de tests.
Axiome 2: *Non-Exhaustive Applicability*
Il doit être parfois possible de tester un programme de façon adéquate sans que l'ensemble des tests ne soit exhaustif.
Axiome 3: *Monotonicité*
Si T est adéquat pour P, et T est un sous ensemble de T' alors T' est adéquat pour P.
Axiome 4: *L'ensemble vide est inadéquat*
L'ensemble vide est un ensemble de test inadéquat pour tout programme.

Table 3.7. Axiomes de 1 à 4.

Soit un programme Q et soit x un identificateur dans Q. Si l'on remplace toutes les instances de l'identificateur x par un identificateur y qui n'existe pas dans Q, on dit que x a été renommé. Un programme P est un renommage d'un programme Q si P est identique à Q à l'exception près d'un ensemble d'identificateurs de Q renommés dans P.

Axiome 5: *Renommage*
Soit P un renommage de Q; T est adéquat pour P si et seulement si T est adéquat pour Q.
Axiome 6: *Complexité*
Pour tout n, il existe un programme P qui est adéquatement testé par un ensemble de test de taille n mais par aucun ensemble de test de taille n-1.
Axiome 7: *Couverture d'instructions*
Si un ensemble de test T est adéquat pour un programme P, alors T entraîne l'exécution de tout traitement exécutable de P.
Axiome 8: *Anti-extensionalité*
Si deux programme P et Q ont les mêmes fonctionnalité, un ensemble de test adéquat pour P n'est pas nécessairement adéquat pour Q.

Table 3.8. Axiomes de 5 à 8.

Deux programmes ont la même forme (on dit aussi syntaxiquement similaire) si on peut passer de l'un à l'autre par l'application des règles suivantes :

a- remplacer l'opérateur relationnel $r1$ dans un prédicat par l'opérateur relationnel $r2$.

b- remplacer la constante $c1$ dans un prédicat ou une affectation par la constante $c2$.

c- remplacer l'opérateur arithmétique $a1$ dans une affectation par l'opérateur arithmétique $a2$.

Axiome 9: *Changement Général et Multiple*
Si deux programme *P* et *Q* ont la même forme, un ensemble de test adéquat pour *P* n'est pas nécessairement adéquat pour *Q*.
Axiome 10: *Anti-décomposition*
Si un programme est adéquatement testé ceci n'entraîne pas que toutes les composantes de ce programme ont été adéquatement testées pour d'autres contextes.
Axiome 11: *Anti-composition*
Si toutes les composantes d'un programme ont été adéquatement testées ceci n'entraîne pas que le programme a été adéquatement testé.

Table 3.9. Les axiomes de tests de Weyuker de 9 à 11.

- **Facteurs énoncés par Smith et Robson**

Smith et Robson [Smit 90] décrivent certains problèmes rencontrés lors des tests de programmes orientés objet. Durant la phase de maintenance, l'héritage peut accroître les tests requis. Les auteurs affirment qu'il est plus facile à tester des programmes orientés objet dont les classes dérivées héritent toutes les méthodes de la classe parent, c'est-à-dire l'héritage stricte. Les cas de tests d'une classe parent restent valide pour les classes dérivées. Des cas de tests additionnels sont requis pour les classes ayant de nouvelles méthodes ou des méthodes redéfinies.

3.4.10. Contrôlabilité et Observabilité

Une grande partie des travaux sur la testabilité du logiciel évoquent l'observabilité et la contrôlabilité comme caractéristiques importantes qui peuvent influencer le coût des tests. Ces facteurs ne sont pas nouveaux; ils ont été utilisés dans le domaine du test de matériel [Will 73]. Initialement, ces facteurs avaient une origine mathématique dans la théorie des systèmes [Kalm 69].

- **Dssouli et Fournier**

Dssouli et Fournier [Dsso 91a] décrivent les avantages de l'approche modulaire de développement du logiciel. Cette approche améliore la qualité de la spécification ainsi que de l'implémentation. Pour la spécification, cette approche améliore la clarté, diminue la complexité et augmente la flexibilité pour d'éventuels extensions. D'autres parts, cette approche permet une implémentation graduelle, produit un noyau exécutable dans un délai raisonnable et elle accélère le processus de développement en permettant le partage de la tâche de codage sur plusieurs personnes. D'un autre coté la modularité n'a pas été très exploitée dans la phase de tests. Dssouli et Fournier ont analysé cet aspect ; elles ont commencé par décrire les objectifs et les problèmes de tests. Par la suite, elles ont présenté une nouvelle définition de la testabilité (voir section 2.2) basée sur l'instrumentation. Elles ont proposé une extension du cycle de vie du protocole prenant en considération les problèmes des tests. les auteurs ont présenté trois techniques utilisées lors de la spécification pour rendre les points d'interaction (entre les modules) observables:

- L'addition de primitives: cette technique consiste à étendre l'aspect fonctionnel des points d'interactions en leur rajoutant des primitives. Le rôle de ces dernières est de capter la trace de tous les événements qui se passent au niveau de ces points.
- Diffusion sélective: par définition, un point d'interaction relie deux modules ensemble. Les auteurs proposent d'étendre cette définition de façon à ce qu'elle prenne en considération un troisième module appelé observateur. Dans un système contenant des observateurs, lorsqu'un module envoie un message vers un module destinataire, le message est diffusé en même temps au module destinataire et à l'observateur.
- Appel directe de procédure de trace: cette technique est aussi appelée instrumentation du code [Prob 82]. Elle consiste à rajouter dans le code des appels de procédure de trace. Le choix de l'emplacement de ces appels correspond aux points d'interaction.

Les auteurs ont défini et classifié les notions d'observation et d'observateurs. Elles ont montré l'utilité de rendre la structure interne du logiciel visible. Elles ont fait la relation entre la granularité de l'observation et la facilité du test. Elles ont proposé quatre niveaux d'observation:

- Observation structurelle: ce type d'observation permet d'observer et/ou de contrôler tous les points d'interaction d'un système issu d'une décomposition modulaire.

- Observation des transitions: elle permet d'observer des transitions spécifiques pour des implémentations basées sur le modèle d'automates.
- Observation des entrées/sorties: les points d'interaction sont insérés entre l'implémentation et son environnement ou entre les différentes composantes d'un module pour capter la trace des entrées/sorties.
- Observation fonctionnelle: ce type d'observation est basé sur l'instrumentation du code [Prob 82]. Il utilise une décomposition fonctionnelle du code basé sur la recherche des invariants.

- **Probert**

Probert et al. [Prob 82] et [Prob 90] proposent une technique qui améliore l'observabilité du système. Cette dernière est une technique de tests boîte grise appelée instrumentation sémantique. Elle consiste à insérer dans le code des appels de procédure dont l'objectif est de capter et de rapporter la trace d'exécution du système.

Les emplacements des appels de procédure de traces sont déterminés lors de la phase de conception. Pour cela, les auteurs [Prob 90] proposent une technique de décomposition du comportement de module en classes d'équivalences appelées *ECB* (*Equivalence Classes of Behavior*). L'appel de procédure de traces est inséré dans le code au niveau de chaque comportement représentant une classe d'équivalence de la conception. Ces appels permettent de rapporter l'exécution de la partie du code visée et de l'associer à L'ECB correspondant.

Cette technique nous donne de l'information pertinente sur la couverture des tests en termes d'*ECB* couverts. Les auteurs ont appelé ce type de couverture la couverture sémantique. Cette information peut être utilisée pour guider la construction de tests additionnels.

Les auteurs [Prob 90] ont illustré (manuellement) le déroulement des différentes étapes de cette technique sur l'exemple de la spécification du protocole du bit alternant. Ils ont exprimé la couverture sémantique des tests de ce protocole.

- **Freedman**

Freedman dans "*Testability of software components*" [Free 91] a présenté un nouveau concept de la testabilité d'une composante d'un programme relatif au domaine

(*domain testability*). Deux types de composantes de programmes ont été considérées: les procédures et les fonctions. Une composante d'un programme est dite testable relativement à son domaine si elle est observable et contrôlable. Une composante est dite observable si pour chaque valeur d'entrée il ne correspond qu'une et une seule valeur de sortie. Une composante est dite contrôlable si pour chaque valeur de sortie, il existe une valeur d'entrée permettant de générer cette valeur de sortie (voir section 3.3).

- **Saleh**

Dans son travail "*Testability-directed service definitions and their synthesis*" [Sale 92], l'auteur a proposé d'étudier la testabilité durant la définition du service. Les besoins de la testabilité ont été considérés comme des fonctions spéciales qui sont rajoutées au service de base du protocole. Ce rajout permet d'obtenir des spécifications de service testables. La spécification est alors dérivée à partir des services testables en utilisant les techniques de synthèse et de raffinement proposées par Probert [Prob 91a]. Cette méthode permet d'obtenir des spécifications qui sont correctes par construction. Elles ne nécessitent pas de grands efforts de vérification comme dans le cas où les mécanismes de testabilité sont introduites au niveau des procédures de conception. Dans ce dernier cas, la spécification devient complexe et difficile à vérifier.

Saleh a commencé par introduire les aspects architecturaux et fonctionnels qui améliorent l'observabilité et la contrôlabilité durant le processus de tests. Il a défini l'observabilité comme étant l'aptitude d'observer le comportement d'une entité de protocole en utilisant les points d'accès au service inférieurs et supérieurs.

Au niveau des aspects architecturaux, l'auteur a proposé l'introduction de nouveaux points d'accès au service appelés *T-SAP* (*Tests Service Access Point*). Ces points d'accès ne sont utilisés que durant le processus de tests et ont des propriétés identiques à ceux qui sont définis dans l'architecture *ISO*. Ces points permettent d'améliorer l'observabilité et la contrôlabilité. Du point de vue fonctionnalité, l'auteur propose d'introduire de nouvelles primitives qui ne sont accessibles qu'à partir des *T-SAPs*. Ces primitives ont pour objectifs, d'une part, d'améliorer l'observabilité par des mécanismes de lecture de l'état et de trace (*Probe, Trace*), d'autre part, d'améliorer la contrôlabilité par des mécanismes de réinitialisation du système et de positionnement dans un état particulier (*Reset* et *Set*).

- **Kim et Chanson**

Dans leur travail commun *"Design for testability of protocols based on formal specifications"* [Kim 95], Kim et Chanson ont traité l'instrumentation de l'implémentation de protocoles pour améliorer leurs observabilité et contrôlabilité. Même si l'approche proposée n'est pas introduite lors de la définition du service, elle reste assez similaire à celle qui est proposée par Saleh. Elle consiste à améliorer la testabilité d'un protocole en rajoutant des primitives de contrôle et d'observation. Cette méthode s'applique aussi bien aux implémentations issues de spécifications décrites en Estelle normalisé qu'aux spécifications séquentielles et concurrentes. Les auteurs attirent l'attention des lecteurs que l'instrumentation telle qu'elle est proposée, permet d'améliorer l'observabilité et la contrôlabilité, mais peut interférer avec le comportement normal du protocole.

Pour les spécifications séquentielles, les auteurs proposent d'améliorer l'observabilité en introduisant deux nouvelles primitives. Ces primitives sont *TEST_observe* et *TEST_unobserve* et sont utilisées comme un interrupteur pour activer ou désactiver l'observation. L'observation consiste à rapporter les traces d'exécution grâce à des instructions de sorties rajoutées à la fin de chaque transition conditionnelle. Ce mécanisme permet au testeur d'identifier la transition qui a été exécutée. Pour l'amélioration de la contrôlabilité, les auteurs proposent l'introduction de deux nouvelles primitives *TEST_control* et *TEST_uncontrol*. Ces dernières, servent comme un interrupteur pour activer ou désactiver le contrôle des transitions. Ce mécanisme est utilisé en vue de positionner le système dans un état permettant l'exécution d'une transition.

En ce qui concerne les spécifications concurrentes, les auteurs définissent la spécification globale comme une collection de modules écrits en *Estelle* normalisé. L'implémentation conserve la structure de la spécification et des modules qui la composent. L'observation consiste à associer le comportement de modules individuels dans une implémentation concurrente à celui qui est défini par les modules de la spécification de référence. La comparaison du comportement de l'implémentation vis à vis de sa spécification s'effectue en utilisant une relation d'implémentation [Broo 84]. Pour la contrôlabilité, les auteurs utilisent les points d'interaction internes qui permettent de tester individuellement les modules.

3.4.11. Méthodes formelles

La capacité de description et les caractéristiques des méthodes de description formelles (*FDT: Formal Description Techniques*) influencent les méthodes de génération et d'application des tests. Dans ce contexte, Larsen [Lars 90] propose la testabilité comme l'un des quatre critères de comparaison des méthodes formelles. Les autres facteurs sont les capacités d'expression, les capacités de composition et la décidabilité.

- **Yu**

En se basant sur la définition de la *spécification orientée testabilité* qu'il a proposé (Définition 10), l'auteur [Yu 91] identifie deux caractéristiques que doit satisfaire le formalisme de spécification pour aboutir à une spécification testable. Ces caractéristiques sont les suivantes:

- l'aptitude d'exprimer tous les aspects de communication, de synchronisation et de comportement concurrent observé lors des tests.
- la méthode de spécification est basée sur un modèle temporel qui facilite la génération et l'application des cas de test.

Comme application de ces caractéristiques, l'auteur a étendu le travail de Lamport [Lamp 78] sur les horloges logiques pour l'utiliser comme base pour spécifier et tester les systèmes distribués. Afin de spécifier un module ou un système à partir de son comportement externe observable et du temps relatif, l'auteur a utilisé le langage *ETAL* (*Extended Trace Assertion Language*) qui est une extension du langage d'assertion temporel *TAL* (*Trace Assertion Language*) [Parn 89]. Les horloges relatives sont utilisées comme base pour la définition du modèle des événements globaux. Ce dernier est utilisé pour l'analyse des résultats des tests issus d'une spécification ainsi décrite. Cette technique d'analyse des tests permet d'améliorer la testabilité en identifiant les événements spontanés et en interprétant les tests non concluants qui résultent de collisions.

- **Testabilité et SDL**

European Telecommunication Standard Institute

L'*ETSI* (*European Telecommunication Standard Institute*) [ETSI 94] propose certains principes de la spécification qui peuvent faciliter les tests de conformité d'un protocole. Ces principes constituent la base du choix des concepts du langage de

spécification formel *CCITT SDL* (*Specification and Description Language*) pour l'adoption des standards (*ETS*). Ces concepts sont les suivants:

- la consistance: un *ETS* doit être consistant, c.à.d. qu'il ne devrait pas contenir de contradictions dans le texte, ou entre le texte et les diagrammes, ou entre les différents diagrammes.
- la clarté: un *ETS* doit définir clairement et sans ambiguïté les besoins du produit de télécommunication qu'il spécifie. La spécification doit être bien structurée pour faciliter la révision par des experts humains.
- l'utilisation correcte des formalismes: les diagrammes présentés dans un *ETS* doivent être synthaxiquement et sémantiquement correctes vis à vis du langage formel utilisé. L'utilisation de concepts qui ne sont pas supportés par des outils, doit être évitée. Ce principe est essentiel pour utiliser des outils automatiques de vérification, de validation et de dérivation des tests.
- éviter l'explosion des états: une classe importante de la validation formelle se base sur l'exploration des états. En spécifiant d'importants nombres d'instances de processus ou des types de données infinies, l'espace de représentation des états devient énorme et difficile à explorer. Ce phénomène s'appelle l'explosion d'états, il rend impossible l'utilisation d'outils de validation courantes.
- éviter l'indéterminisme implicite: il est difficile de découvrir l'indéterminisme s'il n'est pas explicitement spécifié. Ce genre d'indéterminisme peut échapper au concepteur de tests. Il est difficile de déterminer s'il est intentionnel ou si c'est une erreur dans le standard.
- indiquer les options d'implémentation: un standard doit clairement décrire les options d'implémentation. Ceci est important pour la sélection des cas de tests.
- indiquer la partie normalisée du standard: un standard doit indiquer quelles sont les parties normalisées du standard. Ceci détermine les besoins de conformité associés au standard.

- l'utilisation d'un seul niveau d'abstraction: un standard ne doit pas spécifier le système en plusieurs niveaux d'abstraction. Ceci est essentiel pour éviter une interprétation fausse des besoins de conformité associés au standard.

Luo

Dans leur article "*Software Testing Based on SDL Specifications with Save*" [Luo 94b], Luo et al. proposent d'appliquer les méthodes de test des *FSMs* sur des spécifications décrites en *SDL*. Pour atteindre cet objectif, ils proposent de transformer une spécification *SDL* en une spécification équivalente en *FSM* sur laquelle ils appliquent les méthodes classiques de test des *FSMs*. Les auteurs identifient la construction *SAVE* de *SDL* comme étant difficile à tester. La transformation qu'ils proposent tient compte de ce fait et produit un *FSM* sans cette construction. La construction *SAVE* peut donc être considérée comme l'un des facteurs de testabilité relié aux constructions même de la technique formelle de description.

Ellsberger

Au cours de leur travail [Ells 92], les auteurs ont pu dégager trois propriétés sémantiques du langage de spécification *SDL* qui donnent lieu à des spécifications difficiles à tester. Ces aspects peuvent donc être considérés comme des facteurs de testabilité. Ces facteurs sont les suivants:

- Communication asynchrone entre les modules: ce facteur influence la testabilité du système [Dsso 90b] puisque dans les spécifications utilisant ce genre de communication on ne peut pas trouver une relation directe entre les actions initiées à partir de l'environnement et la réaction du système. Pour rendre possible une telle relation, le système doit être conçu de façon à répondre dans un délais maximal fixe.

- Le modèle d'utilisation du temps: il est impossible de spécifier des contraintes de temps réel en *SDL*. Ce dernier est un langage de spécification qui n'est pas doté de structures permettant d'exprimer le temps réel. Ceci limite la testabilité de spécifications utilisant les timers. Pour y remédier le concepteur doit définir une relation entre les contraintes de temps réel et la façon de les modéliser en *SDL*.

- L'indéterminisme: comme on l'a déjà présenté, ce facteur est difficilement testable puisque la relation entre les entrées du système et sa réaction n'est pas unique. Pour

garantir l'efficacité des tests, l'auteur propose d'utiliser de l'information supplémentaire et d'imposer des contraintes structurelles à la spécification décrite en *SDL*.

- **Lotos**

Partant de leur expérience dans la spécification et les tests en *Lotos*, les auteurs [Burg 92] ont soulevé le problème de l'écart qui existe entre la théorie et la pratique. Ils ont proposé deux facteurs qui peuvent influencer les tests pratiques. Le premier facteur est l'indéterminisme; il est associé à la spécification. Concernant ce facteur les auteurs proposent de limiter le plus possible l'indéterminisme lors de la spécification. Le deuxième facteur est associé à l'interface de l'implémentation. Les auteurs proposent que les implémentations aient des interfaces dont la description est le plus proche possible de la spécification, c'est-à-dire celles qui acceptent les mêmes séquences d'entrées-sorties.

3.4.12. Influence de l'environnement

Le test de conformité d'une implémentation *I* d'un module *M* est moins puissant lorsque le module est encapsulé dans un certain environnement *E* que lorsqu'il est isolé. Partant de cette constatation, l'auteur [Drir 92] introduit les graphes de refus comme base de l'analyse de testabilité de systèmes de transition étiquetés. L'analyse de testabilité se base sur l'idée intuitive que les tests à travers un environnement ne peuvent que dégrader la testabilité. De tels tests peuvent juger non conformes des implémentations qui sont en réalité conforme. La testabilité du système *M* à travers son environnement *E* est bonne si le test du système global "*I* dans *E*" écarte autant d'implémentation non conformes que le test direct *I*. L'approche proposée permet de retrouver les principaux facteurs de testabilité à savoir la contrôlabilité et l'observabilité du composant sous test. Elle traite de la même manière l'analyse de la testabilité d'un système partiellement ou totalement contrôlable et observable.

3.4.13. Facteurs qui influencent la testabilité des protocoles de communication

Une liste de facteurs influençant la testabilité des protocoles de communication peut être retrouvée dans différents travaux. Parmi cette liste, nous pouvons citer l'observabilité et la contrôlabilité [Dsso 91b] et [Kim 95], la perte de coordination [Lour

96], etc. Ces facteurs ont des origines mathématiques et matériel [Kalm 69] et [Will 73] ; ils ont été adapté aux protocoles de communication. D'autres facteurs un peu plus spécifiques ont été aussi présentés, nous citerons par exemple, la coordination [Vuon 93] et l'indéterminisme, les méthodes formelles [Burg 92, ETS 94, Luo 94a, etc.], l'environnement [Drir 92], etc. Les facteurs évoqués pour le logiciel classique restent valables.

3.4.14. Discussion

Les facteurs influençant les tests sont aussi liés à la phase du cycle de développement dont l'auteur s'est intéressé. Beaucoup d'auteurs sont d'accord que la testabilité est une propriété composée de qualité de logiciel. Elle dépend de plusieurs facteurs qui peuvent être regroupés en classes qui ne sont pas nécessairement indépendantes:

- les facteurs généraux: la description de ces facteurs n'est pas très formelle. Leur interprétation et leur compréhension ne sont pas uniques, elles peuvent être vue de différentes manières et leur quantification n'est pas directe. Dans cette classe de facteurs, nous pouvons citer à titre d'exemple: la simplicité de la conception et la conception bien documentée [86], l'efficacité [Perr 87]. etc.

- les facteurs spécifiques: ces facteurs décrivent des propriétés intrinsèques du système qui influencent les tests. À titre d'exemple nous pouvons citer l'indéterminisme et la coordination [Vuon 93], les routines de récupération pour les facteurs à risques [86], etc.

- les facteurs liés à l'environnement de développement: cette classe de facteurs dépend des moyens utilisés pour le développement du produit. Dans cette classe, nous pouvons citer par exemple les *FDT*s [Lars 90].

- les facteurs liés à l'environnement de fonctionnement et de test du produit: cette classe de facteurs regroupe les facteurs qui sont indépendants des caractéristiques intrinsèques du produit développé. Ces facteurs dépendent de l'environnement dans le quel le produit opérera et/ou sera testé [Drir 92].

- l'observabilité et la contrôlabilité: ce sont les facteurs qui influencent l'observabilité et la contrôlabilité du produit. L'accès aux points d'interaction entre les modules du

système est l'un des facteurs les plus importants dans cette classe [Prob 82], [Dsso 91b], [Sale 92] et [Kim 95], etc.

- les facteurs associés à l'orienté objet: cette classe regroupe les facteurs qui influencent les tests en orienté objet. L'héritage est l'un des facteurs les plus important dans cette classe [Smit 90].

L'ensemble des facteurs de testabilité et particulièrement les facteurs spécifiques ne sont pas complètement cernés. Les rares travaux qui existent ont évoqué certains de ces facteurs sans pour autant donner comment les améliorer. Dans le reste de cette thèse, nous poursuivrons dans cette direction pour proposer de nouveaux facteurs spécifiques associés aux modèles de spécification utilisés. Nous proposerons des évaluations de ces facteurs afin de les utiliser comme guide pour l'amélioration de la testabilité.

3.5. Moyens utilisés pour améliorer la testabilité

La phase de conception a une grande influence sur la qualité du produit final. Elle peut être considérée comme le pivot du processus de développement. Elle constitue la phase de transition de la théorie vers les résultats tangibles.

Pour que le processus de développement de logiciel aboutisse à un produit testable, il est nécessaire d'intervenir lors de la phase de conception. Les moyens utilisés pour y arriver ne sont pas encore bien définis. Dans ce qui suit nous allons en citer les rares techniques rapportées dans la littérature.

- **Révision de la conception et simulation:** Les tests doivent être développés en parallèle avec l'étape de conception fonctionnelle. L'essai sur quelques scénarios permettra de juger si l'application des tests sera satisfaisante. L'utilisation des méthodes formelles de spécification (*Estelle*, *SDL*, *Lotos*, etc.) et des outils de simulation associés nous facilitent la tâche. En effet, la simulation est un excellent moyen pour avoir une idée assez précise sur la performance et le degré de testabilité du système à développer. Elle permet de suivre le comportement des différents modules d'une conception et les interactions internes sous divers scénarios. Cette technique permet de détecter les situations anormales de fonctionnement du système. Elle peut révéler certaines propriétés pouvant affecter la complexité des tests comme:

l'indéterminisme, le synchronisme entre les processus, la complétude des spécifications, etc.

- **Application des métriques:** Plusieurs recherches se sont basées sur les métriques pour l'amélioration de la testabilité de la conception ou de l'implémentation [Voas 92], [Petr 93a] et [Voas 93]. Au cours de la section 3.3, nous avons présenté les principales métriques de testabilité. Ces dernières sont utilisées pour détecter les parties difficiles à tester du système considéré. Ceci constituera une base pour transformer le système pour le rendre mieux testable. Pour l'illustration, nous pouvons citer le travail de Freedman [Free 91] dans lequel l'auteur présente une définition de la testabilité des composantes de programmes (les procédures et les fonctions). Par la suite, en se basant sur cette définition, il présente deux métriques relatives à l'observabilité et la contrôlabilité du système. Ces dernières sont utilisées pour étendre la description des procédures ou des fonctions en vue d'améliorer leur testabilité.

- **Instrumentation:** L'instrumentation consiste à doter la conception ou l'implantation de mécanismes qui améliorent sa testabilité. Ces mécanismes sont essentiellement des procédures de trace insérées dans la conception et/ou le code [Prob 82] et [Dsso 91a]. Saleh [Sale 92] et Kim [Kim 95] ont proposé d'ajouter des primitives qui ne sont utilisées que pour les fins de tests. Ces dernières sont évoquées pour rapporter l'état du système (observabilité) ou pour accéder à un état particulier du système (contrôlabilité). Dans cette même direction, certains protocoles de signalisation d'ISDN ou d'ATM sont initialement spécifiés avec un PDU spécial "*status*" permettant de rapporter l'état du système. D'un autre côté, Probert [Prob 82] a proposé une nouvelle approche de tests appelée les tests *boîte grise*. Elle consiste à laisser la structure modulaire de la conception observable lors de l'implémentation. D'un point de vue formel, cette méthode est une technique d'instrumentation de la conception. Elle consiste à rajouter à la spécification des points d'observation et/ou de contrôle au niveau des interfaces des composantes non observables (respectivement non contrôlables) du système.

- **Approche modulaire de développement:** Cette approche de développement du logiciel améliore la qualité du produit résultant. Cette technique améliore la clarté, diminue la complexité et augmente la flexibilité du système à développer. Elle permet

d'implémenter graduellement les différents modules du système et de les tester avant de les intégrer. Cette technique n'a pas été trop exploitée au cours de la phase de tests. Seulement quelques travaux [Prob 82], [Dsso 91a] et [Kim 95] ont proposé des mécanismes d'observation et de contrôle au niveau des points d'interactions entre les modules. D'un autre point de vue, cette approche est la base même du paradigme orienté objet. La personne qui développe de nouveaux systèmes peut utiliser des objets testés contenus dans des librairies.

- **Synthèse à partir du service testable:** Cette technique consiste à étudier les besoins de la testabilité lors de la définition du service de base du protocole. La spécification est alors générée à partir de services testables en utilisant les techniques de synthèse et de raffinement proposées par Probert et al. [Prob 91a]. Cette approche nous évite de grands efforts de vérification puisque la spécification générée par les techniques de synthèse est correcte par construction. Saleh [Sale 92] a étudié cette approche et a proposé d'étendre le service avec des primitives d'observation et de contrôle. Ces dernières sont prises en compte dans la spécification du protocole et dans l'implantation du système. Elles permettent d'avoir une meilleure observabilité et contrôlabilité du système à tester.

Chapitre 4

Prise en compte de la testabilité dans le cycle de développement du protocole[1]

4.1. Introduction

Dans le cadre du chapitre 3 (section 3.2.3), nous avons dressé une liste de définitions de la testabilité. La testabilité y est considérée comme une propriété de qualité qui peut caractériser chaque produit du cycle de développement de logiciel. La testabilité peut être intégrée dans le cycle de développement du logiciel et en particulier lors des premières étapes du cycle. Au cours des chapitres 5, 6, 7 et 8, nous proposerons des techniques de conception et d'analyse des besoins qui permettent d'améliorer la testabilité du produit final.

Ce chapitre est composé de deux parties principales. Lors de la première partie, nous proposons de considérer la testabilité comme une activité intégrante du cycle de développement du protocole. Dans la deuxième partie, nous expliquons comment cette activité peut améliorer certaines activités du cycle (génération et application des tests) sans en altérer les autres.

4.2. Prise en compte de la testabilité dans le cycle de développement du logiciel

L'étude de la testabilité d'un produit peut être considérée comme un processus de raffinement itératif (voir figure 4.1) constitué des étapes suivantes: l'évaluation de la testabilité, l'étude des facteurs qui influencent la testabilité et le choix de transformations

---

[1] Les résultats de ce chapitre sont publiés dans: IEEE GLOBECOM'96, Londre, Novembre 1996 [Karo 96a], Article Invité IWPTS'95, France, 1995 [Dsso 95].

adéquates, l'application des transformations, une nouvelle évaluation de la testabilité et finalement le remplacement du produit initial par le produit transformé. La figure 4.1 illustre un tel processus de raffinement. Si on rencontre l'une des conditions suivantes ce processus s'arrête :

- après une série de transformations, on obtient une testabilité optimale,
- il n'existe plus de transformations possibles pouvant améliorer la testabilité,
- le niveau de testabilité du produit résultant est jugé satisfaisant par la personne en charge de l'étape.

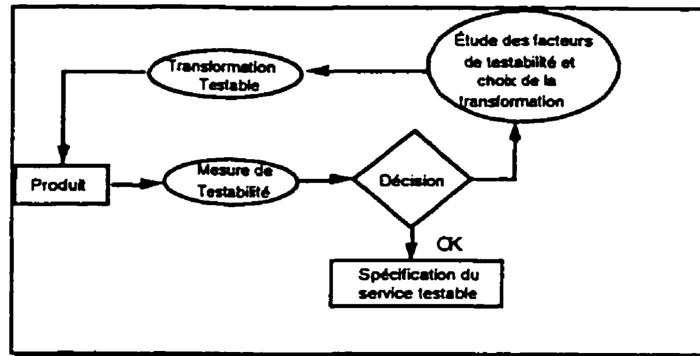


Figure 4.1. Processus de raffinement de la testabilité.

Comme nous l'avons présenté dans le chapitre 2, le cycle de vie du logiciel est composé d'un certain nombre d'étapes séquentielles. Le passage d'une étape à une autre se fait via l'application de certaines activités spécifiques. Ces dernières donnent lieu à un nouveau produit.

L'étude de la testabilité peut être intégrée dans le cycle de développement du logiciel en tant que nouvelle activité (voir figure 4.2). Cette activité peut être prise en considération à chaque étape du cycle. L'étude de la testabilité précédera à chaque fois les autres activités classiques du cycle de développement et génère un produit intermédiaire testable. On appliquera sur ce dernier les activités usuelles du cycle de développement pour générer un nouveau produit et passer à l'étape suivante.

Cette approche d'extension du cycle de développement du logiciel est valable quelque soit le type du logiciel et en particulier, pour les protocoles de communication. En

effet, les protocoles de communications font partie de la classe des systèmes réactifs et donc aux logiciels d'une façon générale.

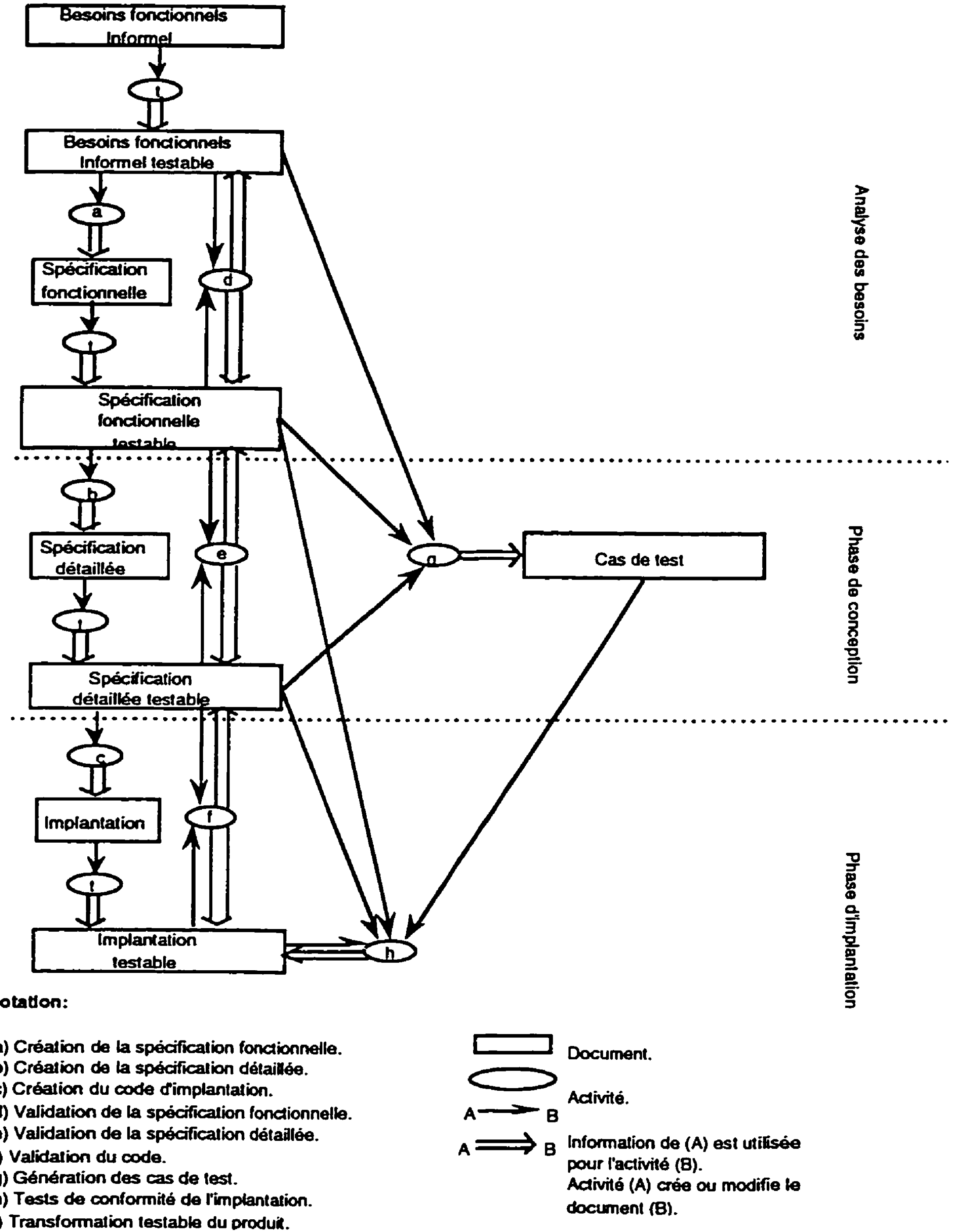


Figure 4.2. Cycle de vie étendu.

4.3. Considération précoce de la testabilité et influence sur l'implantation

L'un des objectifs de la testabilité est la réduction du nombre de tests requis pour affirmer l'exactitude d'un logiciel. Pour atteindre cet objectif, Voas et al. [Voas 95], présentent deux solutions:

- la sélection d'un ensemble de tests ayant une grande capacité de révéler des fautes,
- la conception de logiciels dont l'exécution révèle automatiquement des défectuosités si le code contient des fautes.

Parmi ces deux solutions, les auteurs choisissent la deuxième. Pour accomplir cet objectif, la conception doit avoir certaines propriétés qui seront propagées dans l'implémentation et qui permettent:

- d'atteindre le plus de parties possibles du code en variant les entrées du système,
- le programme doit contenir des constructions (exemple: assertions) qui rendent l'état du programme incorrect si les constructeurs eux même sont incorrects,
- le programme doit être capable de propager des états incorrects en révélant des défectuosités.

La génération du code à partir d'une spécification de référence est un processus de raffinement successif. Ce processus permet de passer graduellement d'un certain niveau d'abstraction à un autre moins élevé. Comparée à la spécification initiale, l'implémentation peut être considérée comme une spécification moins abstraite. La formalisation et l'automatisation de ce processus permet de préserver le comportement et les propriétés de qualité (dont la testabilité) de la spécification. Plus la différence d'abstraction entre la spécification et l'implémentation est grande, plus ce processus est complexe.

Pour parvenir à la construction de programmes à partir de spécifications formelles, on peut distinguer quatre approches [Chop 88]:

"- Synthèses de programme: cette approche vise à générer automatiquement des programmes à partir de formules logiques et de traces. Dans le domaine des logiciels de communication la synthèse de protocoles se fait à partir de la spécification du service.

- Approche fondée sur la notion de preuve: un programme est développé puis il y a une preuve a posteriori démontrant que le programme satisfait bien sa spécifications de référence.

- Une approche uniquement basée sur la notion de transformations: à partir de la spécification de référence, le programme est dérivé par des transformations systématiques successives.

- Une approche mixte que l'on pourrait appeler construction assistée de programmes: une squelette ou certaines parties du programme sont dérivés systématiquement des spécifications, les autres parties sont développées à la main puis prouvées."

Il existe certaines règles formelles qui garantissent que les approches adoptées pour dériver les programmes préservent les propriétés requises. Si telles règles n'existent pas, il faut prouver que ces transformations préservent ces propriétés. La preuve consiste à vérifier si les comportements du programme et de sa spécification de référence sont équivalents. Elle dépend alors étroitement du modèle et de l'approche de dérivation utilisés.

Pour les automates à états finis avec entrées-sorties, la spécification et l'implantation sont représentées par leurs comportements observables. Dans ce cas, la preuve consiste à trouver une correspondance adéquate formelle entre la spécification la moins abstraite et la spécification la plus abstraite. Cette correspondance est appelée une relation d'implantation. Elle associe les objets modélisant les implantations et ceux modélisant les spécifications. L'association entre ces deux représentations est appelée conformité et est décrite dans plusieurs travaux dont [Ledu 91], [Tret 92], [Drir 92], etc.

4.4. Approche adoptée

Dans les chapitres qui suivent nous allons proposer un ensemble de techniques permettant l'étude des facteurs qui influencent les tests, leur évaluation respective et leur utilisation comme guides pour l'amélioration de la testabilité. Ces techniques sont appliquées lors des premières étapes du cycle de développement du protocole tel qu'il est présenté dans les figures 4.1 et 4.2.

L'application de ces techniques a pour objectif de produire des spécifications détaillées pourvues de propriétés de testabilité. La façon dont est rajoutée la testabilité (figure 4.2) n'altère pas les autres activités classiques du cycle de vie : création de la spécification formelle, création de la spécification détaillée, création du code et validation, etc. Elle permet, par contre, de faciliter l'activité de génération des cas de test et celle des tests de conformité.

La testabilité telle qu'elle sera présentée dans les chapitres qui suivent, analyse particulièrement des propriétés d'association entre les entrées et les sorties observables (simples ou séquences) des systèmes. Si la méthode utilisée pour la dérivation du code préserve les propriétés de traces (équivalence de traces), elle pourra automatiquement propager les propriétés de testabilité considérées lors de la spécification. Ceci car, les propriétés de testabilité considérées sont des propriétés de trace.

La phase de conception est une étape composée du cycle de développement de logiciel. C'est un processus de raffinement permettant de passer d'une conception de haut niveau à une conception détaillée (spécification détaillée) [Sinc 84]. Au cours des chapitres qui suivent, nous proposerons certaines propriétés de testabilité spécifiques associées à trois modèles de spécification différents. Ces trois modèles peuvent être utilisés dans la phase de conception à des niveaux d'abstraction différents. Nous montrerons comment ces propriétés de testabilité influencent les activités de génération et d'application des tests. Nous appellerons ces propriétés les *facteurs de testabilité* et nous proposerons pour chacune d'elles une évaluation. Nous utiliserons les facteurs de testabilité ainsi que les métriques associées pour classer les spécifications (d'après leur degré de testabilité). Le concepteur les utilisera comme guide afin de raffiner ou de transformer (dans la mesure du possible) les spécifications pour améliorer leur testabilité.

Chapitre 5

Étude de la testabilité basée sur le modèle relationnel[1]

5.1. Introduction

Les différentes techniques de spécification ont été introduites (au niveau de la conception) pour réduire l'ambiguïté que peut engendrer l'utilisation du langage naturel, comme l'anglais, et aussi pour systématiser le plus possible chacune des étapes et activités du cycle de développement (voir figure 2.1).

Le modèle relationnel tel qu'il est présenté par Tarski [Tars 41], Sanderson [Sand 80], Mili [Mili 90] ou Jaoua [Jaou 92] est un des modèles utilisé pour décrire la spécification d'un logiciel. Ce modèle est basé sur une théorie simple, souple et puissante. En plus de sa souplesse d'utilisation, le modèle relationnel dispose d'une multitude d'opérateurs simples dont l'utilisation est facile à comprendre et permet le développement de mesures simples. Mili [Mili 90] est l'un de ceux qui ont utilisé ce modèle pour décrire les spécifications de programme. Il a défini une spécification comme un objet qui est décrit par deux composantes: un espace S et une relation R. L'espace S défini les variables manipulées par un programme. La relation R contient tous les couples d'entrée/sortie considérés correctes par l'utilisateur. R est définie sur deux ensembles: le domaine et l'image. Le domaine contient tous les états d'entrée légaux qu'un utilisateur peut soumettre à un programme. L'image contient tous les états de sortie qu'un utilisateur considère correcte et qui sont associés aux états d'entrée. Cette technique de modélisation permet d'analyser certains aspects des spécifications et/ou des programmes en se basant

---

[1] Les résultats de ce chapitre sont publiés dans: Publication départememtales no: 921 [Karo 94] et 1050 [Karo 96b].

sur certaines propriétés des relations associées. Par exemple, Mili [Mili 90] a proposé les propriétés de *symétrie*, de *déterminisme*, ou de régularité des relations pour analyser la tolérance aux fautes de programmes. D'un autre côté, Jaoua [Jaou 92] a utilisé d'autres propriétés comme la *difonctionnalité* et la *rectangularité* des relations pour étudier la décomposition et les dépendances dans les bases de données relationnelles. Dans ce qui suit, nous allons utiliser ce modèle pour introduire et analyser la testabilité des protocoles de communication au niveau de la spécification. Dans le domaine des protocoles de communication, la spécification du service d'un protocole sous forme relationnelle donne au concepteur des outils pour l'identification de la plate-forme idéale de test et l'amélioration de la testabilité. Selon l'information disponible, nous pouvons utiliser ce modèle à des différents niveaux d'abstraction. À un niveau d'abstraction élevé, nous pouvons décrire une spécification d'un protocole par ses primitives d'entrées-sorties. Plus on descend dans l'abstraction, plus on peut considérer les valeurs des variables internes d'un système et en particulier son état.

Dans la suite de ce chapitre, nous utiliserons ce modèle pour étudier et améliorer la testabilité d'un protocole de communication au niveau de la spécification de service. Nous adapterons la définition de testabilité de programmes proposée par Freedman [Free 91] à fin de l'utiliser pour les spécifications de service. Pour les besoins de l'analyse de testabilité nous utiliserons parfois certains détails de la spécification du protocole impliquant les messages échangés. Nous donnerons souvent des exemples qui ne représentent qu'une vue partielle du service, c'est-à-dire une composition particulière (partielle) des relations qui le composent.

5.2. Définitions de base

On présente ci-dessous, un bref aperçu de quelques définitions de bases nécessaires à la description de notre modèle. Nous utilisons la notation proposée par Mili [Mili 90]: Soit la relation $R: E \rightarrow S$,

- *Domaine* de R : $Dom(R) = \{e \in E \mid \exists s \in S : (e, s) \in R\}$,
- *Image* de R : $Im(R) = \{s \in S \mid \exists e \in E : (e, s) \in R\}$,
- Relation *identité* : $Id = \{ (e, e) \mid e \in E\}$,
- Relation *vide* : $\emptyset = \{ \}$,
- Relation *inverse* : $R^{-1} = \{(s, e) \mid (e, s) \in R\}$,
- *Produit* de relations : Le produit de deux relations R_i et R_j est noté

$R_i * R_j = \{(e, s) \mid \exists\, e' : (e, e') \in R_i \text{ et } (e', s) \in R_j\}$,

- *Union* de relations : L'union de deux relations R_i et R_j est noté $R_i \parallel R_j = \{(e, s) \mid (e, s) \in R_i \text{ ou } (e, s) \in R_j\}$,
- *Pré-application de R à entrée*: La pré-application d'une relation R à une entrée $e \in E$ est l'ensemble $R(e) = \{s \in S \mid (e, s) \in R\}$,
- *Puissance de relations* : $R_i^1 = R_i$ et $R_i^n = R_i * R_i^{n-1}$ pour n > 1,
- *Cardinalité de R* : $Card(R)$ = nombre d'éléments de R.

Nous proposons un ensemble de nouvelles définitions nécessaires à la suite du travail:

- *Pré-application de R à un ensemble* : La pré-application d'une relation R à un ensemble d'entrées $A \subseteq E$ est l'ensemble $R(A) = \{s \in S \mid (e, s) \in R \text{ et } e \in A\}$
- *Entrées indéterministes* d'une relation : $ind(R) = \{e \in Dom(R) \mid Card(R(e)) > 1\}$.
- *Indéterminisme* d'une relation : $Ind(R) = \{(e, s) \in R \mid e \in ind(R)\}$.
- *Sorties convergentes* d'une relation : $conv(R) = \{s \in Im(R) \mid Card(R^{-1}(s)) > 1\}$.
- *Convergence* d'une relation : $Conv(R) = \{(e, s) \in R \mid s \in conv(R)\}$.
- *Masquage* d'une relation : soit une relation $R = R_i * R_j$, le masquage de R est l'ensemble des entrées indéterministes de R_i qui ne sont plus indéterministes dans la relation composée R. Ceci est dû au fait que l'indéterminisme est caché par des sorties convergentes de R_j (c'est-à-dire $R_i(e) \subseteq R_j^{-1}(s)$). Nous définissons le masquage comme suit : $masq(R) = \{e \in ind(R_i) \mid \exists\, (e, s) \in R \text{ et } s \in ind(R_j^{-1}) \text{ et } R_i(e) \subseteq R_j^{-1}(s)\}$.

5.3. Le modèle proposé

Les concepts de service et de protocole sont souvent confus. Le service définit les opérations qu'une couche est prête à offrir à ses utilisateurs (la couche au dessus) mais ne décrit en rien la façon dont ces opérations sont réalisées. En contre partie, le protocole est un ensemble de règles et de conventions que les entités homologues utilisent pour la réalisation du service.

Du point de vue de l'utilisateur, le fournisseur de service est une boîte noire qui offre une série d'interactions avec un autre utilisateur (voir figure 5.1). L'utilisateur est concerné par le comportement observable du fournisseur de service via les points d'accès au service (*SAP: Service Access Point*).

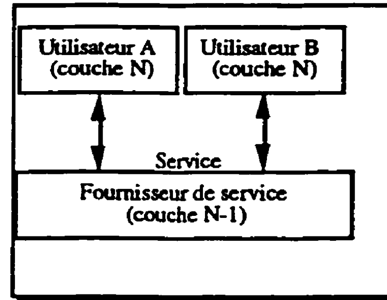


Figure 5.1. Modèle du fournisseur de service.

Le comportement observable d'un logiciel peut être représenté par différentes sortes de diagrammes. Parmi les plus connus, nous pouvons citer les machines à états finis (*FSMs*), les diagrammes de séquencement temporel (*time sequencing diagram*) ou les diagrammes de séquencement de messages (*MSC ou Message Sequencing Chart*). Ces diagrammes permettent de représenter la séquence de réactions d'un système suite à la réception d'une entrée ou d'une séquence d'entrées. Comme exemple de systèmes où ces diagrammes peuvent être utilisés, nous pouvons considérer le service associé à un protocole. Dans ce dernier cas, le diagramme adopté permet d'associer les primitives envoyées à celles qui sont reçues. La figure 5.2 illustre par un *MSC* le service associé à un protocole spécifique. Cette association entre des entrées et des sorties d'un système nous rappelle les notions de bases des relations où à chaque élément du domaine de la relation on associe une ou plusieurs images (voir section 5.2).

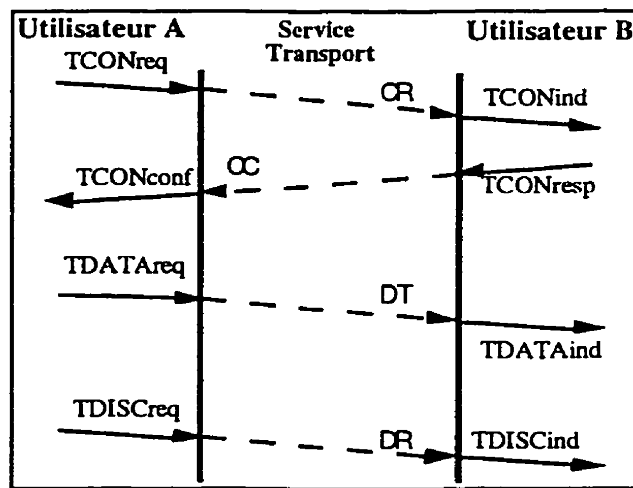


Figure 5.2. Séquences de primitives du service de transport d'*OSI*.

Dans ce qui suit nous allons utiliser le modèle relationnelle pour la description des spécifications de service d'un protocole. Nous considérons qu'un système est modulaire. L'interaction entre les modules permet l'accomplissement des fonctionnalités du système global. La façon dont les modules d'un système sont reliés est appelée architecture du système. À chaque module M du système on associe une relation R qui à chaque entrée du module associe une ou plusieurs sorties (voir figure 5.3). Un module M est représenté par une relation R définie comme suit: R: $E=(\mathcal{E} \cup \varepsilon) \rightarrow S=(\mathcal{S} \cup \varepsilon)$ où, E est le domaine de la relation, $\mathcal{E}$ un ensemble fini d'entrées, $\mathcal{E}=\{e_k$ avec $1 \leq k \leq n\}$ et $\mathcal{E} \neq \emptyset$, S est l'image de la relation, $\mathcal{S}$ un ensemble fini de sorties, $\mathcal{S}=\{s_j$ avec $1 \leq j \leq m\}$ et ε représente l'entrée ou la sortie *nulle*. Cette entrée (resp. sortie) fait toujours partie du domaine (resp. l'image) de la relation.

On supposera que la relation R associée à la spécification du service d'un module M est non vide. On peut rajouter aux définitions de base de la section 5.2, les notions d'entrée et de sortie observables Input(R) et Output(R):
$Input(R) = \{e \neq \varepsilon \in E: R(e) \neq \varepsilon\}$ et $Output(R) = \{s \in S: R^{-1}(s) \neq \varepsilon\}$.

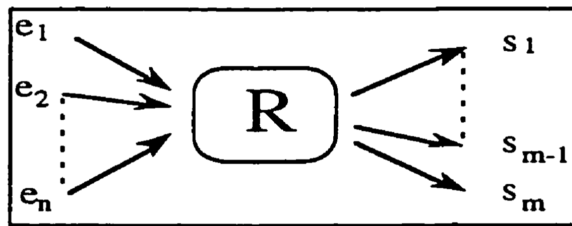


Figure 5.3. Relation associée à un module.

La relation représentant le service est généralement composée de plusieurs sous-relations. Chacune de ces sous-relations représente une vue du service à une interface particulière du système. La figure 5.4 illustre une relation composée qui représente le service de la figure 5.2. Ce modèle de représentation a l'avantage, d'une part de permettre l'étude individuelle des relations qui décrivent une vue particulière du service. D'autre part ce modèle permet l'étude de la relation composée qui représente le service global. Cette propriété permet d'analyser le service à des niveaux d'abstraction différents. Cette possibilité de composition/décomposition du service et de passer d'un niveau d'abstraction à un autre est très importante pour l'analyse de la testabilité. Elle a le grand avantage de faciliter la recherche des causes d'une faible testabilité à une étape précoce du cycle de développement du protocole.

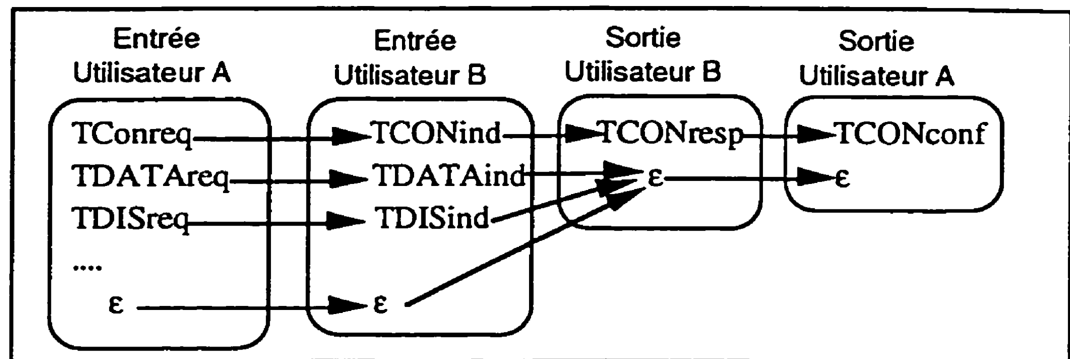


Figure 5.4. Représentation relationnelle du service au niveau de l'utilisateur A.

5.4. Testabilité d'un module

Au cours de ce chapitre nous nous baserons sur la définition de la testabilité présentée par Freedmann [Free 91] (Définition 5, chapitre 3). Nous choisissons cette définition parce que: d'une part, elle est fondée sur un ensemble de propriétés caractérisant les entrées/sorties d'une composante logicielle. Ces propriétés peuvent être propagées de la spécification à l'implantation. D'autres parts, même si elle traite de l'implantation de composantes logicielles, elle peut être adaptée à la spécification de modules d'un protocoles (Définition 4.1). Selon l'auteur, une expression ou une commande est dite testable si elle est observable et contrôlable. L'expression ou la commande est observable si elle produit des sorties différentes pour des entrées différentes. Par contre, elle est contrôlable si pour chaque sortie, il existe au moins une entrée qui force le programme à produire cette sortie.

Définition 4.1: Observabilité, Contrôlabilité et Testabilité

Soit la relation $R\ E \rightarrow S$ associée à un module M,

1- R est dite ***observable*** si elle donne des sorties distinctes pour chaque entrée distincte. Plus la cardinalité de *Input*(R) est proche de celle de $\mathcal{E}$ plus la relation est observable. Pour des fins de comparaison, une relation R est dite plus observable que R': $E \rightarrow S$ si *Input*(R')$\subset$*Input*(R).

2- De la même manière, une relation R est dite ***contrôlable*** si pour chaque sortie, il existe au moins une entrée qui force le programme à produire cette sortie. Plus la cardinalité de *Output*(R) est proche de celle de $\mathcal{S}$ plus la relation est contrôlable. Une relation R est dite plus contrôlable que R': $E \rightarrow S$ si *Output*(R')$\subset$*Output*(R).

3- La définition de la testabilité découle directement de celle de la contrôlabilité et de l'observabilité. Une relation R est dite *testable* si elle est contrôlable et observable. Plus la cardinalité de *Output*(R) est proche de celle de $\mathcal{S}$ et plus la cardinalité de *Input*(R) est proche de celle de $\mathcal{E}$ plus la relation est testable. Une relation R est dite plus testable que R': $E \rightarrow S$ si *Output*(R')$\subset$*Output*(R) et *Input*(R')$\subset$*Input*(R).■

Dans ce qui suit, nous appellerons toutes les propriétés des relations que nous présenterons et que nous associeront à la testabilité *facteurs de testabilité*. La définition 4.1, décrit informellement des associations entre les entrées-sorties d'un module. Nous proposerons de formaliser cette définition (voir définition 4.5.1) en utilisant les quatre facteurs de testabilité suivants (voir sections 5.4.1., 5.4.2., 5.4.3 et 5.4.4.): le *degré de définition* de R, le *degré d'information* de R, l'*indéterminisme* de R et la *convergence* de R. Nous associerons à ces facteurs des métriques bornées entre 0 (mauvais) et 1 (idéal). L'utilisation de ces métriques permettra de classer les spécifications d'après les facteurs qui leur sont associées. Plus l'évaluation des facteurs est proche de 1 plus la spécification est testable et réciproquement, plus c'est proche de 0 moins c'est testable. Dans ce qui suit, nous donnerons une description de ces facteurs, de leur influence sur les tests ainsi que des métriques associées.

5.4.1. Degré de définition de R

La relation R: $E \rightarrow S$ associée à un module M est dite complètement définie si, quel que soit l'entrée du module $\neq\varepsilon$, la relation associée R lui assure au moins une image $\neq\varepsilon$. Une entrée d'un module est dite non définie si elle n'appartient pas à l'ensemble des entrées observables (*Input*(R)) de la relation R (voir section 5.3).

En réalité, une entrée non définie cause une réaction interne non observable à partir de l'interface du module. En général, les actions internes augmentent la latence des fautes, c'est-à-dire, les défaillances surviennent longtemps après l'occurrence de fautes. Dans ce cas, les activités de test et de diagnostic sont complexes, il est plus difficile d'interpréter les résultats des tests, c'est-à-dire d'associer les bonnes entrées aux bonnes sorties. Cette propriété contribue à la réalisation de l'observabilité du module (Définition 4.1). En effet, si la sortie nulle n'est pas facile à discerner, l'application d'une séquence d'entrées (cas de test) provoque une séquence de sorties où il est difficile d'associer les entrées aux bonnes sorties. Ceci va à l'encontre de la première règle décrite par la Définition 4.1. Nous

appellerons ce facteur ***degré de définition*** de R, nous le noterons $Def(R)$ et nous le calculerons comme suit :

$$Def(R) = \frac{Card(Input(R))}{Card(Dom(R)) - 1}, 0 \leq Def(R) \leq 1 \quad (1)$$

La figure 5.5 illustre ce facteur. Elle représente l'un des scénarios associé à l'émetteur du service du protocole de contrôle d'erreur du type *selective repeat* [Hals 92]. Dans ce protocole, il n'y a pas d'association temporelle entre l'envoi de trames et la réception d'accusé de réception. Après l'émission d'une trame, l'accusé de réception doit être reçu dans un délai fixe. Ceci peut entraîner que des entrées soient non définies (exemple $I(N+1)$). Le degré de définition de la relation R_A de la figure 5.5 est $Def(R_A)=1/3$. Les relations associées aux autres scénarios possibles sont généralement non complètement définies aussi.

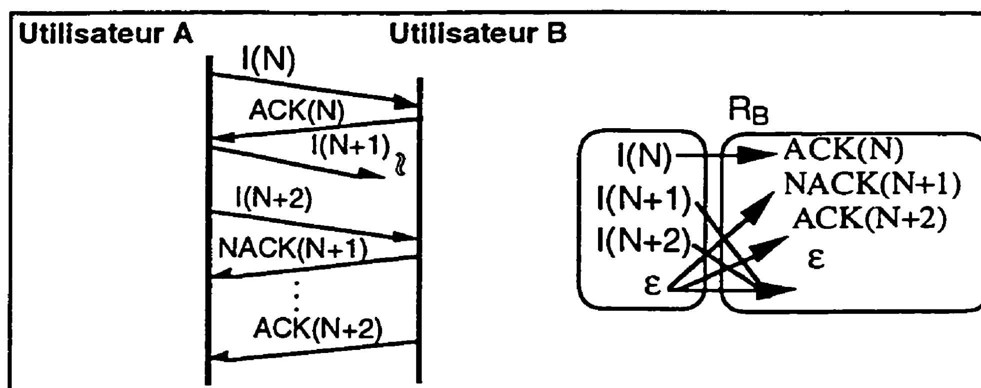


Figure 5.5. Exemple de relation non complètement définie.

5.4.2. Degré d'information de R

La relation R: $E \cup \varepsilon \rightarrow S \cup \varepsilon$ associée à un module M est dite complètement informée si quel que soit la sortie $\neq\varepsilon$ du module, il existe au moins une entrée $e \neq\varepsilon$ appartenant à E qui puisse provoquer la sortie. Une sortie d'un module est dite non informée si elle est provoquée par l'entrée nulle ε uniquement. Les sorties non informées sont notées $Noninf(R)=\{s \in S \mid R^{-1}(s)= \{\varepsilon\}\}$. Les situations induites par de telles sorties non informées ressemblent à celles produites par les transitions spontanées dans le domaine des tests de protocoles de communication. Dans notre cas, les sorties dont au moins un des antécédents est différent de l'entrée nulle ε sont considérées comme informées. De telles situations ont un impact négatif sur les tests puisque nous ne

pouvons pas trouver des entrées observables qui peuvent produire ces sorties. Par rapport à la Définition 4.1, cette propriété agit sur la contrôlabilité d'un module (deuxième règle). Nous noterons ce facteur $Inf(R)$ et nous le calculerons comme suit :

$$Inf(R) = \frac{Card(Output(R))}{Card(Im(R)) - 1}, \ 0 \leq Def(R) \leq 1 \ (2)$$

L'exemple de la figure 5.5 reste valide pour illustrer le degré d'information puisque R_A n'est pas complètement informée. Par exemple, l'accusé de réception $ACK(N+2)$ n'est pas informé. Le degré d'information de cette relation est $Inf(R_A)=1/3$.

5.4.3. Déterminisme de *R*

La relation R associée à un module M est dite déterministe si quel que soit l'entrée définie ($\neq\varepsilon$) appliquée au module, on obtient une seule image par R. Ce facteur agit sur la contrôlabilité telle qu'elle est présentée dans la Définition 4.1. Si le nombre de sorties associées à une entrée définie ($\neq\varepsilon$) est supérieur à 1, l'entrée est dite indéterministe et les sorties associées sont difficiles à produire. Le test des protocoles de communication indéterministe est un problème complexe [Fuji 91b] et [Luo 93]. Si la relation associée à un module est indéterministe, il est difficile de forcer le module à exécuter certaines fonctionnalités à tester à partir de son interface d'entrée. Dans un pareil cas, il faut supposer que les techniques et le formalisme utilisées pour la description du module ainsi que son environnement offrent la propriété d'équité. Cette propriété coûteuse, assure l'exécution de chaque fonctionnalité à tester en soumettant le module aux mêmes entrées un certain nombre de fois. L'indéterminisme a donc un impact important sur la testabilité. Nous noterons ce facteur $D(R)$ et nous le calculerons comme suit:

$$D(R)=1-\frac{Card(Ind(R))}{Card(R)-1}, \ 0 \leq D(R) \leq 1 \ (3)$$

Cette formule compare le nombre d'instances de la relation dont les entrées ont plus qu'une sortie au nombre total d'instances de la relation. Pour illustrer ce facteur, nous pouvons prendre l'exemple d'un protocole orienté connexion. Dans cet exemple, le résultat de la fonction d'établissement de connexion n'est pas déterministe, c'est-à-dire la réponse à une requête de connexion peut être positive ou négative suivant le service offert par les couches plus basses (voir figure 5.6). La figure 5.7 représente la relation (R_I)

associée à une vue du service relative à la partie qui demande la connexion. Dans un pareil cas $D(R_I)=0$.

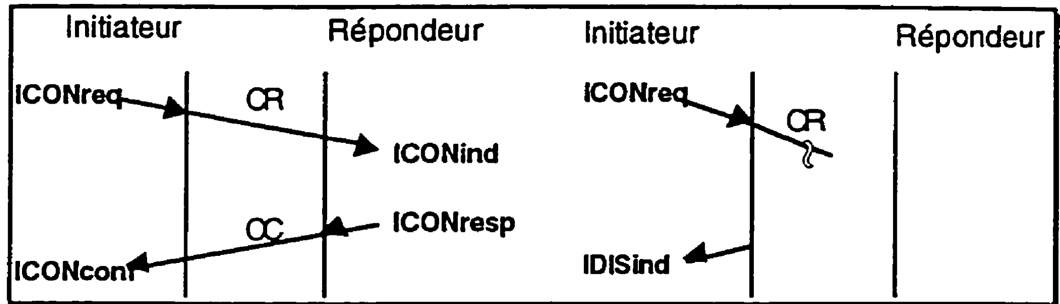


Figure 5.6. Service associé au processus d'établissement de la connexion.

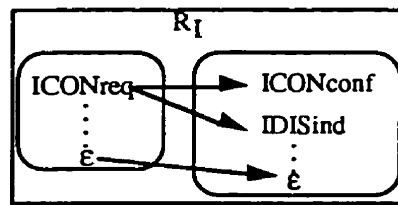


Figure 5.7. Relation associée au service de la figure 5.6.

5.4.4. Convergence de *R*

Ce facteur est assez similaire au déterminisme de R. La seule différence, c'est qu'il caractérise la relation inverse R^{-1} de R. En d'autres termes, les entrées et les sorties sont inversées de façon à ce qu'on puisse raisonner sur R^{-1}. Cette propriété affecte particulièrement l'observabilité des modules (règle 1 de la Définition 4.1). Les protocoles qui associent plusieurs entrées à une seule sortie ($\neq \varepsilon$) perdent de l'information durant leur exécution. En effet, ce type de protocole ne communique pas toutes les informations sur leur états internes [Voas 93]. Lors des tests, cette perte d'information peut masquer un état interne incorrecte et diminuer donc la probabilité d'observation des erreurs. Ce facteur est noté $D(R^{-1})$ et est calculé comme suit :

$$D(R^{-1}) = 1 - \frac{Card(Ind(R^{-1}))}{Card(R) - 1} \quad 0 \leq D(R^{-1}) \leq 1 \quad (4)$$

L'exemple de la figure 5.8 montre un exemple d'une relation convergente. La convergence d'une telle relation est égale à $D(R^{-1})=1/4$

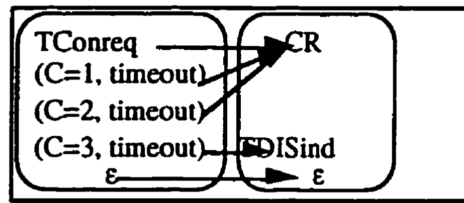


Figure 5.8. Exemple de relations convergente.

5.4.5. Mesure Unique de la testabilité

La relation globale associée au service d'un module peut être caractérisée par un vecteur de quatre facteurs qui influencent la testabilité du module. Ce vecteur est appelé *vecteur de testabilité*. Il est constitué des facteurs suivants: le déterminisme, la convergence, le degré de définition et le degré d'information. Le vecteur de testabilité est noté ainsi:

$$C(R) = \langle D(R), D(R^{-1}), Def(R), Inf(R) \rangle.$$

Ce vecteur de mesures donne au concepteur une idée détaillée sur les facteurs qui peuvent rendre les tests difficiles. En fonction de la valeur associée à ces facteurs, le concepteur peut raffiner la spécification en considérant de nouvelles informations. Vue le niveau d'abstraction adopté, les informations en question n'étaient pas considérées. Les métriques associées à ce vecteur n'exigent aucune hypothèse sur les caractéristiques internes du module. En prenant en considération les facteurs de testabilité, la Définition 4.1 peut être réécrite comme suit:

Définition 4.5.1:

1- R est dite observable si elle est complètement définie et R^{-1} est déterministe. Autrement dit, son vecteur de testabilité est : $C(R) = \langle D(R), 1, 1, Def(R) \rangle$,

2- R est dite contrôlable si elle est complètement informée et R est déterministe. Son vecteur de testabilité est donc : $C(R) = \langle 1, D(R^{-1}), Def(R), 1 \rangle$,

3- R est dite testable si elle est complètement informée, complètement définie, R est déterministe et R^{-1} est déterministe, donc $C(R) = \langle 1, 1, 1, 1 \rangle$. ■

Chacun des facteurs précédants est borné entre 0 et 1. Le cas où $C(R) = \langle 1, 1, 1, 1 \rangle$ représente le cas optimal (Définition 4.5.1) et réciproquement, plus la valeur de $C(R)$ est proche de $\langle 0, 0, 0, 0 \rangle$ plus le module est difficile à tester.

Les quatre mesures que nous venons de présenter peuvent être pondérées et combinées pour produire une mesure unique. Il existe différentes manières de les combiner. Le choix de l'une ou de l'autre dépend de l'importance accordée à chacun des facteurs. Dans ce qui suit nous présentons l'une des manières de le faire. La mesure que nous proposons est appelée *mesure de testabilité combinée CTM(R)* (*Combined Testability Measure*). Cette mesure se base sur la distance du vecteur de testabilité d'un module par rapport au meilleur cas ($C(R) =< 1, 1, 1, 1>$):

$$\mathrm{CTM(R)} = 1 - \frac{\sqrt{(1-D(R))^2+(1-D(R^{-1}))^2+(1-Def(R))^2+(1-Inf(R))^2}}{2} \quad , 0 \leq CTM(R) \leq 1 \quad (5)$$

Cette métrique donne une idée globale de la testabilité. Cette dernière ainsi les quatre précédantes (Formules (1), (2), (3) et (4)) sont toujours applicables et ne nécessite aucune hypothèse sur les caractéristiques internes du module. La précision de ces mesures dépend en fait du niveau d'abstraction au quel elles sont appliquées.

5.5. Classification des modules d'après leur testabilité

L'étude de la testabilité telle qu'elle est décrite dans les sections précédantes mène à une classification de la testabilité du service des modules d'après les caractéristiques des relations associées. Cette classification est basée sur les quatre éléments du vecteur de testabilité $C(R)$. La variation des valeurs de chacun des facteurs de testabilité ($0 \leq D(R) \leq 1$, $0 \leq D(R^{-1}) \leq 1$, $0 \leq Def(R) \leq 1$ et $0 \leq Inf(R) \leq 1$) peut être utilisée pour classifier les spécifications. La classification que nous proposons est en 9 classes (voir table 5.1): les relations, les fonctions, les fonctions inverses, les applications, les relations observables, les relations contrôlables, les surjections, les injections et les relations testables. Parmi les 16 classes possibles, on s'est limité à 9 pour deux raisons: on a choisi les classes de relations qui sont très utilisées et celles de relations qui modélisent les différentes composantes de la testabilité. Cette décomposition varie de la classe des spécifications de service faiblement testables ou *relations quelconques* à la classe des spécifications de service testables ou *relations testables*. La classe des relations quelconques contient les spécifications de service dont aucun élément de $C(R)$ n'est optimal (égal à 1) (voir table 5.1): $D(R)< 1$, $D(R^{-1})< 1$, $Def(R)< 1$ et $Inf(R)< 1$. En contre partie, la classe des relations testables contient les spécifications de service dont la valeur de la métrique associée à chaque facteur de testabilité est optimale(voir table 5.1): $D(R)= 1$, $D(R^{-1})= 1$, $Def(R)= 1$ et $Inf(R)= 1$. Cette dernière classe contient les spécifications de service dont les

relations associées sont bijectives. Dans la réalité, de telles spécifications sont quasiment impossible à obtenir. Lors de la conception, nous ne chercherons pas donc à produire de telles spécification, mais plutôt à optimiser la valeur associée à chacun des quatre facteurs de testabilité afin de se rapprocher le plus possible de la relation testable.

	$D(R)$	$D(R^{-1})$	$Def(R)$	$Inf(R)$
Relations	–	–	–	–
Fonctions	×	–	–	–
Fonctions inverses	–	×	–	–
Applications	×	–	×	–
Relations Observables	–	×	×	–
Relations Contrôlables	×	–	–	×
Surjections	×	–	×	×
Injections	×	×	×	–
Relations Testables	×	×	×	×

×: évaluation égale à 1 du facteur de testabilité. –: évaluation inférieure à 1 du facteur de testabilité.

Table 5.1. Classement de quelques types de relations.

5.6. Analyse de la testabilité

Si on analyse le vecteur de testabilité associé au service d'un module, on peut s'apercevoir qu'un des facteurs de testabilité est faible. Ceci peut être dû, soit aux propriétés de la relation globale du système, soit aux propriétés des différentes relations qui composent le système. Très souvent on aura besoin de raffiner la relation en vue de travailler sur un niveau d'abstraction plus adéquat. Cela nécessite souvent de regarder les interactions au niveau du protocole (*PDUs*).

Après avoir détecté le facteur de testabilité qui a une faible valeur et après avoir isolé la (ou les) relation(s) responsable(s) de cette faiblesse, nous raffinerons ces relations en vue d'améliorer le facteur de testabilité en question. Ceci est accompli par des transformations dont l'objectif est d'améliorer la testabilité. Elles consistent à raffiner et étendre (dans la mesure du possible) la relation en question en vue d'améliorer sa classe de testabilité (table 5.1). Ces changements ne sont pas toujours des modifications majeures de la conception. Il s'agit très souvent de l'identification de la plate-forme idéale de test (meilleure interface de test). Très souvent les informations requises existent, il

s'agit d'en tenir compte lors du test. Cet objectif peut être atteint par un processus de raffinements itératifs qui a comme guide la valeur associée à chacun des facteurs de testabilité ou la valeur de la mesure de testabilité unique *CTM* (voir figure 4.1). Ce processus se termine si on a atteint un niveau de testabilité que l'on juge acceptable (classe acceptable) ou si aucune amélioration de la testabilité n'est encore possible. Ceci évoque les questions suivantes:

1. Quel genre de transformations peut-on utiliser pour l'amélioration de la testabilité?
2. Quels sont les mesures adéquates de testabilité ?
3. Quand est ce que ce processus se termine ?

En général, une transformation qui améliore la testabilité doit être guidée par les facteurs qui peuvent influencer les tests. Dans le cas d'une dégradation de la testabilité, le concepteur doit déterminer les facteurs responsables et les utiliser pour le choix des transformations. La liste de ces facteurs peut inclure [Dsso 91b]: le langage de description formel, la complexité de la spécification, l'indéterminisme, la plate-forme de test ou le niveau d'abstraction utilisé pour les tests, l'architecture de test, la stratégie de test adoptée.

Malheureusement, la plupart des protocoles existants ont été conçus et documentés sans penser aux exigences des tests. Ce n'est pas non plus réaliste de changer les fonctions de bases du protocole pour améliorer sa testabilité. La marge de manoeuvre du concepteur est mince, car il doit agir sous des contraintes afin d'améliorer la testabilité.

Pour la question 2, il existe quelques travaux [Free 91], [Petr 93a] et [Karo 94a] qui se sont intéressés aux mesures de la testabilité. Les mesures qui y sont proposées ne s'intéressent qu'à un aspect particulier de la testabilité et à une phase spécifique du cycle de développement. Elles ne peuvent pas être généralisées et utilisées dans toutes les étapes du cycle. L'utilisation d'un modèle permettant la mesure de la testabilité d'un logiciel pour des fins de comparaison, peut aider le concepteur à résoudre plusieurs problèmes. Idéalement, ce modèle doit refléter tous les aspects et les facteurs qui influencent les tests. Dans ce cas, la mesure de testabilité peut être vue comme un vecteur dont chacun des éléments est une mesure individuelle d'un aspect de la testabilité [Karo 94a]. Malheureusement l'idée d'un modèle est idéaliste, chaque stratégie de test peut nécessiter l'adoption d'une mesure de testabilité particulière.

Pour la dernière question, la terminaison du processus de transformations et de raffinement dépend de la satisfaction des critères de testabilité adoptés. Le degré de testabilité que le concepteur juge adéquat pour la stratégie de test doit être utilisé comme seuil.

En général, le choix des transformations dépend des propriétés de la spécification initiale. Cet aspect peut être considéré comme un problème de classification des spécifications. Nous allons présenter ci-dessous, quatre transformations qui agissent chacune sur l'un des facteurs de testabilité déjà présentés. Ces transformations ne sont pas applicables systématiquement, elles dépendent des cas et nécessitent l'intervention du concepteur pour décider de l'opportunité de les appliquer. Dans les sections qui suivent, nous avons choisi exprès certains exemples qui montrent que ce n'est pas toujours opportun d'appliquer les transformations.

5.6.1. Transformation permettant d'éviter l'indéterminisme

L'une des causes de l'indéterminisme est le manque d'information dans la description du module [Vuon 93]. Pour éviter cet indéterminisme, il est nécessaire de prendre en compte plus d'informations pertinentes dans la description du module. Ceci conduit à l'adoption d'un autre niveau d'abstraction de la spécification. On doit considérer des informations qui puissent diviser les entrée indéterministes en plusieurs copies non identiques. Chaque nouvelle entrée sera associée à une branche de l'indéterminisme. Cette opération est appelée le ***dépliage*** des entrées (*input unfolding*)

Si le système à modéliser est séquentiel, l'ordre de réception des entrées est très important. Une primitive n'est acceptée (à l'entrée d'un module) que lorsque le module est dans un état spécifique. Dans un tel système, l'état est défini comme étant une valeur particulière de certaines variables appelées *variables d'état*. Dans ce cas, l'entrée du module peut être considérée comme une composition de l'état du module et de la primitive d'entrée (figure 5.9). Pour éviter l'indéterminisme d'un système séquentiel, on raffinera soit les états soit les primitives d'entrée du système. Pour raffiner les états, il est nécessaire de considérer de nouvelles variables d'état dans sa description initiale. Ceci cause la décomposition de chaque état indéterministe $état_i$ en un ensemble d'états $\{état_{i,j}\}$ où j est le nombre de valeurs que peuvent prendre les variables nouvellement considérées. Chaque nouvelle copie de l'état $état_i$ prendra en charge une branche de l'indéterminisme. La même approche peut être appliquée aux primitives d'entrée. Chaque primitive

correspondante à une entrée indéterministe peut être déployée en plusieurs copies en lui associant de nouveaux paramètres. Il y aura autant de copies que de valeurs de paramètres nouvellement considérées.

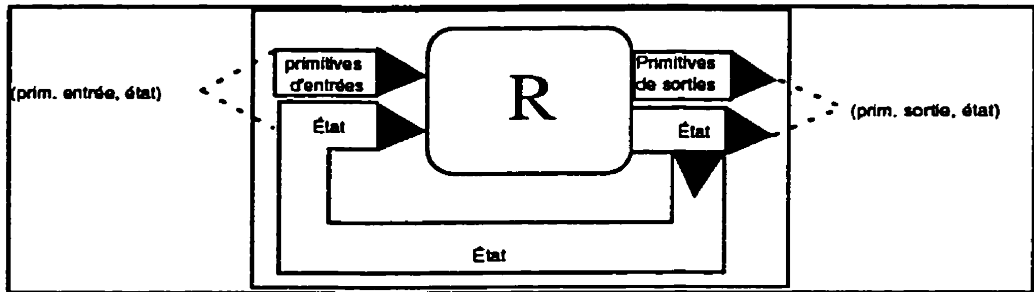


Figure 5.9. Modèle associé aux processus séquentiels.

Pour l'illustration de cette transformation nous prenons l'exemple du service du protocole *INRES* (figure 5.10). Au niveau de l'initiateur, la réponse à l'entrée *ICONreq* est soit *ICONconf* (figures 5.6 et 5.7) si la connexion est établie, soit *IDISind* si la connexion n'arrive pas à être établie. Cette situation est indéterministe, pour la résoudre on doit d'abord détecter les relations fautives. Pour ce faire, on devrait voir ce qui se passe à un niveau plus bas et plus particulièrement au niveau protocole. La figure 5.11.a. illustre ceci; elle représente par une composition de relations le service offert. Sur cette figure, les primitives en gras représentent les primitives observables à travers les interfaces de l'initiateur et du répondeur.

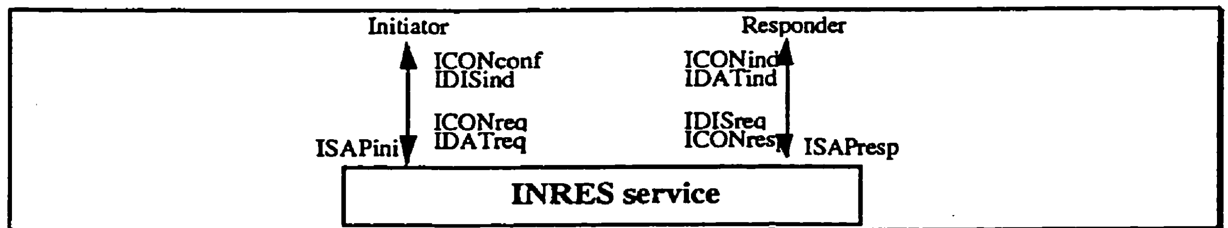


Figure 5.10. Le service d'*INRES*.

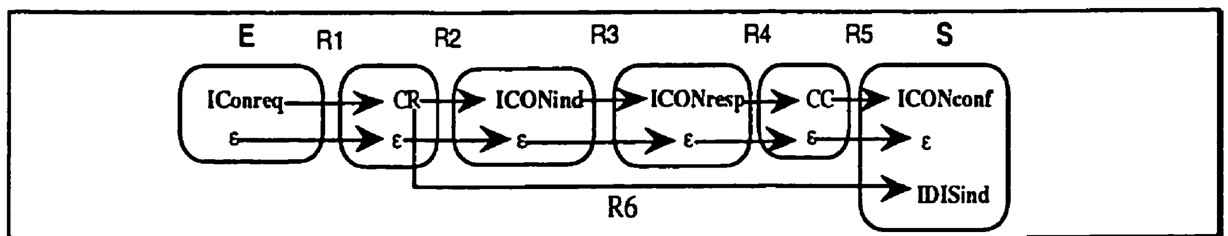


Figure 5.11.a. La relation décomposée d'*INRES*.

L'analyse du système composé montre que l'indéterminisme est dû à la sous-relation *R*2 (figure 5.11.a). Cet indéterminisme est dû à un manque d'information ; pour y remédier, il faut adopter un niveau d'abstraction plus bas. On doit déployer l'état (si cela est possible) en prenant en considération deux nouvelles variables d'état *C* et *T* qui représentent respectivement le *compteur* et le *temporisateur* (figure 5.11.b). Pour la variable *T*, il n'est pas nécessaire de connaître toutes ces valeurs. Il suffit de la prendre comme une variable booléenne initialisée à *faux* et qui prend la valeur *vrai* quand il y a un *timeout*.

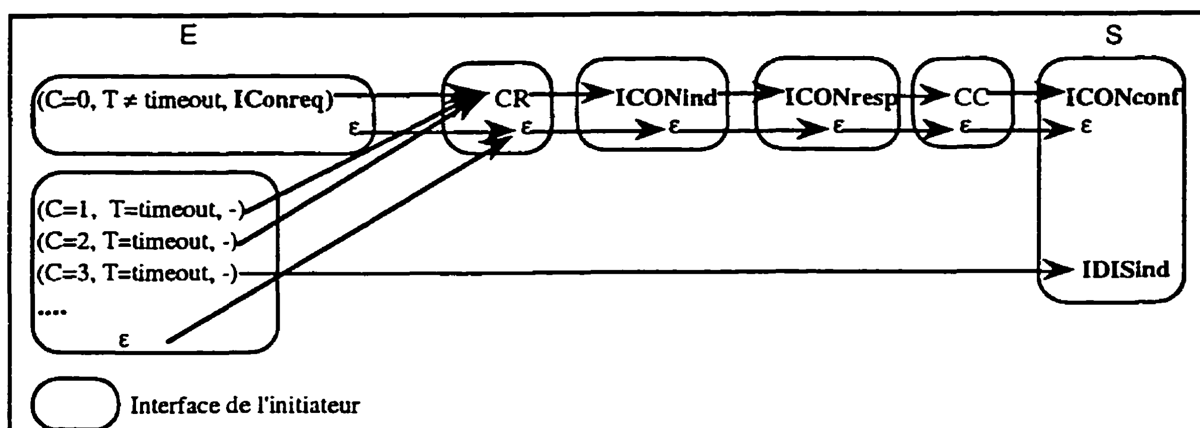


Figure 5.11.b. Considération de nouvelles variables d'état.

Pour mieux comprendre ce qui se passe, reprenons le mécanisme d'établissement de la connexion de l'initiateur. Lorsque l'initiateur reçoit une requête de connexion *ICONreq* de son utilisateur, il procède comme suit :

```
Début
C<--0;
Fanion:
Si C< 3,
  il envoi un PDU du type CR vers le répondeur,
  déclenche(T), C<-- C+1,
  Si il reçoit un CC à un temps t<T
     il envoi un ICONconf à son utilisateur et passe à la phase de transfert de données,
  Sinon Si il reçoit un timeout
     il revient à Fanion
  FinSi
Sinon
  il envoi IDISind à son utilisateur
FinSi
Fin
```

Cette transformation ne résout pas l'indéterminisme, car l'initiateur n'a pas de moyens pour contrôler et observer les valeurs du compteur et du temporisateur. Pour résoudre cette situation, on doit rendre la valeur du compteur et du temporisateur observables à partir de l'interface de l'initiateur. Du point de vue relationnel, cela consiste à étendre le domaine de la relation globale de l'initiateur. Cette extension consiste en :

- rajouter à chaque primitive d'entrée une description de l'état, c'est-à-dire la valeur de certaines variables d'état,

- rajouter de nouvelles entrées constituées de la description de l'état et de la primitive nulle (aucune entrée). Ces nouvelles entrées permettent de prendre en charge les différentes branches de l'indéterminisme.

Cette opération nécessite l'introduction de points d'observation *PO* (*Point of Observation*) [Dsso 91a]. La figure 5.11.c, représente la version finale d'*INRES* après avoir rendu le compteur et le temporisateur observables. Le déterminisme *D*(*R*) est amélioré de la valeur 0 à 1.

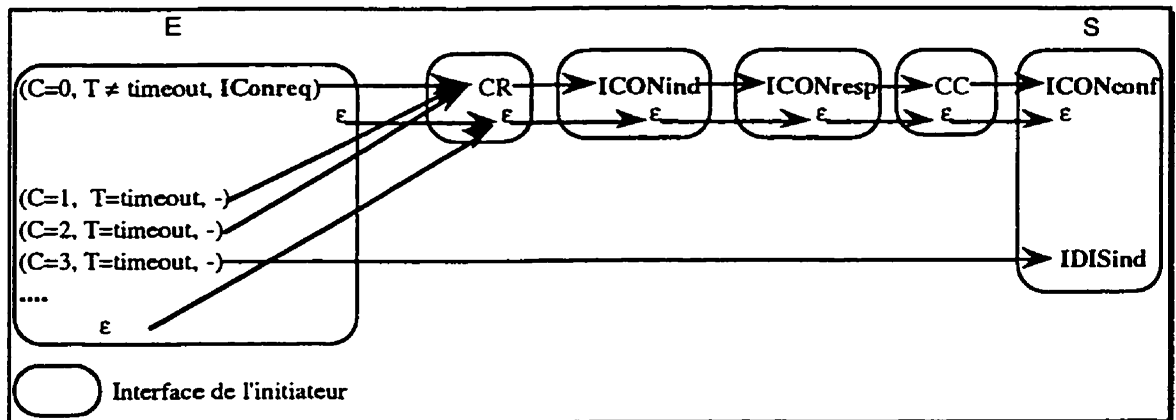


Figure 5.11.c. La composition de relations après avoir rendu quelques variables observables.

5.6.2. Transformation permettant d'améliorer la convergence

Cette transformation ressemble à celle du déterminisme. Elle est due a un manque d'information des sorties du module. Cette transformation nécessite l'introduction de plus d'information sur les sorties de la spécification. Elle décompose les sorties en prenant en considération plus de paramètres dans sa description. Pour illustrer cette transformation,

nous reprenons l'exemple de la figure 5.11.c. Nous remarquons qu'il y a une situation de convergence due au fait qu'un ensemble d'entrées produisent un *CR* comme sortie. Cette situation peut être évitée de la même façon que pour l'indéterminisme, c'est-à-dire en considérant si cela est possible le compteur et le temporisateur dans la description des sorties. Ceci produit une nouvelle relation représentée dans la figure 5.12. Cette solution décale le problème de convergence vers une autre sous-relation. La nouvelle situation est résolue de la même manière.

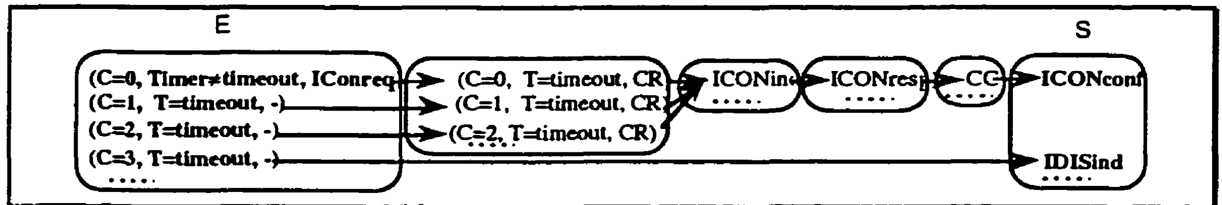


Figure 5.12. Considération de nouvelles variables pour résoudre la convergence.

5.6.3. Transformation permettant d'améliorer le degré de définition

Cette transformation consiste à compléter la spécification de façon à associer dans la mesure du possible à chaque entrée non définie une sortie. Le choix de la sortie est important, il doit préserver les autres propriétés de la spécification du service et particulièrement la convergence. Si cela est possible, nous choisirons une nouvelle sortie ou encore mieux des sorties non informées. Tout ceci doit tenir compte de la sémantique des opérations du protocole. Cette transformation peut améliorer en même temps le degré de définition et le degré d'information.

Pour illustrer cette transformation, on peut reprendre l'exemple des protocoles de contrôle d'erreur du type *selective repeat* ou *go-back N* [Hals 92]. Pour ce genre de protocoles, l'acquittement de la réception de trames de données n'est pas immédiate (figure 5.5). On peut envoyer autant de trames de données que la taille de la fenêtre coulissante le permet sans pour autant recevoir d'acquittement. Si on modélise ce service par une relation on obtient un ensemble d'entrées non définies et des sorties non informées. La relation R_A de la figure 5.5, illustre de tels cas. Par exemple la trame *I(N*+1) est non définie et l'acquittement négatif *NACK(N*+1) est une sortie non informée. Dans ce cas, l'idée d'associer une nouvelle sortie (resp. entrée) à l'entrée *I(N*+1) (resp. sortie *NACK(N*+1)) n'est pas faisable, puisqu'en réalité la sortie existe (*NACK(N*+1)) mais elle n'est pas obtenue directement après l'entrée *I(N*+1). Pour corriger une telle

situation, si cela est possible, nous associons l'entrée $I(N+1)$ à la sortie $NACK(N+1)$ et $I(N+2)$ à la sortie $ACK(N+2)$. Ceci donne lieu à une nouvelle spécification de service modélisée par la relation R'_A où il y a correspondance directe entre les entrées et les sorties.

L'application de ces changements sur l'exemple de la figure 5.5, n'est pas sans implications. Cette transformation change le protocole à répétition sélective en un protocole où l'attente d'acquittement est obligatoire avant l'envoi du message suivant (*Stop and Wait*) (voir figure 5.13). Dans un certain contexte d'utilisation, cette transformation produit un protocole qui est moins performant, mais qui a l'avantage d'être plus testable. En effet, le protocole *Stop and Wait* nécessite la vérification d'un seul cas, par contre, le protocole à répétition sélective est plus difficile à tester, il nécessite la vérification de tous les cas d'ordonnancement entre données et acquittements (si l'ordonnancement est important).

Nous avons introduit l'exemple précédent exprès car il nous permet de montrer que dans certains cas la testabilité d'un logiciel peut avoir un impact sur sa performance. La testabilité n'est en fait qu'un critère de qualité parmi d'autres qu'il faut prendre en considération lors de la phase de conception. Ces critères sont quelques fois incompatibles et il est difficile d'agir sur un critère tout en préservant les autres [Perr 87].

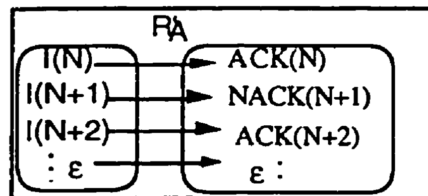


Figure 5.13. Extension de la Relation R_A de la figure 5.5.

5.6.4. Transformation permettant d'améliorer le degré d'information

Cette transformation est assez similaire au degré de définition. On analysera les causes de cette convergence et si c'est possible on complétera la relation inverse (R^{-1}). On associera à chaque sortie non informée une entrée. Le choix de l'entrée doit préserver la propriété du déterminisme. Il est préférable de choisir une entrée non définie qui améliore en même temps le degré d'information et le degré de définition. Tout ceci doit tenir compte de la sémantique des opérations du protocole. L'exemple de la figure 5.13 reste

valide puisque cette transformation améliore aussi bien le degré de définition que le degré d'information.

5.6.5. Composition de transformations

L'objectif de la *DFT* est de concevoir un logiciel testable. Cet objectif peut être atteint en plusieurs étapes. Le concepteur commence par produire une première version de la spécification, analyse sa testabilité et la classe selon les valeurs de ses facteurs de testabilité (voir table 5.1). Si la relation associée appartient à la classe des relations testables, le concepteur peut alors passer à l'étape suivante soit la validation et l'implantation. Sinon, il peut (dans la mesure du possible) raffiner la spécification par les transformations précédemment décrites pour améliorer la classe d'appartenance de la spécification. Le processus de transformations peut continuer jusqu'à obtenir une testabilité acceptable ou lorsque aucune transformation n'est encore applicable (figure 4.1). Les transformations décrites ci-dessus, si elles sont applicables, peuvent être composées de différentes façons. Le choix des transformations à appliquer et leur ordre d'application dépendent de la classe de la spécification initiale du service, de la liberté du concepteur de modifier le service et enfin du degré de testabilité escompté. Le graphe de la figure 5.14 représente un sous-ensemble des séquences de transformations permettant d'améliorer la classe de testabilité de la spécification du service. On a choisi de ne représenter qu'un sous-ensemble de séquences car les autres séquences de transformations aboutissent à des classes de relations que nous n'avons pas considéré dans notre classification (voir section 5.5). Les noeuds de ce graphe représentent les neuf classes de spécifications considérés. Les arcs orientés étiquetés représentent les transformations à appliquer pour passer d'une certaine classe de spécification à une classe plus testable. Par exemple, pour passer de la classe des fonctions à la classe des spécifications contrôlables, il faut appliquer la transformation permettant d'améliorer le degré d'information (figure 5.14).

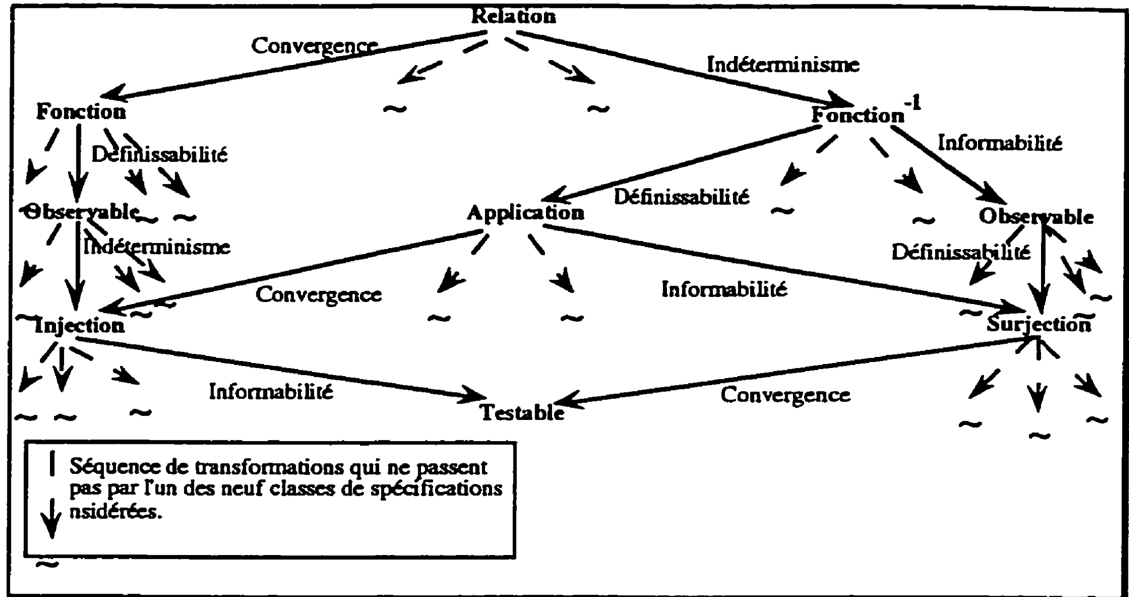


Figure 5.14. Passage d'une classe de testabilité à une autre.

5.6.6. Application aux spécifications de protocoles

Dans les sections précédantes, nous avons proposé une nouvelle approche d'analyse, d'évaluation et d'amélioration de la testabilité des spécifications de service des protocoles de communication. Cette approche n'est pas restreinte aux spécifications de service, elle peut être utilisée pour étudier la testabilité des spécifications de protocoles de communication. Pour illustrer ceci, nous allons appliquer notre approche sur une spécification simplifiée du protocole X.25.

5.7. Application au protocole *X.25*

Comme application de l'analyse de testabilité ci-dessus présentée, nous avons pris l'exemple du protocole *X.25*. Pour une meilleur illustration de l'approche que nous proposons, nous avons simplifié ce protocole. Dans ce qui suit, nous allons présenter les quelques restrictions adoptées pour simplifier *X.25* (niveau paquets). Nous décrirons le comportement de la *DTE* sous les hypothèses suivantes :

- Le *NetworkManager* n'est pas considéré,
- On ne considère que des connexions simples. Avant la déconnexion, l'utilisateur ne peut pas ouvrir une deuxième connexion,
- *N_CONNECT_ACK* est utilisé pour confirmer l'acceptation de la connexion,

- Après qu'une primitive du type requête est envoyée (à l'interface utilisateur de la *DTE*), l'utilisateur ne peut envoyer d'autres primitives que lorsqu'il reçoive une primitive d'indication ou de confirmation.

5.7.1. Transformations préliminaires

Le protocole de la deuxième couche de *X.25* est appelé *LAPB*. Dans ce dernier, le contrôle d'erreur y est fait par retransmission continue du type *Go-Back-N.* Comme nous l'avons précisé dans la section 5.6.2 et 5.6.3, ce genre de retransmission induit beaucoup d'entrées non définies et de sorties non informées. Pour éviter cette situation nous avons assumé que les paquets de données ne sont envoyés qu'après la réception de l'accusé de réception du dernier paquet de données. Nous avons aussi considéré que le flux de données n'est que dans un seul sens. Ceci évitera l'indéterminisme causé par le *piggybacking* si le flux est dans les deux sens [Hals 92].

5.7.2. Spécification du service *X.25*

Le protocole *X.25* est un système séquentiel, pour le représenter, nous avons étendu les entrées et les sorties du module en prenant en considération l'état. La relation R que nous avons choisie est la suivante: $R \subseteq (E \cup \varepsilon) \times (S \cup \varepsilon)$ où $E = Q \times I$, $S = Q \times O$, Q est l'ensemble des états, I l'ensemble des entrées (primitives et PDUs) et S l'ensemble des sorties (primitives et PDUs). Sous les hypothèses décrites précédemment, nous avons défini 18 états, 28 primitives d'entrée et 31 de sorties (voir annexe A). Pour simplifier la représentation et le calcul, nous avons décomposé R en sous-relations R_i tel que $R = \cup R_i$. Chaque relation R_i représente le comportement au niveau d'un état particulier q_i. La Figure 5.15 représente l'exemple de la relation associée à l'état 13. Pour plus d'information sur ces relations, nous référons le lecteur à l'annexe A.

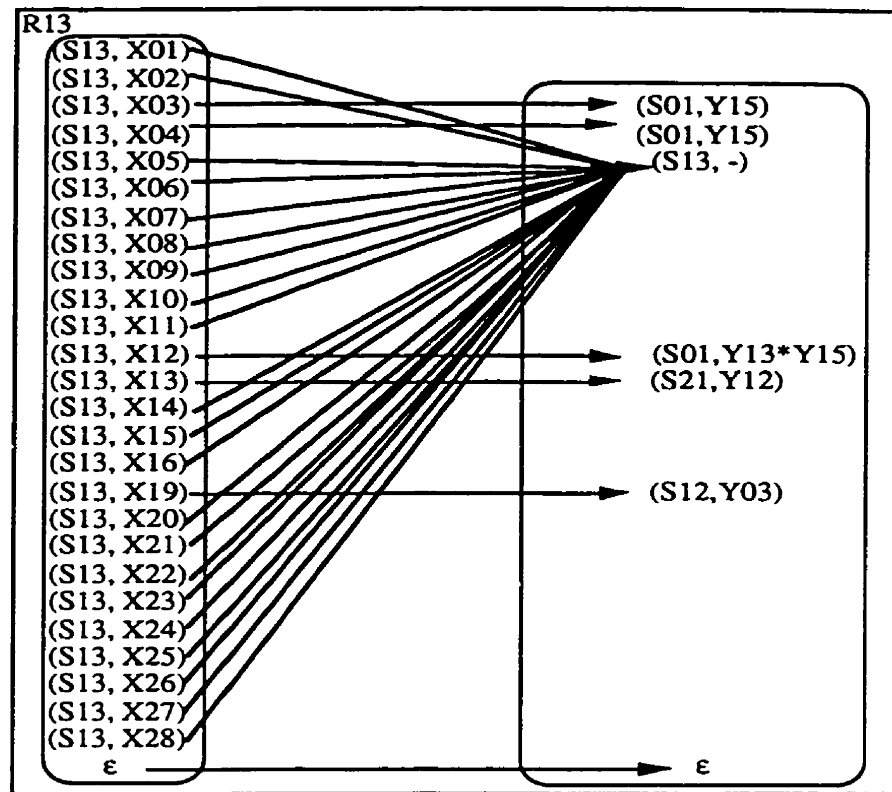


Figure 5.15. La relation associée à l'état 13.

5.7.3. Interprétation des résultats

Si on considère la spécification de *X*.25 sous les hypothèses précédantes, on peut calculer notre vecteur de testabilité : $C(R)$ = <1,.21,.43, 1> et dégager les remarques suivantes :

- le protocole *X*.25 tel qu'il est spécifié est complètement contrôlable puisqu'il est :
  - complètement déterministe $D(R) = 1$,
  - complètement informé $Def(R) = 1$,
- l'observabilité de *X*.25 n'est pas très bonne car :
  - R^{-1} est faiblement déterministe $D(R) = .21$,
  - R n'est pas trop définie $Def(R) = .43$.

Nous avons rassemblé les résultats de l'analyse de testabilité de *X*.25 dans le graphique de la figure 5.16. Ce dernier, représente les facteurs de testabilité des sous-relations R_i composant R (associée à *X*.25). Ce graphique peut aider à identifier exactement dans quel état les facteurs de testabilité sont faibles (surtout l'observabilité).

Pour y remédier, on peut identifier des états qui causent la dégradation de la testabilité, ensuite, suivant la liberté dont nous disposons, nous pouvons appliquer les transformations adéquates.

Par exemple la relation *R*13 associée à l'état 13 (figures 5.15 et 5.16) a un très mauvais degré de définition (*Def*(*R*)=.19) et convergence ($D(R^{-1})$=.12). La cause de cela est que beaucoup de transitions sont des *self-loops* qui acceptent des primitives d'entrée et ne produisent pas de sorties et mènent vers le même état. Cette situation peut être corrigée en associant des sorties à ces entrées (en tenant compte de la sémantique des opérations). Ceci corrigera en même temps le degré de définition et la convergence.

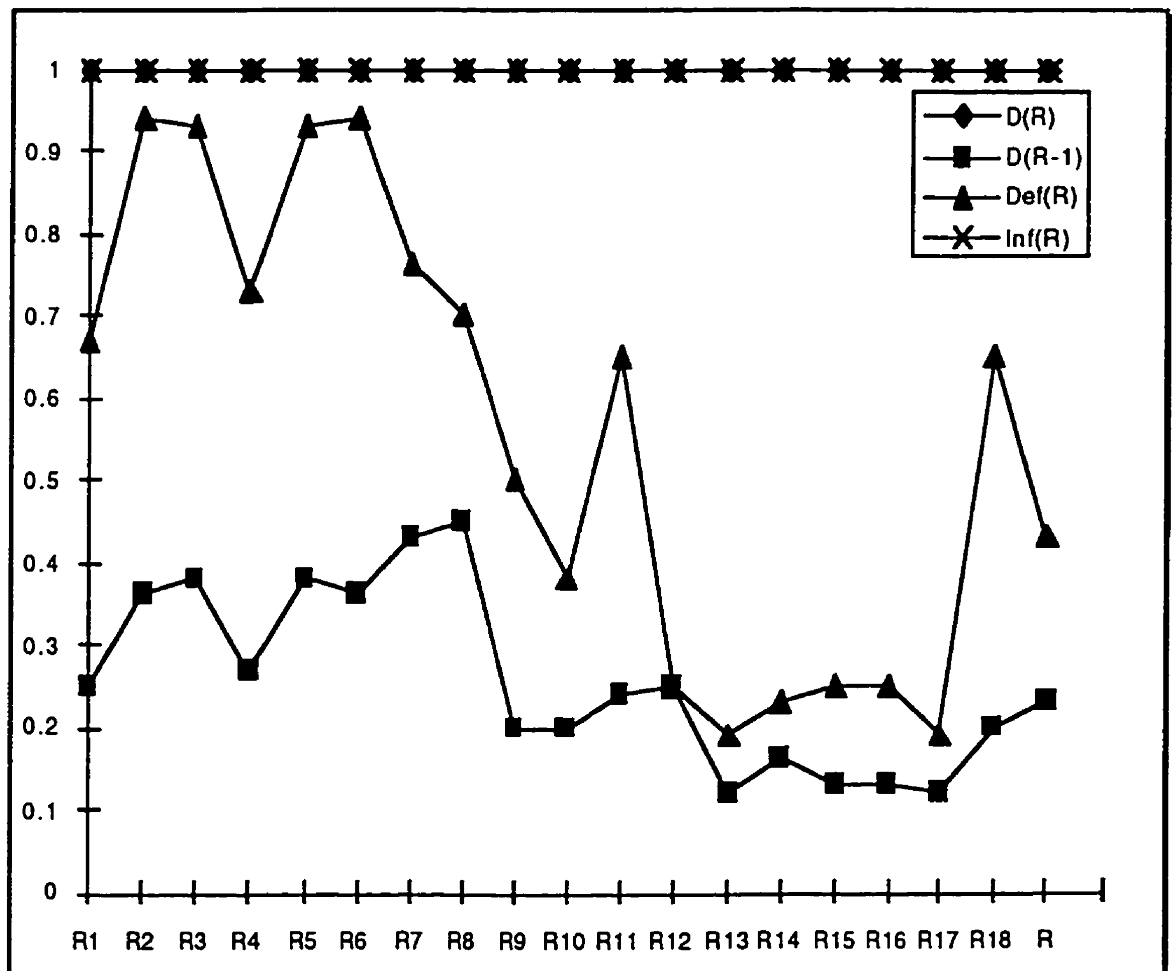


Figure 5.16. Résultats de l'analyse de la testabilité de *X*.25.

5.8. Testabilité d'une composition de modules

Après avoir étudié la testabilité d'un module unique, nous nous tournons maintenant vers la composition de modules. La composition de modules est utile pour la définition d'un système au complet:

Définition 8.1: Un **système de communication** est un ensemble de modules interagissant ensemble pour l'accomplissement de certaines fonctionnalités de communication. Le système reçoit ses entrées et délivre ses sorties à l'environnement. L'ensemble des modules est partitionné en sous-ensembles appelés **sous-systèmes.** Un **sous-système** rassemble un ou plusieurs modules interconnectés ensemble. La façon dont les modules sont rassemblés ensemble en sous-systèmes respecte certains critères appelés **critères de décomposition** [Karo 97d].

Au niveau le plus bas de décomposition, le système est composé de sous-systèmes contenant chacun un seul module. En appliquant un critère de décomposition au système initial, on obtient une autre architecture où chaque sous-système contient des modules liés entre eux par un critère de décomposition. Un système initial S et son environnement Env est modélisé par un graphe orienté $(M \cup \{M_Env\}, E \cup E_Env)$ où M est un ensemble fini de noeuds représentant les k modules du Système et M_Env un noeud spécial représentant l'environnement. E est un ensemble fini d'arcs orientés étiquetés représentant l'ensemble des interactions reliants les différents modules appartenant à M et E_Env est un ensemble fini d'arcs orientés étiquetés représentant l'ensemble des interactions reliants les différents modules de M au noeud représentant l'environnement M_Env. Après l'application d'un critère de décomposition DC_i, le système et son environnement Env sera modélisé par un graphe orienté $(SS_i \cup \{M_Env\}, E_i \cup E_Env)$ où $SS_i = \{ss_{i,1}, ss_{i,2}, \ldots, ss_{i,p}\}$ avec $ss_{i,j} \cap ss_{i,k} = \emptyset$ pour $j \neq k \leq p$, est un ensemble fini de noeuds représentant les sous-systèmes ; SS_i est une partition de l'ensemble M en p parties suivant le critère de décomposition DC_i tel que $1 \leq p \leq k$. $E_i \subseteq E$ est l'ensemble des interactions reliant des modules appartenant à des sous-systèmes différents. On fera abstraction des différents modules d'un même sous-système et des interactions les reliant (voir figure 5.17).

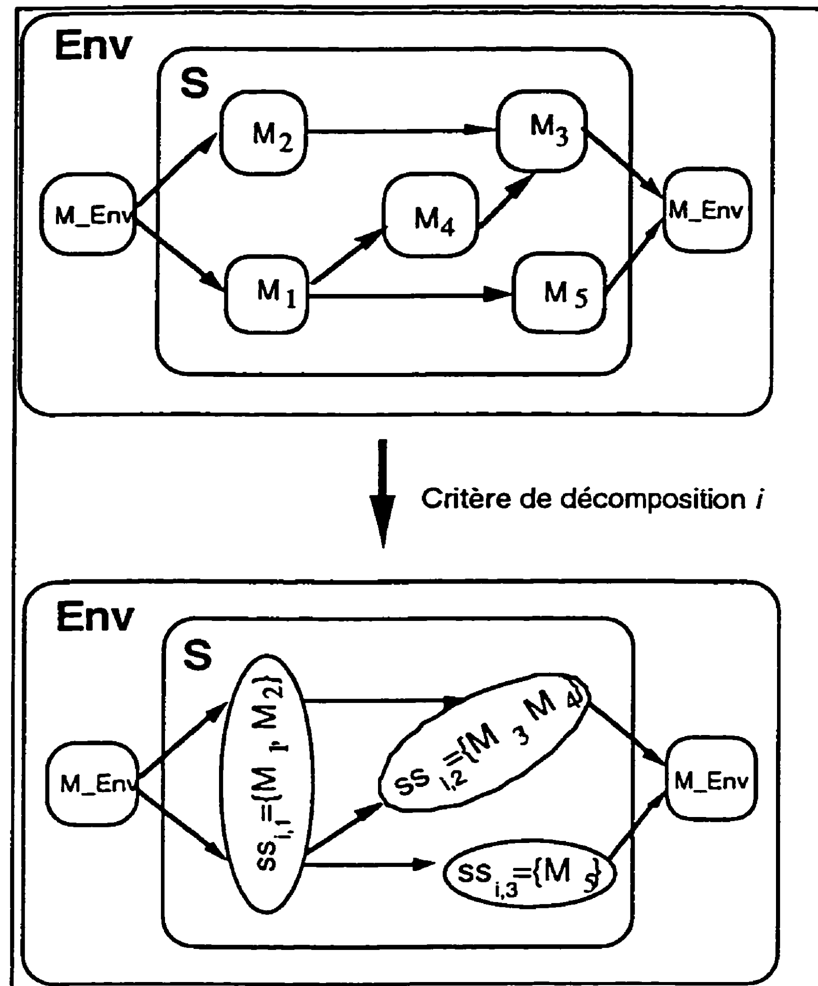


Figure 5.17. Exemple de décomposition de système.

Il existe plusieurs critères de décomposition dont par exemple: le temps d'invocation des modules, les fonctionnalités des modules, la synchronisation des modules, etc. Dans ce qui suit, nous étudierons la testabilité comme critère de décomposition.

5.9. Utilisation de la testabilité comme critère de composition/décomposition

Lors des tests, il est parfois possible d'accéder aux interfaces de certains sous-systèmes (tests boîte grise). Ceci est fait en ajoutant des points d'observation ou de contrôle (*PO* et *PCO*). Il n'est pas nécessaire de mettre des points d'observation et/ou de contrôle entre chaque paire de modules. Pour minimiser le nombre de ces points, on peut

rassembler des modules dont la composition donne une bonne testabilité. Ceci minimisera la redondance d'information entre les points possibles. Dans ce cas, la testabilité est utilisée comme critère de composition/décomposition. Une mauvaise testabilité d'un sous-système peut être due aux propriétés de l'une des sous-relations composant le système. La réciproque n'est pas nécessairement vraie, c'est-à-dire si une sous-relation a de mauvaises propriétés de testabilité cela n'implique pas que la relation globale le sera (voir Proposition 9.1). Ceci car certaines compositions vont cacher la testabilité individuelle des modules.

Proposition 9.1

Soit R la relation globale associée au service d'un sous-système SS.

1- Si une des sous-relations qui composent R est indéterministe ceci n'implique pas nécessairement que R est indéterministe.
2- Si une des sous-relations qui composent R est convergente ceci n'implique pas nécessairement que R est convergente.
3- Si une des sous-relations qui composent R est non complètement définie ceci n'implique pas nécessairement que R est non complètement définie.
4- Si une des sous-relations qui composent R est non complètement informée ceci n'implique pas nécessairement que R est non complètement informée. ■

Preuve.

Pour prouver les quatre propriétés de cette proposition, il suffit de donner des contre-exemples.

Pour la première et la seconde propriétés, nous prenons l'exemple de la relation $R = R_1 * R_2$, où $R_1 = \{(a, b) ; (a, c) ; (\varepsilon, \varepsilon)\}$, $R_2 = \{(b, d) ; (c, d) ; (\varepsilon, \varepsilon)\}$ et donc $R = \{(a, d) ; (\varepsilon, \varepsilon)\}$. On peut voir que R est déterministe et non convergente bien que R_1 est indéterministe et R_2 est convergente. Ceci est due au masquage de la relation (voir section 5.2).

Pour la troisième propriété, nous prenons l'exemple de la relation $R = R_1 * R_2$, où $R_1 = \{(a, b) ; (a, c) ; (\varepsilon, \varepsilon)\}$, $R_2 = \{(b, d) ; (c, d) ; (e, \varepsilon) ; (\varepsilon, \varepsilon)\}$ et donc $R = \{(a, d) ; (\varepsilon, \varepsilon)\}$. R est complètement définie bien que R_2 ne l'est pas.

Pour la troisième propriété, nous prenons l'exemple de la relation $R = R_1 * R_2$, où $R_1 = \{(a, b) ; (a, c) ; (\varepsilon, f) ; (\varepsilon, \varepsilon)\}$, $R_2 = \{(b, d) ; (c, d) ; (\varepsilon, \varepsilon)\}$ et donc $R = \{(a, d); (\varepsilon, \varepsilon)\}$. R est complètement informée bien que R_2 ne l'est pas. ■

Pour pouvoir étudier la testabilité d'une composition de modules, nous allons étudier deux types de composition et leur testabilité respectives.

5.9.1. Testabilité de la composition de modules en série

Définition 9.1.1

Une relation R_k est dite en série avec R_i si R_i fournit des entrées au module R_k. La relation R qui représente la composée (R_k en série avec R_i) est égale à: $R = R_i * R_k$. ■

La figure 5.18 illustre la définition précédente. Les parties a et b de cette figure représentent la composition de deux relations en série.

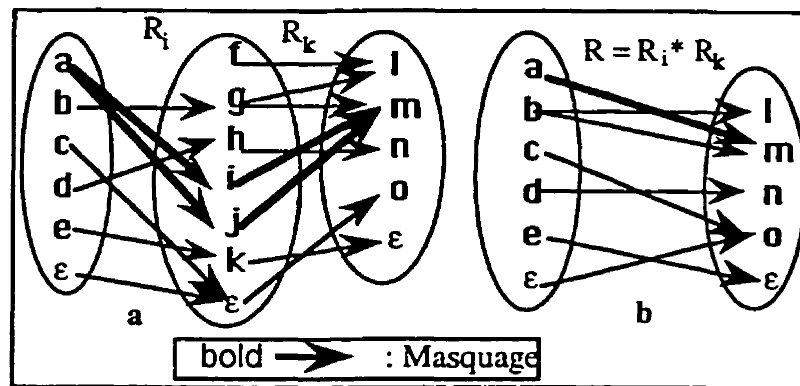


Figure 5.18. Illustration du masquage d'une relation composée.

- **Le degré de définition de la composition de modules en série**

Soit $R = R_1 * R_2 * ... * R_k$ la composition série de k relations, l'ensemble des entrées non définies de R est calculé comme suit: $Nondef(R)=\{e \in Dom(R_1) \mid R_k(R_{k-1}(...*R_1(e)))...) =\{\varepsilon\}\}$. L'ensemble des entrées non définie de R_1 est toujours inclus dans celui de R. Il se peut qu'il existe des entrées initialement définies dans R_1 mais qui ne provoquent pas de sorties observables dans R. Cette propriété peut être utilisée comme guide de la composition des modules. En effet, la composition ne peut pas améliorer le degré de définition des composantes. Ça ne sert à rien de se baser sur le degré de définition pour faire une composition série si le premier module à un degré de définition optimale.

La figure 5.18 illustre une telle propriété, au départ la relation $Nondef(R_i)=\{c\}$, à la suite de la composition, l'ensemble des éléments non définies s'est agrandi $Nondef(R)=\{c, e\}$.

- **Le degré d'information de la composition de modules en série**

L'ensemble des sorties non informées d'une relation $R = R_1 * R_2 * ... * R_k$ est calculé comme suit: $Noninf(R)=\{s \in Im(R_k) \mid R_1^{-1}(R_2^{-1}(... * R_k^{-1}(s)))...) =\{\varepsilon\}\}$. L'ensemble des sorties non informées de R_k est toujours inclus dans celui de R. Un ensemble de sorties initialement informées dans R_1 risquent de ne plus l'être dans R. La composition ne peut produire qu'un degré d'information inférieure ou égale à celui du dernier module de la composition. On ne peut pas utiliser le degré d'information comme critère de composition si le dernier module a un degré d'information optimal.

- **Le déterminisme de la composition de modules en série**

L'un des phénomènes rencontrés lors de la composition série est le masquage de la relation. D'une façon non formelle, on dira qu'il y a masquage dans une relation composée $R=R_i * R_k$, s'il existe au moins une entrée e indéterministe de la relation R_i qui n'est pas indéterministe dans R. Ceci est du au fait que quelque soit l'image s de e par R_i $R_k(s)$. Cette propriété est appelée masquage de la relation. Elle influe sur le degré de testabilité des relations en diminuant le nombre d'entrées indéterministes, c'est-à-dire, le nombre d'entrées indéterministes de la relation globale est inférieur à celui des relations composantes. Une définition formelle du masquage de la relation est donnée dans la section 5.2. La composition série de relations peut donc améliorer le degré du déterminisme.

La figure 5.18 illustre un cas de masquage dû à la composition des relations R_i et R_k. L'entrée a de la relation R_i a deux images i et j, ceci constitue un indéterminisme pour R_i. Cet indéterminisme est masqué puisque les images de m par la relation R_k^{-1} correspondent aux images de a par la relation R_i.

- **La convergence de la composition de modules en série**

L'étude de la convergence de la composition en série de modules ressemble à l'étude du déterminisme. Elle s'applique aux relations inverses des relations utilisées dans la composition. Le masquage des relations inverses peut diminuer le nombre de sorties

convergentes d'une relation. La composition série peut donc améliorer le degré de convergence. La figure 5.18 reste valable pour illustrer la convergence. En effet, la sortie m qui était non informée par R_k, ne l'est plus dans la relation composée R.

5.9.2. Testabilité de la composition de parallèle de modules

Définition 9.2.1

Une relation R_i est dite en parallèle avec R_k s'il existe une troisième relation R_j qui fourni des entrées aux deux relations R_i et R_k. Nous définissons la composition parallèle de relations comme étant l'union de relations ($R_i \parallel R_k$ voir section 5.2). ■

Pour illustrer cette définition, nous présentons dans la figure 5.19 un exemple de composition parallèle $R = R_i \parallel R_k$.

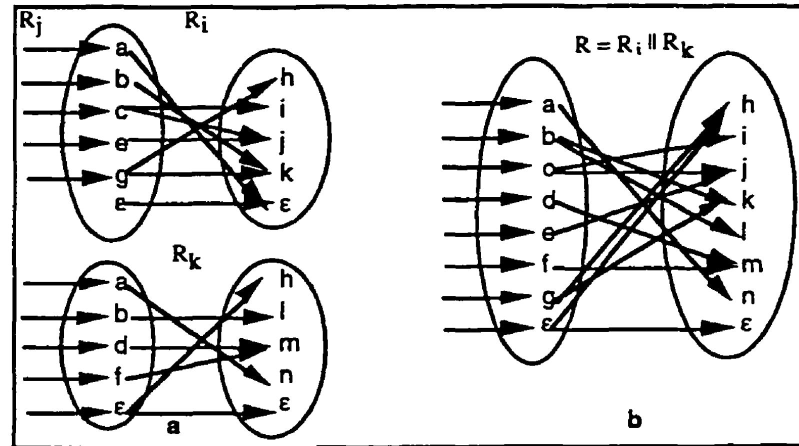


Figure 5.19. $R = R_i \parallel R_k$.

- **Le degré de définition de la composition parallèle de modules**

Soit $R = R_1 \parallel R_2 \parallel ... \parallel R_k$ la composition parallèle de k relations, l'ensemble des entrées non définies de R est calculé comme suit: $Nondef(R)=\{e\in \bigcup_{i=1}^{k} Dom(R_i) \mid R_1(e)=\varepsilon, R_2(e)=\varepsilon ... \text{et } R_k(e)=\varepsilon\}$. L'ensemble des entrées non définie de R est toujours inclus dans $\bigcup_{i=1}^{k} Nondef(R_i)$. En composant parallèlement des relations on peut diminuer le nombre d'entrées non définies, car une entrée non définie d'une des relations composantes peut ne pas l'être dans une autre relation. Cette propriété peut être utilisée comme guide de la

composition de modules. En effet, on composera des relations ensemble si la composée améliore le degré de définition.

La figure 5.19 illustre une telle propriété ; au départ l'ensemble des entrées non définies de R_i est $Nondef(R_i)=\{a\}$, à la suite de la composition il n'y a plus d'entrées non définies.

- **Le degré d'information de la composition parallèle de modules**

L'ensemble des sorties non informées d'une relation $R = R_1 \| R_2 \| ... \| R_k$ est calculé comme suit: $Noninf(R)=\{s \in \bigcup_{i=1}^{k} Im(R_i) \mid R_1^{-1}(s)=\varepsilon \text{ et } R_2^{-1}(s)=\varepsilon ... \text{et } R_k^{-1}(s)=\varepsilon\}$.

L'ensemble des sorties non informée de R est toujours inclus dans $\bigcup_{i=1}^{k} Noninf(R_i)$ La composition parallèle ne peut qu'améliorer le degré d'information des relations composantes. Cette propriété peut être utilisée comme guide de la composition de modules en parallèle. Les relations sont composées si la composée améliore le degré de définition.

La figure 5.19 reste valable pour cette propriété. L'ensemble des sorties non informées de R_k est $Noninf(R_k)=\{h\}$, par contre la composée n'admet plus de sorties non informées.

- **Le déterminisme de la composition parallèle de modules**

L'ensemble des entrées indéterministes d'une relation $R = R_1 \| R_2 \| ... \| R_k$ est inclus dans la réunion de tous les ensembles des entrées indéterministes des relations composantes. En effet, à part les entrées indéterministes des relations composantes, d'autres entrées indéterministes peuvent se rajouter si: il existe au moins une entrée appartenant au domaine de plusieurs relations composantes dont l'image par ces relations est différentes. Un exemple d'une telle situation est donné dans la figure 5.19, l'entrée b, originalement déterministe dans R_i et R_k, devient non-déterministe dans R. La composition parallèle influe donc négativement sur l'indéterminisme.

- **La convergence de la composition parallèle de modules**

Comme pour le cas du déterminisme, si une sortie est convergente dans l'une des relations composante, elle le restera dans $R= R_1 || R_2 . || ... || R_k$. En plus d'autres sorties convergentes peuvent se rajouter. Ceci est le cas lorsqu'une sortie est l'image de deux ou plusieurs entrées appartenant à plusieurs relations composantes. La composition parallèle influe donc négativement sur la convergence.

5.9.3. Influence des facteurs de testabilité sur la composition de modules

Dans cette section nous allons récapituler dans une table unique (voir table 5.2) les résultats des sections 5.9.1 et 5.9.2. Cette table montre l'influence du type de composition sur les facteurs de testabilité.

	Composition série	Composition parallèle
Deg. de Déf.	-	+
Degré d'inform.	-	+
Déterminisme	+	-
Convergence	+	-
(-) influence négative, (+) influence positive.		

Table 5.2. Influence de la composition sur la testabilité.

Les informations contenues dans cette table peuvent être utilisées par un algorithme de composition/décomposition de systèmes afin de donner les points d'observation et de contrôle. Des algorithmes comme celui présenté dans [Raz 93], peuvent être adaptés pour accomplir cette tâche.

5.10. Conclusion

Dans ce chapitre, nous avons proposé une approche pour l'analyse de la testabilité basée sur le modèle relationnel. Nous avons dégagé quatre facteurs qui influencent la testabilité. Comparativement à la méthode d'évaluation de la testabilité des protocoles de communication basée sur la longueur de la suite de test [Petr 93a], notre méthode n'exige pas d'hypothèses sur la spécification. Elle permet d'identifier les causes d'une mauvaise testabilité grâce à l'analyse du vecteur de testabilité $C(R)$: $C(R)=< D(R), D(R^{-1}), Def(R),$

Inf(*R*)>. Il est cependant important de noter que la granularité de notre analyse peut différer comparativement à la méthode de Petrenko. La précision de notre métrique dépend aussi du niveau d'abstraction auquel nous nous sommes intéressés.

Nous avons proposé une classification des spécifications basée sur les quatre facteurs de testabilité. Nous nous sommes basés sur cette classification pour utiliser des méthodes de raffinement de la spécification en vue d'améliorer sa testabilité.

Enfin, nous avons fait le même travail d'analyse et d'évaluation de la testabilité pour la composition de modules.

Chapitre 6

Étude de la testabilité basée sur le modèle des automates à états finis[1]

6.1. Introduction

Les automates sont des modèles mathématiques adaptés à la spécification de systèmes réactifs à un niveau d'abstraction satisfaisant. Historiquement, les automates ont été beaucoup utilisés comme modèles pour l'étude des propriétés des circuits électroniques dont les sorties sont synchrones avec les entrées [Gill 62], [Star 72] et [Koha 78]. Cependant, depuis les années 70 plusieurs travaux sur la spécification des protocoles de communication se sont basés sur ce modèle [Boch 90, Petr 92, Luo 93, etc.]. Plus récemment, ce modèle a été utilisé dans le domaine des tests des protocoles de communication [Chow 78], [Sidh 89], [Fuji 91a] et [Turn 93b].

Deux modèles d'automates particuliers ont été largement utilisés dans le domaine des tests de conformité des protocoles: les *FSMs* (*Finate State Machine*) et les systèmes de transitions étiquetées. Ce sont des modèles utilisés soit directement pour la spécification de protocoles, soit indirectement pour la description de la sémantique d'un certain nombre de langages de spécification, comme *SDL* [Beli 89], [Sari 93], etc., *ESTELLE* [Budk 86] et [ISO9074 89], *CCS* [Miln 80], *CSP* [Hoar 85] ou *LOTOS* [ISO8807 89]. Les automates permettent de représenter aussi bien le comportement d'un module isolé, qu'un ensemble de modules communicants, décrits dans ces langages. Ils sont également utilisés dans des travaux sur les tests [Brin 88], [Brin 89], [Yao 93b] et [Tan 95].

---

[1] Les résultats de ce chapitre sont publiés dans: IEEE GLOBECOM'96, Londres 96 [Karo 96a], IFIP IWPTS'95, France 1995 [Yevt 95].

Au cours de ce chapitre, nous ne nous intéresserons qu'au modèle des *FSMs*. Nous commencerons par rappeler les définitions et les principales propriétés de ce modèle. Nous poursuivrons le chapitre avec une étude détaillée de la testabilité fondée sur ce modèle. Cette étude comporte une analyse des facteurs de testabilité, une évaluation de ces facteurs et enfin des transformations qui lorsqu'elles sont applicables permettent d'améliorer la testabilité de tels systèmes.

6.2. Modèle des machines à états finis avec entrées-sorties

Une machine à états finis M , est définie par le quintuplé $M=(Q, I, O, h, D_M)$, où Q est un ensemble fini d'états incluant l'état initial q_o, I un ensemble fini de symboles d'entrée, O un ensemble fini de symboles de sorties, h: $D_M \subseteq Q \times I \rightarrow \mathbb{P}(Q \times O)$ est la fonction de comportement, où $\mathbb{P}(Q \times O)$ est l'ensemble des parties de $Q \times O$ et D_M est le domaine de M ou tout simplement l'ensemble des transitions définies de M.

Une séquence $i_1 ... i_k \in I^*$ est dite *acceptable* à l'état q, s'il existe k états $q_1,...,q_k$ de Q tel que $h(q, i_1) = (q_1, o_2)$ et $h(q_j, i_{j+1}) = (q_{j+1}, o_{j+2})$, j = 1, ..., k-1. $I_q{}^*$ est l'ensemble de toutes les séquences acceptables en q et $I_M{}^*$ est l'ensemble de toutes les séquences acceptables par l'automate M en son état initial. Nous pouvons étendre la fonction de comportement h sur $D_M = \bigcup_{q \in Q} (q \times I_q^*)$, on gardera le même nom pour le prolongement, c'est-à-dire, h: $D_M \rightarrow \mathbb{P}(Q \times O^*)$.

On note h^1, h^2 la première, la seconde projection de h. La fonction h^1 est dite la fonction prochain état, alors que h^2 est dite la fonction de sortie de M,
$h^1(q, \alpha) = \{ q' \mid \exists \ \gamma \in O^* \ (q', \gamma) \in h(q, \alpha)\}$, $h^2(q, \alpha) = \{ \gamma \mid \exists \ q' \in Q \ (q', \gamma) \in h(q, \alpha)\}$.

Une machine M est dite complètement spécifiée (*CFSM*) si $D_M = Q \times I^*$. La machine M est dite *déterministe* (*DFSM*) si $|h(q, \alpha)| = 1$ pour tout $(q, \alpha) \in D_M$.

Une *FSM* $M' = (Q', I, O, h', D_{M'})$ est appelée une *sous-machine* de M si : $Q'' \subseteq Q$ et $h'(q, i) \subseteq h(q, i)$ pour tout $(q, i) \in D_{M'}$.

Pour une *FSM* M, deux états q_i et q_j sont dits équivalents si $Iq_i{}^* = Iq_j{}^*$ et $\forall \ \omega \in I_M^*$, $h^2(q_i, \omega) = h^2(q_j, \omega)$, sinon q_i et q_j sont dits *distingables*. Si tous les états de M sont distingables deux à deux alors M est dite *réduite*. Cette notion peut être généralisée aux

machines indéterministes (*CNFSMs*), deux machines $M=(Q, I, O, h, D_M)$ et $M' = (Q', I, O, h', D_{M'})$ sont dites ***équivalentes***, si leur états initiaux sont équivalents: $\forall \alpha \in I^*$ $(h^2(q_o, \alpha) = h'^2(q_o', \alpha))$. La relation d'***équivalence*** entre *FSMs* est parfois appelée *équivalence de traces*. Les traces d'une machine sont les séquences entrée/sortie telles que si l'on applique la séquence d'entrée I à partir de l'état initial, la machine génère la séquence de sortie O. Des machines équivalentes se comportent de façon identique, c'est-à-dire elles exécutent les mêmes traces.

Pour une machine M, deux états q_i et q_j sont dits *compatibles* si et seulement si $\forall\ \omega \in Iq_i^* \cap Iq_j^*$, $h^2(q_i, \omega) = h^2(q_j, \omega)$ et $h^1(q_i, \omega)$ et $h^1(q_j, \omega)$ sont *compatibles*.

Nous dirons que l'état q_i de M_1 couvre l'état q_j de M_2 si et seulement si quelque soit la séquence applicable à q_j elle est aussi applicable à q_i, et l'application aux deux machines M_1 et M_2 (quand ils sont dans leur états q_i et q_j respectifs), produit des séquences de sorties identiques chaque fois que les sorties de M_2 sont spécifiés. Une machine M_1 couvre une machine M_2 si et seulement si, quelque soit l'état q_j de M_2, il existe un état correspondant q_i de M_1 tel que q_i couvre q_j.

Un FSM $M'=(Q, I', O, h', D'_M)$ est une projection de $M = (Q, I, O, h, D_M)$ sur $I' \subseteq I$ si $D'_M \subseteq D_M$ et $h' \subseteq h$, M' est dite projection unitaire de M si M' est une projection de M et $|I'| = 1$, on note $M_1, M_2, ..., M_p$ les projections unitaires de M sur $i_1, i_2, ..., i_p$ respectivement.

6.3. Les tests de conformité pour les machines à états finis

Les machines à états finis ont été largement utilisées dans le test de conformité des protocoles de communication pour représenter le comportement du système sous test considéré comme une boîte noire qui interagit avec son environnement.

6.3.1. Le modèle des fautes

Soit S une spécification et soit $\mathfrak{I}$ l'ensemble de toutes les machines à états finis. Le but du test est de vérifier qu'une certaine relation de conformité *conf* est vraie pour une implantation $Imp \in \mathfrak{I}$ et S. La spécification S et la relation de conformité *conf* partitionnent l'ensemble $\mathfrak{I}$ en deux sous ensembles, le premier sous ensemble est celui des implantations conformes à S alors que le second est celui des implantations qui ne sont pas conforme à S. Ce dernier s'appelle l'ensemble des mutants de S ou l'univers des

fautes. Les modèles de fautes introduit des classes d'équivalences de fautes connues afin de réduire l'ensemble des tests et surtout pour savoir ce qui a été testé. À partir d'une classe d'équivalence de fautes particulière on peut isoler un sous ensemble de mutants de la spécification S noté $\mathfrak{F}(S)$: $\mathfrak{I} \longrightarrow \mathbb{P}(\mathfrak{I})$. $\mathfrak{F}(S)$ représente un sous-ensemble des implantations non-conformes à S, il est aussi appelé le domaine de fautes de S. Du point de vue des tests, les modèles de fautes limitent le genre de fautes à détecter et ils diminuent donc le nombre de cas de tests. Dans le domaine des *FSMs*, on utilise les approches suivantes pour l'obtention de domaines de fautes finis :

- énumération explicite de tous les mutants possibles [Poag 64],
- fautes de sortie seulement [Nait 81],
- Fautes groupées spécifiées par une fonction de fautes [Petr 92],
- limitation du nombre d'états dans l'implantation [Gill 62].

Cette dernière approche est la plus utilisée. Nous considérons une spécification S ayant n états et l'on prend $\mathfrak{F}(S)=\mathfrak{I}_m$, où $\mathfrak{I}_m$ est l'ensemble de toutes les implantations ayant un nombre d'états inférieur ou égal à m. Pour réussir une bonne couverture de fautes (voir chapitre 2) avec une suite de tests minimale, le nombre d'états de l'implémentation ne doit pas dépasser de beaucoup celui de sa spécification de référence. En effet, quelque soit le type de spécification utilisé (déterministe ou non, réduite ou pas, etc.) toutes les formules qui décrivent la longueur de la suite de tests dépendent de m (voir sections 6.3.2, 6.3.3, etc.). Une suite de test *TS* est dite complète, si pour chaque implémentation I non conforme à sa spécification de référence S, il existe un cas de test appartenant à *TS* qui peut détecter ce mutant.

6.3.2. Machine déterministe complètement spécifiée

- **Relation de conformité**

Les machines déterministes complètement spécifiée (*CDFSM*) constituent une classe restreinte. La dérivation des tests de conformité à partir d'une spécification *CDFSM* est basée sur la relation d'équivalence.

Soit deux FSMs complètement spécifiés $A=(Q, I, O, h, q_0)$ et $B=(T, I, O, H, t_0)$. La relation d'*équivalence* entre deux états q de A et t de B, notée $q \cong t$ est vraie, si et

seulement si : $\forall \alpha \in I^*$ ($h^2(q, \alpha) = H^2(t, \alpha)$). Les FSMs A et B sont équivalentes ($A \cong B$) si leurs états initiaux sont équivalents.

Si A est une spécification *CDFSM* et B est une implantation *CDFSM* équivalente à A, alors cette implantation est conforme à la spécification et "$\cong$" est la relation de conformité la plus fine qu'on puisse avoir pour le modèle des machines déterministes complètement spécifiées. Chaque implantation *CDFSM* qui est distingable de la spécification *CDFSM*, représente une implantation non-conforme.

- **Bornes supérieures pour une suite de test complète d'une CDFSM**

Les suites de test complètes pour une spécification *CDFSM* donnée sont faciles à dériver si l'on ne s'intéresse pas à les optimiser. Deux *CDFSMs* réduits avec m et n états peuvent être distinguées par une séquence d'entrées de longueur $l \leq (m + n - 1)$ [Gill 62]. L'ensemble I^{m+n+1} de toutes les séquences d'entrées de longueur $(m + n - 1)$ est une suite de test complète pour toute *CDFSM* ayant n états et I pour alphabet d'entrée et toute implantation *CDFSM* ayant m états et le même alphabet d'entrée. Cet ensemble peut être réduit en considérant l'ensemble $VI^{m-n+k+1}$, où V est l'ensemble d'accessibilité [Vasi 73], [Chow 78] et [Fuji 91a] et k un paramètre qui donne la longueur maximale des séquences d'entrées nécessaires pour séparer tous les états distingables [Yevt 89].

Pour tout *CDFSM* réduit nous avons: $1 \leq k \leq n-1$ [Gill 62]. La longueur maximale d'un élément de V est inférieure ou égale à $n-1$. Dans le pire cas, la longueur totale d'une suite de test est inférieure ou égale à mn^2p^{m-n+1}, où $p=|I|$ [Vasi 73]. Les bornes supérieures pour une suite de test complète présentées précédemment supposent l'existence de la fonction *reset* dans l'implantation à tester ou l'*IUT* (*Implementation Under Test*).

6.3.3. Machine déterministe partiellement spécifiée

- **Relation de conformité**

Étant donnée $A = (Q, I, O, \delta, \lambda, q_0)$, une machine déterministe partiellement spécifiée (*PDFSM*) avec un domaine de spécification D_A et $B=(T, I, O, \Delta, \Lambda, t_0)$ un *PDFSM* arbitraire, B est dite *quasi-équivalente* à A sur le domaine D_A, noté $B \equiv_{D_A} A$, si $\forall \alpha \in I^*_A$ $(\Delta(t_0,\alpha) = \lambda(q_0,\alpha))$. B est dite distingable de A, et nous le notons $B \not\equiv_{D_A} A$, si elle n'est pas quasi-équivalente à A.

La dérivation d'une suite de test complète pour cette classe de machines est plus compliquées, car la machine n'est pas nécessairement réduite [Yevt 89], [Yevt 90a] et [Petr 91].

Deux états q_i et q_j d'une *PDFSM* $A = (Q, I, O, \delta, \lambda, q_0)$ sont dits distingables si :

$$\exists \alpha \in I_i^* \cap I_j^* \quad (\lambda(q_i, \alpha) \neq \lambda(q_j, \alpha)),$$

où I_i^* (respectivement I_j^*) est l'ensemble de toutes les séquences d'entrées acceptées par l'état q_i (respectivement q_j). Sinon ils sont compatibles [Koha 78]. *A* est réduite si tous ses états sont deux à deux distingables, sinon elle est non-réduite. Si on regroupe des états compatibles dans un seul état ceci pourra entraîner l'apparition de nouvelles traces non présentes dans la *PDFSM* initiale. Il ne faut pas réduire la *PDFSM* pour éviter qu'une implantation conforme ne puisse passer les tests.

Machine déterministe partiellement spécifiée	quasi équivalence
Machine déterministe complètement spécifiée	équivalence

Table 6.1 : Relation de conformité pour des machines déterministes

- **Bornes supérieures pour une suite de test complète pour une PDFSM**

Les *PDFSM*'s peuvent nécessiter des séquences de test plus longues afin d'obtenir une suite de test complète. Une telle séquence peut atteindre une longueur égale à $m.n$, où n est le nombre d'états de la *PDFSM* et m est définie dans $\Im_m$. L'ensemble $I^{mn} \cap I_\lambda^*$ est une suite de test complète pour n'importe quelle *PDFSM A* ayant n états [Yevt 89].

Il est possible de réduire une *PDFSM* donnée *A* pour obtenir l'ensemble $VI^{mf-n+k+1}$, où V est un ensemble d'accessibilité, k un paramètre qui donne la longueur maximale des séquences d'entrées nécessaires pour séparer tous les états distingables et f est le nombre de blocs de la partition minimale de l'ensemble des états en sous-ensembles d'états deux à deux distingables (*fuzziness*) [Yevt 89] et [Luo 93]. Le paramètre k peut atteindre la valeur $n(n-1)$, alors que z est compris entre 1 et n inclusivement. Si $z=n$ la machine est réduite et si $z=1$ la machine est non-réduite et tous ses états sont deux à deux compatibles.

6.3.4. Machines indéterministes

- **Relation de conformité**

La *NFSM* $B=(T, I, O, H, t_0)$ est une réduction de la *NFSM* $A=(Q, I, O, h, q_0)$, notée $B \leq A$, si :

$$I_A^* \subseteq I_B^* \text{ et } \forall \alpha \in I_A^* \ (H^2(t_0, \alpha) \subseteq h^2(q_0, \alpha));$$

sinon, B n'est pas une réduction de A, notée $B \not\leq A$. Si B est déterministe et $B \leq A$, alors B est dite une *réduction* déterministe de A.

Cette relation de conformité est basée sur le fait que toutes les séquences de sorties produites par l'implantation en réponse à toutes les séquences d'entrées acceptées par la spécification doivent être décrites par la spécification [Boch 93].

La *NFSM* $B=(T, I, O, H, t_0)$ est quasi-équivalente à la *NFSM* $A=(Q, I, O, h, q_0)$ sur le domaine D_A, et nous le notons $B \cong_{D_A} A$, si :

$$I_A^* \subseteq I_B^* \text{ et } \forall \alpha \in I_A^* \ (H^2(t_0, \alpha) = h^2(q_0, \alpha));$$

B est dite distingable de A, et nous le notons $B \not\cong_{D_A} A$, si elle n'est pas quasi-équivalente à A. Cette relation de conformité exprime le fait que toutes les séquences de sortie décrite par la spécification et seulement celles-ci doivent être produites par l'implantation en réponse à toutes les séquences d'entrées acceptées par la spécification.

Si A et B sont des machines indéterministes complètement spécifiées (*CNFSM*) alors:

$$A \leq B \text{ et } B \leq A \Leftrightarrow B \cong A,$$

où "$\cong$" est la relation d'équivalence comme dans le cas déterministe.

La relation de quasi-équivalence est un cas particulier de la relation de réduction. Si les machines sont déterministes alors les deux relations se confondent dans la relation de quasi-équivalence pour les machines partiellement spécifiées ou dans la relation d'équivalence pour les machines complètement spécifiées.

Machine indéterministe partiellement spécifiée	Réduction	quasi-équivalence
Machine indéterministe complètement spécifiée	Réduction	équivalence

Table 6.2 : Relation de conformité pour des machines indéterministes

- **Bornes supérieures pour une suite de test complète**

Pour distinguer deux états qui ne sont pas équivalents dans une machine indéterministe complètement spécifiée ayant n états, nous devons appliquer une séquence d'entrée dont la longueur est inférieure ou égale à $2^n -2$; alors que pour distinguer deux machines indéterministes ayant respectivement n et m états, nous avons besoin d'une séquence de longueur inférieure ou égale à $2^{n+m}-2$ [Star 72]. Pour les machines observables complètement spécifiées, les bornes sont plus raisonnables : n-1 et $n+m$-1 respectivement.

Il faut noter que pour tester des implantations indéterministes, il faut supposer que la propriété d'équité est implantée. Dans ce cas, il est nécessaire de répéter l'exécution des tests afin de s'assurer que toutes les transitions de l'implantation ont été visitées pendant le test. La complexité du test est donc affectée par le nombre maximal de répétitions requises.

Nous regroupons dans la table suivante les bornes supérieures pour une suite de test complète dans le cas des machines indéterministes. Nous supposons que les machines ont un alphabet d'entrée commun I.

	longueur maximale d'une séquence distinguant deux machines ayant n et m états	suite de test m-complète pour une machine ayant n états
Machine observable complètement spécifiée	$n+m-1$	I^{n+m-1}
Machine observable partiellement spécifiée	$n.m$	$I^{n.m} \cap I_A^*$
Machine quelconque	$2^{n+m} -2$	$I^{2^{n+m}-2}$

Table 6.3 : bornes supérieures pour le test

6.3.5. Méthodes de dérivation de tests pour les *FSMs*

• **Hypothèses de test**

La dérivation des tests de conformité à partir du modèle des machines à états finis est basée sur le fait que la détection de fautes dans une implantation représentée par une boîte noire est en général indécidable sauf si nous introduisons certaines hypothèses. Un ensemble minimal d'hypothèses sur la classe de *FSMs* doit assurer que toute implantation peut être représentée par un seul *FSM*. Cette dernière, complètement spécifiée, réduite, et connectée, a le même alphabet d'entrée que la spécification et que son nombre d'états est inférieur où égale à une valeur donnée. Si l'implantation est indéterministe alors l'hypothèse qu'après un nombre donnée de répétition des tests, tout le comportement de la machine doit être produit doit être ajoutée.

Pour garantir une couverture de fautes totale pour un domaine de fautes prédéfini, certaines méthodes de dérivation de tests imposent d'autres contraintes telles que: l'existence d'une fonction de remise à l'état initial (*reset*) et le nombre d'états de l'implantation n'est pas plus grand que celui de la spécification ou les erreurs dans l'implantation sont restreintes à un type de fautes donné.

6.3.6. Description des méthodes

Les méthodes développées pour les machines à états finis déterministes complètement spécifiées, sont toutes basées sur une approche de vérification des transitions [Henn 64]. Chaque méthode requiert un type distinct d'ensemble de caractérisation pour l'identification des états. Les méthodes sont aussi applicables aux machines déterministes partiellement spécifiées [Yevt 90a] et [Petr 91] ; elles ont été étendues par la suite aux machines indéterministes observables. La production de suites de tests complètes pour les implantations indéterministes peut être réalisée grâce à la méthode *HSI* pour la relation de quasi-équivalence [Luo 93] et la méthode *SC* pour la relation de réduction [Petr 93b]. La figure 6.1, donne une classification de ces méthodes d'après la longueur de la suite de test qu'ils produisent et sa couverture.

L'avantage de telles techniques est la possibilité de les automatiser. Les cas de test ainsi générés sont appliqués à l'implantation pour tester toutes les transitions. Généralement, les cas de test servent à accomplir les trois étapes suivantes :

- atteindre et confirmer l'état initial de la transition à tester,
- exécuter et valider la transition,
- confirmer l'état final de la transition à tester.

Dans ce qui suit, nous allons présenter les techniques les plus intéressantes pour la génération des cas de test :

Tour de transitions [Nait 81]:

Cette méthode consiste à chercher dans la spécification une séquence de transitions appelée tour de transitions. La particularité d'une telle séquence, est qu'elle passe par chaque transition de la spécification au moins une fois. L'extraction d'une telle séquence est un problème mathématique, il revient au problème classique de la théorie de graphes appelé le problème du postier chinois. Cette méthode nécessite que la spécification sous forme d'un *FSM* soit fortement connexe.

Séquence de distinction [Gone 70]:

Cette méthode consiste à chercher dans la spécification une séquence d'entrées appelée séquence de distinction. Cette séquence permet de distinguer tous les états de la *FSM*. Dépendamment de l'état à partir duquel elle est appliquée, elle donne une séquence de sortie distincte. Cette méthode nécessite que la *FSM* soit fortement connexe, complètement spécifié et possédant une séquence de distinction.

Méthode des séquences uniques d'entrées-sorties (*UIO*) [Sabn 88]:

Cette méthode consiste à déterminer pour chaque état de la spécification une séquence d'entrées qui permet de le distinguer de tous les autres états. Ces séquences sont appelés les *UIO* (*Unique Input-Output*). Cette méthode permet de distinguer les états de la *FSM* suivant leur réaction aux *UIO*. Le calcul de la signature est une technique pour la détermination des *UIO*. La méthode des *UIO* nécessite que la *FSM* soit fortement connexe, complètement spécifié et possédant des *UIO*.

Méthode *UIOv* [Vuon 89]:

Cette méthode est une variante de la méthodes des *UIO*. Elle est dans l'ensemble identique à l'*UIO*, mais ajoute une étape de vérification. Cette vérification permet de vérifier si les *UIO* extraites de la spécification sont valables pour l'implantation.

Méthode *W* [Chow 78]

Cette méthode consiste à construire deux ensembles *W* et *P* où *W* est l'ensemble de séquences d'entrée permettant de distinguer entre les états de la *FSM* et *P* l'ensemble de couverture des transitions de la *FSM*, c'est-à-dire l'ensemble de séquences d'entrée qui permettent de passer de l'état initial de la *FSM* à l'état de départ de la transition à tester. Une certaine concaténation des deux ensembles forme l'ensemble des séquences de test. Cette méthode permet de résoudre le problème des *FSMs* qui n'ont ni de séquences de distinction ni des séquences *UIO*. Elle exige par contre que la spécification sous forme de *FSM* soit réduite, complètement spécifiée, déterministe et fortement connexe.

Méthode *Wp* [Fuji 90]:

Cette méthode est une optimisation de la méthode *W*. Elle permet d'obtenir des séquences de test de longueur inférieure ou égale à celle de la méthode *W*. Au lieu d'utiliser l'ensemble *W* pour tester chaque état atteint, elle se limite à un sous-ensemble de *W* qui dépend de l'état atteint. Le sous-ensemble ainsi utilisé s'appelle "***ensemble d'identification***" de l'état.

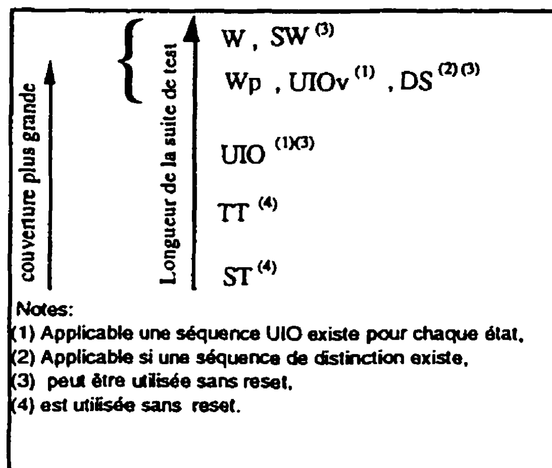


Figure 6.1. Relation entre la longueur d'une suite de test et sa couverture

6.4. Classification des Spécifications

La longueur de la suite de tests peut être considérée comme une bonne mesure de testabilité [Petr 93a]. Plus la suite de test est longue, plus les tests sont difficiles et coûteux. Certaines propriétés de la spécification (voir table 6.3) peuvent influencer la longueur des suites de test. Les automates indéterministes sont difficiles à tester ; ils sont généralement moins testables que les automates déterministes. Parmi l'ensemble des automates indéterministes, ceux qui sont non-observables sont moins testables que les automates observables. L'indéterminisme avec ces deux types observable et non-observable est l'une des propriétés qui influencent la contrôlabilité des automates. Nous savons aussi que la génération de la suite de test à partir d'une spécification non-réduites est plus complexe que celle à partir de spécifications réduites. Ceci pose le problème de distinction entre les différents états de la spécification. La propriété de réduction est aussi appelée la minimalité de la machine ; elle influence l'observabilité des automates. Finalement une machine partiellement spécifiée possède moins de transitions qu'une machine complète, mais l'identification de ses états peut exiger des séquences de distinction plus longues et donc elle est moins testable. Le déterminisme, la minimalité et la complétude sont donc trois propriétés qui peuvent être utilisées pour classer les spécifications. L'appartenance d'une spécification à une classe particulière peut donner une idée sur sa testabilité.

Si c'est possible le concepteur applique des transformations à la spécification pour la raffiner et améliorer sa testabilité. Ces raffinements ont pour objectif d'améliorer la classe d'appartenance (niveau de testabilité) de la spécification initiale. Nous proposons une classification tridimensionnelle basée sur les propriétés de déterminisme, de minimalité et de complétude (voir figure 6.2). Cette classification comporte douze classes. Les classes varient de la classe la moins testable, comportant les spécifications partielles, non-réduites, indéterministes et non-observables, à la classe des spécifications les plus testables et qui comporte les machines complètes, réduites, et déterministes.

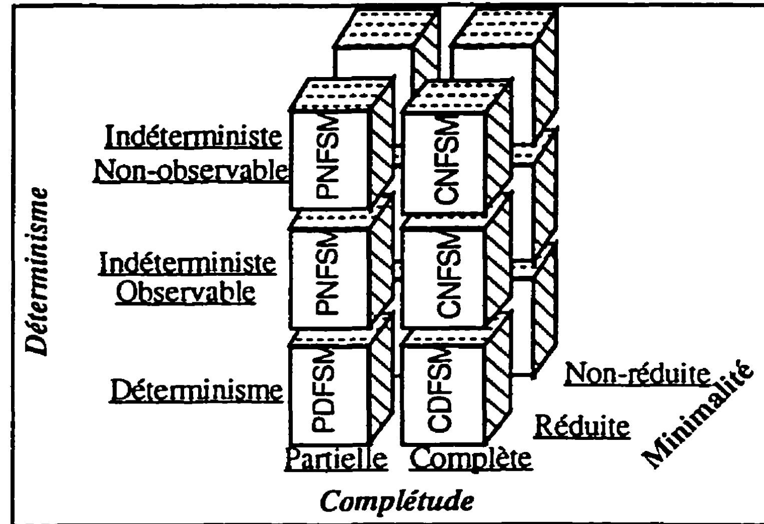


Figure 6.2. Classification des *FSMs*.

6.5. Mesure de testabilité proposée

La longueur d'une suite de test peut être associée à l'effort nécessaire pour tester un protocole, plus la suite est longue, plus les tests sont coûteux. Elle peut donc être considérée comme un bon indicateur de la testabilité d'un protocole [Petr 93a]. Elle est étroitement dépendante des propriétés de la spécification de référence. Pour un *FSM* M à n états, d transitions, et une implantation avec $m \geq n$ états, la suite de test complète d'après la méthode W est égale à [Boch 94b]: $TS = V.(I^{fm-n+1} \cap I_M^*).W$ (1)

où $V.(I^{fm-n+1} \cap I_M^*).W$ est la concaténation des ensembles de séquences V, $I^{fm-n+1} \cap I_M^*$ et W, f est le degré de *fuzziness*, I l'ensemble des entrées, V est l'ensemble d'accessibilité, $I^{fm-n+1} \cap I_M^*$ est l'ensemble des séquences de longueur $fm-n+1$ acceptable par M, et W l'ensemble des séquences de distinction.

Pour évaluer la testabilité d'une spécification (*CDFSM*), Petrenko [Petr 93a] a proposé l'utilisation d'une mesure qui est basée sur la borne supérieure de la longueur de la suite de test nécessaire au test complet d'un protocole. Pour un *CDFSM*, la borne de la suite de test (TS) est de: $Longueur(TS) \leq mn^2p^{m-n+1}$. À partir de cette borne, l'auteur a proposé la mesure de testabilité suivante: $T=|O|/mn^2I^{m-n+1}$. Cette mesure n'est valide que pour la classe de spécification complète et déterministe *CDFSM*, elle ne peut pas être appliquée aux *PNFSMs*. Pour cette classe d'automates aucune évaluation de la suite de

test complète n'a été jusqu'à maintenant proposée; la testabilité ne peut donc pas être évaluée qu'après avoir extrait la suite de test au complet. Au cours des sections prochaines, nous proposerons un ensemble de mesures de testabilité plus générales et qui peuvent donner plus d'information sur les faiblesses d'une spécification.

6.5.1. Nouvelle mesure de testabilité

La suite de test telle qu'elle est présentée dans la formule 1, est fonction de plusieurs éléments de bases (V, I, W, f, etc.). Ces éléments sont soit des ensembles, soit des variables, leurs propriétés ou leurs valeurs sont importantes pour caractériser l'effort de test, ils font aussi partie des facteurs de testabilité (voir chapitre 4).

Pour éviter d'évaluer la longueur de la suite de test, nous proposons d'évaluer individuellement quatre facteurs de testabilité extraits de la formule 1. Les facteurs que nous évaluons (voir sections suivantes) sont: le *degré de contrôlabilité* ($C(M)$) associé aux caractéristiques de l'ensemble V, le *degré de flou* ($flou(M)$) associé à la valeur de f, le *degré de distinction des états* ($W(M)$) associé aux caractéristiques de l'ensemble W et le *degré d'abstraction* ($AB(M)$) associé aux valeurs de m et de n. Cette approche a l'avantage, d'une part, de généraliser la mesure donnée par Petrenko [Petr 93a], et d'autre part, de connaître avec plus de précision les causes d'une éventuelle mauvaise testabilité. Les évaluations des facteurs de testabilité mentionnées ci-dessus sont rassemblées au sein d'un même vecteur appelé *vecteur de testabilité*:

$$TV(M) = \langle C(M), flou(M), W(M), AB(M) \rangle \qquad (2)$$

Dans les sections qui suivent, nous donnerons plus de détail sur chaque élément de $TV(M)$ et nous le lui associerons une évaluation. Nous nous intéresserons particulièrement aux *FSMs* observables (voir section 6.2).

- **Le degré de contrôlabilité**

L'ensemble d'accessibilité V est constitué d'un ensemble de séquences appelées *séquences de transfert* ou *préambules*. Lors du processus de tests, ces séquences permettront d'accéder aux différents états de la spécification. L'ensemble V constitue un squelette pour la suite de tests, il garanti l'accès à chacun des états pour exécuter et tester toutes les transitions. Pour un *FSM* donné l'ensemble d'accessibilité n'est pas unique,

nous choisissons en général l'ensemble le plus court. L'effort nécessaire pour accéder aux états contribue au coût des tests, nous l'appellerons *degré de contrôlabilité*.

Soit V l'ensemble d'accessibilité minimal d'un *FSM M* complètement connecté. Pour chaque état q de M, il existe une séquence de transfert $v \in V$ tel que $q \in h^1(q_o, v)$. Les caractéristiques de l'ensemble V influencent l'effort des tests. Ces caractéristiques sont la longueur de toutes les séquences de transfert et le nombre de séquences de sorties produites par chaque séquence de transfert. Le nombre de séquences de sorties produites par chaque séquence de transfert est un indicateur de la difficulté d'atteindre l'état q. Lors de l'étape de test, si l'automate est indéterministe, il faut appliquer plusieurs fois la même séquence pour pouvoir atteindre l'état q. Ceci augmente la longueur et la complexité des tests. Nous intégrons la longueur de toutes les séquences de transfert et le nombre de séquences de sorties produites par chaque séquence de transfert en une seule formule appelée poids de V [Dsso 95b]: $\omega(V) = \sum_{v \in V} L(v)|h^2(q_1, v)|$.

Dans le meilleur cas, pour chaque état d'une spécification (à part l'état initial q_o), il existe une entrée simple qui nous permet d'accéder d'une façon déterministe à l'état $\omega(V)=n\text{-}1$. En contre partie, dans le pire cas, une séquence de transfert peut avoir une longueur de n-1, et le nombre de sorties produites par une telle séquence peut être égal à $A^{|v|}_{|v|.|O|} = \frac{|v|.|O|!}{(|v|.(|O|-1))!}$ (les arrangements de longueur |v| des symboles de sorties). Le poids de V a donc les bornes suivantes:

$$\omega_l(V)=n\text{-}1 \leq \omega(V) \leq \omega_u(V)=\sum_{p=1}^{n-1} p.\, A^{p}_{p.|O|}$$

Posons $c_l(V)= 1/\omega_u(V)$, $c_u(V)= 1/\omega_l(V)$, et $c(V)=1/\omega_{min}(V)$, où $\omega_{min}(V)=n\text{-}1$ est la valeur minimale du poids de V, c'est-à-dire le cas de l'ensemble d'accessibilité V contient les séquences les plus courtes et dont les éléments produisent le moins de sorties possibles. Nous définissons le *degré de contrôlabilité* d'une machine M comme suit:

$$C_T(M)=\frac{c(V) - c_l(V)}{c_u(V) - c_l(V)} \quad \text{, et } 0 \leq C_T(M) \leq 1. \qquad (3)$$

À partir de cette formule nous pouvons déduire que la contrôlabilité d'une machine déterministe (*DFSM*) est la suivante: $c_u(V)=1/n\text{-}1$, $c_l(V)=2/n(n\text{-}1)$, et $C_T(M)=[n(n\text{-}1)c(V)\text{-}2]/(n\text{-}2)$.

- **Le degré de *flou***

Au cours du processus de test, certains problèmes peuvent survenir si les états ne sont pas deux à deux distingables, c'est-à-dire si la machine n'est pas réduite. Dans ce cas, si la spécification est complètement spécifiée (*CFSM*), nous devons réduire la machine avant de générer la suite de test. Pour les machines partiellement spécifiées (*PFSMs*), nous pouvons, dans certains cas, étendre et appliquer les méthodes de test relatives aux spécifications complètes [Boch 94b]. Dans le cas où la machine est partielle et non-réduite, nous pouvons rassembler le maximum d'états distingables ensemble pour former des partitions [Yevt 89]. Le nombre minimal des partitions constitue une des propriétés qui peut caractériser l'effort des tests.

Soit un *FSM* $M=(Q, I, O, h, q_0)$, l'ensemble Q est partitionné en sous-ensembles $\Pi=\{Q_1, ..., Q_k\}$, chaque sous-ensemble contient des états distingables. Le nombre minimum de sous-ensembles de la partition minimale Π_{min} est appelé *fuzziness* [Yevt 89] de M. Ce facteur est représenté par $f=(min(k))$ dans la formule 1. Il donne une idée de la difficulté d'identification des états. Plus f est grand plus la suite de test est longue et les tests difficiles. Comme pour les métriques précédantes, nous proposons une nouvelle métrique bornée entre 0 et 1 et qui représente le *fuzziness* d'une machine. Cette nouvelle mesure sera appelée le *degré de flou* d'une machine $flou(M)$ et est évaluée comme suit:

$$flou(M) = (n - f)/(n-1) \text{ et } 0 \leq flou(M) \leq 1. \qquad (4)$$

Dans le pire cas, tous les états ne sont pas deux à deux distingables, il y aura donc autant de partitions que d'états $f = n$ et $flou(M) = 0$. Par contre, si la machine est réduite tous les états sont distingables, alors $f = 1$ et $flou(M) = 1$.

- **Le degré de distinction des états**

La majorité des méthodes de test des *FSMs* sont basées sur la distinction des états [Gone 70], [Chow 78], [Sabn 88] et [Vuon 89], etc. Pour pouvoir distinguer les différents états d'un *FSM* nous avons besoin de leur appliquer une ou plusieurs séquences d'entrée qui peuvent produire pour chaque état des séquences de sortie différentes. De telles séquences d'entrée sont appelées *séquences de distinction*, elles sont rassemblées dans un ensemble unique appelé *ensemble de caractérisation* ou *de distinction*. Les caractéristiques de cet ensemble influencent la difficulté des tests. Les caractéristiques de

cet ensemble sont la longueur et le nombre des séquences de distinction. Plus il contient de séquences et plus ces séquences sont longues, plus le processus de test est difficile. Nous appelons ce facteur le *degré de distinction* des états.

Étant donné un *FSM M*, et la partition minimale de ces états $\Pi_{min}=\{Q_1, ..., Q_k\}$ (voir le degré de flou de la section précédante), pour chaque $Q_i \in \Pi_{min}$, nous pouvons générer un ensemble de séquences de distinction W_i qui permet de distinguer tous les états de Q_i. Tous les W_i sont, à leur tour, rassemblés dans un ensemble unique W que nous appellerons ensemble de caractérisation. Cet ensemble n'est pas unique, il est l'union de tous les W_i. Le nombre de cas de test est proportionnel au nombre de séquences de W. D'autre part, plus les séquences de W sont longues plus les cas de test le sont aussi. Le nombre de séquences $|W|$ de W ainsi que la somme des longueurs des séquences $L(W)$ de W peuvent caractériser la difficulté des tests. Nous combinons $L(W)$ et $|W|$ dans une formule unique $\omega(W)$ appelée poids de W [Dsso 95b]: $\omega(W)=|W|\ L(W)$.

Dans le meilleur cas, une entrée simple peut distinguer tous les états de la machine $\omega(W)=1$. Dans le pire cas, pour une machine réduite partiellement spécifiée, chaque paire d'états est distingable par une séquence unique w_i. Dans ce cas, $|W| = C_n^2 = n(n-1)/2$, et

$$L(W)= \sum_{i=1}^{n(n-1)/2} L(w_i) = \sum_{i=1}^{n(n-1)/2} \frac{i(i-1)}{2} \ .$$

Pour les machines non-réduites, les w_i n'existent pas toujours, alors $\omega(W)=+\infty$.

Le *degré de distinction* des états de la machine est dénoté $W(M)$ et est défini alors comme $1/\omega_{min}(W)$, où $\omega_{min}(W)$ est la valeur minimale du poids de tous les ensembles de caractérisation possible:

$$0 \leq W(M)=1/\omega_{min}(W) \leq 1. \qquad (5)$$

- **Le degré d'abstraction**

La spécification et l'implantation peuvent être considérées comme des représentations différentes d'un même comportement à des niveaux d'abstraction différents. Une implantation est moins abstraite, elle possède plus d'état que sa spécification de référence. Il est claire que plus la différence entre le nombre d'état de l'implantation (m) et de la spécification (n) est grand, plus la suite de test est longue (voir Formule 1). La comparaison entre le nombres d'états de la spécification et de

l'implantation donne une idée sur les niveaux d'abstraction. Nous appelons ce facteur de testabilité *degré d'abstraction* et nous l'évaluons comme suit:

$AB(M) = n/m$ and $0 < AB(M) \leq 1$ (6).

- **Mesure de testabilité unifiée**

Chacun des facteurs de testabilité précédents est quantifié individuellement par une métrique ayant comme bornes inférieure et supérieure 0 et 1 (Formules 3, 4, 5 et 6) . Dans le meilleur cas, le vecteur de testabilité aura la valeur $TV(M) = \langle 1, 1, 1, 1 \rangle$. Nous appellerons cette valeur optimale de *TV(M)* *BTV* (*Best Testability Vector*). Pour obtenir une seule mesure de testabilité basée sur les évaluations individuelles des quatre facteurs de testabilité, nous proposons de combiner *TV(M)* et *BTV*. La mesure produite est appelée ***mesure unifiée de testabilité*** notée *ACTM(M)* (*Automata Combined Testability Measure*). *ACTM(M)* est basée sur la distance entre *TV(M)* et *BTV*:

$$ACTM(M) = 1 - \frac{\sqrt{(1-C(M))^2 + (1-flou(M))^2 + (1-W(M))^2 + (1-AB(M))^2}}{2} \qquad (7)$$

Le principal avantage de cette mesure est qu'elle ne nécessite pas de fortes hypothèses sur les propriétés de la spécification (le déterminisme et la complétude, etc.).

- **Lien entre les facteurs de testabilité et les propriétés d'une spécification**

Les propriétés de déterminisme, de minimalité et de complétude utilisés pour classer les spécifications ont une influence directe sur l'évaluation de nos facteurs de testabilité.

- Influence du déterministe sur le degré de contrôlabilité: si la spécification est déterministe, la composante de la formule de *C(M)* qui représente le nombre de sorties associée aux séquences de transfert ($h^2(v)$) sera toujours égale à 1. Ceci améliore l'évaluation de *C(M)*.

- Influence de la minimalité sur le degré de flou: si la spécification est minimale, *f* est égal au nombre d'états de la spécification et donc le degré de flou est optimal $flou(M) = 1$. En contre partie, si la spécification n'est pas minimale, $f > 1$ et donc $flou(M) < 1$.

- La complétude influence le degré de distinction des états: si la spécification est complète, pour chaque sous-ensemble d'états distingables, nous pouvons trouver une séquence unique de distinction. Ceci minimise la composante $|W|$ et donc peut minimiser $W(M)$.

- Le degré d'abstraction est un bon indicateur de la longueur d'une suite de test. Ce facteur ne peut pas être associé à aucune des propriétés précédantes de la spécification car il implique aussi bien des propriétés de la spécification que celle de l'implantation.

- **Signification des valeurs associées aux facteurs de testabilité**

Les métriques que nous venons d'introduire ont pour bornes les valeurs 0 et 1. Des valeurs très proche de 0 ou de 1 peuvent nous indiquer que la valeur de ce facteur est très mauvaise ou très bonne. Nos métriques peuvent être aussi utiles pour des fins de comparaison. Dans le cycle de développement, elles peuvent être utilisées dans les deux directions suivantes:

- Choix de spécification: plusieurs spécification valides peuvent décrire un même comportement exigé par l'utilisateur. Nos métriques peuvent être utilisées comme guide pour choisir l'une d'elles, c'est-à-dire celle qui est le plus testable.

- Jugement des résultats d'une transformation: certains raffinements ou transformations peuvent améliorer la testabilité d'une spécification. À partir d'une spécification initiale, une transformation produit une nouvelle spécification. Nos métriques peuvent être utilisés pour juger si la spécification produite est plus testable que l'initiale. Dans le reste de ce chapitre, nos métriques sont utilisées dans ce sens. Nous proposons un ensemble de transformations qui peuvent améliorer les trois propriétés de la spécification. Nous jugeons, de l'amélioration de la testabilité apportées par nos transformations en évaluant les métriques associés à nos facteurs.

6.6. Transformations testables d'une spécification

Certaines propriétés de la spécification peuvent être améliorées afin d'augmenter la testabilité d'un protocole [Yevt 95a] et [Karo 96a]. Au cours de la section 6.5, nous en avons présenté trois de ces propriétés: le déterminisme, la complétude et la minimalité. Dépendamment du degré de testabilité ainsi que des propriétés de la spécification désirées, le concepteur peut raffiner la spécification en appliquant des transformations appropriées.

Les transformations peuvent être regroupées en trois classes dépendament de la propriété de la spécification sur laquelle elles agissent.

Dans ce qui suit, nous présenterons ces trois classes et nous proposerons pour chaque type une ou deux transformations. Finalement, nous illustrerons l'utilité de ces transformations en les appliquant à une spécification (figure 6.3.a).

6.6.1. Transformations de complétude

L'architecture en couches est largement utilisée dans le domaine des protocoles de communication. Les protocoles d'une couche spécifique reçoivent leurs entrées des couches adjacentes. Dans ce contexte, certaines séquences d'entrées ne peuvent jamais être appliquées à un protocole déterminé. Dans ce cas, il peut exister des transitions qui ne peuvent jamais être exécutées ; ces transitions sont appelées les *don't care transitions*. Lors de l'étape de la conception, le concepteur est libre de les assigner de la manière qui l'arrange le plus. Dans notre cas, nous essayerons de les assigner de façon à faciliter les tests.

Dans les sections qui suivent, nous présenterons deux algorithmes qui complètent les *don't care* transitions de la spécification de façon à améliorer le degré de distinction des états.

- **Transformation basée sur les *Self-loops* (*S-trans*)**

Cette transformation (*S-trans*) consiste à assigner aux *don't care transitions* d'une spécification des *self-loops* . Dans la mesure du possible, les sorties associées à ces transitions sont choisies de façon à améliorer les degrés de distinction des états et/ou le degré de flou de la spécification ($W(M), f(M)$). Comme première étape, nous avons opté pour l'amélioration de $f(M)$ et ceci en essayant de diminuer le nombre de classes de la partition minimale $\Pi_{min}=\{Q_1, ..., Q_k\}$ (voir section sur le degré de flou).

Nous commençons par extraire de Π_{min} un ensemble Q_i contenant le moins d'états possibles et ayant au moins une transition à compléter. Soit x et $q \in Q_i$ respectivement l'entrée et l'état de départ d'une des transitions *don't care* t qu'on désire assigner. Soit $Q_j \in \Pi_{min}$ une deuxième classe contenant le plus d'états possible et dont tous les états acceptent l'entrée x. L'objectif de *S-trans* serait d'assigner t en associant à x une sortie distinguant q de tous les autres états de Q_j. L'idéal serait de pouvoir distinguer q de tout

autres états de Q_j par la seule entrée x, c'est-à-dire la réponse à l'entrée x dans l'état q est différente de celles qui pourrait être produites dans n'importe quel état de Q_j. Si cela, n'est pas possible, nous essayerons la même technique sur un autre ensemble $Q_p \in \Pi_{min}$ dont le nombre d'états est le plus proche de Q_j. Cette opération se poursuit jusqu'à ce qu'on puisse trouver un ensemble Q_r où il est possible d'assigner t. Sinon nous passerons à une autre transition *don't care* à compléter. Ce processus est itératif, il se termine lorsque toutes les transitions *don't care* ont été essayées. Cette transformation est bénéfique si elle arrive à assigner au moins une transition, sinon on ne change pas la spécification. *S-trans* préserve la majeure partie des propriétés de la spécification initiale: nombre d'états, degré de contrôlabilité, etc.

Pour illustrer cette transformation, nous l'avons appliqué à la spécification de la figure 6.3.a. Cette spécification étant partielle, *S-trans* la complète en rajoutant des *self-loops*. Les résultats de cette application sont présentés dans la table 6.4. Dans l'annexe B, nous donnerons un algorithme de *S-trans*.

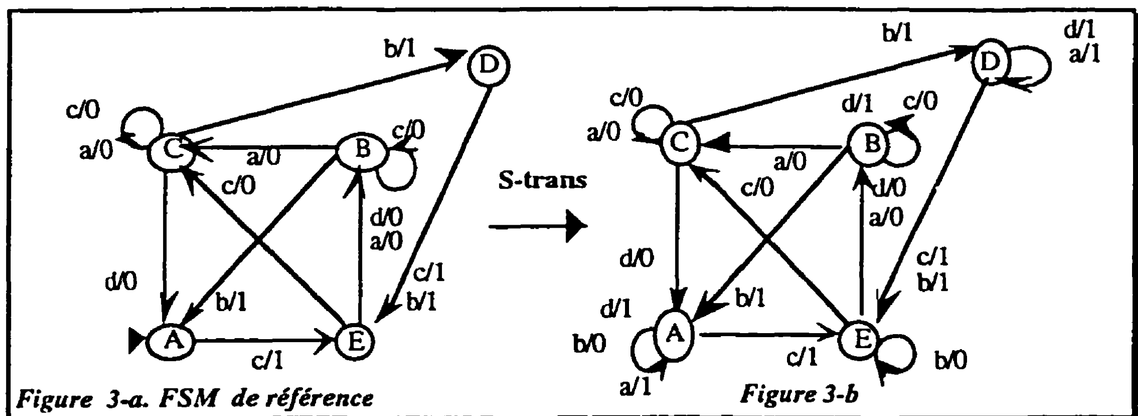


Figure 6.3. application de *S-trans*.

- **Transformation basée sur la séquence de distinction unique homogène (*H-trans*)**

Cette transformation consiste à compléter les transitions "*don't care*" de la spécification [Yevt 95a] et [Karo 96a] de façon à trouver une séquence de distinction homogène unique (une séquence constituée par la répétition du même symbole d'entrée) qui distingue tous les états de la spécification. Cette transformation est notée *H-trans*.

Pour trouver une séquence de distinction homogène i^* à un *FSM M*, il faut que la projection de *M* sur *i* soit réduite (section 6.3). Si aucune des projections de *M* sur un de ses symboles d'entrée n'est réduite, alors cette transformation n'est pas applicable. Par contre, s'il existe au moins une projection unitaire réduite de *M* sur une entrée *i*, alors la séquence de distinction homogène unique existe (Propositions 6.1.1 et 6.1.2).

Proposition 6.1.1 : Soit $Pr = \{M_1, M_2, ..., M_p\}$ l'ensemble des projections unitaires d'un *FSM M* sur respectivement $i_1, i_2, ...$ et i_p. Pour que *M* puisse avoir une séquence de distinction homogène unique il faut qu'au moins une de ses projections unitaires $M_j \in Pr$ soit réduite.

Preuve: Un *FSM M* est réduit si aucun de ces états n'accepte le même ensemble de séquences d'entrée-sorties. Supposons qu'aucun des éléments de *Pr* n'est réduit, cela veut dire que pour chaque $M_j \in Pr$, il existe au moins deux états qui répondent de la même manière à une séquence i_j^*, nous ne pouvons donc trouver une séquence homogène de distinction unique. ■

Proposition 6.1.2 : Soit M_j une projection unitaire minimale d'un *FSM M* sur i_j. Soit (q, i) une transition *don't care* de M_j, il est toujours possible d'assigner la transition (q, i) de façon à obtenir une séquence homogène de distinction unique de *M*

Preuve: Dans [Yevt 95a] (voir annexe B) nous avons prouvé que nous pouvons assigner *q* de façon à trouver une séquence homogène de distinction α qui peut distinguer *q* de tous les autres états de M_j. Puisque M_j est minimale alors la séquence la plus longue entre α et la séquence qui distinguait les états avant l'assignation de (q, i) constitue une séquence homogène de distinction unique de M_j et donc de *M* ■

Pour prendre en considération cette condition (proposition 6.1.1), nous avons étendu l'algorithme présenté dans [Yevt 95a]. Nous appellerons cet algorithme (transformation) *H-trans*. Nous donnerons ci-dessous, les étapes de bases de *H-trans*, l'algorithme au complet pourra être trouvé dans l'annexe B.

```
Début
Si l'automate de référence M admet un ensemble de projections unitaires
réduites,
Alors Si plusieurs de ses projections ont le même W(M) minimal,
Alors Si plusieurs de ses projections ont le même C(M)
     minimal,
       Alors Si toutes les projections ont le même
             nombre de transitions à compléter
                    Alors nous choisissons
```

```
                  une projection aléatoirement
            Sinon nous compléterons celle
                  qui possède le moins de transitions
                  à compléter
            FinSi,
      Sinon nous compléterons celle qui possède le C(M)
        minimal
      FinSi
Sinon nous compléterons celle qui possède le W(M) minimal
FinSi
FinSi
Fin
```

H-trans ne change ni le nombre d'états de la spécification ni la structure initiale de la spécification. *H-trans* peut être étendue pour obtenir une spécification complètement spécifiée. Ceci est fait en complétant les transitions non-spécifiées restantes (après avoir rajouté la transition *don't care*) par des self-loops. Cette extension de *H-trans* est notée *H-trans*.

Pour illustrer *H-trans*, nous l'avons appliqué à la spécification M de la figure 6.3.a. Il n'existe que deux projections unitaires de M sur b et d respectivement). M_2 et M_4 ont le même $W(M)$. Nous compléterons M_2 parce qu'elle offre un meilleur degré de contrôlabilité. Les états A et E sont complétés comme suit: $h^1(A, b) = C$, $h^2(A, b) = 0$ et $h^1(E, b) = A$, $h^2(E, b) = 0$ (figure 6.4.a). Le reste des transitions non-spécifiées est complété par des self-loops: $h^1(A, a)= A$, $h^2(A, a)= 1$; $h^1(A, d)= A$, $h^2(A, d)= 1$; $h^1(B, d)= B$, $h^2(B, d)= 1$; $h^1(D, a)= D$, $h^2(D, a)= 1$; $h^1(D, d)= D$, $h^2(D, d)= 1$.

Bien qu'elle ait augmenté le nombre de transitions spécifiées (de 13 à 20), cette transformation a:

1- produit un ensemble d'accessibilité $V = \{\varepsilon, ca, b, bc, c\}$ ayant de meilleurs propriétés que l'ensemble initial $V = \{\varepsilon, ca, cb, cbb, c\}$.

2- produit une séquence de distinction homogène unique $W = \{bbb\}$ bien que dans la spécification initiale les états ne soient pas deux à deux distingables.

3- amélioré $f(M)$ puisque tous les états sont devenus distingables. La suite de test produite est $TS = \{abbb, caabbb, babbb, bbabbb, cabbbb, bbbbbb, cbbbb, cacbbb, bcbbb, bbcbbb, ccbbb, dbbb, cadbbb, bdbbb, bbdbbb, cdbbb\}$ et donc la longueur $L(TS) = 86$ et la testabilité est améliorée de $ACTM = .34$ à $.57$ (Formule 7). Nous présentons dans la table 6.4, le résultat de l'application de cette transformation sur la spécification initiale.

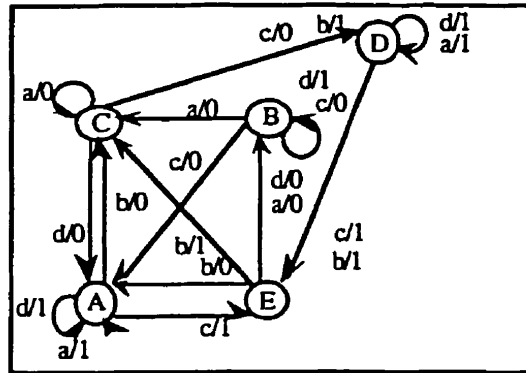


Figure 6.4.a. Application de *H-trans* sur la spécification de la figure 6.3.a.

Pour illustrer cette approche, on l'a appliqué à l'entité du protocole d'*INRES* relative au répondeur [Yevt 95a]. Le comportement du répondeur est spécifié par l'automate de la figure 6.4.b (flèches normales). L'alphabet d'entrée de cette entité de protocole est le suivant: 1- *CR*, 2 - *IDISr*, 3 - *ICONrsp*, 4 - *DT*0, 5 - *DT*1. L'alphabet de sortie est le suivant: 1 - *ICONi*; 2 - *DR*; 3 - *CC*; 4 - *ACK*0; 5 - *ACK*0, *IDATi*; 6 - *ACK*1; 7 - *ACK*1, *IDATi*; 8 - *null*. Si les transitions manquantes sont considérées comme des *self-loops* ayant des sorties nulles, l'application de la méthode *Wp* génère une suite de test de longueur 76 et qui comprend 19 cas de test. Par contre, si on applique *H-trans*, nous obtenons une suite de test de longueur 49 et qui comprend 17 cas de test. Les transitions que nous rajoutons sont indiquées par des lignes grasses dans la figure 6.4.b

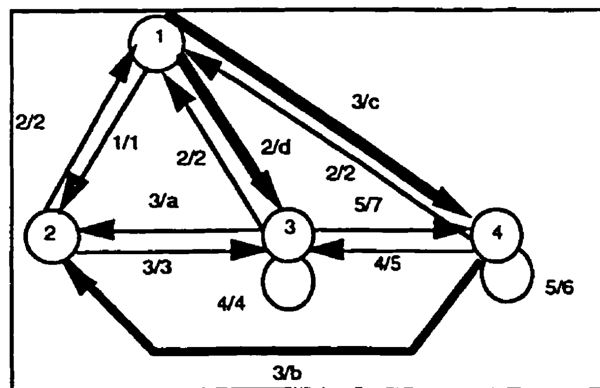


Figure 6.4.b. L'entité de protocole du répondeur d'*INRES*.

Pour montrer l'efficacité de cette approche, on a construit un outil pour énumérer toutes les machines possibles générées en complétant la spécification du répondeur. Cet outil a généré $(1+48)^4$ = 1,185,921 machines ; pour chacune d'elles, on a dérivé une suite

de test. Les résultats de cette expérience sont concluants puisqu'aucune des machines générée n'a une suite de test de longueur inférieure à 49.

6.6.2. Transformation permettant de rendre déterministe une spécification

Il existe plusieurs types d'indéterminisme [Vuon 93] (voir section 3.4.7.). Dans cette section, nous traiterons l'indéterminisme due au niveau d'abstraction adopté, c'est-à-dire l'indéterminisme dû au manque d'information. Pour l'éliminer, il est nécessaire de raffiner la spécification en rajoutant plus d'informations pertinentes. Dans un travail précédant [Karo 96a], nous avons proposé une transformation *D-trans* qui accompli cette tâche.

D-trans

D-trans élimine d'abord l'indéterminisme non-observable avant de s'attaquer à l'indéterminisme observable. L'indéterminisme non-observable est éliminé en utilisant un algorithme classique présenté dans [Star 72] et [Luo 93], etc. Pour l'indéterminisme observable, nous proposons principalement deux techniques. Le *dépliage de l'état* qui consiste à décomposer l'état indéterministe en plusieurs copies dont chacune est associée à une branche de l'indéterminisme. La seconde technique est appelée *dépliage des transitions* et consiste à rajouter de nouveaux paramètres aux primitives associées à des transitions indéterministes pour distinguer les différentes branches de l'indéterminisme.

6.6.3. Transformation permettant de minimiser la spécification

L'objectif de cette transformation est de fusionner tout le comportement redondant de la spécification. Pour atteindre cet objectif, il existe un algorithme [Koha 70] qui permet de fusionner les états compatibles des machines partiellement spécifiées. Cet algorithme consiste à compléter les transitions non-spécifiées de chacun des états appartenant à un ensemble d'états compatibles de façon à le rendre équivalent avec les autres états de l'ensemble. Cet algorithme n'admet pas une solution unique. Il produit un ensemble de spécifications dont certaines sont indéterministes. Le comportement de chacune des spécifications produites couvre le comportement de la spécification initiale. Pour résoudre le problème du choix de la meilleure spécification à considérer, nous avons proposé la transformation *C-trans* [Karo 96a].

Transformation basée sur la fusion des états compatibles (*C-trans*)

C-trans fusionne les états compatibles d'une spécification partielle et produit une spécification minimale unique. Pour le choix de la spécification, cette transformation prend en considération les deux critères suivants:

- L'ensemble des séquences d'entrée acceptées par la nouvelle spécification couvre celui de la spécification initiale. Notre transformation doit produire une spécification dont l'ensemble des séquences d'entrée est le plus proche possible de la spécification initiale. Dans notre cas, ceci consiste à rajouter le minimum de nouveaux chemins à la spécification initiale. Ce critère assure que le comportement de la nouvelle spécification soit proche de la spécification initiale.
- La nouvelle spécification doit améliorer la testabilité (*ACTM*) de la spécification initiale.

À partir de la spécification initiale, *C-trans* commence par construire un ensemble contenant un nombre minimal de compatibles couvrant tous les états de la spécification. Les compatibles sont des ensembles dont les éléments sont des états deux à deux compatibles. À partir de cet ensemble, *C-trans* extrait de nouveaux ensembles fermés de compatibles qui couvrent tous les états de la spécification [Koha 78]. Un ensemble de compatibles est dit fermé si, pour chacun de ses éléments, ses images par la fonction de comportement sont aussi contenues dans le même ensemble. Chacun de ces ensembles de compatibles constitue une machine réduite dont les états sont les compatibles et dont les transitions sont produites par la fonction de comportement de la spécification initiale. Après cette première étape, *C-trans* se base sur les deux critères précédants (comportement et testabilité) pour choisir l'une des machines réduites. Pour cela, nous proposons de choisir une machine qui d'une part contient le moins de chemins indépendants possible [McCa 89] et d'autre part qui améliore notre mesure *ACTM*(*M*). Cette tâche est délicate puisqu'elle consiste à maximiser un facteur et à minimiser un autre. Pour une première version de l'algorithme, nous avons choisi de donner la priorité à la testabilité. Nous construirons tout d'abord un ensemble de spécifications réduites qui améliorent le plus la testabilité. À partir de cet ensemble, nous sélectionnerons la spécification qui possède le moins de chemins indépendants possibles (voir annexe B).

Pour compléter la spécification, *C-trans* peut être étendue en rajoutant des self-loops pour les transitions qui sont restées non-spécifiées. *C-trans* étendue est notée $\mathcal{C}$*-trans*. *C-*

trans améliore la testabilité de la spécification initiale en agissant sur le nombre d'états et de transitions à tester de la spécification. Cette transformation est assez complexe, elle peut changer la structure de la spécification initiale (nombre d'états et de transitions) (voir figure 6.5).

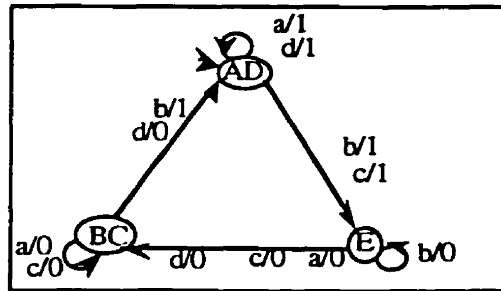


Figure 6.5. Application de *C-trans* sur la spécification de la figure 6.3.a.

6.6.4. Combinaison de transformations (*K-trans*)

De meilleurs résultats peuvent être obtenus en combinant plusieurs transformations. Dans cette section, nous proposons d'étudier une transformation composée notée *K-trans*. Cette dernière est la combinaison des transformations précédantes: *D-trans*, *C-trans*, *H-trans* et *S-trans*. *K-trans* a comme entrée une *PNFSM* non-réduite M_0, applique *D-trans* pour obtenir une spécification déterministe M_1, puis *C-trans* pour obtenir une spécification minimale M_2, puis *H-trans* pour avoir une séquence de distinction homogène unique M_3, et finalement *S-trans* pour compléter la spécification et obtenir une spécification déterministe, minimale et complète M_4 (M_0--*K-trans*-->M_4). Si toutes les transformations composant *K-trans* sont applicables, cette transformation produit une spécification appartenant à la meilleur classe de testabilité (d'après la classification de la section 6.5). Dans l'annexe B, nous donnerons un algorithme de *K-trans* .

K-trans améliore la testabilité d'une spécification en améliorant un ou plusieurs facteurs de testabilité $W(M)$, $f(M)$, $C(M)$. Si *C-trans* est applicable alors la spécification produite a moins d'états et de transitions. Ceci rend plus facile toutes les activités de validation, d'implantation et de tests du cycle de développement du protocole.

Pour illustrer *K-trans*, nous l'avons appliqué à la spécification de la figure 6.3.a, la spécification produite est représentée dans la figure 6.6. La table 6.4 résume les résultats

de l'application de *K-trans, S-trans, H-trans* et *C-trans* . Ces résultats sont calculés à partir des formules 3, 4, 5, 6 et 7.

	Spéc.	*S-trans*	*H-trans*	*C-trans*	*K-trans*
Figure n°	3.a	3.b	4	5	6
# transitions	13	20	20	12	12
nbre d'états	5	5	5	3	3
C(M)	1/6	1/6	4/9	0	1
f(M)	3/4	1	1	1	1
W(M)	0	1/3	1/3	1/2	1/2
AB(M)	1	1	1	1	1
L(TS)	Difficile	112	86	41	38
ACTM(M)	.34	.46	.57	.44	.75

Table 6.4. Résultats de l'application des transformations.

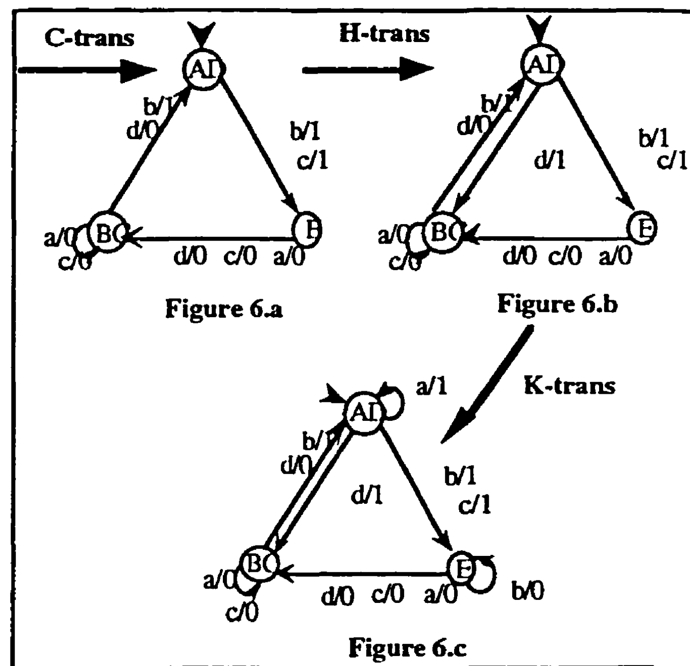


Figure 6.6.Application de *K-trans* sur la spécification de la figure 6.4.a..

6.7. Conclusion

Dans ce chapitre, nous avons présenté quatre nouveaux facteurs de testabilité. Ces derniers, influencent la longueur de la suite de test d'une spécification décrite sous forme de *FSM*. Nous avons évalué individuellement chacun de ces facteurs et nous les avons combinées en une seule formule qui donne une vue globale de la testabilité. Toutes les

mesures que nous avons présentées ne nécessitent pas de fortes hypothèses sur les propriétés de la spécification initiale. Par la suite, nous nous sommes basés sur ces facteurs ainsi que sur une décomposition (en six classes de testabilité) des spécifications de protocoles, pour proposer quatre transformations d'une spécification qui ont pour objectif d'améliorer la classe d'appartenance d'une spécification. Nous avons montré que si on combine ces transformations et on les applique dans un certain ordre, nous pouvons obtenir de meilleurs résultats. Dans ce même contexte, nous avons présenté une cinquième transformation qui est une combinaison des quatre transformations précédantes.

Chapitre 7

Étude de la testabilité basée sur le modèle des automates à états finis étendus[1]

7.1. Introduction

Les Techniques de Description Formelles (*FDTs*: *Formal Description Techniques*) sont utilisées de plus en plus dans la spécification des fonctionnalités des protocoles de communication. Certaines de ces techniques comme *ESTELLE* [ISO9074 89], *SDL* [ITU, 96] et *LOTOS* [ISO8807 89] ont été standardisées et utilisées pour la description des standards des grandes organisations de normalisation mondiales.

Plusieurs raisons encouragent l'utilisation des méthodes formelles pour la description des spécifications de protocoles de communication. La principale raison est d'éviter d'avoir plusieurs versions incompatibles du même protocole. Ce problème est dû à l'ambiguïté des méthodes informelles et aux différentes interprétations que des implanteurs peuvent donner à une même spécification. La deuxième raison est l'identification et la correction de certains types d'erreurs avant la phase d'implantation. Cette tâche peut être automatisée en utilisant certains outils de vérification et de simulation. Il existe d'autres raisons qui renforcent l'idée d'utiliser les *FDTs*, nous citerons par exemple: la génération automatique des suites de test, la génération automatique du code à partir de la spécification formelle, etc.

La génération automatique des suites de test à partir d'une spécification décrite par des *FDTs* reste encore un problème ouvert [ETS 94]. Les quelques méthodes de test existantes commencent par traduire la spécification en un modèle formel comme les *FSMs*

---

[1] Les résultats de ce chapitre sont publiés dans: the Eighth SDL Forum 97, INT Evry France, 22-26 Septembre 1997 [Karo 97a].

(*Finate State Machines*), les *EFSMs* (*Extended Finate State Machines*) ou les *graphes de flux de données*, puis extraient les tests de la spécification générée [Ural 87a], [Chan 93], [Sari 93], [Huan 95] et [Kim 95]. Au cours de ce chapitre, nous allons poursuivre cette direction: nous traduirons une spécification *SDL* en un *EFSM*, nous étudierons et améliorerons la testabilité sur l'*EFSM* ainsi généré, puis nous le retraduirons en *SDL*.

Nous avons choisi *SDL* parce qu'il est très répandu (dans la recherche et l'industrie), sa version graphique est facile à utiliser et sa syntaxe est très proche du modèle des *EFSM*. En plus, la manière dont y sont interprétées les transitions non-spécifiées est compatible avec la méthode que nous présentons. Finalement, *SDL* possède de très bons outils automatiques (de spécification, de validation, de simulation et d'implantation automatique). Ces derniers peuvent faciliter les différentes étapes du cycle de développement de protocoles.

Généralement, les protocoles sont des systèmes partiellement spécifiés [Petr 93b, Boch 94a]. Un processus décrit en *SDL* ne bloque pas à la réception d'entrées non-spécifiées. Pour *SDL*, les transitions non-spécifiées sont toujours considérées comme des "*self-loops*". L'approche que nous proposons consiste à utiliser ces transitions pour améliorer la testabilité des protocoles. Ce travail est la continuité de celui entamé au chapitre 6. En effet après avoir étudié l'aspect de contrôle d'une spécification, nous nous tournons maintenant vers l'aspect de données. Nous travaillerons uniquement sur des *EFSMs* obtenus en traduisant une spécification *SDL* normalisée [Sari 93]. Un *EFSM* ainsi généré sera complété de sorte à obtenir des tests plus courts de flux de données sans pour autant changer le comportement initial du protocole.

7.2. Définitions de base

Les *EFSMs* constituent le modèle théorique du langage de spécification *SDL*. Certains travaux [Chan 93], [Huan 95], [Kim 95] et [Rama 95] ont formellement décrit ce modèle et l'ont utilisé dans la génération des suites de test.

L'une des principales préoccupations de ce modèle est l'exécutabilité des cas de test [Chan 93] et [Huan 95]. En général, les méthodes de test existantes n'ont pas défini formellement l'exécutabilité des séquences de test. Cette notion est importante pour l'identification (automatique) des cas de test exécutables parmi toutes les séquences de test

possibles à générer. Dans cette section nous allons étendre les définitions traditionnelles de l'*EFSM* pour prendre en considération l'exécutabilité des séquences de test. Enfin, nous introduirons les propriétés de base de *SDL*.

7.2.1. Le modèle des *EFSMs*

Un *EFSM E* est représenté par un 5-tuplet $E = (S, X, Y, T, s_o)$ [Rama 95], où S est un ensemble fini non vide d'*états* incluant l'état s_0 appelé *état initial*, X est un ensemble fini non vide d'*interactions d'entrée* incluant l'interaction x_0 représentant l'absence d'interactions en entrée, Y est un ensemble fini non vide d'*interactions de sortie* incluant l'interaction y_0 qui représente l'absence d'interactions de sortie, T est un ensemble fini non vide de *transitions*. Chaque interaction d'entrée $x \in X$ est représentée par *?x(inlist)*, où x est le nom de l'interaction d'entrée et *inlist* est un ensemble de variables appelées *variables d'entrée* appartenant à un ensemble I; pour chaque élément $i \in I$, $d(i)$ représente un domaine fini de i. On rassemblera dans un ensemble J toutes les collections de valeurs de variables d'entrée possibles. Chaque interaction de sortie $y \in Y$ est représentée par *!y(outlist)*, où $y \in Y$ est le nom de l'interaction et *outlist* est un sous-ensemble de variables appartenant à un ensemble V des *variables internes* du protocole. Pour chaque élément $v \in V$, $d(v)$ représente le domaine fini de v. L'union des ensembles I et V est l'ensemble des variables de l'*EFSM*. Pour simplifier notre analyse, on supposera que les ensembles I et V sont disjoints.

Chaque transition $t \in T$ est un 6-tuplet $t=(s, s', x, y, p, a)$, où $s, s' \in S$, sont respectivement les *états de départ* et *d'arrivée* de la transition t, $x \in X$, $y \in Y^*$, p est un *prédicat* appartenant à un ensemble fini de prédicats P; chaque $p \in P$ opère sur les variables de l'*EFSM*; P inclus un prédicat spécial p_0 appelé le *prédicat identité*, p_0 est vrai quelque soit la configuration des variables de l'*EFSM*; a est appelée la *procédure* de t, a consiste en un ensemble fini d'actions de l'ensemble d'actions A, a opère sur les variables de l'*EFSM* et produit un changement des valeurs des variables internes; l'ensemble A inclue une procédure a_0 qui préserve la valeur courante des variables. Le couple (x, p) est appelé *partie conditionnelle* de la transition t, tandis que (y, a) est appelé *partie action* de la transition.

Nous définissons une fonction de transition H de l'*EFSM E* qui associe les éléments du domaine de E: $D_E \subseteq S \times X \times P$ à l'ensemble $S \times Y^* \times A$. Un triplet $(s, x, p) \in D_E$ s'il existe une transition $t \in T$ dont l'état de départ est s, et la partie conditionnelle (x, p).

$H(s, x, p)$, pour $(s, x, p) \in D_E$ produit un triplet (s', y, a) s'il existe une transition $t \in T$ tel que $t = (s, s', x, y, p, a)$.

L'*EFSM E* est *complètement spécifié* si $D_E = S{\times}X{\times}P$, sinon l'*EFSM E* est dit *partiel*. Une transition dont l'état de départ est $s \in S$ est dite *non-spécifiée* s'il existe $x \in X$ et/ou $p \in P$ tel que le triplet $(s, x, p) \notin D_E$. Nous dirons qu'une transition dont l'état de départ est $s \in S$ est *I-inspécifiée* s'il existe un $x \in X$ tel que le triplet $(s, x, p) \notin D_E$ pour chaque $p \in P$. Dans ce cas, l'état s est dit *I-inspécifié par rapport à x*.

Une *configuration d'état* est un $(k+1)$-tuplet $(s, b_1,..., b_k)$, où $s \in S$, le k-tuplet $(b_1,..., b_k) \in C$, où $C = \{(b_1,..., b_k) | \; b_i \in d(v_i), v_i \in V\}$ est l'ensemble de toutes les configurations de variables, C comprend l'élément c_0 appelé *configuration initiale*: $c_0 = (b_{1,0},..., b_{k,0})$ où $b_{i,0} \in d(v_i)$, $v_i \in V$. Soit $?x_j(inlist_j)$ avec $|inlist_j| = m \leq |I|$, une interaction d'entrée, nous appellerons *configuration d'entrée* une collection $(x_j, vinlist_j)$ où $vinlist_j=(c_1,..., c_m)$, $|vinlist_j|=m$, est une collection de valeurs associés aux variables de $inlist_j$. Soit $S{\times}C$ l'ensemble de toutes les configurations d'état possible, une configuration d'entrée $(x_j, vinlist_j)$ est dite acceptable dans la configuration d'état $(s, b_1,..., b_k) \in S{\times}C$ s'il existe un prédicat p évalué à vrai pour le tuplet $(vinlist_j, b_1,..., b_k)$.

Pour un *EFSM E* on associe un *PFSM* $E_D = (S{\times}C, X{\times}J, Y, h_D, D_D, s_0c_0)$ (voir chapitre 6), où, pour chaque configuration d'état $(s, b_1,..., b_k) \in S{\times}C$ et une configuration d'entrée acceptable $(x, vinlist) \in X{\times}J$ en $(s, b_1,..., b_k)$ on a:
$h_D[(s, b_1,..., b_k), (x, vinlist)] = \{[(H^s(s, x, p), \mathfrak{C}(a, x, vinlist, b_1,..., b_k)), H^y(s, x, p)]$ tel que $(H^s(s, x, p), H^y(s, x, p)) \in H(s, x, p)\}$, où $H^s(s, x, p)$, $H^y(s, x, p)$, et $H^a(s, x, p)$ représentent respectivement les projections de $H(s, x, p)$ sur l'état, la sortie, et l'action et $c(a, x, vinlist, b_1,..., b_k)$ représente le résultat de l'application de la procédure $H^a(s, x, p)$ sur les variables $(v_1,..., v_k)$. Si une configuration d'entrée $(x, vinlist)$ n'est pas acceptable dans la configuration d'état $(s, b_1,..., b_k)$ nous supposerons que cette transition est non spécifiée dans le *PFSM* E_D. Une séquence acceptable du *PFSM* E_D à l'état $(s, b_1,..., b_k)$ est définie de la manière classique (voir chapitre 6).

Une configuration d'état de l'*EFSM E* est dite *accessible* si elle est accessible à partir de l'état initial du *PFSM* E_D. Pour chaque état s, on note R_s l'ensemble de toutes les configurations d'états accessibles $(s, b_1,..., b_k)$ de E.

Une séquence de transitions $\alpha = T_1,..., T_n$ d'un *EFSM E* est dite ***exécutable*** si pour l'état de départ s de la transition T_1, il existe une configuration d'état accessible (s, $b_1,..., b_k$) tel qu'il existe une séquence $(x_1, vinlist_1)...(x_n, vinlist_n)$ de configurations d'entrée ($x_1,..., x_n$, sont respectivement les interactions d'entrée des transitions $T_1,..., T_n$), qui est une séquence d'entrée acceptable du *PFSM* E_D à l'état (s, $b_1,..., b_k$). L'état s est appelé état de départ de α et α est dite exécutable par rapport à la configuration d'état (s, $b_1,..., b_k$).

7.2.2. Le langage *SDL*

SDL (*Spécification and Description Language*) est un langage de spécification basé sur le modèle des *EFSMs*. Après une période d'investigations, le développement de *SDL* a commencé en 1972 . En 1976 la première version du langage est apparue, suivie par d'autres versions chaque quatre ans (la dernière en 96). Les deux versions de 1984 et 1988 sont une extension considérable du langage et lui permettent d'arriver à une certaine maturité. Par contre les deux dernières versions, celle de 1992 et celle de 1996, ont vu naître la version orientée objet de *SDL*. La dernière version, *SDL96* a été approuvée en 1995 et est définie dans [ITU, 96]. *SDL* donne un choix entre deux formes syntaxiques équivalentes : une représentation graphique (*SDL/GR*), et une représentation textuelle (*SDL/PR*). La syntaxe graphique facilite l'utilisation de *SDL* pour des personnes non habituées à la programmation. Les concepts de bases de *SDL* sont les suivants [Beli 89] et [Sari 93]:

- **Le système:** Une spécification (un système) en *SDL* est un nombre de modules (blocks) interconnectés. Le comportement de chaque "block" est modelé par un ou plusieurs processus.
- **La communication:** La communication entre les processus se fait de façon asynchrone par l'intermédiaire d'une file *FIFO* non bornée.
- **La structure:** Un système contient des blocs interconnectés par des canaux. Le canal est le moyen à travers lequel un bloc communique avec un autre ou avec son environnement. Un canal peut être unidirectionnel ou bidirectionnel. Un canal bidirectionnel peut être considéré comme deux canaux unidirectionnels indépendants. Un canal est typé s'il ne peut contenir que des messages d'un certain type. Dans *SDL*, les messages sont appelés des signaux.

Un bloc peut contenir, soit une sous-stucture de blocs, ou un ou plusieurs processus interconnectés par des "signal-routes" (chemin de signal). Le chemin de signal est le moyen à travers lequel un processus peut communiquer avec un autre processus à l'intérieur du bloc, ou avec son environnement (tout ce qui est en dehors du bloc). Un chemin de signal est, comme un canal, une file *FIFO* typée non bornée.

- **Le comportement:** Dans *SDL*, le comportement est défini par un ensemble de processus qui s'exécutent en parallèles. A chaque processus est associé exactement une file d'entrée, dans laquelle tous les signaux qui arrivent sont déposés.

  Dans *SDL*, un processus est décrit par une machine à états finis étendue *EFSM* (voir figure 7.1). Pour chaque état, il y a un ensemble de symboles d'entrées, chacun avec une transition associée. Lorsque dans un état, un processus attend une entrée, il est suspendu. La réception d'un signal est le seul événement qui peut permettre la transition vers un nouveau état. Le signal à recevoir est spécifié dans les symboles d'entrées. A chaque symbole d'entrée une condition peut être associée. Le signal spécifié dans la file d'entrée peut être reçu, si la condition est évaluée à vrai.

  Tous les signaux envoyés à un processus dans *SDL* sont stockés dans la file d'entrée dans l'ordre de leur arrivée, jusqu'à ce que le processus soit dans l'état où il peut accepter l'entrée. Il y a alors quatre possibilités:

  - le signal à la tête de la file est spécifié comme le début d'une transition, dans ce cas le signal est consommé par le processus, il est enlevé de la file et le processus exécute la transition associée.
  - le signal à la tête de la file est spécifié dans un "*save construct*", dans ce cas le signal reste dans la file et on examine le signal qui est après lui.
  - le signal à la tête de la file n'est dans aucun des deux cas précédent, dans ce cas le signal est enlevé de la file et on examine le signal suivant.
  - la file est vide, dans ce cas le processus est suspendu jusqu'à l'arrivée d'un signal.

Une transition permet de changer d'état et d'effectuer une suite d'actions. Les actions peuvent être des sorties ou des "*tasks*". Une action *sortie* transmet un signal à un autre processus. Un "*task*" est une manipulation des données internes.

La plupart des méthodes de test développées pour *SDL* ne génèrent pas directement les cas de test à partir de la spécification *SDL*. Ils transforment la spécification en un modèle mathématique approprié (les *FSMs* [Luo 94], les *EFSMs* ou les *graphes de*

flux de données [Ural 87a]) qu'ils utilisent pour l'extraction des cas de test. Cette transformation n'est pas systématique si la spécification *SDL* n'est pas écrite dans un style normalisé [Ural 87a] et [Sari 93].

La normalisation est une technique d'analyse statique des spécifications Estelle et SDL [Sari 93]. Elle permet de transformer une spécification initiale en une deuxième spécification dont chacune des transitions est accessible par un chemin unique. La spécification produite a le même comportement et plus de transitions que la spécification initiale. Les objectifs de la normalisation est de: identifier tous les chemins menant à une transition, définir un prédicat (évaluation symbolique) pour chacun de ces chemins et remplacer toutes les variables incluant les paramètres de sortie par leur valeur symboliques respectives. Dans le cas d'*Estelle*, la normalisation est achevée en ignorant les constructions qui nécessitent une analyse dynamique. Ces constructions sont: les pointeurs de *Pascal*, *priority*, *attach/detach*, et *connect/disconnect*. On assume aussi que les procédures sont non-récursives. Les transitions sont dépliées en considérant des valeurs de variables héritées à partir d'autres transitions. La normalisation de *SDL* est assez similaire à *Estelle*. Parmi les différences, on peut citer l'élimination de la construction *Save* [Luo 94b] et l'extension des procédures. En conclusion, la normalisation permet de rendre plus claire les transitions de la spécification: état de départ, entrée, prédicat, actions, sorties et état d'arrivée. Ceci correspond exactement au modèle des *EFSMs*. Pour illustrer cette transformation, prenons l'exemple de la spécification *SDL* de la figure 7.1.a. Après l'application de la procédure de normalisation, une nouvelle spécification équivalente est obtenue. Dans cette dernière, la structure *save* est éliminée et elle est remplacée par un ensemble de transitions (figure 7.1.b). À partir de la spécification résultante, on peut facilement dériver l'*EFSM* associé (figure 7.1.c).

Comme la procédure de normalisation est systématique, nous supposerons que toutes les spécifications *SDL* que nous utilisons sont normalisées. Ceci nous permet de travailler indifféremment sur la spécification *SDL* ou l'*EFSM* associé.

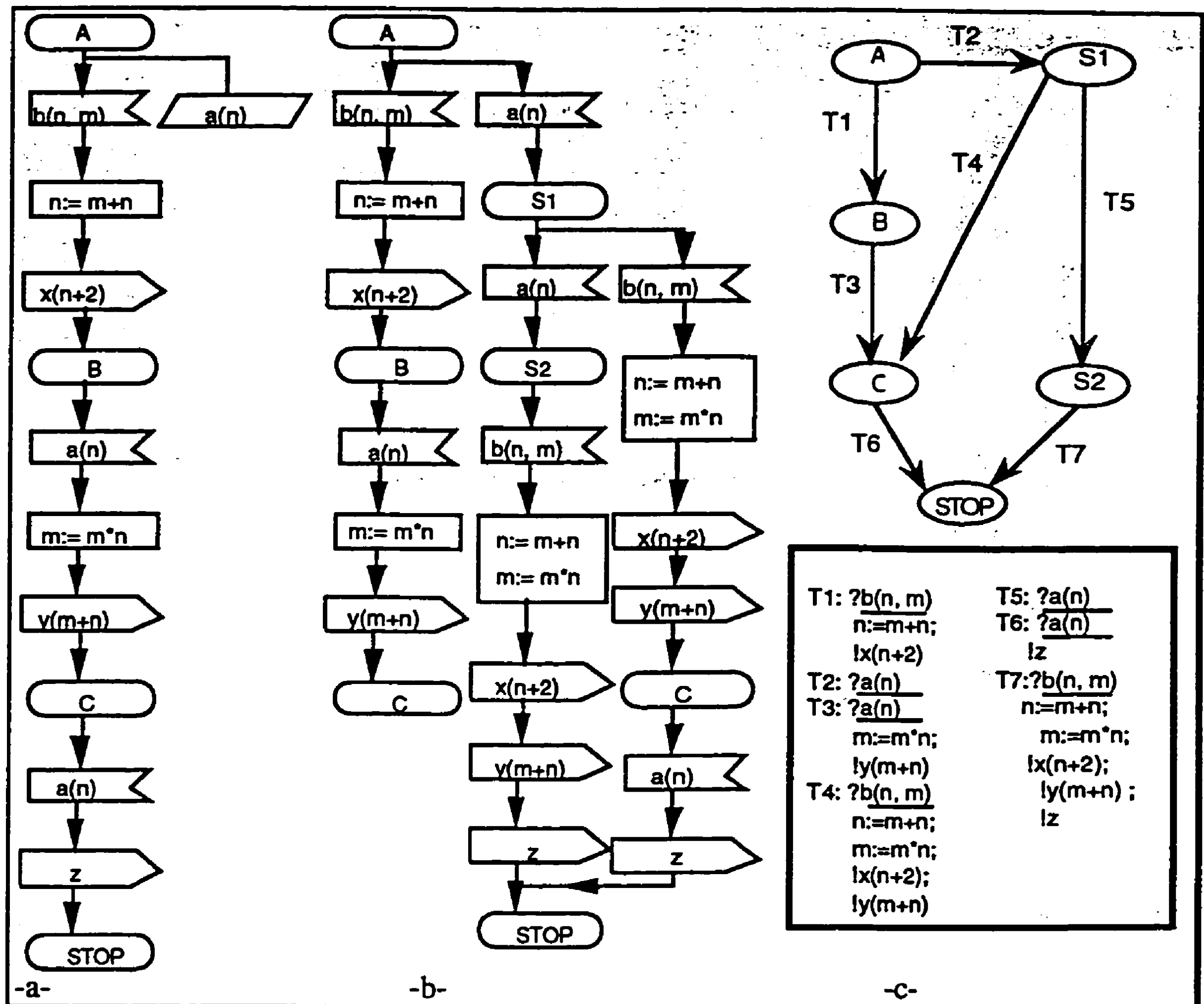


Figure 7.1. Procédure de transformation d'une spécification *SDL* en un *EFSM*.

7.2.3. Les tests de flux de données

Les tests de flux de données sont utilisés pour vérifier l'exactitude des différentes valeurs que peuvent prendre les variables d'un système. Pour des systèmes de taille réelle, ces tests ne peuvent pas vérifier l'exactitude de toutes les valeurs de variables. La cardinalité du domaine des variables est parfois trop importante (infini des fois). Certains auteurs , [Ural 86a], [Sari 93], [Weyu 93], [Huan 95] et [Rama 95] ont proposé de tester quelques propriétés spécifiques associées au flux de données du logiciel. Ces propriétés sont basées sur un ensemble de règles appelées *critères de test*. Chaque critère de test est caractérisé par une couverture particulière des aspects de flux de donnée des protocoles. Weyuker [Weyu 93] a proposé une classification hiérarchique de la couverture de ces

critères. La figure 7.2 illustre cette classification, les noeuds de l'arbre représentent les critères de test et les arcs orientés montrent une relation entre les noeuds fils et les noeuds pères. Cette relation signifie que si une suite de test satisfait le critère père, elle satisfait nécessairement le fils. En d'autres termes les chemins couverts par le critère du noeud fils sont inclus dans ceux du noeud père.

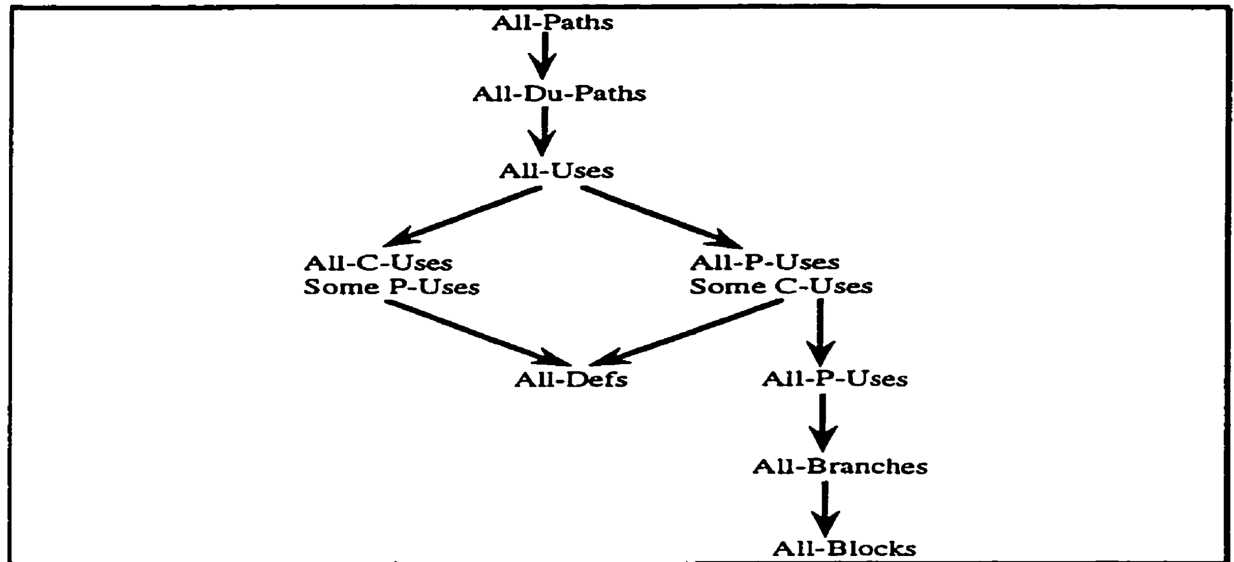


Figure 7.2. Critères de test du flux de données.

Nous présentons ci-dessous, brièvement les critères de test du flux de données tels qu'ils sont présentés par Rapps [Rapp 85]. Nous détaillerons un nouveau critère appelé *DO-paths* [Huan 95] que nous utiliserons plus tard pour la génération des suites de test. Avant cela, nous commençons par présenter deux définitions (2.1 et 2.2) qui nous seront utiles pour la description des critères de test du flux de données.

- **Définitions**

Définition 7.1: l'utilisation d'une variable x dans une transition t est dite en *Def-use* si elle est en *A-use* ou en *I-use*. x est dite en *A-use* dans t si x apparaît dans la partie gauche d'une instruction d'assignation de l'une des actions de t. x est en *I-use* si x apparaît dans l'interaction d'entrée de t. x est en *P-use* si elle apparaît dans le prédicat de t. x est en *O-use* si x apparaît dans l'interaction de sortie de t. x est dite en *C-use* dans t si x apparaît dans la partie droite d'une instruction d'assignation de l'une des actions de t. q

Définition 7.2: Une séquence de transitions $T_1, ..., T_k$ est dite un *chemin def-clear* par rapport à x si les transitions $T_2, ..., T_{k-1}$ ne contiennent aucun *Def-use* de x. q

Critères de test de flux de données

all-Nodes: les cas de test couvrent toutes les instructions du programme.

all-Edges: les cas de test couvrent toutes les branches de contrôle du programme.

all-Defs: les cas de test sont des chemins *def-clear* qui couvrent toutes les instructions du programme où des variables sont en *Def-use.*

all-P-uses: les cas de test couvrent tous les chemins du programme dans lesquels il existe un sous-chemin *def-clear* commençant par un *Def-use* d'une variable x et finissant par un *P-use* de la même variable.

all-C-uses/some-P-uses: les cas de test couvrent tous les chemins du programme dans lesquels il existe un sous-chemin *def-clear* commençant par un *def-use* d'une variable x et finissant par un *C-use* de la même variable. Si dans un chemin une variable est définie mais n'est jamais utilisée en *C-use* on prendra un sous-chemin qui se termine par un *P-use* de la variable.

all-P-uses/some-C-uses: les cas de test couvrent tous les chemins du programme dans lesquels il y a un sous-chemin *def-clear* commençant par un *def-use* d'une variable et finissant par un *P-use* de la variable. S'il existe une variable qui est définie mais qui ne sera jamais utilisée en *P-use* on prendra un sous-chemin qui se termine par un *C-use* de la variable.

all-Uses: les cas de test couvrent tous les chemins du programme comprenant un sous-chemin *def-clear* commençant par un *Def-use* d'une variable x et finissant par un *P-use* ou par un *C-use* de la même variable.

all-DU-paths: ce critère est le même que *all-Uses*, sauf qu'on prendra en considération tous les sous-chemin *def-clear* sans boucle menant d'un *Def-use* à un *P-use* ou un *C-use* d'une variable x .

all-paths: les cas de test couvrent tous les chemins du programme.

DO-paths: les *DO-paths* [Huan 95] sont un ensemble de chemins de la spécification qui commencent par une transition où certaines variables sont en *A-use* ou en *I-use* et se terminent par une transition où il y a une interaction de sortie dont les variables dépendent de l'assignation qui a été faite dans la première transition. Les Définitions 2.1, 2.2, et 2.3 décrivent formellement les *DO-paths* [Huan 95]. Dans [Huan 95], les auteurs ont proposé une méthode originale pour générer un ensemble de séquence de test qui couvre tous les *DO-paths* exécutables d'un *EFSM*. Chaque cas de test est constitué de trois parties: le préambule, le *DO-path*, et le postambule, c'est-à-dire une séquence d'interactions d'entrée qui permet d'accéder à l'état de départ du *DO-path* à tester, d'exécuter le *DO-path*, et de retourner à l'état initial.

Dans la section 7.3, nous montrerons comment il est possible de réduire la longueur de la suite de test en assignant certaines transitions non-spécifiées appelées *don't care*.

Définition 7.3: Une séquence de transitions $T_1, \ldots, T_k$ est dite un chemin *DO-path* par rapport à une variable x si x est une variable globale qui est en *Def-use* dans T_1 et soit: (1) x est en O-use dans T_k et la séquence $T_1, \ldots, T_k$ est *def-clear* par rapport à x ou (2) T_k où $2 \leq k \leq n-1$, contient un P-use de x, et la séquence $T_1, \ldots, T_k, \ldots, T_{n-1}$ est *def-clear* par rapport à x, tel que (i) il y a un O-use d'une variable dans T_n et (ii) l'exécution de T_n dépend de l'exécution de T_k q

7.3. Amélioration de la testabilité en complétant la spécification

Dans le chapitre 6, nous avons proposé une nouvelle approche d'analyse, d'évaluation et d'amélioration de la testabilité basée sur l'extension des spécifications. Cette approche ne traite que des aspects de flux de contrôle. Pour pouvoir appliquer cette approche aux EFSMs, il faudrait transformer l'EFSM en FSM. Cette transformation peut être faite de deux manières:

- faire abstraction des données et ne considérer que l'aspect contrôle de la spécification [Boch 97]. Pour extraire les cas de test, on utilise les méthodes usuelles des FSMs en les complétant manuellement de façon à assurer leur exécutabilité.

- déplier les états et les transitions de l'EFSM en prenant en considération les données et les valeurs des variables et des paramètres utilisés [Faro 90]. Pour des cas réels, cette transformation induit une explosion combinatoire des états et des transitions de l'EFSM. Les cas de tests générés à partir de la nouvelle FSM sont automatiquement exécutables.

Dans les sections qui suivent, nous allons compléter l'approche du chapitre 6 en considérant les aspects de flux de données. L'approche que nous proposons peut être considérée comme une amélioration de certains travaux [Dsso 91a], [Sale 92] et [Kim 95]. Ces derniers proposent de rajouter de nouvelles primitives d'entrée et de sortie à la description du protocole en vue d'améliorer sa testabilité. Ces primitives sont utilisées pour améliorer l'observabilité et la contrôlabilité du protocole. Pour l'observabilité, ces primitives sont généralement utilisées pour rapporter l'état interne du protocole ou pour faire la trace des transitions exécutées. En ce qui concerne la contrôlabilité, ces primitives sont utilisées pour mettre le protocole dans un état particulier ou pour choisir la prochaine transition à exécuter. Kim et Chanson [Kim 95] ont utilisé ce genre de primitives comme un interrupteur qui active ou désactive les options d'observabilité ou de contrôlabilité rajoutées. L'inconvénient d'une telle approche c'est qu'elle étend le comportement du protocole. Cependant, notre méthode permet d'améliorer la testabilité d'un système en complétant le comportement non spécifié. Cette approche ressemble à celle de Kim et Chanson [Kim 95], elle étend le comportement du protocole, sans pour autant rajouter de nouvelles primitives.

Nous considérons que le coût et la complexité du processus de test sont inversement proportionnels à la longueur de la suite de test [Petr 93a]. Nous proposons de compléter certaines transitions non-spécifiées de la spécification afin de raccourcir la longueur de la suite de test. Pour éviter que nos extensions puissent se répercuter sur l'utilisation normale du protocole (c'est-à-dire qu'un utilisateur non averti puisse activer de telles transitions), nous proposons les approches décrites dans la section suivante.

7.3.1. Approches pour maintenir le comportement normale du protocole

Première approche

Certaines séquences d'entrée ne peuvent jamais être appliquées à l'entrée d'un module appartenant à un système plus large. Ce cas est typique aux modules encapsulés dans un certain environnement, c'est-à-dire les modules qui reçoivent leurs entrées à partir

des sorties d'autres modules (exemple: architecture en couches). Ceci implique que certaines transitions spécifiées ou non ne pourront jamais être exécutées. Dans une réalisation physique d'un tel type de module, l'assignation de ces transitions n'est pas importante. Les transitions non-spécifiées de ce genre sont appelées *don't care*. Le concepteur est libre de les arranger de manière à satisfaire certains critères de qualité prédéfinis sans pour autant changer le comportement du système. Dans notre cas, le critère qui nous interresse est la testabilité des systèmes. Nous avons choisi d'assigner ces transitions pour écourter la suite de test d'un protocole. Notre approche prend en considération le cas où le module peut être testé en isolation, c'est-à-dire on ne peut tester le module qu'en utilisant une architecture locale (voir chapitre 2).

Seconde approche

La seconde approche consiste aussi à compléter des transitions non-spécifiées de la spécification. La différence est que cette dernière ne pose aucune hypothèse sur l'environnement d'utilisation du module. L'assignation des transitions est faite en fonction de deux modes d'utilisation du système [Kim 95]: le mode test et le mode opérationnel. L'exécution des transitions nouvellement assignées est contrôlée par une variable booléenne appelée *test-guard* qui agit comme un interrupteur pour passer d'un mode à un autre. Lors du mode de test, nous assignons à cette variable la valeur *vraie* pour permettre d'exécuter les nouvelles transitions. Par contre, pour le mode opérationnel, nous assignons à cette variable la valeur *faux*, ce qui désactivera ces transitions pour qu'ils deviennent des *self-loops* sans sorties ni actions (figure 7.3). Cette seconde approche permet de compléter toutes les transitions non-spécifiées. L'avantage de cette méthode est qu'elle permet au module de ne pas être bloqué lors de la réception d'événement imprévus par la spécification initiale. Lors de l'implantation d'un module, on fait toujours de sorte que le comportement soit complètement spécifié. Cela évite le blocage du module suite à la réception d'un événement non prévu par la spécification initiale. L'approche proposée délivre donc au programmeur une spécification qui est déjà complète et il ne lui reste qu'à l'implanter comme elle est.

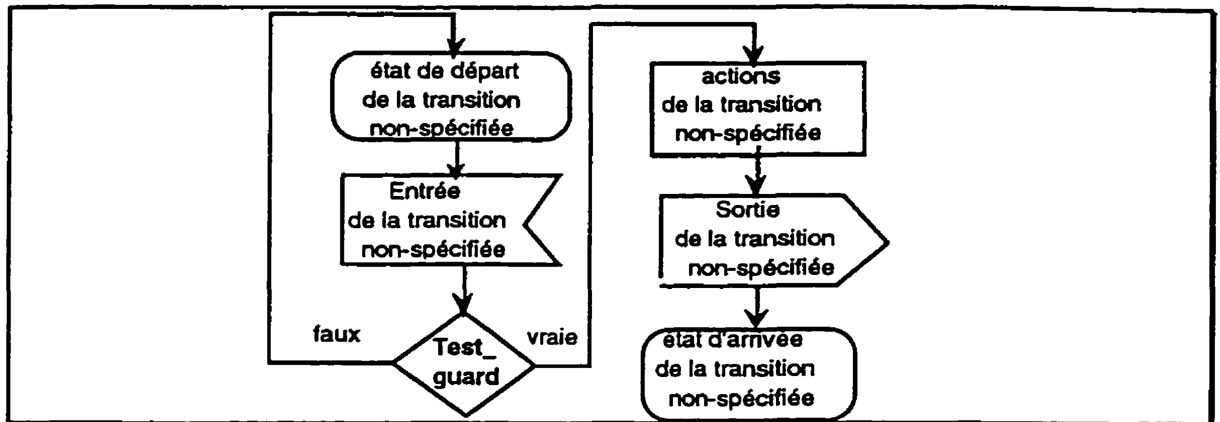


Figure 7.3. Méthode d'assignation des transitions non-spécifiées en *SDL*.

7.3.2. Avantages de ces approches

Comparativement aux méthodes précédantes [Sale 92] et [Kim 95], notre méthode a les avantages suivants:

- Elle ne nécessite pas l'utilisation de nouvelles primitives d'entrée qui n'appartiennent pas au domaine des entrées.
- Elle donne lieu à des modules qui ne bloquent pas lors de la réception d'événements non prévus par la spécification initiale.

Dans les sections qui suivent, nous proposons une technique pour compléter un *EFSM* afin de réaliser les deux approches précédantes.

7.3.3. Technique utilisée pour compléter un *EFSM*

- **Description formelle de la technique**

Une même séquence de transitions α, avec les états de départ s et d'arrivée s', peut être incluse dans plusieurs cas de test d'un *EFSM*. Sous certaines conditions, α peut être remplacée par une nouvelle transition t, de s à s', tel que l'exécution de α ou de t à partir d'une même configuration d'état $(s, b_1,..., b_k)$ mène à une même configuration d'état $(s', b'_1,..., b'_k)$. Au cours du processus de test, l'utilisation de t à la place de α est très bénéfique. Elle rend les tests plus courts et permet d'accéder plus rapidement aux parties du protocole accessibles via la configuration d'état $(s', b'_1,..., b'_k)$.

La technique que nous proposons consiste à assigner certaines des transitions *I-undefined* de façon à remplacer certaines sous-séquences redondantes des cas de test. Après l'assignation des transitions, nous évaluerons l'amélioration de la testabilité en comparant les longueurs des suites de test de l'E*FSM* initial et de L'E*FSM* augmenté.

Nous utiliserons les *DO-paths* (voir section 7.2.3) comme critère de génération des suites de test. Les deux solutions proposées dans la section 7.3.1 assurent que les transitions nouvellement assignées ne seront jamais exécutées dans le fonctionnement normal du protocole. Nous supposons que le module contenant les transitions rajoutées peut être testé en isolation (architecture locale). Le testeur supérieur et inférieur sont spécifiés de façon à pouvoir profiter des nouvelles transitions. Avant de donner plus de détail sur la technique, nous présentons deux définitions (Définitions 3.1 et 3.2), qui nous sont utiles pour le reste de ce chapitre.

Définition 7.4: Soient $E = (S, s_o, X, Y, T)$ et $E' = (S', s_o, X, Y, T')$ deux *EFSMs*, E' est appelée ***supermachine*** de E si $T \subseteq T'$ et $S \subseteq S'$, E est appelée *sousmachine* de E'. q

Définition 7.5: Soit E un *EFSM*, deux séquences de transitions α et β avec le même état de départ s sont dits *interchangeables* par rapport à la configuration d'état $(s, b_1, ..., b_k)$ si leur exécution à partir de $(s, b_1, ..., b_k)$ mène à une même configuration d'état q

La technique consiste à assigner chaque transition *I-undefined t interchangeable* avec une sous-séquence α ayant le même état de départ. Plus α est longue et plus elle est utilisées dans les cas de test, plus l'assignation de transitions permet d'améliorer la testabilité. L'exécution de t doit produire la même configuration d'état que α. Ceci n'est pas toujours possible et il faut poser certaines hypothèses sur t et α..

Hypothèses

Soit $\alpha = T_1, ..., T_n$ et I_j $(1 \leq j \leq n)$ l'ensemble des variables d'entrée utilisés dans les interactions d'entrée d'une transition T_j, soit i l'ensemble des variables d'entrée utilisés dans l'interaction d'entrée de la transition t:

- l'interaction d'entrée de la transition t: doit avoir le même ensemble de variables d'entrée que α soit $i = \bigcup_{j=1}^{n} I_j$,
- chaque variable d'entrée utilisée dans une des transitions de α doit être utilisée au plus une fois dans une intéraction d'entrée: $\bigcap_{j=1}^{n} I_j = \emptyset$.

Construction d'une transition à assigner

Soit $\alpha = T_1,..., T_n$ une séquence de transitions avec l'état de départ s, l'état d'arrivée s' et la configuration initiale $(b_1,..., b_k)$ en s. Soit a_j (resp. y_j) la procédure d'actions (respectivement l'interaction de sortie) de la transition T_j $(1 \leq j \leq n)$. Soit L la liste des variables utilisées en *Def-use* dans α. Soit t une transition *I-undefined* par rapport à l'interaction d'entrée x et avec le même état de départ que α. Sous les hypothèses décrites précédemment, nous construisons t comme suit: $t=(s, s', x, y^{\#}, p_0, a)$ où s' est l'état d'arrivée de α, p_0 est le prédicat identité si on implante la première approche (section 7.3), sinon p_0 est le test-guard (figure 7.3), $y^{\#}$ est la séquence de toutes les interactions de sortie de α. Dans le cas où $y^{\#}$ ne contienne aucune intéraction de sortie, nous lui rajouterons une intéraction de sortie spéciale de façon à ce que t deviennent un *DO-path*. Ceci permet de générer un cas de test spécifique pour la vérification de t. La procédure d'actions a est choisie de façon à mener à la même configuration d'arrivée que α. La façon la plus simple de construire a est de concaténer les différentes actions des transitions de α dans le même ordre de leur apparition dans α. L'exécution de a remplacera l'exécution séquentielle de tous les actions de α. a est construite comme suit:

Soit *actions* = $\boxed{\mathit{sequencing}}_{j=1}^{n} a_j$, la concaténation de toutes les procédures d'actions de $T_1,..., T_n$. Soit $\mho(actions, x, b_1,..., b_k)$ une fonction qui donne les nouvelles valeurs des variables $(v_1,..., v_k)$ suite à l'application de *actions* et la réception de l'interaction d'entrée x. La procédure d'actions a de la transition t sera une liste d'assignations $v_1 := \mho^1(actions, x, b_1,..., b_k) ... v_k := \mho^k(actions, x, b_1,..., b_k)$, où $\mho^i(actions, x, b_1,..., b_k)$ est la transformation de valeur de la variable v_i suite à l'application de *actions* et la réception de l'interaction d'entrée x.

Il est évident qu'une transition t, construite comme décrit précédemment, est *interchangeable* avec $\alpha = T_1,..., T_n$.

- **Algorithme**

L'amélioration de la longueur de la suite de test peut être accomplie en agissant sur l'une des trois parties de chaque cas de test, c'est-à-dire, le préambule et/ou le DO-path et/ou le postambule. Dans cette section, nous prenons le cas de l'amélioration de la longueur des préambules en complétant une spécification partielle. L'algorithme substitue (chaque fois que c'est possible) chaque sous-séquence α, appartenant à l'ensemble des préambules, par une transition qui lui est *interchangeable*. Le même algorithme peut être légèrement modifié pour raccourcir les ensembles prédéfinis de postambule ou de DO-paths.

Algorithme. Augmentation d'un *EFSM E*.
Input. Un *EFSM E*, un ensemble de préambules, et un ensemble *DT* des transitions à assigner.

Début
E'<--E; /* affectation de E à E'
seq <-- séquence vide;
Initialisation des variables seq et Trans_ajoutée;
Pour chaque préambule T_i, T_{i+1},...T_{j-1}, T_j
Faire
Pour k=i à j
Faire
Si DT=$\emptyset$ Alors exit
Sinon
Si l'état de départ de T_k appartient à un état de départ d'une transition t de DT
Alors
Par <-- les variables d'entrée t
I <-- I_k; /* les variables d'entrée de T_k */
I_{k+1} <-- les variables d'entrée de T_{k+1} ;
Act <-- Act_k /*la procédure d'actions de T_k */
l <--k;
Tantque $I \cap I_{l+1} = \emptyset$ et $I \cup I_{l+1} \subseteq Par$
Alors
I<-- $I \cup I_{l+1}$;
Act <-- Act *sequencing* Act_{l+1};
l<--l+1;
I_{l+1} <-- les variables d'entrée de T_{l+1} ;
FinTantque
Si (l-k)$\geq$2
Alors
t.état_départ <-- état de départ de T_k;
t.état_arrivée <-- état d'arrivée de T_l;
t.param <-- I;
t.Action <-- Act

```
                Si seconde approche
                Alors
                        t.condition <-- Test_guard
                FinSi
                Trans_ajoutée <-- Trans_ajoutée ∪t
                DT <-- DT - t; k<-- l
         FinSi
        FinSi
        FinSi
        seq<-- substituer(preambule, Tk...Tl, t); /* remplacer Tk...Tl par t */
        k<--k+1
  FinPour
  prochain préambule
 FinPour
Fin
```

- **Gain de testabilité**

Comme nous l'avons déjà précisé, le même algorithme peut être modifié pour raccourcir aussi bien les préambules que les postambules ou les *DO-paths*. Pour pouvoir utiliser une nouvelle transition *t*, il faut d'abord la tester. La façon dont elle a été construite génère un nouveau *DO-path* constitué uniquement de *t*. Pour tester *t*, on a besoin de générer un nouveau cas de test *tc* en utilisant la même méthode des *DO-paths* [Huan 95]. Ceci veut dire que, l'ajout d'une transition *t* n'est pas sans conséquences, en éliminant la redondance dans les cas de test, on a besoin d'un nouveau cas de test *tc*. Nous laisserons aussi un des plus courts cas de test inchangé pour tester α. Pour évaluer la rentabilité de l'assignation d'une transition nous proposons une nouvelle formule appelée *gain*:

$$gain = p.(longueur(\alpha)-1)-longueur(tc). \qquad (1)$$

où p est le nombre de fois que α a été utilisée comme sous-séquence dans l'ensemble des cas de test initial.

Plus la valeur de p et de $longueur(\alpha)$ sont importants, mieux est le gain de testabilité. Pour des spécifications indéterministes, nous savons qu'elles sont généralement moins testables que les spécifications déterministes. Dans ce cas, l'un des facteurs qui influence les tests est l'effort nécessaire pour atteindre les *DO-paths*. Si l'*EFSM* est indéterministe, une séquence α peut mener à plus qu'un état valide et produire plus qu'une séquence de sortie valide, tandis qu'une transition interchangeable *t* ne mène qu'à un seul état et ne produit qu'une seule séquence de sortie. Dans ce cas, le gain est

double: on diminue la longueur des cas de test (Formule (1)) et on facilite l'accès aux parties du protocole à tester. Concernant le deuxième côté du gain, nous savons que l'effort nécessaire pour atteindre un état s' dépend de deux facteurs: la longueur de la séquence α qui mène à s'' et le nombre de séquences de sorties produites. Dans un travail précédant [Karo 96], nous avons intégré ces deux facteurs dans une formule unique que nous avons appelé degré de contrôlabilité (voir chapitre 6). Cette formule est basée sur la notion de poids de l'ensemble d'accessibilité [Dsso 95b] et [Karo 96]. Nous pouvons étendre cette formule pour le cas d'une séquence de transitions α ayant comme état de départ s et d'arrivée s':

$$\omega(s,\alpha, s')=longueur(\alpha).sorties(s,\alpha) \qquad (2),$$

où, $sorties(s,\alpha)$ est le nombre de séquences de sorties produite lorsque α est appliquée à l'état s. Cette formule est utile pour mesurer combien il est difficile d'atteindre un état s' à partir d'un état s en utilisant la séquence α. Plus ce poids est grand plus les tests des parties du protocole accessibles via s' sont difficiles. Dans notre cas, il est évident qu'en utilisant une transition unique à la place de α, on diminue le poids. L'évaluation des Formules (1) et (2) à l'avance peuvent nous aider à décider si la transition à assigner est rentable.

7.4. Application

Dans cette section, nous allons appliquer notre algorithme sur l'*EFSM E* [Huan 95] de la figure 7.4. La spécification *SDL* associée à cet *EFSM* est donnée en annexe B. Les séquences de test respectant le critère de *DO-paths* de cet *EFSM* sont données dans la Table 7.1 [Huan 95] (colonne de gauche). La longueur de ces séquences est de 271.

Pour l'amélioration de la testabilité, nous adopterons la seconde approche de la section 7.3, puisque nous n'avons aucune information sur l'environnement d'utilisation du protocole. Sous les hypothèses précédantes, la transition t avec l'interaction d'entrée *U.DATA(SDU, segment, blockbound)* à partir de l'état *Idle* est *I-undefined.* Nous supposerons que t soit l'unique transition à assigner. Notre algorithme nous permet de construire la transition *t interchangeable* avec (*T1, T2, T3*):

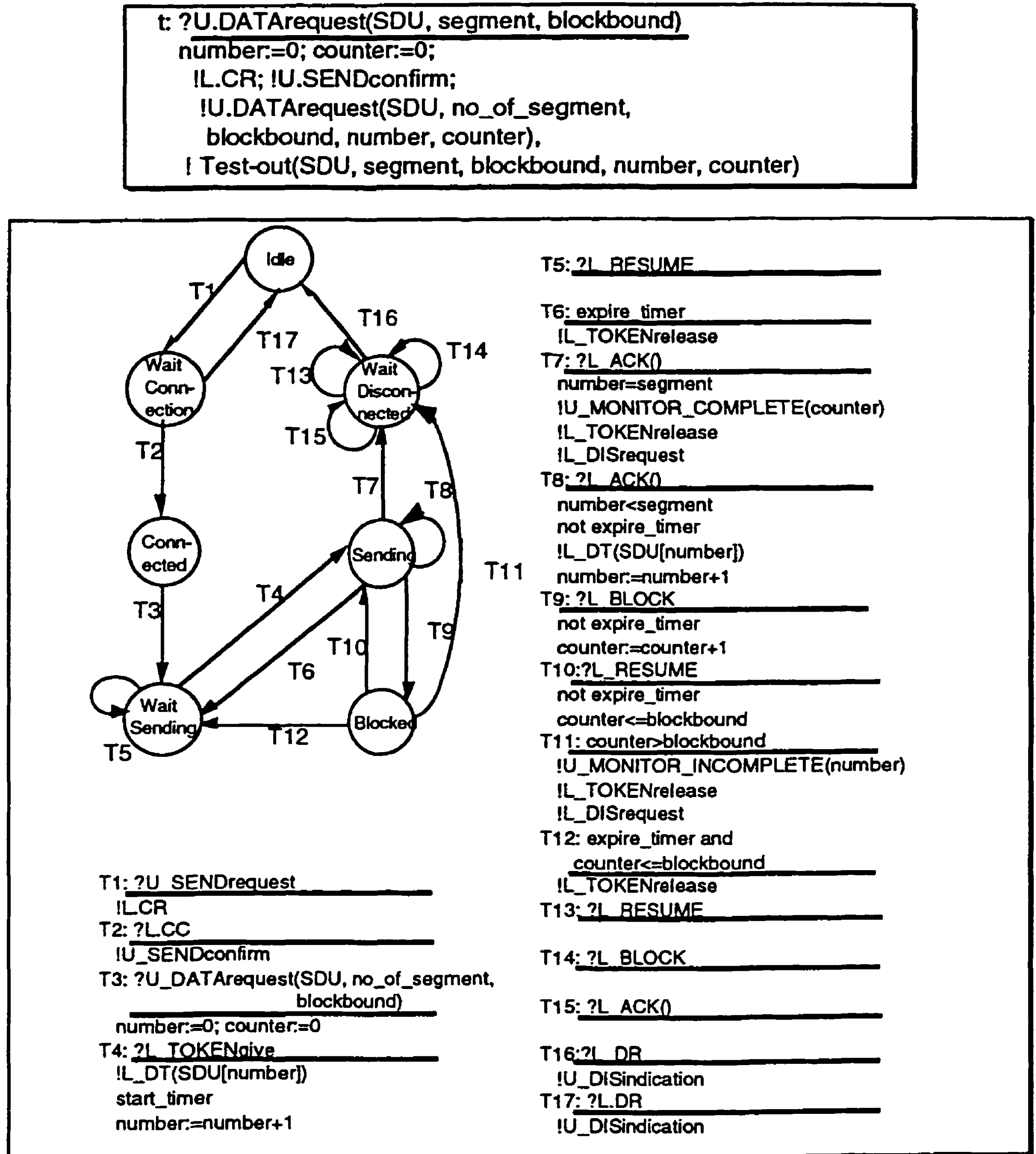

t: ?U.DATArequest(SDU, segment, blockbound)
number:=0; counter:=0;
!L.CR; !U.SENDconfirm;
!U.DATArequest(SDU, no_of_segment,
blockbound, number, counter),
! Test-out(SDU, segment, blockbound, number, counter)
Idle
Wait Conn-ection
Conn-ected
Wait Sending
Sending
Blocked
Wait Discon-nected
T1
T2
T3
T4
T5
T6
T7
T8
T9
T10
T11
T12
T13
T14
T15
T16
T17
T5: ?L_RESUME
T6: expire_timer
!L_TOKENrelease
T7: ?L_ACK()
number=segment
!U_MONITOR_COMPLETE(counter)
!L_TOKENrelease
!L_DISrequest
T8: ?L_ACK()
number<segment
not expire_timer
!L_DT(SDU[number])
number:=number+1
T9: ?L_BLOCK
not expire_timer
counter:=counter+1
T10:?L_RESUME
not expire_timer
counter<=blockbound
T11: counter>blockbound
!U_MONITOR_INCOMPLETE(number)
!L_TOKENrelease
!L_DISrequest
T12: expire_timer and
counter<=blockbound
!L_TOKENrelease
T13: ?L_RESUME
T14: ?L_BLOCK
T15: ?L_ACK()
T16:?L_DR
!U_DISindication
T17: ?L.DR
!U_DISindication
T1: ?U_SENDrequest
!L.CR
T2: ?L.CC
!U_SENDconfirm
T3: ?U_DATArequest(SDU, no_of_segment,
blockbound)
number:=0; counter:=0
T4: ?L_TOKENgive
!L_DT(SDU[number])
start_timer
number:=number+1

Figure 7.4. Exemple de spécification décrite sous forme d'un *EFSM* .

Si *t* est assignée, nous aurons besoin d'un cas de test *tc* pour la vérifier: *tc= t.T4.T7.T16*. En plus, nous garderons le cas de test le plus court *(T1, T2, T3, T4, T8, T7, T16)* inchangé pour le test de *(T1, T2, T3)*.D'autre part, la séquence *(T1, T2, T3)*

est utilisée 23 fois dans la partie préambule des cas de test (voir table 7.1). Lors des tests, si *t* est utilisée à la place de (*T1, T2, T3*), nous améliorerons la testabilité, puisque le gain (Formule 1) est positif:

gain = 23(3-1)-4 = 42 > 0. La formule 2 n'est pas applicable puisque la spécification est déterministe. En assignant une seule transition nous avons gagné plus que le 1/6 du coût des tests (effort de test). L'algorithme substitue chaque occurrence de (*T1, T2, T3*) par *t*. Après avoir appliquer l'algorithme, nous obtenons de nouvelles séquences de test représentée dans la deuxième colonne de la table 7.1. La nouvelle longueur de cette suite de test est 229 au lieu de 271. Dans *SDL*, la transition *t* est rajoutée avec la variable booléenne *test_guard* pour contrôler son exécutabilité. La figure 7.5 illustre la partie de la spécification (*SDL* et *EFSM*) où la transition *t* est rajoutée.

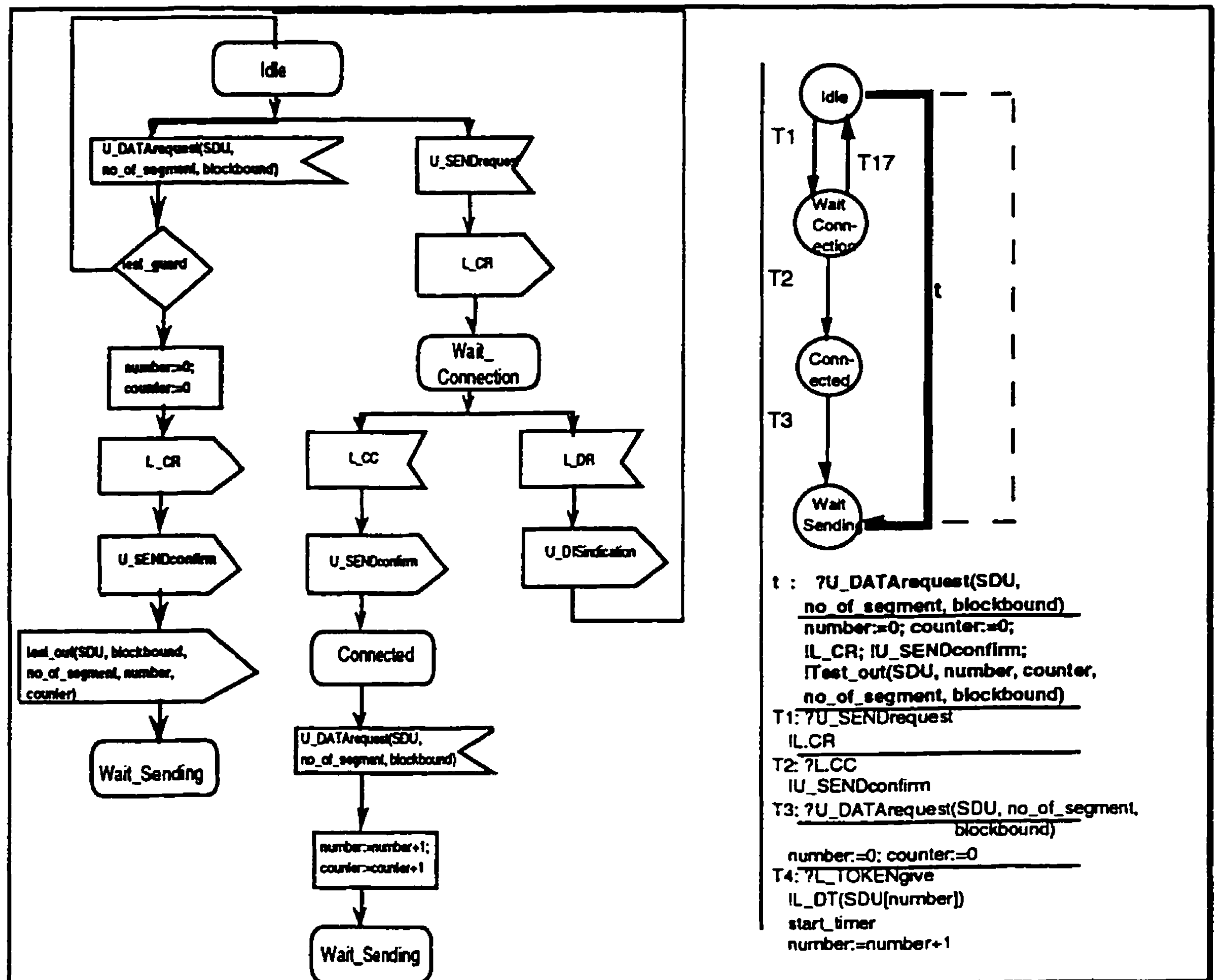


Figure 7.5. Assignation d'une transition *I-undefined.*

N°	Executable Test Sequences	Ext. Executable TestSequences
1	(T1, T2, T3, T4, T8, T7, T16)	(T1, T2, T3, T4, T8, T7, T16) /* vérification de α
2	(T1, T2, T3, T4, T6, T4, T7, T16)	(t, T4, T6, T4, T7, T16)
3	(T1, T2, T3, T4, T6, T4, T9, T10, T7, T16)	(t, T4, T6, T4, T9, T10, T7, T16)
4	(T1, T2, T3, T4, T6, T4, T9, T10, T9, T10, T7, T16)	(t, T4, T6, T4, T9, T10, T9, T10, T7, T16)
5	(T1, T2, T3, T4, T6, T4, T9, T10, T9, T10, T9, T11, T16)	(t, T4, T6, T4, T9, T10, T9, T10, T9, T11, T16)
6	(T1, T2, T3, T4, T8, T9, T10, T7, T16)	(t, T4, T8, T9, T10, T7, T16)
7	(T1, T2, T3, T4, T8, T9, T10, T9, T10, T7, T16)	(t, T4, T8, T9, T10, T9, T10, T7, T16)
8	(T1, T2, T3, T4, T8, T9, T10, T9, T10, T9, T11, T16)	(t, T4, T8, T9, T10, T9, T10, T9, T11, T16)
9	(T1, T2, T3, T4, T9, T10, T9, T10, T9, T11, T16)	(t, T4, T9, T10, T9, T10, T9, T11, T16)
10	(T1, T2, T3, T4, T8, T9, T10, T6, T4, T9, T10, T9, T11, T16)	(t, T4, T8, T9, T10, T6, T4, T9, T10, T9, T11, T16)
11	(T1, T2, T3, T4, T8, T9, T10, T9, T10, T6, T4, T9, T11, T16)	(t, T4, T8, T9, T10, T9, T10, T6, T4, T9, T11, T16)
12	(T1, T2, T3, T4, T8, T9, T10, T9, T12, T4, T9, T11, T16)	(t, T4, T8, T9, T10, T9, T12, T4, T9, T11, T16)
13	(T1, T2, T3, T4, T9, T10, T6, T4, T7, T16)	(t, T4, T9, T10, T6, T4, T7, T16)
14	(T1, T2, T3, T4, T9, T10, T9, T10, T6, T4, T7, T16)	(t, T4, T9, T10, T9, T10, T6, T4, T7, T16)
15	(T1, T2, T3, T4, T9, T10, T8, T9, T10, T7, T16)	(t, T4, T9, T10, T8, T9, T10, T7, T16)
16	(T1, T2, T3, T4, T9, T10, T8, T9, T10, T9, T11, T16)	(t, T4, T9, T10, T8, T9, T10, T9, T11, T16)
17	(T1, T2, T3, T4, T9, T10, T9, T10, T8, T7, T16)	(t, T4, T9, T10, T9, T10, T8, T7, T16)
18	(T1, T2, T3, T4, T9, T10, T9, T10, T8, T9, T11, T16)	(t, T4, T9, T10, T9, T10, T8, T9, T11, T16)
19	(T1, T2, T3, T4, T9, T12, T4, T7, T16)	(t, T4, T9, T12, T4, T7, T16)
20	(T1, T2, T3, T4, T9, T12, T4, T9, T10, T6, T4, T9, T11, T16)	(t, T4, T9, T12, T4, T9, T10, T6, T4, T9, T11, T16)
21	(T1, T2, T3, T4, T9, T12, T4, T9, T10, T7, T16)	(t, T4, T9, T12, T4, T9, T10, T7, T16)
22	(T1, T2, T3, T4, T9, T12, T4, T9, T10, T9, T11, T16)	(t, T4, T9, T12, T4, T9, T10, T9, T11, T16)
23	(T1, T2, T3, T4, T9, T10, T9, T12, T4, T7, T16)	(t, T4, T9, T10, T9, T12, T4, T7, T16)
24	(T1, T2, T3, T4, T9, T10, T9, T12, T4, T9, T11, T16)	(t, T4, T9, T10, T9, T12, T4, T9, T11, T16)
25	Test case added for the test of t	(t, T4, T7, T16) /* test de *t*

Table 7.1. table des séquences de test avant et après l'assignation de la transition.

Après avoir extrait la nouvelle suite de test, nous pouvons utiliser l'outil *SDT* (*Specification Description Tool*) pour générer une implémentation (le code *C*) de la spécification étendue. *SDT* est un outil qui permet d'éditer, de vérifier (syntaxe et sémantique), de simuler, de valider et de générer le code à partir d'une spécification *SDL*. La génération du code est faite avec la valeur de la variable *test_guard* à *vrai*, ce qui permet d'appliquer les cas de test. Si les tests sont concluants, on remet *test_guard* à *faux* et on regénère le code définitif.

7.5. Conclusion

Dans ce chapitre, nous avons présenté deux méthodes qui permettent d'étendre une spécification *SDL* pour améliorer sa testabilité. L'avantage de ces approches est qu'elles ne changent pas le comportement du protocole. Les extensions sont uniquement

utilisées au cours du processus de test, après cela, ils seront transparent dans le fonctionnement normal du protocole. Nous avons suggéré un algorithme qui, sous certaines hypothèses, améliore la longueur des cas de test.

Chapitre 8

Influence des facteurs de testabilité sur le diagnostic[1]

8.1. Introduction

Le problème de diagnostic et de localisation des fautes a été peu étudié dans le domaine des protocoles de communication [Vuon 91] et [Ghed 93d]. Cependant, il a été largement utilisé dans d'autres domaines (diagnostic), tel que l'intelligence artificielle, la médecine et la mécanique. En fonction du système à diagnostiquer, différentes techniques et méthodes peuvent être employées. Pour les systèmes représentés par des *FSMs*, plusieurs techniques de détection de fautes existent déjà (voir chapitre 6). Ces mêmes méthodes peuvent être exploitées pour le diagnostic. Ces méthodes diffèrent par leur capacité respective de localisation de fautes.

Les séquences de test générées par les méthodes basées sur les FSMs (voir chapitre 6) ne garantissent pas, en général, la localisation des fautes. Pour cela, des tests supplémentaires de diagnostic sont requis. Il est important de noter que les méthodes de test avec une meilleure couverture de fautes peuvent requérir moins de tests additionnels de diagnostic.

Dans ce chapitre, nous étudierons l'effet de certains facteurs qui influencent les tests [Karo 96] sur le diagnostic. Ces facteurs n'affectent pas de la même façon les tests le diagnostic. Nous adapterons ces facteurs aux exigences du diagnostic. Finalement, nous proposerons certaines idées de conception qui permettent de faciliter le processus de diagnostic.

---

[1] Les résultats de ce chapitre sont publiés dans: le 8th Annual Oregon Workshop on Software Metrics, Coeur d'Alene, Idaho USA, 11-12 Mai 1997 [Karo 97d], Soumis à METRICS'97, Albuquerque, New Mexico USA, 5-7 Novembre 97 [Karo 97b],Publication départementale no 1048.

8.2. Le diagnostic

Pour effectuer un diagnostic, nous assumons la disponibilité de l'implantation ainsi que de la spécification de référence. Les entrées-sorties de l'implantation sont observées (cet tâche est appelée *observation*) et sont comparées à ceux de la spécification de référence (cet tâche est appelée *prédictions*). L'observation des entrées-sorties de l'implantation montre le comportement du système, tandis que les prédictions nous informent sur ce que l'implantation est supposée faire. Les différences entre les prédictions et les observations sont appelées *symptômes*. Les symptômes assurent l'existence d'une ou de plusieurs défaillances du système; ils sont utilisés pour guider la recherche des fautes responsables des défaillances (voir figure 8.1) [Ghed 93d].

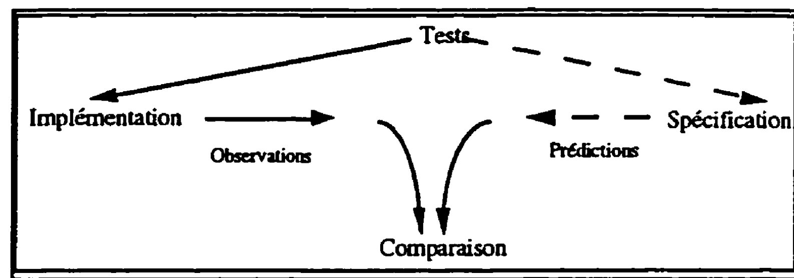


Figure 8.1. Interactions entre les prédictions et les observations.

Un *diagnostic* est alors défini comme la différence minimale entre le système et sa spécification de référence ou modèle. Cette différence doit expliquer les observations. Pour extraire ces différences, le processus de diagnostic doit effectuer deux principales tâches [Ghed 93]: la génération des candidats et la discrimination entre les candidats.

Première tâche: Génération de candidats

Ce processus se base sur les symptômes et la spécification de référence pour générer des candidats. Chaque candidat est défini comme la différence minimale entre l'implantation et sa spécification de référence capable d'expliquer les symptômes. Chaque transition d'un candidat est appelée *transition candidate*. Un bon algorithme de génération de candidats doit être complet, non redondant et optimal. Il est complet s'il génère tous les candidats qui peuvent expliquer les observations. Il est non redondant, s'il ne génère pas plusieurs fois un même candidat. Finalement, il est optimal s'il ne génère que les candidats minimaux.

Deuxième tâche: Discrimination entre les candidats

L'étape de génération de candidats résulte souvent en un grand nombre de candidats. Pour réduire ce nombre, deux techniques peuvent être utilisées. La première technique, consiste à sélectionner et appliquer un ensemble de tests additionnels appelés *"distinguishing tests"* [Gene 84]. La seconde technique consiste à insérer de nouveaux points d'observation dans l'implantation à diagnostiquer.

8.3. Influence des facteurs de test sur le diagnostic

Le coût du processus de diagnostic dépend de la complexité des tâches de génération et de discrimination entre les candidats. Ces deux tâches dépendent de la qualité du test original. Plus la suite de test est détaillée et complète, moins il y a de candidats et par conséquent moins d'effort pour la discrimination entre les candidats.

Dans le processus de diagnostic, une longueur minimale de la suite de test n'est pas aussi importante que pour les tests. Les facteurs de testabilité (voir chapitres 5 et 6) n'agissent pas de la même manière sur les tests et sur le diagnostic. Au cours de ce chapitre, nous analyserons l'influence de ces facteurs sur le processus de diagnostic. Nous ajusterons ces facteurs aux problèmes du diagnostic.

8.3.1. Le degré de contrôlabilité

L'ensemble d'accessibilité de longueur minimale [Karo 96] est une propriété intéressante dans le processus de détection de fautes. Elle permet d'écourter les cas de test. Cette propriété n'est pas suffisante dans le processus de localisation de fautes (diagnostic). Pour localiser une faute, la séquence de transfert permettant d'accéder à l'état initial d'une transition suspect ne doit pas contenir d'autres fautes, si non la faute en question ne peut pas être traitée par les méthodes de diagnostic actuelles [Ghed 93] et [Vuon 90] (voir exemple 1). Plus il y a des chemins de la spécification menant à une transition à vérifier, plus on a de chance de trouver dans l'implantation un chemin sans fautes menant à l'état de départ de cette transition. Lors de la génération de cas de tests on associera à chaque états q un ensemble A_q de séquences qui permettent d'accéder à l'état en question. La plus courte de ces séquences est utilisée pour des fins de tests (détection de fautes) tandis que les autres pourront être utilisées pour la localisation de fautes. Plus explicitement, si une transition t ayant comme état de départ q est suspectée, on cherchera

dans A_q l'une des séquences sans fautes qui mène à q. Cette séquence est utilisée comme préambule pour des tests additionnels permettant de vérifier t. Pour savoir si une séquence d'accessibilité est sans fautes, on l'applique à l'implémentation et on compare ses sorties avec celles prévues par la spécification. Le nombre de séquences d'accessibilité (chemins menants aux différents états) est donc un facteur d'influence de la qualité du diagnostic. Il est difficile de calculer le nombre optimal de chemins de la spécification qui doivent mener aux transitions. Pour ce faire, nous proposons une estimation de ce nombre basée sur des expériences d'implantation ultérieures. Soit r le nombre moyens de fautes détectées dans des expériences d'implantations ultérieures, au pire des cas ces r fautes peuvent être distribuées sur tous les chemins qui mènent à une transition à tester. Si pour chaque état q, la spécification peut assurer un certain nombre $p > r$ de chemins menant à q, nous somme presque sure de l'existence d'au moins un chemin sans fautes dans l'implantation qui mène à cet état. Les chemins sans fautes sont importants puisqu'ils peuvent être utilisés pour tester des transitions suspectes. Ceci nous conduit à un nouveau facteur que nous appelons *degré de contrôlabilité du diagnostic* $C_D(M)$. Ce facteur est évalué comme suit: pour chaque état q_i de la spécification nous associons une mesure binaire $Cposs_i$ qui dépend du nombre de chemins qui mènent à q_i (chemins sans boucles [Nejm 88]), si $p > r$ alors $Cposs_i = 1$ sinon $Cposs_i = 0$. $C_D(M)$ est alors calculée comme la moyenne des $Cposs_i$:

$$C_D(M) = \frac{\sum_{i=1}^{n} Cposs_i}{n-1}, \text{ et } 0 \leq C_D(M) \leq 1, \quad (1)$$

où n est le nombre d'états de la spécification.

Exemple 1

La figure 8.2 et 8.3 représentent un exemple de spécification de référence et une de ses implémentations possibles. Cette implémentation contient deux fautes: $f1$ dont la transition responsable est accessible par un chemin sans fautes et $f2$ non accessible par des chemins sans fautes. La faute $f2$ ne peut pas être traitée par les méthodes de diagnostic actuelles [Ghed 93] et [Vuon 90]. Si par exemple $r = 1$, le degré de contrôlabilité du diagnostic n'est pas parfait, il est évalué à $C_D(M) = 2/3$. Par contre, le degré de contrôlabilité de la spécification de la figure 8.5 est parfaite ($C_D(M)$=1). ■

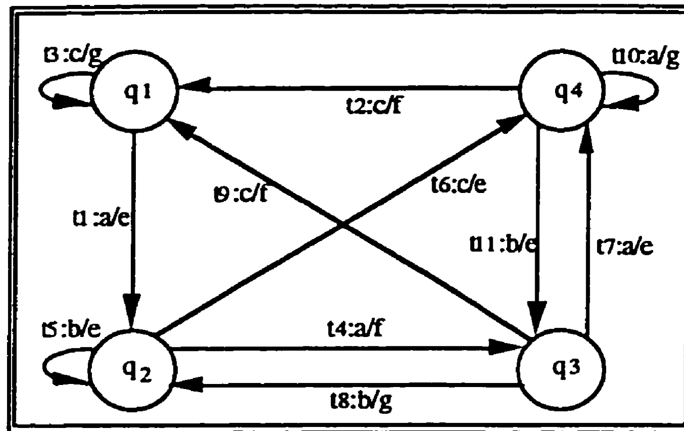


Figure 8.2. La spécification *S*.

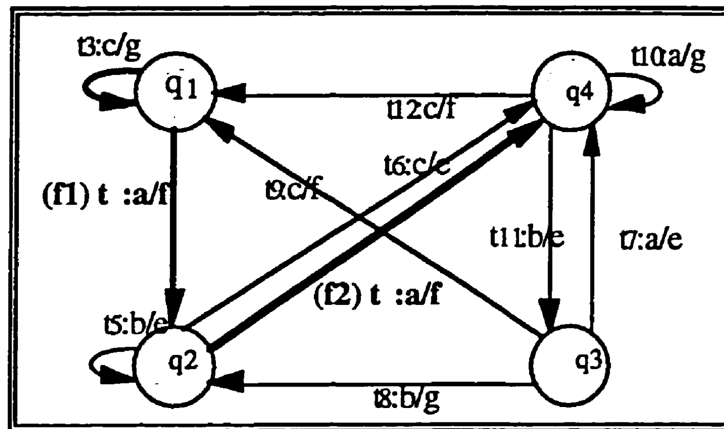


Figure 8.3. L'implantation *Im*1.

8.3.2. Le degré de Flou

Ce facteur (voir chapitre 6) affecte de la même façon les tests et le diagnostic, mais son optimalité n'est pas suffisante pour assurer la qualité du diagnostic. En effet, même si la spécification est réduite, on ne pourrait jamais garantir que l'implantation le sera. Ceci car les fautes peuvent rendre des états de l'implantation équivalents. Ce facteur a deux volets: le premier peut être géré en améliorant la distingabilité des états (dès l'étape de conception) et peut être évalué de la même façon que pour les tests (voir chapitre 6). Le deuxième volet est aléatoire, il dépend du type de fautes et son évaluation est donc probabiliste. Elle peut être étudiée en analysant la probabilité qu'une faute affecte la distingabilité des états.

8.3.3. Le degré de distingabilité des états

La majorité des méthodes de test existantes se basent sur les séquences de distinction des états. Si une machine n'est pas réduite, on partitionne l'ensemble des états Q en des sous-ensembles $\Pi=\{Q_1, ..., Q_k\}$. Chaque sous-ensemble Q_i ($|Q_i|>1$) inclus des états deux à deux distingables (voir chapitre 6) par un ensemble de séquences W_i. Une faute peut rendre des états de Q_i non distingables avec W_i. Dans ce cas, W_i devient inutile durant le processus de diagnostic (voir Exemple 2). Pour le diagnostic, il est donc mieux d'avoir plus qu'un seul ensemble de distinction W_i pour chaque Q_i. Ceci augmentera les capacités de localisation des fautes.

Le processus de diagnostic, n'est pas seulement influencé par la longueur et le nombre de séquences des ensembles W_i. Il est aussi influencé par le nombre de possibilité d'ensembles $W_{i,1}, W_{i,2},..., W_{i,p}$ associés à chaque Q_i. Nous appellerons ce nouveau facteur le degré de distinction des états du diagnostic et nous le noterons $W_D(M)$. Pour le calcul de ce facteur nous avons besoin du nombre d'ensembles de séquences de distinction $W_{i,1}, W_{i,2},..., W_{i,p}$ associés à chaque Q_i . Pour calculer ce nombre, nous agissons comme nous l'avons fait pour le degré de contrôlabilté, c'est-à-dire nous utilisons une estimation basée sur le nombre moyen r de fautes dans des expériences d'implantation ultérieures. Dans le pire cas, les r fautes peuvent être distribuées sur toutes les possibilités de séquences de distinction d'un même état. Si pour chaque sous-ensemble Q_i, on arrive à trouver un nombre d'ensembles de séquences de distinction supérieur à r, nous serons presque sûr que l'un d'eux restera valide dans l'implantation. Nous associerons à chaque Q_i, une mesure binaire $Wposs_i$ qui dépend du nombre d d'ensembles de séquences de distinction $W_{i,1},W_{i,2},..., W_{i,p}$ associés à Q_i, $Wposs_i = 1$ si $d > r$ sinon $Wposs_i = 0$. $W_D(M)$ sera calculée comme la moyenne de tous les $Wposs_i$:

$$W_D(M)=\frac{\sum_{i=1}^{k} Wposs_i}{k}, \quad 0 \le W_D(M) \le 1 \quad (2).$$

Exemple 2

Soient la spécification et les deux implantations fautives (figures 8.4). Si nous utilisons la méthode W pour la sélection des cas de test on obtient la suite de tests suivante: l'ensemble de caractérisation $W=\{a, b\}$, la suite de tests $TS=\{aa, ab, baa, bbb,$

cab, cca, ccb, cba, bab, bba, caa, cbb}, les sorties de *S*={*ef, ef, fff, fff, ffe, ffe, efe, eef, eef, eee, fff, fff, eff, eef*}.

L'application de *TS*, aux implantations *Im*1 et *Im*2 des figures 8.4.b et 8.4.c, génère dans les deux cas la même séquence de sortie {*ef, ef, ffe, fff, eef, eee, eef, eff, fff, fff, eef, eff*} et donc *TS* ne peut être utilisée pour distinguer entre *Im*1 et *Im*2. Ceci est dû au fait que *W* n'est plus un ensemble de distinction de *Im*2 et l'état *q*2 n'est plus accessible dans *Im*1. ■

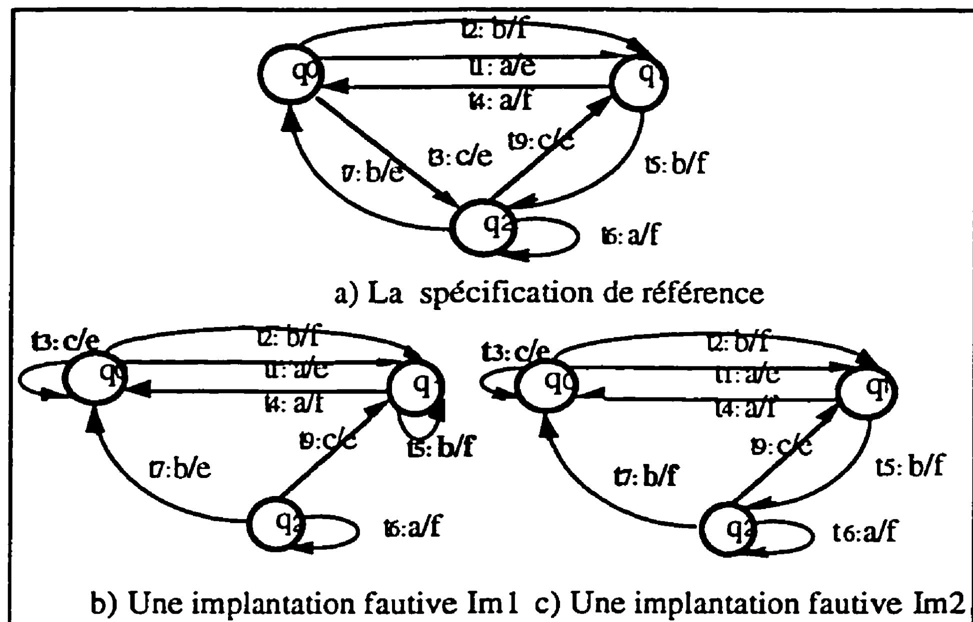


Figure 8.4. Une spécification et deux implantations fautives non distingables.

8.3.4. Le degré d'abstraction

Ce facteur [Karo 96a] influence de la même manière les tests et le diagnostic. Il est difficile de localiser des fautes dans une implantation contenant plus d'informations (plus d'états) que la spécification. C'est pour cette raison, nous avons besoin de plus d'effort pour adapter les tests à un niveau de détail acceptable par l'implantation. La mesure utilisée dans le chapitre 6 reste donc valide.

8.3.5. Facteurs dépendants des fautes

Les tests et le diagnostic sont influencés par d'autres facteurs aléatoires. Ces facteurs sont difficiles à étudier car ils sont en relation directe avec les fautes. En effet,

certains facteurs comme le nombre, le genre et la distribution des fautes doivent être pris en considération lors des tests ou lors du processus de diagnostic. Pour comprendre comment ces facteurs peuvent influencer le diagnostic, nous pouvons utiliser la formule de complexité du processus de diagnostic présentée dans [Ghed 93a]:

$$O(I.Lc^{S}.n^{F+2}) \quad (3),$$

où I est l'ensemble des symboles d'entrée de la spécification, Lc est le nombre d'entrées du plus long cas de test, S est l'estimation du nombre maximal de séquences de test ayant des symptôme et F le nombre maximum de fautes de transfert. À partir de cette formule, nous pouvons remarquer que le nombre de fautes influence énormément (exponentiel) la complexité du processus de diagnostic.

8.3.6. Discussion

À partir des formules de complexité du diagnostic (Formule (9)) et celle de la longueur de la suite de tests (voir Formule 1, chapitre 6), on peut déduire que la facilité des tests et du diagnostic dépendent des trois classes de propriétés suivantes:

1) les caractéristiques intrinsèques de la spécification, représentées par I et n dans les formules,
2) la qualité des suites de test, représentée par V, W, Lc et S dans les formules,
3) les caractéristiques des fautes, représentées par F dans la formule.

Les deux premières classes ont été discutées dans les sections 8.3.1, 8.3.2, 8.3.3, et 8.3.4. Tandis que le troisième facteur avantage d'une part les tests et d'autre part désavantage le diagnostic. Pour être plus précis, plus il y a de fautes dans l'implantation, plus il est facile et plus rapide de conclure quant au verdict des tests. D'autre part, plus il y a de fautes dans l'implantation, plus l'accès aux états et/ou la distinction des états est difficile, par conséquent la localisation de fautes est complexe.

8.4. La conception de systèmes faciles à diagnostiquer

Dans les chapitres précédants, nous avons montré qu'il est possible de remédier aux difficultés des tests lors de l'étape de conception (*DFT*). Nous avons proposé un ensemble de transformations qui améliorent le processus de test. De la même manière, les problèmes du diagnostic peuvent être étudiés lors de l'étape de conception. Nous

proposerons des idées de transformations qui réduisent le coût du processus de diagnostic. Ces transformations sont guidées par les nouvelles mesures de contrôlabilité et de distingabilité des états du diagnostic présentées dans les sections précédantes.

8.4.1. Amélioration de la contrôlabilité

Un préambule est un chemin permettant d'atteindre une transition à tester. En implantant ce chemin, on peut introduire des erreurs. Il peut alors contenir des fautes et sera inutilisable pour tester une partie du système accessible via ce chemin. Pour résoudre ce problème, nous avons besoin de plus qu'un préambule qui puisse nous permettre d'accéder à chacune des transitions à tester. Lors du processus de génération de la suite de tests, nous avons intérêt à générer plusieurs ensembles de séquences d'accessibilité des états. Même s'ils ne sont pas nécessairement minimaux, ils peuvent être utiles dans le processus de diagnostic. Dans le cas où la spécification ne possède qu'un seul ensemble de séquences d'accessibilité, nous essayons de la compléter de façon à en avoir plus.

Les plus courtes des séquences sont utilisées pour des fins de tests (détection de fautes) tandis que les autres pourront être utilisées pour la localisation de fautes. Pour savoir si une séquence d'accessibilité est sans fautes, on l'applique à l'implémentation et on compare ses sorties avec celles prévues par la spécification.

Pour améliorer la contrôlabilité du diagnostic, nous pouvons donc compléter la spécification de façon à améliorer $C_D(M)$. Ceci peut être fait en assignant les transitions *don't care* des états qui les contiennent vers des états accessibles par un nombre minimal de préambules. L'exemple 3, illustre cette idée.

Exemple 3:

L'état $q2$ de la figure 8.2 est accessible via un unique préambule. Si une faute se glisse dans ce préambule (figure 8.3), alors toutes les transitions accessibles via $q2$ deviennent difficiles à diagnostiquer.

Si nous assignons la transition non-spécifiée ayant comme entrée b de l'état $q1$ vers l'état $q2$ (figure 8.5), l'état $q2$ devient accessible via deux préambules différents. Les fautes, comme celles de la figure 8.6, peuvent être alors facilement localisées. Si r=1, $C_D(M)$ est améliorée de 2/3 à 1.

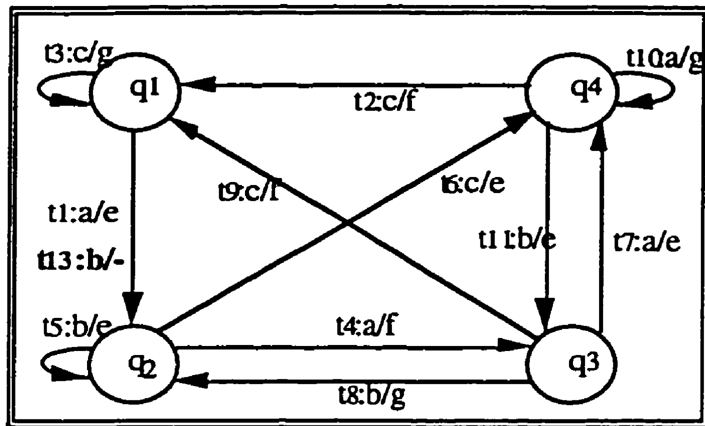


Figure 8.5. Une spécification transformée.

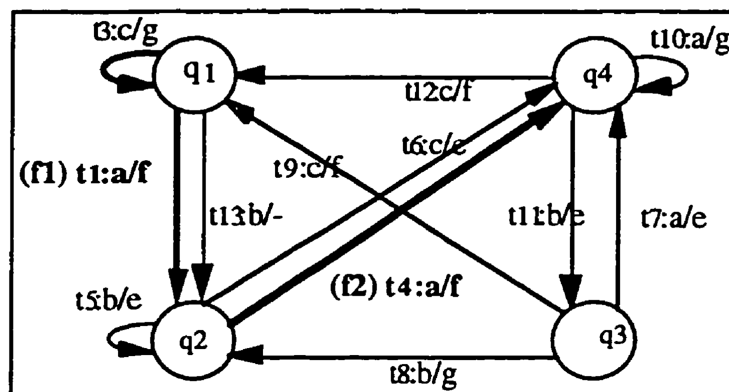


Figure 8.6. Une implantation de la spécification de la figure 8.5.

8.4.2. Amélioration de la distinction des états

Il n'est pas suffisant d'avoir un seul ensemble de distinction W_i pour chaque ensemble d'états distingables Q_i. Une faute peut rendre des états de Q_i non distingables par W_i qui ne sera plus valide. Pour améliorer le processus de diagnostic, il faut extraire le maximum de séquences de distinction possibles (lors du processus de génération de suite de test). Si les W_i sont uniques, on peut essayer de compléter la spécification de façon à en avoir plus.

Nous compléterons les transitions *don't care* d'une spécification partiellement spécifiée de manière à augmenter le nombre de séquences de distinction possibles. Pour atteindre cet objectif, nous proposons d'étendre les techniques présentées dans le chapitre 6, comme *H-trans* par exemple. L'exemple 4 illustre cette idée.

Généralement le processus de diagnostic abouti à un ensemble de machines candidates qui peuvent expliquer la ou les éventuelles erreurs de fonctionnement. Pour réduire le nombre de machines candidates, on a besoin d'appliquer des tests supplémentaires afin de chercher les candidats qui peuvent expliquer le comportement de l'implémentation. Pour cela, Ghedamsi [Ghed 93] a proposé l'utilisation de l'algorithme de Gill [Gill 62].

Pour réduire le nombre de machines candidates, nous suggerons l'extraction de plus qu'une suite de test à partir d'une même spécification. Les suites de test sont construites à partir des différents ensembles d'accessibilité et des différentes séquences de distinction tels que nous l'avons présenté. Nous proposons d'appliquer ces suites de tests à l'implémentation du protocole. Chacune des suites donne une indication supplémentaire sur la ou les fautes de l'implémentation. Les machines candidates doivent expliquer les sorties anormales de chaque suite. Le diagnostic final provient donc de l'intersection des résultats de l'application de chaque suite. Plus on applique de suites de test, plus l'intersection est fine et plus le résultat du processus de diagnostic est meilleur.

Exemple 4:

Les fautes des implantations *Im*1 et *Im*2 des figures 8.b et 8.c ne peuvent pas être localisées. $W1 = \{a, b\}$ n'est plus un ensemble de caractérisation des états de *Im*2. Si nous assignons à la transition non spécifiée de l'état $q1$ une *self-loop* avec l'entrée c et la sortie f (figure 8.7.a) et si cette transition est implantée correctement (figures 7.b et 7.c), la séquence $W1=\{cc\}$ devient un nouveau ensemble de caractérisation. Dans ce cas, si r=1 $W_D(M)$ est améliorée de 0 à 1, nous n'aurons plus qu'à appliquer le cas de test additionnel *abcc* pour distinguer la spécification, *Im*1 et *Im*2. Les réponses de la spécification, *Im*1, et *Im*2 seront respectivement: *efee*, *efff*, et *efef*.

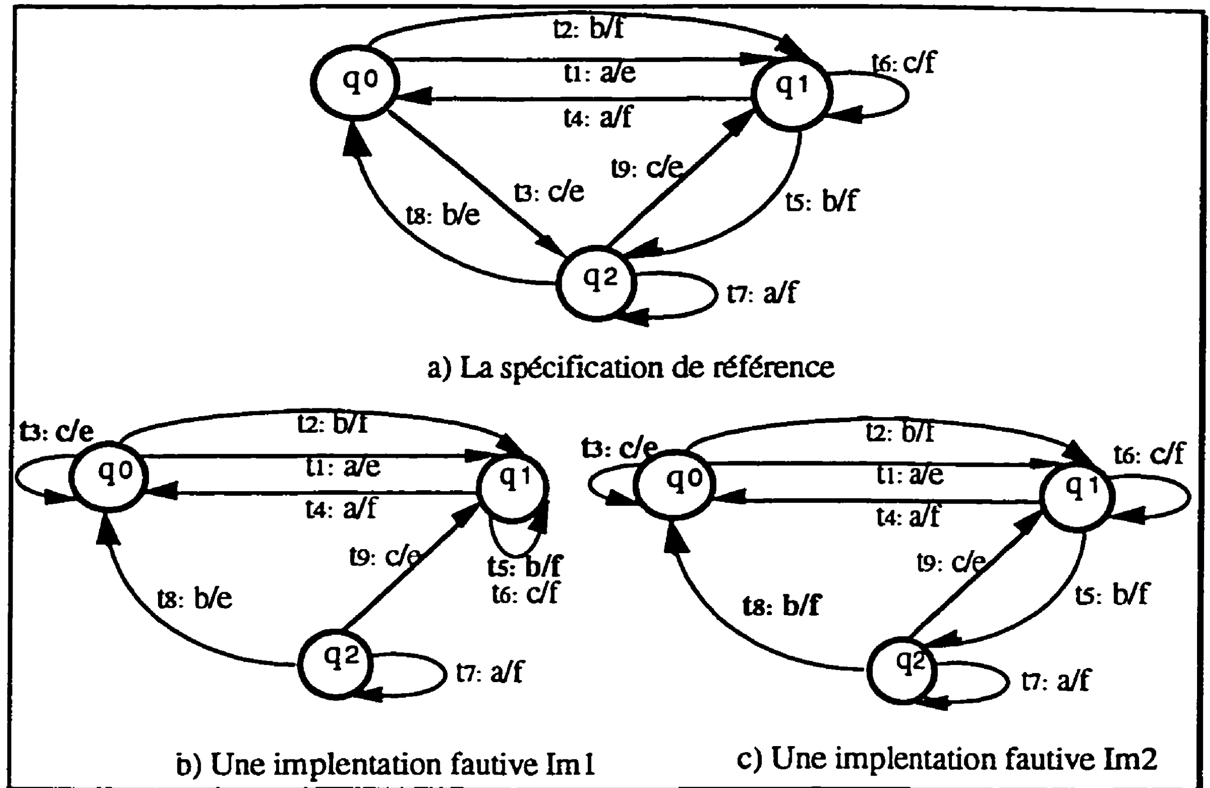


Figure 8.7. Amélioration de la distingabilité des états.

8.5. Conclusion

La détection et la localisation de fautes sont des problèmes de nature fortement liés. Au cours de ce chapitre, nous avons montré que les facteurs d'influence du test sont valides pour le diagnostic mais ne sont pas suffisants. Nous avons changé ces facteurs pour qu'ils prennent en considération les problèmes du diagnostic. Finalement, nous avons proposé des idées pour l'amélioration du diagnostic applicables lors de la conception.

Chapitre 9

Conclusion

Le modèle de spécification constitue l'un des facteurs qui influence la testabilité d'un logiciel (voir chapitre 3). Dans ce travail, nous avons étudié la testabilité des protocoles de communication pour trois modèles différents de spécification: les relations, les automates à états finis et les automates à états finis étendus. L'utilisation d'un modèle ou d'un autre dépend du niveau d'abstraction adopté. À un haut niveau d'abstraction, le modèle des relations fournit une bonne plate-forme pour l'étude de la testabilité. Vu le niveau d'abstraction qu'il offre, ce modèle peut être même utilisé au niveau de l'analyse des besoins. Celui-ci se base sur une théorie mathématique très riche et souple à utiliser. À un niveau d'abstraction inférieur, nous avons utilisé les automates à états finis comme modèle de base pour la spécification et l'étude de la testabilité des protocoles de communication. Il s'agit d'un modèle qui a été largement exploité pour la spécification des protocoles de communication et la génération automatique des séquences de test. Le modèle des *FSMs* ne peut être associé qu'aux aspects de contrôle d'une spécification. Pour pouvoir analyser les aspects de données, nous avons utilisé le modèle des automates à états finis étendus. Ce dernier offre une représentation de la spécification sous un niveau d'abstraction encore plus bas et qui prend en considération les aspects de données.

Au cours de ce travail, nous avons proposé des nouvelles approches pour étudier et améliorer la testabilité. Nos contributions concernent les quatre points suivants: le processus de développement du protocole, l'identification des facteurs influençant la testabilité, les mesures de testabilité et les transformations visant à améliorer la testabilité.

Le processus de développement du protocole:

La testabilité est l'un des attributs de qualité de logiciel. Son analyse vise à produire des logiciels faciles à tester. Cette propriété de qualité peut être intégrée dans le cycle de développement du logiciel. Sa prise en compte lors des premières étapes du cycle peut résoudre d'importants problèmes de test. En effet, toute amélioration de la testabilité lors de la conception du logiciel va se répercuter sur toutes ses implémentation possibles. Par contre, si les améliorations sont apportées à l'étape d'implantation, elles ne peuvent viser qu'une implantation unique du logiciel. Dans le chapitre 4, nous avons proposé une façon de considérer la testabilité comme une activité intégrante du cycle de développement du protocole. Nous avons expliqué comment cette activité peut améliorer le processus de génération et d'application des tests. Elle peut être considérée comme un processus itératif de raffinement et de transformation de la spécification visant à améliorer la testabilité. Ce processus se termine lorsque la testabilité est jugée adéquate ou lorsque aucun changement n'est plus applicable. Ce processus est constitué des étapes suivantes: évaluation des facteurs de testabilité de la spécification initiale, choix et application d'une transformation de la spécification, évaluation nouvelle de la testabilité et décision (fin du processus ou continuer le raffinement).

L'identification des facteurs influençant la testabilité:

Pour les deux premiers modèles adoptés, soit le relationnel et les FSMs, nous avons proposé un ensemble de facteurs qui peuvent influencer la complexité et le coût des tests (chapitres 5 et 6). Pour le modèle relationnel, nous avons proposé quatre facteurs qui sont en réalité des propriétés intrinsèques des relations. Nous avons déduit ces facteurs à partir d'une définition de la testabilité de programmes [Free 91] que nous avons adapté à nos besoins. Ces facteurs représentent la nature de l'association entre les entrées/sorties observables d'un module. Pour le modèle des automates, nous avons proposé aussi quatre facteurs qui peuvent influencer la longueur d'une suite de test. Chacun de ces facteurs représente un des éléments de la formule qui exprime la longueur d'une suite de test générée par la méthode W. Dans le chapitre 8, nous avons étudié l'influence des facteurs influençant les tests que nous avons étendu pour les besoins du diagnostic.

Les mesures de testabilité:

L'évaluation de la testabilité, telle que nous l'avons proposée, est composée de plusieurs mesures décrivant les différents aspects de la testabilité (chapitre 5 et 6). L'avantage de cette approche (par rapport à l'évaluation unique) est qu'elle donne une idée détaillée sur les faiblesses (par rapport aux tests) de la spécification. Ceci permet une intervention précise pour une éventuelle correction du facteur de testabilité en question. Nous avons utilisé ces mesures soit pour prédire la testabilité du produit final, soit pour guider un processus de raffinement ou de transformation de la spécification. Dans le chapitre 7 nous avons proposé deux autres mesures visant à estimer le gain de testabilité obtenu après une transformation de la spécification. Dans le chapitre 8, nous avons proposé de nouvelles mesures associées au processus de diagnostic.

Les transformations visant à améliorer la testabilité:

Pour chacun des modèles utilisés (relations, FSMs et EFSMs), nous avons proposé un ensemble de transformations permettant d'améliorer la testabilité du produit final. Pour le modèle relationnel, ces transformations visent à rendre unique (bijection) l'association entre chaque entrée et chaque sortie du module. Pour les modèles des FSMs et les EFSMs, ces transformations complètent la spécification en vue de raccourcir les cas de test.

Travaux futurs

La notion de qualité d'un protocole n'est pas encore bien définie. La testabilité n'est en fait qu'un critère de qualité parmi d'autres qu'il faut prendre en considération lors de la phase de conception. Sa prise en compte peut avoir un impact négatif sur d'autres attributs de qualité. À part les besoins de l'utilisateur, les attributs de qualité désirés doivent être acheminés au concepteur. Ce dernier utilise cette information pour guider sa conception. Dans le cas où les attributs de qualité visés sont incompatibles, le concepteur fera soit des compromis, soit favorisera certains attributs parmi l'ensemble désiré. Par exemple, nous avons vu dans le chapitre 6, que la testabilité et la performance ne sont pas vraiment compatibles. L'association entre ces deux critères de qualité pourrait faire l'objet d'une investigation ultérieure. Le concepteur devrait analyser les exigences de l'utilisateur et les moyens mis à sa disposition pour décider du niveau de performance acceptable du système à développer (temps de réponse, espace mémoire, etc.). Si la relation entre la

performance et la testabilité était bien déterminée, ceci permettrait d'avoir une idée sur la marge de manoeuvre pour les interventions de testabilité. Il nous semble impérieux d'étudier d'autres attributs de qualité du protocole afin de mieux connaître l'impact qu'ils ont les uns sur les autres et de les classer suivant leur compatibilité respective. Il faudrait prévoir un mécanisme qui permet d'informer le concepteur sur les attributs désirés.

On peut poursuivre ce travail en étudiant l'impact d'autres types de facteurs sur la testabilité. Nous pouvons citer comme exemple l'effet de l'environnement sur les tests et en particulier l'effet des architectures de test. Dans ce cas, nos facteurs de testabilité doivent être généralisés aux systèmes comprenant son environnement. Ces nouvelles mesures peuvent être utilisées pour concevoir des systèmes dont la dérivation de l'environnement de test est plus facile.

On peut aussi penser à compléter le travail sur la testabilité des EFSMs. Nous n'avons pas étudié, jusqu'à présent, les méthodes de test des EFSMs. Nous n'avons pas étudié les facteurs de testabilité associés aux données qui font qu'une suite de test est difficile à générer, à compléter ou à appliquer (exécutabilité). Ces deux points pourraient faire l'objet de travaux futurs.

Bibliographie

[Abra 90] M. Abramovici, M. A. Breuer et A. D. Friedman, Digital Systems Testing and Testable Design, Raymond L. Pickholtz, Series Editor, 1990.

[ANSI 87] ANSI/IEEE-1016 Software Design Descriptions, RECOMMENDED PRACTICE FOR SOFTWARE DESIGN DESCRIPTIONS (IEEEComputer Society Document), 1987.

[ANSI 84] ANSI/IEEE-830 IEEE Guide to Software Requirements Specifications. The Institute of Electrical and Electronics Engineers, Inc. 345 East 47th Street, New York, NY 10017,USA, 1984.

[Bard 81] P. H. Bardell et W. H. McAnney, A View from the Trenches: Production Testing of Family of VLSI Multichip Modules, Dig., 11th Annu. Int. Symp. Fault-Tolerant Comput., Portland, Me., pp. 281-283, June 24-26, 1981.

[Beli 89] F. Belina et D. Hogrefe, The CCITT-Specification and Description Language SDL, Computer Networks and ISDN Systems, Vol. 16, pp.311-341, 1989.

[Benn 94] R. G. Bennetts, Progress in design for test : A personal view, IEEE Design and Test of Computers1994.

[Bera 91c] E. Berard, Motivation for an Object-Oriented Approach to Software Engineering, Berard Software Engineering, Inc. 101 Lakeforest Blvd, Suite 360, Gaithersburg, Maryland 20877, 1991.

[Bert 96] A. Bertolino et a. L. Strigini, On the Use of Testability Measures for Dependability Assessment, IEEE Transactions on Software Engineering, Vol. 22, No. 2, Février 1996.

[Bind 94] R. V. Binder, Design for Testability in Object-Oriented Systems, Communications of the ACM Vol. 37 No. 9 pp. 87-101, 1994.

[Boch 80b] G. v. Bochmann, Specification and Verification of Computer Communication Protocols, chap. 5 in "Advances in Data Communications Management", ed. T.A. Rullo, Heyden Publ. 1980.

[Boch 89] G. v. Bochmann, R. Dssouli et J. R. Zhao, Trace analysis for conformance and arbitration testing, IEEE Tr. on Soft. Eng., Vol.15, no.11, pp.1347-1356, Nov. 1989.

[Boch 90] G. v. Bochmann, Protocol Specification for OSI, Computer Networks and ISDN Systems, Vol. 18, pp. 167-184, 1989-90.

[Boch 91d] G. v. Bochmann, A. Das, R. Dssouli, M. Dubuc, A. Ghedamsi et G. Luo, Fault Models in Testing, Proc. IFIP Intern. Workshop on Protocol Test Systems, Netherlands, pp. (II-17)-(II-32), Octobre 1991.

[Boch 93] G. v. Bochmann et R. Gotzhein, Specialization of Object Behaviors and Requirement Specifications, Publication #853, DIRO, Université de Montréal, 1993.

[Boch 94a] G. v. Bochmann et A. Petrenko, Protocol Testing : Review of Methods and Relevance for Software Testing, publication #923, DIRO, Université de Montréal, 1994.

[Boch 94h] G. v. Bochmann, A. Petrenko et M. Yao, Fault Coverage of Tests based on Finite State Models, (invited paper) IFIP Intl Workshop on Protocol Test Systems, Tokyo 1994.

[Boch 97] G. v. Bochmann, A. Petrenko, O. Belal et S. aMaguiragua, 'Automating the Process of Test Derivation from SDL Specification', à paritre dans the Eighth SDL Forum 97, INT Evry France, 22-26 Septembre 1997.

[Boeh 78] B. Boehm et a. al, Characteristics of Software Quality, North-Holland, New York, 1978.

[Boeh 81] B. Boehm, Software Engineering Economics, Prentice-Hall, 1981.

[Bolo 87] T. Bolognesi et E. Brinksma, Introduction to the ISO Specification Language LOTOS, Computer Networks and ISDN Systems, vol. 14, no. 1, pp.25-59, 1987.

[Brin 88] E. Brinksma, A Theory for the Derivation of Tests, Proc. IFIP Symposium on Protocol Specification, Testing and Verification, Atl. City, 1988.

[Brin 89] E. Brinsma, R. AlderdeTesting, Proceedings of the International Workshop on Protocol Test Systems, Berlin (West), Germany, pp.311-325, Octobre 1989.

[Brin 93] E. Brinksma, Sur la couverture des validations partielles, Actes du colloque Francophone sur l'Ingénierie des Protocoles, CFIP'93, Montréal, 1993.

[Broo 84] S. D. Brookes, C. A. R. Hoare et A. W. Roscoe, A Theory of communicating sequential processes, Journal of the ACM, 31(3), pp 560-599, Mars 1984.

[Budk 86] S. Budkowski, P. Dembinski et J. P. Ansart, ESTELLE, UN LANGAGE DE SPECIFICATION DES SYSTEMES DISTRIBUES, 3eme Congrès 'De nouvelles Architectures pour les communications' Paris 28-30 Octobre1986.

[Burg 92] S. P. v. d. Burgt, J. Kroom et A. M. Peters, Testability of Formal Specifications, In R. J. Linn and M. U. Uyar, editors, Protocol Specification Testing and Verification XII, Amsterdam, North-Holland, 1992.

[Chan 93] S. T. Chanson et J. Zhu, A unified Approaches to Protocol Test Sequence Generation, Proceedings of IEEE INFOCOM, pp. 106-114, 1993.

[Chop 88] C. Choppy et M.-C. Gaudel, Impact des Spécifications Formelles sur le Développement de Logiciels, Bigre no. 58, Spécification de logiciel, Janvier 1988.

[Chow 78] T. S. Chow, Testing Software Design Modeled by Finite-State Machines, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Vol. SE-4, No. 3, Mai 1978.

[Clar 76] L. Clarke, A System to Generate Test data and Symbocally Execute Programs, IEEE Transactions on Software Engineering, 2(3), Septembre 1976.

[Clur 86] C. M. Clure et J. Martin, Software Maintenance : the Problem and its Solution, Prentice-Hall, 1986.

[Dijk 74] E. W. Dijkstra, Self-stabilizing Systems in Spite of Distributed Control, Communications of the ACM, pp.643-644, Novembre 1974.

[Drir 92] K. Drira, Transformation et Composition de Graphes de Refus: Analyse de la Testabilité. Thèse de Doctorat de l'Université Paul Sabatier de Toulouse, France, 1992.

[Dsso 86] R. Dssouli, Étude des Méthodes de test pour les implantations de protocoles de communication basées sur les spécifications formelles, Thèse de Doctorat, Décembre 1986.

[Dsso 86a] R. Dssouli et G. v. Bochmann, Conformance testing with multiple observers, Proc. IFIP Workshop on Prot. Specification, Testing and Validation, 1986, North-Holland Publ. pp. 217-229, 1986.

[Dsso 90b] R. Dssouli, R. Fournier et G. v. Bochmann, Conformance testing with FIFO queues, Proc. of the Third International Conference on Formal Description Techniques (FORTE'90), Madrid, pp.359-380, Novembre 1990.

[Dsso 91a] R. Dssouli et R. Fournier, Communication Software Testability, Protocol Test Systems III - IFIP 1991.

[Dsso 91b] R. Dssouli, Les facteurs influançant l'observation de fautes dans les logiciels de communication, Colloque Francophone sur l'Ingénierie des Protocoles, Hermes, pp.185-200, 1991.

[Dsso 95b] R. Dssouli, K. Karoui, A. Petrenko et O. Rafiq, Towards Testable Communication Software, Invited paper in proceedings of IWPTS'95, France, 1995.

[Dubu 92] M. Dubuc, R. Dssouli et G. v. Bochmann, TESTL: A Tool for the Analysis of Test Sequences based on Finite-State Model, IWPTS'92, IFIP Transactions, Protocol Test Systems IV, North Holland Publ. pp.195-206, 1992.

[ElMa 93] K. El Maadani, Identification de systèmes séquentiels structurés. Application à la validation du test, Thèse INSA Toulouse, 1993.

[Ells 92] J. Ellsberger et F. Kristoffersen, Testability in the context of SDL, In R. J. Linn and M. U. Uyar, editors, Protocol Specification Testing and Verification XII, Amsterdam, North-Holland, 1992.

[ETSI 94] ETSI, Methods for Testing and Specification (MTS); Use of SDL in European Telecommunication Standards; Rules for Testability and facilating validation, European Telecommunication Standards Institute, DRAFT prETS 300 414, Mai 1994.

[Faro 90] A. Faro et A. Petrenko, Sequence Generation From EFSMs For Protocol Testing, COMNET'90, Budapest Hongrie, Mai 1990.

[Free 91] R. S. Freedman, Testability of Software Components, IEEE Transactions on Software Engineering. Vol. 17 No. 6, Juin 1991.

[Fuji 91b] S. Fujiwara et G. v. Bochmann, Testing Non-deterministic Finite State Machines, publication départementale P#758, Dépt. IRO, université de Montréal, 1991.

[Fuji 90] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou et A. Ghedamsi, Test selection based on finite state models, Publication départementale #716, DIRO, Université de Montréal, Février 1990.

[Fuji 91a] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou et A. Ghedamsi, Test selection based on finite state models, IEEE Transactions on Software Engineering, Vol.17, no.6, pp. 591-603, Juin 1991.

[Gene 84] M. R. Genesereth, The Use of Design Descriptions in Automated Diagnosis, Artificial Intelligence, Vol.24, no.3, pp. 411-436, 1984.

[Ghed 93] A. Ghedamsi, G. v. Bochmann et R. Dssouli, Multiple Fault Diagnostics for Finite State Machines, publication départementale P#859, Dépt. IRO, Université de Montréal, janvier 1993, Proc. IEEE INFOCOM'93, San Francisco, USA, March 93.

[Ghed 93a] A. Ghedamsi, G. v. Bochmann et R. Dssouli, Diagnosis for Single Transition Faults in Communicating Finite State Machines, IEEE International Conference on Distributed Computing Systems (ICDCS'93), Pittsburgh, USA, Mai 1993.

[Ghed 93d] A. Ghedamsi, G. v. Bochmann et R. Dssouli, Diagnosing Distributed Systems Modeled by Communicating Finite State Machines, Revue Réseaux et Informatique Répartie, Vol.3, No.4, éditions Hermes, 1993.

[Gill 62] A. Gill, Introduction to the Theory of Finite State Machines, Mc Graw-Hill Book Company, Inc, 1962.

[Gone 70] G. Gonenc, A Method for the Design of Fault Detection Experiments, IEEE Transactions Computer, Vol. C-19, pp. 551-558, Juin 1970.

[Goud 85] M. G. Gouda and J.-Y. Han, Protocol Validation by Fair Progress State Exploration, Computer Networks and ISDN System 9, pp. 353-361, 1985.

[Hals 92] F. Halsall, Data Communication, Computer Networks and Open Systems, Addison-Wesley, Troisième édition, 1992.

[Haml 93b] D. Hamlet et a. J. M. Voas, Faults on its Sleeve : Amplifying Software reliability Testing, International Symposium on Software Testing and Analysis (ISSTA'93). ACM SIGSOFT, 1993.

[Henn 64] F. C. Hennie, Fault Detecting Experiments for Sequential Circuits, Proc. of the IEEE 5th Ann. Symp. on Switching Circuits Theory and Logical Design, 1964.

[Hoar 85] C. A. R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[Howd 82] W. E. Howden, Weak Mutation Testing and Completeness Tests, IEEE Transaction on software Engineering, vol. 8, No 2, pp. 371-379, Juillet 1982.

[Huan 95] C.-M. Huang, Y. Lin et M. Jang, An Executable Protocol Test Sequence Generation Method for EFSM-Specified Protocols, IWPTS'95, Evry, France, Septembre 1995.

[IEEE 90] ANSI/IEEE, IEEE Standard Glossary of Software Engineering Terminology, ANSI/IEEE Standard 610.12-1990, IEEE Press, New York, 1990.

[ISO8807 89] ISO8807, LOTOS: A Formal Description Technique.

[ISO9074 89] ISO9074, Estelle: A Formal Description Technique based on an Extended State Transition Model.

[ISO 9646] ISO9646, Information Processing Systems - Open Systems Interconnection - OSI OSI Conformance Testing Methodology and Framework, Part 1: General Concepts, Part 2: Abstract Test suite Specification, Part 3: The Tree and Tabular Combined Notation (TTCN), Part 4: Test Realization, Part 5: Requirements on Test Laboratories and Clients for the Conformance Assessment Process.

[ITU 96] ITU, Specification and Description Language SDL, Recommendation Z.100, Blue Book, 1996.

[Jaou 92] A. Jaoua, N. Belkhiter, J. Desharnais et T. Moukam, Propriétés des dépendances difonctionnelles dans les bases de données relationnelles, INFOR Vol. 30, No. 3, pp. 297-316, Aout 1992.

[Jone 86] C. B. Jones, Systematic Software Development Usimg VDM, Prentice-Hall, 1986.

[Kalm 69] R. E. Kalman, P. L. Falb et M. A. Arbib, Topics in Mathematical System Theory, McGraw-Hill, New York, USA, 1969.

[Kark 92] Y. Karkouri, Analyse et génération de vecteurs de tests pour les pannes multiples. thèse de PhD, IRO - Université de Montréal, Décembre 1992.

[Karo 94] K. Karoui, R. Dssouli, O. Cherkaoui et A. Khoumsi, Estimation de la testabilité d'un logiciel modélisé par les relations, publication départementale P#921, Dépt. IRO, Université de Montréal, Juin 1994.

[Karo 96a] K. Karoui, R. Dssouli et O. Cherkaoui, Specification Transformations and Design for Testability, IEEE GLOBECOM'96, Londre, Novembre 1996.

[Karo 96b] K. Karoui et R. Dssouli, Testability Analysis of the Communication Protocols Modeled by Relations, publication départementale P# 1050, Dépt. IRO, Université de Montréal, Novembre 1996.

[Karo 97a] K. Karoui, R. Dssouli et N. Yevtushenko, Design for Testability of Communication Protocols Based on SDL, à paraitre dans the Eighth SDL Forum 97, INT Evry France, 22-26 Septembre 1997.

[Karo 97b] K. Karoui, A. Ghedamsi et R. Dssouli, A Study of Some Influencing Factors in Testability and Diagnostics Based on FSMs, publication départementale P#1048, Dépt. IRO, Université de Montréal, Soumis au 4th International Symposium on Software Metrics (METRICS'97), Albuquerque, New Mexico USA, 5-7 Novembre 97.

[Karo 97d] K. Karoui, N. Tagoug, F. Lustman et R. Dssouli, Design Metrics that Predict Maintainability, 8th Annual Oregon Workshop on Software Metrics, Coeur d'Alene, Idaho USA, 11-12 Mai 1997.

[Karo 97e] K. Karoui, R. Dssouli et Y. Iraqi, Design for Testability of Software, Soumis à HASE'97 (IEEE High-Assurance Systems Engineering Workshop), Washington, DC USA, 11-12 Aout 1997.

[Kim 95] M. Kim et S. T. Chanson, Design for Testability of Protocols based on Formal Specifications, Proceedings of the 8th International Workshop on Protocol Test Systems, Evry, France, Septembre 1995.

[Koha 70] Z. Kohavi, Switching and Finite Automata Theory, McGraw-Hill, Inc. 1970.

[Koha 78] Z. Kohavi, Switching and Finite Automata Theory, McGRAW-HILL COMPUTER SCIENCE SERIES, 1978.

[Lamp 78] L. Lamport, Time, Clocks, and the Ordering of Events in a Distributed System, Communication of the ACM, Vol. 21, pp. 558-565, Juillet 1978.

[Lapr 91] J. C. Laprie, Dependability: Basic Concepts and Terminology, IFIP WG 10.4, Dependable Computing and Fault Tolerance, Springer-Verlag Wien New York, 1991.

[Lars 90] K. G. Larsen, Ideal Specification Formalism = Expressivity + Composionality + Decidability + Testability ... J.C.M. Baeten and J.W. Klop, editors, CONCUR'90: Theory of Concurrency-unification and Extension, pp. 35-56, vol. 458, Lecture Notes in Computer Science, Springer-Verlag, 1990.

[Ledu 91] Guy Leduc, On the Role of Implementation Relations in the Design of Distributed Systems using Lotos, Thèse de Doctorat, Université de Liége, 1991.

[Lour 96] A. A. F. Loureiro, Design For Testability of Communication Protocols, Thèse de Doctorat, Université British Columbia, 1996.

[Luo 93] G. Luo, A. Petrenko et G. v. Bochmann, Selecting Test Sequences for Partially-specified Nondeterministic Finite State Machines, Publication #864, 1993.

[Luo 94a] G. Luo, G. v. Bochmann et A. Petrenko, Test Selection Based on Communicating Nondeterministic Finite State Machines Using a Generalised Wp-Method, IEEE Transaction On Software Engineering, Vol. 20, No. 2, Février 1994.

[Luo 94b] G. Luo, A. Das et G. v. Bochmann, Software Testing based on SDL, IEEE Transactions on Software Engineering, SE-20(1), pp. 72-87, Janvier 1994.

[Mart 83] J. Martin et a. C. McClure, Software Maintenance, the Problem and its Solutions, Prentice-Hall.

[Mays 90] R. G. Mays et e. al. Experience with Defect Prevention, IBM Systems Journal, Vol. 29, No. 1, 1990.

[McCa 89] T. J. McCabe et C. W. Butler, Design Complexity Measurement and Testing, Communication of the ACM, Vol. 32, No 12, pp 1415-1424, Décembre 1989.

[McCa 77] J. A. McCall, Concepts et Definitions of Software Quality, Factors in Software Quality, NTIS, Vol. 1, 1977.

[Meye 88] B. Meyer, Object Oriented Software Construction, Prentice Hall International Series in Computer Science, C.A.R. HOARE Series Editor, 1988.

[Mili 90] A. Mili, Program Fault Tolerance, Prentice Hall, 1990.

[Mill 93] K. W. Miller et J. M. Voas, Applying a Dynamic Testability Technique To Debugging Certain Classes of Software Faults, Software Quality Journal, Vol. 2, pp 61-75, 1993.

[Mill 87] H. D. Mills, M. Dyer et R. C. Linger, Cleanroom Software Engineering, IEEE Software, Vol. 4, pp. 19-25, Septembre 1987.

[Miln 80] R. Milner, A Calculus of Communicating Systems, Lecture Notes in Computer Science, No. 92, Springer Verlag, 1980.

[Bach 90] R. B. a. M. Müllerberg, Measures of Testability as a Basis for Quality Assurance, Software Engineering Journal, pp. 86-92, Mars 1992.

[Myer 76] G. J. Myers, Software Reliability, A Willey-Inter-Science Publication 1976.

[Myer 79] G. J. Myers, The Art of Software Testing, John Willey & Sons, 1979.

[Nait 81] S. Naito et M. Tsunoyama, Fault Detection for Sequenctial Machines by Transition Tours, Proc. FTCS, pp. 238-243, 1981.

[Nejm 88] B. A. Nejmeh, NPATH: A Measure of Execution Path Complexity and its Applications, Communications of the ACM vol.31, No.2, Février 1988.

[Ould 86] M. A. Ould et a. C. Unwin, Testing in Software Development, 1986.

[Parn 89] D. Parnas et Y. Wang, The trace Assertion Method of Module Interface Specification, Technical Report 89-261, Queen's University, Ontario, Canada, 1989.

[Perr 90] D. E. Perry et G. E. Kayser, Adequate Testing and Object-Oriented Programming, Journal of Object-Oriented Programming, 2(5): 13-19, Jan-Fev 1990.

[Perr 87] W. Perry, Effective Methods of EDP Quality Assurance, Prentice-Hall, seconde édition, 1987.

[Petr 91] A. Petrenko, Checking Experiments with Protocol Machines, IFIP 4th International Workshop on Protocol Test Systems (IWPTS 91), pp. 83-94. North-Holland, 1991.

[Petr 92] A. Petrenko et N. Yevtushenko, Test Suite Generation for a FSM with a Given Type of Implementation Errors, Proc. Protocol Specification, Testing and Verification, Juin 92, Florida, USA, Juin 92.

[Petr 93a] A. Petrenko, R. Dssouli et H. Koenig, On Evaluation of testability of Protocol Structures, Proc. Int. Workshop on Protocol Test Systems (IFIP), Pau, France, 1993.

[Petr 93c] A. Petrenko, G. v. Bochmann et R. Dssouli, Conformance Relations and Test Derivation, (invited paper), Proc. Int. Workshop on Protocol Test Systems (IFIP), O. Rafiq (ed.), North Holland Publ., 1993.

[Petr 94b] A. Petrenko, N. Yevtushenko et R. Dssouli, Testing Strategies for Communicating FSMs, Proc. of the International Workshop on Protocol Test Systems (IWPTS'94), Tokyo, Japan, Nov. 1994, , Japon, Nov. 1994.

[Petr 93b] A. Petrenko, N. Yevtushenko, A. Lebedev et A. Das, Nondeterministic State Machines in Protocol Conformance Testing, Proceedings of the IFIP Sixth

International Workshop on Protocol Test Systems, Pau, France, pp. 363-378, Septembre 1993.

[Prad 86] D. K. Pradhan, Fault-Tolerant Computing Theory and Techniques. volume 1, Prentice Hall, 1986.

[Pres 92] R. S. Pressman, Software Engineering a Practitoner's Approach, McGraw-Hill, Inc., 1992.

[Prob 82] R. L. Probert, Life-cycle / Grey Box Testing, Congressus Numerantium, volume 34, Winnipeg, Canada, 1982.

[Prob 90] R. L. Probert et C. Geldrez, Communications Systems Design For Testability: Grey Box Testing, publication départementale P#90-34, Dépt. of Computer Science, Université d'Ottawa, Ottawa, Canada, Juillet 1990.

[Prob 91a] R. I. Probert et K. Saleh, Synthesis of Communication Protocols: Survey and Assessment, IEEE Tr. on Computers, Vol. 40, 4, pp. 468-476, Avril 1991.

[Rafi 91] O. Rafiq, Le test de conformité des protocoles, Réseaux et Informatique Répartie, 1(1):101-142, 1991.

[Rama 95] T. Ramalingom, A. Das et K. Thulasiraman, A Unified Test Case Generation Method for the EFSM Model Using Context Independent Unique Sequences, IWPTS'95, Evry, France, Septembre 1995.

[Papp 85] S. Rapps et E. J. Weyuker, Selecting Software Test Data Using Data Flow Information. IEEE Trans. on Software Engineering, vol. SE-11, No. 4, Avril 1985.

[Rayn 87] D. Rayner, OSI Conformance Testing, Computer Networks and ISDN Systems, 14:79-98, 1987.

[Raz 93] T. Raz et A. Yaung, Process Clustering with an Algorithm Based on a Coupling Metric, J. Systems Software, vol. 22, pp. 217-223, 1993.

[Sabn 88] K. K. Sabnani et A. T. Dahbura, A Protocol Testing Procedure, Computer Networks, pp. 285-297, Vol. 15, No. 4, 1988.

[Sale 92] K. Saleh, Testability-directed Service Definitions and their Synthesis, Proceeding of the Eleventh IEEE Phoenix Conference on Computers and Communications, Arizona, USA, Mars 1992.

[Sand 80] G. Sanderson, A Relational Theory of Computing, Lecture Notes in Computer Science, vol 100, Heidelberg: Springer-Verlag, 1980.

[Sari 93] B. Sarikaya, Principles of Protocol Engineering and Conformance Testing, Ellis Horwood Series in Computer Communications and Networking, 1993.

[Sidh 89] D. P. Sidhu et T. K. Leung, Formal Methods for Protocol Testing: a Detailed Study, IEEE Tr. on SE, Vol 15, 4, 1989.

[Sinc 84] R. Sincovec et R. Wiener, Software Engineering with Modula-2 and ADA, Library of Congress Cataloging in Publication Data, 1984.

[Smit 90] M. D. Smith et D. J. Robson, Object-Oriented Programming - the Problems of Validation, Proceedings of the 6th International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos, CA. pp. 272-281, 1990.

[Somm 92] I. Sommerville, Software Engineering, Addison-Wesley, 4eme Édition, 1992.

[Spiv 92] J.M. Spivey, The Z Notation, A Reference Manual, Prentice Hall, 1992.

[Star 72] P. H. Starke, Abstract Automata, American Elsevier Publishing Company, Inc-New York, 1972.

[Tan 95] Q. M. Tan, A. Petrenko, G. v. Bochmann et G. Luo, Testing Trace Equivalence for Labeled Transition Systems, Submitted for Publication, 1995.

[Tars 41] A. Tarski, On The Calculus of Relations, The Journal of Symbolic Logic 6, pp 73-98, 1941.

[Thay 90] R. H. Thayer et a. M. C. Thayer, Glossary, In Richard H. Thayer and Merlin Dorfman Editors, System and Software Requirements Engineering, IEEE Computer Society Press, 1990.

[Tret 92] J. Tretmans, A Formal Approach to Conformance Testing, PhD. Thesis, Twente University, 1992.

[Troy 81] D. A. Troy et S. H. Zweben, Measuring the Quality of Structured Designs, The Journal of Systems and Software, pp. 113-120, 1981.

[Turn 93b] C. D. Turner et D. J. Robson, State-Based Testing and Inheritance, Technical Report TR-1/93, Computer Science Division, School of Engineering and Computer Science (SECS), University of Durham, Durham, England, 1993.

[Ural 86a] H. Ural, An Interactive Test Sequence Generator, Proc. ACM SIGCOMM Symposium 1986, Stowe, Aug. 1986.

[Ural 87a] H. Ural, Test Sequence Selection Based on Static Data Flow Analysis, Computer Communications, Vol.10, No.5, Octobre 1987.

[Ural 87b] H. Ural, A Test Derivation Method for Protocol Conformance Testing, in Proc. IFIP Symp. on Protocol Specification, Testing and Verification VII, North Holland Publ., pp. 347-358, 1988.

[Vasi 73] M. P. Vasilevski, Failure Diagnosis of Automata, Cybernetics, Plenum Piblishing Corporation, New York, No 4, pp. 653-665, 1973.

[Voas 91] J. Voas, L. Morrell et K. Miller, Predicting Where Faults Can Hide From Testing, IEEE Software, Mars 1991.

[Voas 92] J. M. Voas, PIE: A Dynamic Failure-Based Technique, IEEE Transactions on Software Engineering, Vol 18, No. 8, pp. 717-727, Aout 1992.

[Voas 93] J. M. Voas et K. W. Miller, Semantic Metrics for Software Testability, J. Systems Software, Vol.20, pp. 207-216, 1993.

[Voas 95] J. M. Voas et K. W. Miller, Software Testability: The New Verification, IEEE Software, pp. 17-28, Mai 1995.

[Vuon 89] S. T. Vuong, W. W. L. Chan et M. R. Ito, The UIOv Method for Protocol Test Sequence Generation, In the 2-nd International Workshop Protocol Test System, Berlin, Germany, Octobre 3-6 1989.

[Vuon 90] S. t. Vuong and K. c. Ko, A Novel Approach to Protocol Test Sequence Generation, IEEE Global Telecomm. Conference and Exhibition, San Diago, California, Dec. 2-5, 1990, vol. no. 3, 904.1.1 - 904.1.5, 1990.

[Vuon 91] S. T. Vuong et J. Curgus, On Test Coverage Metrics for Communication Protocols, In Proceedings of the IWPTS IV, Leinschendam. The Netherlands, 1991.

[Vuon 93] S. T. Vuong, A. A. F. Loureiro et a. S. T. Chanson, A Frame Work for the Design for Testability of Communications Protocols, Sixth International Workshop on Protocol Test Systems, pp. 89-108, Pau, France, Septembre 93.

[Wass 77] A. Wasserman, On the Meaning of Discipline in Software Design and Development, Software Engineering Techniques, Infotech State of the Art Report, 1977.

[Weyu 88] E. J. Weyuker, Evaluating Software Complexity Measures, IEEE Transactions on Software Engineering, Vol. 14, No. 9, Septembre 88, pp. 1357-1365.

[Weyu 88b] E. J. Weyuker, The Evaluation of Program-Based Software Test Data Adequacy Criteria, Commuinication of the ACM, Vol. 31, No. 6, pp 668-675, 1988.

[Weyu 93] E. J. Weyuker, More Experience with Data Flow Testing. IEEE Transactions on Software Engineering, 19(9):912-919, Septembre 1993.

[Whit 78] C. H. White, System Reliability and Integrity, In Infotech State of the Art Report. Infotech, 1978.

[Will 73] J. F. Williams, Enhancing Testability of Large Scale Integrated Circuits via Test Points and Additional Logic, IEEE Trans. Comput. Vol. C-22, No. 1. pp. 46-60, Janvier 1973.

[Yao 93b] M. Yao, A. Petrenko et G. v. Bochmann, Conformance Testing of Protocol Machines without Reset, IFIP 13th Int. Conference on Protocol Specification, Testing and Verfication, Liege, Belgium, Mai 1993, pp.241-253, Mai 1993.

[Yao 94a] M. Yao, A. Petrenko et G. v. Bochmann, Fault Coverage Analysis in Respect to an FSM Specification, IEEE INFOCOM'94, Toronto, Canada, pp.768-775, Juin 1994.

[Yevt 89] N. Yevtushenko et A. Petrenko, Fault-Detection Capability of Multiple Experiments, Automatic Control and Computer Sciences, Allerton Press, Inc. New York, Vol.23, No.3, pp.7-11, 1989.

[Yevt 90a] N. Yevtushenko et A. Petrenko, Synthesis of Test Experiments in Some Classes of Automata, Automatic Control and Computer Sciences, Allerton Press, Inc. New York, Vol.24, No.4, pp.50-55, 1990 .

[Yevt 95a] N. Yevtushenko, A. Petrenko, R. Dssouli, K. Karoui et S. Prokopenko, On the Design for Testability of Communication Protocols, IFIP Intl Workshop on Protocol Test Systems, France, pp. 271-286, 1995.

[Yu 91] H. Yu, Testability-directed specification of communications software. Mémoire de maitrise, département d'informatique, Université d'Ottawa, Canada.

[Zhu 94] J. Zhu et S. T. Chanson, Toward Evaluating Fault Coverage of Protocol Test Sequences, In Proceedings of the International Symposium on PSTV XIV, Vancouver, Canada, 1994.

Annexe A

Application de l'analyse de la testabilité sur le protocole X.25

États (Q)

S01, S02 DTE ready (p1)
- S01 DTE opened
- S02 DTE listening

S04 DTE waiting (p2)
S04 DCE waiting (p3)
S05 Call collision (p5)
S06,S07,S08 Flow control ready (d1)
- S06 Flow control ready
- S07 Interrupt indication
- S08 Data indication

S09,S10 DTE reset request (d2)
- S09 Normal reset request
- S10 Reset error

S11 DCE reset indication (d3)
S12,S13,S14,S15,S14,S17 DTE clear request (p6)
- S12 DTE clear request by N_DISCONNECT.req
- S13 Clear error at Data transfer (p4)
- S14 Clear error with N_DISCONNECT indication
- S15 Clear error with N_CONNECT_ACK* response
- S16 Clear error without indication but back to DTE listening
- S17 Clear error without indication or DTE clear request by N_DISCONNECT _ACK.rsp

S18 DCE clear indication (p7).

Entrées (I)

- À partir de la couche 2
  X01 Incoming call
  X02 Call connected
  X03 DCE clear indication
  X04 DCE clear confirmation
  X05 DCE data
  X06 DCE interrupt
  X07 DCE interrupt confirmation
  X08 DCE RR
  X09 DCE RNR
  X10 Reset indication
  X11 DCE reset confirmation
  X12 Restart indication
  X13 DCE restart confirmation
  X14 DTE REJ, Unidentifiable packet type, Packet exceed max length, Short packet, Restart request state LCN not 0
  X15 Invalid GFI, Unassigned LCN, LCN 0 for not restart request
  X16 Data with bad P(S) or P(R)

- À partir de l'interface utilisateur:
  X17 N_CONNECT.req
  X18 N_ACCEPT.req
  X19 N_DISCONNECT.req
  X20 N_RESET.req
  X21 N_FLOW_CONTROL.req
  X22 N_FLOW_CONTROL*.req
  X23 N_CONNECT_ACK.rsp
  X24 N_CONNECT_ACK.rsp*
  X25 N_DISCONNECT_ACK.rsp
  X26 N_DATA_ACK.rsp
  X27 N_EXPDATA_ACK.rsp
  X28 N_RESET_ACK.rsp

Sorties (O)

- À la couche 2:
  Y01 Call request

Y02 Call accepted
Y03 Clear request
Y04 DTE clear confirmation
Y05 DTE data
Y06 DTE interrupt
Y07 DTE interrupt confirmation
Y08 DTE RR
Y09 DTE RNR
Y10 Reset request
Y11 DTE reset confirmation
Y12 Restart request
Y13 DTE restart confirmation

- À l'utilisateur de l'interface
  Y14 N_CONNECT.ind
  Y15 N_DISCONNECT.ind
  Y16 N_DATA.ind
  Y17 N_EXPDATA.ind
  Y18 N_RESET.ind
  Y19 N_FLOW_CONTROL.ind
  Y20 N_CONNECT_ACK.cnf
  Y21 N_ACCEPT_ACK.cnf
  Y22 N_DISCONNECT_ACK.cnf
  Y23 N_DATA_ACK.cnf
  Y24 N_EXPDATA.cnf
  Y25 N_RESET_ACK.cnf

- À l'utilisateur de l'interface et la couche 2
  Y26 Call accepted and N_FLOW_CONTROL.ind
  Y27 DTE clear confirmation and N_CONNECT_ACK.cnf
  Y28 DTE restart confirmation and N_DISCONNECT.ind
  Y29 DTE restart confirmation and N_CONNECT_ACK.cnf
  Y30 DTE N_ACCEP_ACK.cnf and N_DISCONNECT_ACK.cnf
  Y31 DTE restart confirmation and N_DISCONNECT_ACK.cnf.

R 0 1

(S01, X01) -----> (S17,Y03)
(S01, X02) -----> (S17,Y03)
(S01, X03) -----> (S01,Y04)
(S01, X04) -----> (S17,Y03)
(S01, X05) -----> (S17,Y03)
(S01, X06) -----> (S17,Y03)
(S01, X07) -----> (S17,Y03)
(S01, X08) -----> (S17,Y03)
(S01, X09) -----> (S17,Y03)
(S01, X10) -----> (S17,Y03)
(S01, X11) -----> (S17,Y03)
(S01, X12) -----> (S01,Y13)
(S01, X13) -----> (S19,Y12)
(S01, X14) -----> (S17,Y03)
(S01, X15) -----> S01
(S01, X16) -----> (S17,Y03)
(S01, X17) -----> (S03,Y01)
(S01, X18) -----> S02
(S01, X19) -----> (S01,Y22)
(S01, X20) -----> (S01,Y25)
(S01, X21) -----> S01
(S01, X22) -----> S01
(S01, X23) -----> S01
(S01, X24) -----> S01
(S01, X25) -----> S01
(S01, X26) -----> S01
(S01, X27) -----> S01
(S01, X28) -----> S01

R 0 1

Card(Ind(R01)) = 0
Card(Ind($R01^{-1}$)) = 21
Card(Dom(R01)) = 28
Card(Im(R01)) = 9
Card(Input(R01)) = 19
Card(Output(R01)) = 9
C(R01)=<1, .25, .67, 1>

R 0 2

(S02, X01) -----> (S04,Y14)
(S02, X02) -----> (S16,Y03)
(S02, X03) -----> (S02,Y04)
(S02, X04) -----> (S16,Y03)
(S02, X05) -----> (S16,Y03)
(S02, X06) -----> (S16,Y03)
(S02, X07) -----> (S16,Y03)
(S02, X08) -----> (S16,Y03)
(S02, X09) -----> (S16,Y03)
(S02, X10) -----> (S16,Y03)
(S02, X11) -----> (S16,Y03)
(S02, X12) -----> (S02,Y13)
(S02, X13) -----> (S20,Y12)
(S02, X14) -----> (S16,Y03)
(S02, X15) -----> S02
(S02, X16) -----> (S16,Y03)
(S02, X19) -----> (S01,Y22*Y2)

R 0 2

Card(Ind(R02)) = 0
Card(Ind($R02^{-1}$)) = 11
Card(Dom(R02)) = 17
Card(Im(R02)) = 7
Card(Input(R02)) = 16
Card(Output(R02)) = 7
C(R02)=<1, .36, .94, 1>

R03

(S03, X01) ------> S05
(S03, X02) ------> (S06,Y20)
(S03, X03) ------> (S17,Y03)
(S03, X04) ------> (S15,Y03)
(S03, X05) ------> (S15,Y03)
(S03, X06) ------> (S15,Y03)
(S03, X07) ------> (S15,Y03)
(S03, X08) ------> (S15,Y03)
(S03, X09) ------> (S15,Y03)
(S03, X10) ------> (S15,Y03)
(S03, X11) ------> (S15,Y03)
(S03, X12) ------> (S01,Y13*Y20)
(S03, X13) ------> (S22,Y12)
(S03, X14) ------> (S15,Y03)
(S03, X15) ------> S03
(S03, X16) ------> (S15,Y03)

R03

Card(Ind(R03)) = 0
Card(Ind($R03^{-1}$)) = 10
Card(Dom(R03)) = 16
Card(Im(R03)) = 7
Card(Input(R03)) = 15
Card(Output(R03)) = 7
C(R03)=<1, .38, .93, 1>

R04

(S04, X01) ------> (S14,Y03)
(S04, X02) ------> (S14,Y03)
(S04, X03) ------> (S18,Y15)
(S04, X04) ------> (S14,Y03)
(S04, X05) ------> (S14,Y03)
(S04, X06) ------> (S14,Y03)
(S04, X07) ------> (S14,Y03)
(S04, X08) ------> (S14,Y03)
(S04, X09) ------> (S14,Y03)
(S04, X10) ------> (S14,Y03)
(S04, X11) ------> (S14,Y03)
(S04, X12) ------> (S01,Y13*Y15)
(S04, X13) ------> (S21,Y12)
(S04, X14) ------> (S14,Y03)
(S04, X15) ------> S04
(S04, X16) ------> (S14,Y03)
(S04, X19) ------> (S04,Y22)
(S04, X20) ------> (S04,Y25)
(S04, X21) ------> S04
(S04, X22) ------> S04
(S04, X23) ------> (S06,Y02*Y19)
(S04, X24) ------> (S17,Y03)
(S04, X25) ------> S04
(S04, X26) ------> S04
(S04, X27) ------> S04
(S04, X28) ------> S04

R04

Card(Ind(R04)) = 0
Card(Ind($R04^{-1}$)) = 19
Card(Dom(R04)) = 26
Card(Im(R04)) = 9
Card(Input(R04)) = 19
Card(Output(R04)) = 9
C(R04)=<1, .27, .73, 1>

R05

(S05, X01) -----> (S15,Y03)
(S05, X02) -----> (S06,Y20)
(S05, X03) -----> (S01,Y20*Y04)
(S05, X04) -----> (S15,Y03)
(S05, X05) -----> (S15,Y03)
(S05, X06) -----> (S15,Y03)
(S05, X07) -----> (S15,Y03)
(S05, X08) -----> (S15,Y03)
(S05, X09) -----> (S15,Y03)
(S05, X10) -----> (S15,Y03)
(S05, X11) -----> (S15,Y03)
(S05, X12) -----> (S01,Y13*Y20)
(S05, X13) -----> (S22,Y12)
(S05, X14) -----> (S15,Y03)
(S05, X15) -----> S05
(S05, X16) -----> (S14,Y03)

R05

Card(Ind(R05)) = 0
Card(Ind($R05^{-1}$)) = 10
Card(Dom(R05)) = 16
Card(Im(R05)) = 7
Card(Input(R05)) = 15
Card(Output(R05)) = 7
C(R05)=<1, .38, .93, 1>

R06

(S06, X01) -----> (S13,Y03)
(S06, X02) -----> (S13,Y03)
(S06, X03) -----> (S18,Y15)
(S06, X04) -----> (S13,Y03)
(S06, X05) -----> (S08,Y16)
(S06, X06) -----> (S07,Y17)
(S06, X07) -----> (S10,Y10)
(S06, X08) -----> (S10,Y10)
(S06, X09) -----> (S06,Y19)
(S06, X10) -----> (S11,Y18)
(S06, X11) -----> (S17,Y03)
(S06, X12) -----> (S01,Y13*Y15)
(S06, X13) -----> (S21,Y12)
(S06, X14) -----> (S10,Y10)
(S06, X15) -----> S06
(S06, X16) -----> (S10,Y10)
(S06, X19) -----> (S12,Y03)
(S06, X20) -----> (S09,Y10)
(S06, X21) -----> (S06,Y08)
(S06, X22) -----> (S06,Y09)
(S06, X23) -----> S06
(S06, X24) -----> S06
(S06, X25) -----> S06
(S06, X26) -----> S06
(S06, X27) -----> S06
(S06, X28) -----> S06

R06

Card(Ind(R06)) = 0
Card(Ind($R06^{-1}$)) = 14
Card(Dom(R06)) = 26
Card(Im(R06)) = 15
Card(Input(R06)) = 19
Card(Output(R06)) = 15
C(R06)=<1, .36, .94, 1>

R07

(S07, X01) ------> (S13,Y03)
(S07, X02) ------> (S13,Y03)
(S07, X03) ------> (S18,Y15)
(S07, X04) ------> (S13,Y03)
(S07, X06) ------> (S10,Y10)
(S07, X07) ------> (S10,Y10)
(S07, X10) ------> (S11,Y18)
(S07, X11) ------> (S10,Y10)
(S07, X12) ------> (S01,Y13*Y15)
(S07, X13) ------> (S21,Y12)
(S07, X14) ------> (S10,Y10)
(S07, X15) ------> S07
(S07, X19) ------> (S12,Y03)
(S07, X20) ------> (S09,Y10)
(S07, X21) ------> (S07,Y08)
(S07, X22) ------> (S07,Y09)
(S07, X23) ------> S07
(S07, X24) ------> S07
(S07, X25) ------> S07
(S07, X27) ------> (S06,Y07)
(S07, X28) ------> S07

R07

Card(Ind(R07)) = 0
Card(Ind($R07^{-1}$)) = 12
Card(Dom(R07)) = 21
Card(Im(R07)) = 12
Card(Input(R07)) = 16
Card(Output(R07)) = 12
C(R07)=<1, .43, .76, 1>

R08

(S08, X01) ------> (S13,Y03)
(S08, X02) ------> (S13,Y03)
(S08, X03) ------> (S18,Y15)
(S08, X04) ------> (S13,Y03)
(S08, X10) ------> (S11,Y18)
(S08, X11) ------> (S10,Y10)
(S08, X12) ------> (S01,Y13*Y15)
(S08, X13) ------> (S21,Y12)
(S08, X14) ------> (S10,Y10)
(S08, X15) ------> S08
(S08, X19) ------> (S12,Y03)
(S08, X20) ------> (S09,Y10)
(S08, X21) ------> (S08,Y08)
(S08, X22) ------> (S08,Y09)
(S08, X23) ------> S08
(S08, X24) ------> S08
(S08, X25) ------> S08
(S08, X26) ------> S06,Y08)
(S08, X27) ------> S08
(S08, X28) ------> S08

R08

Card(Ind(R08)) = 0
Card(Ind($R08^{-1}$)) = 11
Card(Dom(R08)) = 20
Card(Im(R08)) = 12
Card(Input(R08)) = 14
Card(Output(R08)) = 12
C(R08)=<1, .45, .7, 1>

R09

(S09, X01) ------> (S13,Y03)
(S09, X02) ------> (S13,Y03)
(S09, X03) ------> (S18,Y15)
(S09, X04) ------> (S13,Y03)
(S09, X05) ------> S09
(S09, X06) ------> S09
(S09, X07) ------> S09
(S09, X08) ------> S09
(S09, X09) ------> S09
(S09, X10) ------> S09
(S09, X11) ------> (S06,Y25)
(S09, X12) ------> (S01,Y13*Y15)
(S09, X13) ------> (S21,Y12)
(S09, X14) ------> (S17,Y03)
(S09, X15) ------> S09
(S09, X16) ------> S09

R09

Card(Ind(R09)) = 0
Card(Ind($R09^{-1}$)) = 11
Card(Dom(R09)) = 16
Card(Im(R09)) = 7
Card(Input(R09)) = 8
Card(Output(R09)) = 7
C(R09)=<1, .20, .5, 1>

R10

(S10, X01) ------> (S13,Y03)
(S10, X02) ------> (S13,Y03)
(S10, X03) ------> (S18,Y15)
(S10, X04) ------> (S13,Y03)
(S10, X05) ------> S10
(S10, X06) ------> S10
(S10, X07) ------> S10
(S10, X08) ------> S10
(S10, X09) ------> S10
(S10, X10) ------> (S06,Y18)
(S10, X11) ------> (S06,Y18)
(S10, X12) ------> (S01,Y13*Y15)
(S10, X13) ------> (S21,Y12)
(S10, X14) ------> S10
(S10, X15) ------> S10
(S10, X16) ------> S10
(S10, X19) ------> (S12,Y03)
(S10, X20) ------> (S09,Y10)
(S10, X21) ------> S10
(S10, X22) ------> S10
(S10, X23) ------> S10
(S10, X24) ------> S10
(S10, X25) ------> S10
(S10, X26) ------> S10
(S10, X27) ------> S10
(S10, X28) ------> S10

R10

Card(Ind(R10)) = 0
Card(Ind($R10^{-1}$)) = 21
Card(Dom(R10)) = 26
Card(Im(R10)) = 8
Card(Input(R10)) = 10
Card(Output(R10)) = 8
C(R10)=<1, .2, .38, 1>

R11

(S11, X01) ------> (S13,Y03)
(S11, X02) ------> (S13,Y03)
(S11, X03) ------> (S18,Y15)
(S11, X04) ------> (S13,Y03)
(S11, X05) ------> (S10,Y10)
(S11, X06) ------> (S10,Y10)
(S11, X07) ------> (S10,Y10)
(S11, X08) ------> (S10,Y10)
(S11, X09) ------> (S10,Y10)
(S11, X10) ------> S11
(S11, X11) ------> (S10,Y10)
(S11, X12) ------> (S01,Y13*Y15)
(S11, X13) ------> (S21,Y12)
(S11, X14) ------> (S10,Y10)
(S11, X15) ------> S11
(S11, X16) ------> (S10,Y10)
(S11, X19) ------> (S12,Y03)
(S11, X20) ------> (S06,Y10)
(S11, X21) ------> S11
(S11, X22) ------> S11
(S11, X23) ------> S11
(S11, X24) ------> S11
(S11, X25) ------> S11
(S11, X26) ------> S11
(S11, X27) ------> S11
(S11, X28) ------> (S06,Y11)

R11

Card(Ind(R11)) = 0
Card(Ind($R11^{-1}$)) = 20
Card(Dom(R11)) = 26
Card(Im(R11)) = 9
Card(Input(R11)) = 17
Card(Output(R11)) = 9
C(R11)=<1, .24, .65, 1>

R12

(S12, X01) ------> S12
(S12, X02) ------> S12
(S12, X03) ------> (S01,Y15)
(S12, X04) ------> (S01,Y22)
(S12, X05) ------> S12
(S12, X06) ------> S12
(S12, X07) ------> S12
(S12, X08) ------> S12
(S12, X09) ------> S12
(S12, X10) ------> S12
(S12, X11) ------> S12
(S12, X12) ------> (S01,Y13*Y22)
(S12, X13) ------> (S23,Y12)
(S12, X14) ------> S12
(S12, X15) ------> S12
(S12, X16) ------> S12

R12

Card(Ind(R12)) = 0
Card(Ind($R12^{-1}$)) = 12
Card(Dom(R12)) = 16
Card(Im(R12)) = 5
Card(Input(R12)) = 4
Card(Output(R12)) = 5
C(R12)=<1, .25, .25, 1>

R13

(S13, X01) ------> S13
(S13, X02) ------> S13
(S13, X03) ------> (S01,Y15)
(S13, X04) ------> (S01,Y15)
(S13, X05) ------> S13
(S13, X06) ------> S13
(S13, X07) ------> S13
(S13, X08) ------> S13
(S13, X09) ------> S13
(S13, X10) ------> S13
(S13, X11) ------> S13
(S13, X12) ------> (S01,Y13*Y15)
(S13, X13) ------> (S21,Y12)
(S13, X14) ------> S13
(S13, X15) ------> S13
(S13, X16) ------> S13
(S13, X19) ------> (S12,Y03)
(S13, X20) ------> S13
(S13, X21) ------> S13
(S13, X22) ------> S13
(S13, X23) ------> S13
(S13, X24) ------> S13
(S13, X25) ------> S13
(S13, X26) ------> S13
(S13, X27) ------> S13
(S13, X28) ------> S13

R13

Card(Ind(R13)) = 0
Card(Ind($R13^{-1}$)) = 23
Card(Dom(R13)) = 26
Card(Im(R13)) = 5
Card(Input(R13)) = 5
Card(Output(R13)) = 5
C(R13)=<1, .12, .19, 1>

R14

(S14, X01) ------> S14
(S14, X02) ------> S14
(S14, X03) ------> (S01,Y15)
(S14, X04) ------> (S01,Y15)
(S14, X05) ------> S14
(S14, X06) ------> S14
(S14, X07) ------> S14
(S14, X08) ------> S14
(S14, X09) ------> S14
(S14, X10) ------> S14
(S14, X11) ------> S14
(S14, X12) ------> (S01,Y13*Y15)
(S14, X13) ------> (S21,Y12)
(S14, X14) ------> S14
(S14, X15) ------> S14
(S14, X16) ------> S14
(S14, X19) ------> (S14,Y22)
(S14, X20) ------> (S17,Y25)
(S14, X21) ------> S14
(S14, X22) ------> S14
(S14, X23) ------> S14
(S14, X24) ------> S14
(S14, X25) ------> S14
(S14, X26) ------> S14
(S14, X27) ------> S14
(S14, X28) ------> S14

R14

Card(Ind(R14)) = 0
Card(Ind($R14^{-1}$)) = 22
Card(Dom(R14)) = 26
Card(Im(R14)) = 6
Card(Input(R14)) = 6
Card(Output(R14)) = 6
C(R14)=<1, .16, .23, 1>

R15

(S15, X01) ------> S15
(S15, X02) ------> S15
(S15, X03) ------> (S01,Y20)
(S15, X04) ------> (S01,Y20)
(S15, X05) ------> S15
(S15, X06) ------> S15
(S15, X07) ------> S15
(S15, X08) ------> S15
(S15, X09) ------> S15
(S15, X10) ------> S15
(S15, X11) ------> S15
(S15, X12) ------> (S01,Y13*Y20)
(S15, X13) ------> (S22,Y12)
(S15, X14) ------> S15
(S15, X15) ------> S15
(S15, X16) ------> S15

R15

Card(Ind(R15)) = 0
Card(Ind($R15^{-1}$)) = 14
Card(Dom(R15)) = 16
Card(Im(R15)) = 4
Card(Input(R15)) = 4
Card(Output(R15)) = 4
C(R15)=<1, .13, .25, 1>

R16

(S16, X01) ------> S16
(S16, X02) ------> S16
(S16, X03) ------> S02
(S16, X04) ------> S02
(S16, X05) ------> S16
(S16, X06) ------> S16
(S16, X07) ------> S16
(S16, X08) ------> S16
(S16, X09) ------> S16
(S16, X10) ------> S16
(S16, X11) ------> S16
(S16, X12) ------> (S02,Y13)
(S16, X13) ------> (S20,Y12)
(S16, X14) ------> S16
(S16, X15) ------> S16
(S16, X16) ------> S16

R16

Card(Ind(R16)) = 0
Card(Ind($R16^{-1}$)) = 14
Card(Dom(R16)) = 16
Card(Im(R16)) = 4
Card(Input(R16)) = 4
Card(Output(R16)) = 4
C(R16)=<1, .13, .25, 1>

R17

(S17, X01) ------> S17
(S17, X02) ------> S17
(S17, X03) ------> S01
(S17, X04) ------> S01
(S17, X05) ------> S17
(S17, X06) ------> S17
(S17, X07) ------> S17
(S17, X08) ------> S17
(S17, X09) ------> S17
(S17, X10) ------> S17
(S17, X11) ------> S17
(S17, X12) ------> (S01,Y13)
(S17, X13) ------> (S19,Y12)
(S17, X14) ------> S17
(S17, X15) ------> S17
(S17, X16) ------> S17
(S17, X19) ------> S17
(S17, X20) ------> (S17,Y25)
(S17, X21) ------> S17
(S17, X22) ------> S17
(S17, X23) ------> S17
(S17, X24) ------> S17
(S17, X25) ------> S17
(S17, X26) ------> S17
(S17, X27) ------> S17
(S17, X28) ------> S17

R17

Card(Ind(R17)) = 0
Card(Ind($R17^{-1}$)) = 23
Card(Dom(R17)) = 26
Card(Im(R17)) = 5
Card(Input(R17)) = 5
Card(Output(R17)) = 5
C(R17)=<1, .12, .19, 1>

R18

(S18, X01) ------> (S17,Y03)
(S18, X02) ------> (S17,Y03)
(S18, X03) ------> S18
(S18, X04) ------> (S17,Y03)
(S18, X05) ------> (S17,Y03)
(S18, X06) ------> (S17,Y03)
(S18, X07) ------> (S17,Y03)
(S18, X08) ------> (S17,Y03)
(S18, X09) ------> (S17,Y03)
(S18, X10) ------> (S17,Y03)
(S18, X11) ------> (S17,Y03)
(S18, X12) ------> (S01,Y13)
(S18, X13) ------> (S19,Y12)
(S18, X14) ------> (S17,Y03)
(S18, X15) ------> S18
(S18, X16) ------> (S17,Y03)
(S18, X19) ------> (S01,Y03)
(S18, X20) ------> (S18,Y25)
(S18, X21) ------> S18
(S18, X22) ------> S18
(S18, X23) ------> S18
(S18, X24) ------> S18
(S18, X25) ------> (S01,Y04)
(S18, X26) ------> S18
(S18, X27) ------> S18
(S18, X28) ------> S18

R18

Card(Ind(R18)) = 0
Card(Ind($R18^{-1}$)) = 21
Card(Dom(R18)) = 26
Card(Im(R18)) = 7
Card(Input(R18)) = 17
Card(Output(R18)) = 7
C(R18)=<1, .2, .65, 1>

Annexe B

Algorithme des transformations de testabilité des FSMs

S-trans Algorithm

```
Début
DC <-- Ensemble des transitions à compléter;
 I_d <-- Ensemble des entrées des transitions à compléter;
     Pour chaque i ∈ I_d
        Faire
        P_all <-- {out : ∀qj h^2(qj, i) =out };  P_s <-- {out : h^1(qj, i)=qj et  h^2(qj, i)
=out };
        Pour chaque (q, i) ∈ DC
          Faire
          P <-- {out : h^1(qj, i)=q et  h^2(qj, i) = out };
          h^1(q, i) = q;
          Si  (∃ o ∉ P_all )  Alors h^2(q, i) = o; h^1(q, i)=q
            Sinon Si  (∃ o ∉ P_s) Alors h^2(q, i) = o
                   Sinon Si  (∃ o ∉ P)  Alors h^2(q, i) = o
                          Sinon  h^2(q, i) = null
                         FinSi
                   FinSi
          FinSi
          Suivant q
        FinFaire
        Suivant i
    FinFaire
Fin.
```

C-trans Algorithm

```
Début
S0<-- Spécification initiale;
i=1;
Extraire(Ci);   /*Ci<-- Un ensemble de compatibles [KOHA70]*/
Extraire(Cmax);                    /* Cmax={Ci} */
G(S0)<-- Nombre de chemins indépendants de S0;
P<-- {Pq} /* Extraire de Cmax les ensembles fermés de comp. couvrant les états Q de S0
Pour chaque Pq
        Faire
        Mq <-- Machine extraite de Pq;
        Si UTM(Mq)<UTM(S0)
            Alors
            T(i)<-- (q, UTM(Mq), G(Mq)); /* table contenant l'indexe, la testabilité le
                                          /* nombre de chemins indépendants de la machine réduite
            i<--i+1;
        Finsi;
Suivant Pq;
FinFaire;
Si T est vide Alors sortir /* il n'y a pas de machines minimales qui améliorent la
testabilité
Finsi
Trier(T, UTM, i-1); /*trier la table T suivant la testabilité
i <-- longueur(T) div 2;
Trier(T, G, i); /* trier la première moitié de T suivant le nombre de chemins indépendants
Résultat(T(1));
Fin.
```

H-trans Algorithm

```
Début
 maxdisting <-- ' '; seqdisting <-- ' ';
 DC <-- Ensemble des transitions à compléter;
 Id <-- Ensemble des entrées des transitions à compléter;
 Pour chaque i ∈ Id
 Faire
          Qi <-- l'ensemble des états qui accepte l'entrée i;
          Q'i <-- l'ensemble des états qui accepte la séquence i*;
          Mi<-- (Q, {i}, O, hi, DMi)          /* projection unitaire de M sur i */
          Si Mi not reduced
                     Alors Suivant i;
                     Sinon
                        Pour chaque (q, i) ∈ DC
                        Faire
```

```
                    A <-- L'ensemble des états à partir des quels on peut atteindre q;
                    M'i <-- (Q'i∪A, {i}, O, h'i, DM'i);
                    assigner les transitions relativement au quatre cas (Cas 1 ou
                    Cas2 ou Cas3 ou Cas4) cas qui suivent [Yevt 95]
                    disting <-- calcdisting (M'i, 'i') /* séq. de distinction de M'i */;
                    Si |disting| > |maxdisting| Alors maxdisting <-- disting FinSi;
                    Suivant q
                FinFaire;
                Si |maxdisting| < |seqdisting|
                     Alors seqdisting <-- maxdisting; M' <-- complete(M, M'i)
                FinSi
                Suivant i;
        FinSi
 FinFaire
 Si seqdisting ≠ ' ' Alors retourner (M', seqdisting)
  Sinon chercher le meilleur second cas
 FinSi
Fin.
```

Cas 1. (q, i) est une transition "don't care". Tous les chemins se terminent en un état q. M_i (projection unitaire de M sur i) n'a pas de cycle. On détermine le plus long chemin q_1->q_2->...->q_k->q et on défini une transition de l'état q avec l'entrée i: $h^1(q, i) = q_1$ et $h^2(q, i) = o$, où o est une sortie tel que $h^2(q_1, i)$, ..., $h^2(q_k, i)o$ ne peut pas être représenté comme un préfixe propre répété plusieurs fois. Si $|O|>1$ alors il est toujours possible de trouver cette sortie.

Cas 2. (q, i) est une transition "don't care". M_i possède des cycles, mais tous les chemins se terminent en un état q. Dans ce cas, on prend un cycle arbitraire (q_1->q_2->...->q_k) et on défini une transition de l'état q vers q_1, c. à d., $h^1(q, i) = q_1$, avec la sortie $h^2(q, i) = o$ tel que $o \neq h^2(s_k, x)$.

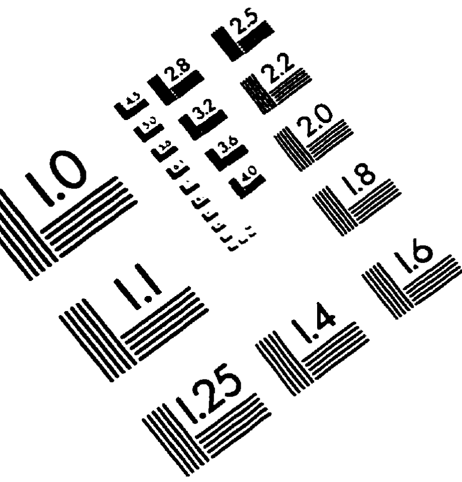
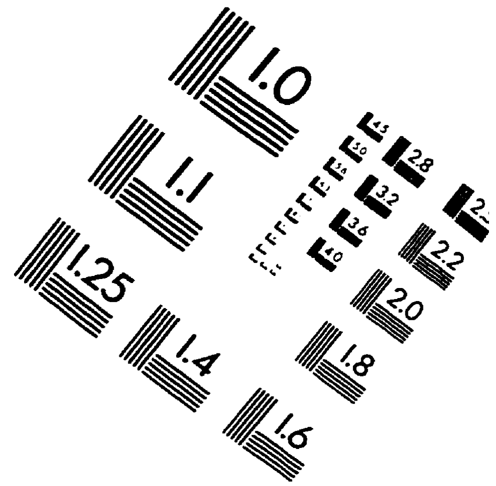
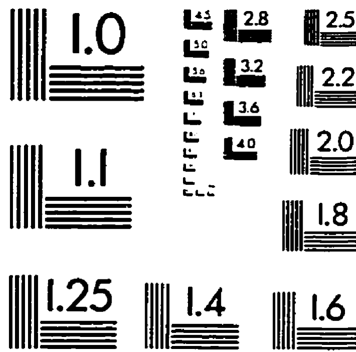
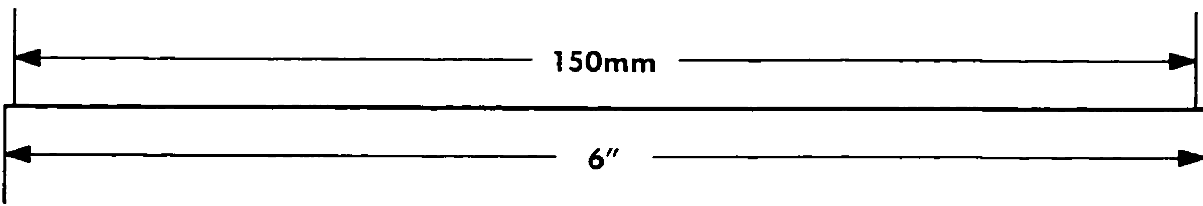
Cas 3. M_i a des chemins cyclants, mais aucun de ces chemins n'a un cycle dominant. Dans ce cas, l'état de départ de caque chemin cyclant est équivalent avec un autre état de ce chemin. Pour assigner la transition (q, i) on choisi un chemin arbitraire (q_1->...->q_{i+1}->...->q_k) qui se termine par le cycle (q_{i+1}->...->q_k), où $i \geq 1$. Puisque q_1 est équivalent à un certain état du chemin, il est aussi équivalent à un état q_j du cycle, $i+1 \leq j \leq k$, on assigne alors: $h^1(q, i) = q_j$ et $h^2(q, i) = o$ tel que $o \neq h^2(q_{j-1}, i)$. Si j=1 alors $o \neq h^2(q_k, i)$.

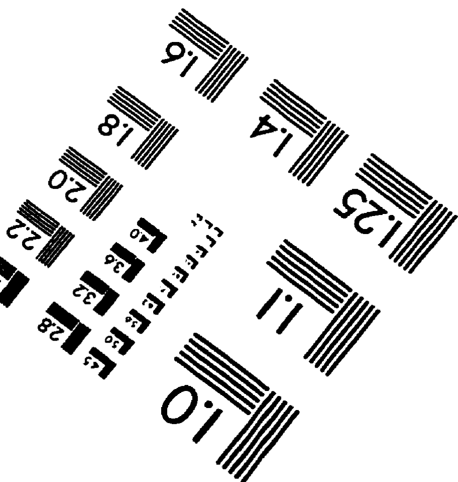
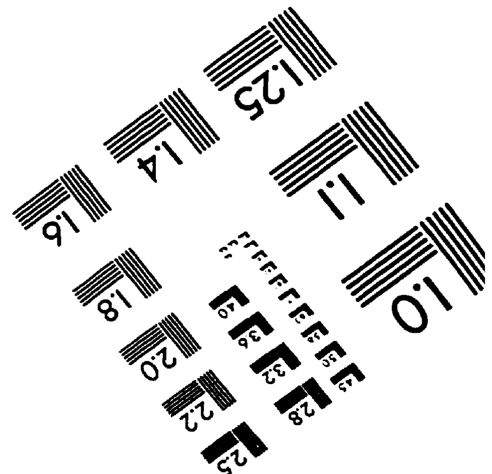
Cas 4. M_i a au moins un chemin avec un cycle dominant. Dans ce cas, il existe un chemin cyclant P = (q_1->...q_{i+1}->...->q_k), $i \geq 1$ tel que q_1 n'est pas équivalent avec aucun des états de P. En plus, seulement un autre état de départ d'un chemin cyclant peut êtr équivalent à q_1. On assigne la transition (q, i) comme suit: $h^1(q, i) = q_1$ et $h^2(q, i)$ n'importe quelle sortie $o \in O$.

K-trans Algorithm

```
Début
   So <--- First_spec; S1 <-- D-trans(So);
   Si det(S1) Alors So <--- S1;  To <--Testability(So) FinSi;  /* Si S1 est déterministe */
   S1 <-- C-trans(So); T1 <-- Testability(S1);
   Si T1>To  Alors So <--- S1;  /* Si testabilité de S1 est meilleure que So */
   To <-- T1
    FinSi;
   S1 <-- H-trans(So); T1 <-- Testability(S1);
   Si T1>To  Alors So <--- S1;  To <--T1   /* Si testabilité est meilleure que So */
         Sinon  S1 <--S-trans(S0)
   FinSi;
Fin
```

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved