N°d'Ordre : 738 Année 2004

THÈSE

préparée au

Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS) du CNRS

en vue de l'obtention du titre de

Docteur de l'Institut National des Sciences Appliquées de Toulouse

Spécialité : Systèmes Informatiques

par **Guillaume Lussier**
Ingénieur INSA Toulouse

Test Guidé par la Preuve

Application à la vérification d'algorithmes de tolérance aux fautes

Soutenue le 17 septembre 2004 devant le jury :

Président	M.	**Jean-Pierre**	**Elloy**
Rapporteurs	M.	**Richard**	**Castanet**
	Mme	**Isabelle**	**Puaut**
Examinateurs	M.	**Bruno**	**Marre**
	Mme	**Pascale**	**Thévenod-Fosse**
Directeur de thèse	Mme	**Hélène**	**Waeselynck**

Cette thèse a été préparée au LAAS-CNRS,
dans le groupe Tolérance aux fautes et Sûreté de Fonctionnement Informatique
7, Avenue du Colonel Roche, 31077 Toulouse Cedex 4

Avant-propos

Les travaux présentés dans ce mémoire ont été effectués au sein du Laboratoire d'Analyse et d'Architecture des Systèmes du Centre National de la Recherche Scientifique (LAAS-CNRS).
J'exprime toute ma reconnaissance à Messieurs Jean-Claude Laprie et Malik Ghallab, qui ont successivement assuré la direction du LAAS-CNRS depuis mon entrée, pour m'avoir accueilli au sein de ce laboratoire.
Je remercie également Messieurs David Powell et Jean Arlat, responsables successifs du groupe de recherche Tolérance aux fautes et Sûreté de Fonctionnement informatique (TSF), pour m'avoir permis de réaliser ces travaux dans ce groupe.

J'exprime ma profonde reconnaissance à Hélène Waeselynck, Chargée de Recherche CNRS, pour l'encadrement de mes travaux de thèse. Sa compétence et sa rigueur intellectuelle ont été un exemple tout au long de cette thèse. Ces travaux n'auraient pas vu le jour sans son concours et ses conseils, qu'elle en soit ici remerciée.

J'exprime également ma gratitude à :

- Monsieur Richard Castanet, Professeur à l'ENSEIRB
- Monsieur Jean-Pierre Elloy, Professeur à l'Ecole Centrale de Nantes
- Monsieur Bruno Marre, Docteur au CEA Centre de Saclay
- Madame Isabelle Puaut, Professeur à l'Université de Rennes I
- Madame Pascale Thévenod-Fosse, Directeur de Recherche CNRS
- Madame Hélène Waeselynck, Chargée de Recherche CNRS

pour l'honneur qu'ils me font en participant à mon Jury, et plus particulièrement à Monsieur Jean-Pierre Elloy, qui m'a fait l'honneur de présider le jury, et à Monsieur Richard Castanet et Madame Isabelle Puaut, qui ont accepté la charge d'être rapporteur, en dépit de leurs nombreuses obligations.

Je remercie également l'ensemble des services techniques et administratifs du LAAS-CNRS, pour leur soutien efficace qui m'a permis de me consacrer pleinement à mes travaux de recherche. Je tiens à remercier personnellement Madame Joëlle Penavayre, Secrétaire du groupe TSF, qui a largement contribué à me procurer ces excellentes conditions de travail par sa disponibilité et sa gentillesse.

Une place spéciale revient aussi à Monsieur Holger Pfeifer, qui m'a fourni le cas d'étude utilisé dans la partie de mes travaux liée aux preuves formelles. Son aide m'a été précieuse pour comprendre et analyser cette preuve.

Mes remerciements s'adressent également à l'ensemble des membres du groupe TSF, permanents, doctorants et stagiaires, avec lesquels j'ai partagé ces années de travail. Je tiens particulièrement à remercier Jean Arlat pour son soutien scientifique et moral, qui m'a été très précieux.
Parmi beaucoup, j'ai partagé toute cette thèse avec Eric Marsden et Cristina Simache, et ils ont toujours été un support pour moi. Mais je pense aussi à tous les autres membres de ce groupe de recherche qui ont compté pour moi durant ces cinqs années : Olfa Abdellatif-Kaddour, Arnaud Albinet, Salimeh Behnia, Claudia Betous-Almeida, Ludovic Courtès, Agnan De Bonneval, Jérémie Guiochet, Marc-Olivier Killijian, Magnos Martinello, Vincent Nicomette, Mourad Rabah, Christophe Zanon, et tous ceux que j'oublie sans doute.
Enfin, je n'oublie pas Karim Guennoun, et Céline Gil, qui ont travaillé tous deux, lors de leurs stages respectifs, sur les expérimentations liées à mes travaux de thèse.

Cet avant-propos ne serait pas complet sans adresser mes profonds remerciements à l'équipe du Département de Mathématique et d'Informatique de l'Université de Toulouse le Mirail, ainsi qu'à l'équipe de recherche ISYCOM. Je remercie notamment leur Directeur, Monsieur Bernard Coulette, pour m'avoir accueilli lors de ma dernière année de thèse en tant qu'Attaché Temporaire d'Enseignement et de Recherche, mais aussi tous les membres de l'équipe pour leur sympathie et leur soutien.

Pour finir, je remercie profondément tous ceux qui ont pu m'apporté courage et gentillesse : mes parents, tous mes amis qui seraient trop nombreux à citer mais que je n'oublie pas, et enfin mon frère Benjamin à qui je souhaite une poursuite heureuse de ses propres travaux de thèse. C'est à eux tous que cette thèse est dédiée.

Toulouse, le 17 janvier 2005

Table des matières

Avant-propos iii

Introduction générale 1

1 Cadre des travaux et état de l'art 5

1.1 Concepts de base de la sûreté de fonctionnement 6

1.2 Élimination des fautes 7

1.2.1 L'analyse statique 8

1.2.2 La vérification de modèle 9

1.2.3 La preuve 11

1.2.3.1 Preuves informelles 11

1.2.3.2 Preuves formelles 12

1.2.4 Le test 16

1.2.4.1 Notions générales 16

1.2.4.2 Test structurel 17

1.2.4.3 Test fonctionnel 18

1.2.4.4 Génération probabiliste d'entrées de test 20

1.2.4.5 Analyse de mutation 22

1.2.5 Couplage de techniques de vérification 23

1.2.5.1 Couplages autour du model-checking 23

1.2.5.2 Couplage entre le test et la preuve 24

1.3 Tolérance aux fautes 26

1.3.1 Aspects généraux de la tolérance aux fautes 26

1.3.2 La tolérance aux fautes dans les systèmes répartis 27

1.4 Validation de la tolérance aux fautes 28

1.4.1 Évaluation de la tolérance aux fautes 29
1.4.2 Vérification de la tolérance aux fautes 30
1.5 Conclusion et approche proposée 31

2 Test guidé par une preuve informelle 33
2.1 Présentation générale de l'approche proposée 34
2.2 Étape 1 : Analyse préliminaire 36
2.3 Étape 2 : Restructuration de la preuve 37
2.3.1 Représentation sous forme d'arbre 37
2.3.2 Exemple d'arbre de preuve 39
2.3.3 Identification des faiblesses de la preuve 40
2.4 Étape 3 : Test Guidé par la Preuve 41
2.4.1 Des faiblesses de la preuve au test de l'algorithme 41
2.4.2 Mise en œuvre du test statistique 43
2.5 Étape 4 : Retour sur la preuve 43
2.6 Conclusion 44

3 Premier cas d'étude d'une preuve informelle : FT-RMS 47
3.1 Présentation du premier cas d'étude : le FT-RMS 48
3.1.1 L'algorithme RMS 49
3.1.2 Deux versions de l'algorithme FT-RMS 50
3.1.3 Vérification du FT-RMS, résultats antérieurs 53
3.2 Étape 1 : Analyse Préliminaire 56
3.3 Étape 2 : Restructuration de la preuve 58
3.3.1 Structure générale de la preuve 58
3.3.2 Preuves des trois lemmes principaux 59
3.3.3 Bilan des faiblesses identifiées 62
3.4 Étape 3 : Test Guidé par la Preuve 62
3.4.1 Critère de test basé sur les cas de preuve 63
3.4.2 Autres critères de test 65
3.4.3 Résultats expérimentaux 65
3.5 Étape 4 : Retour sur la preuve 69
3.6 Conclusion 69

4 Deuxième cas d'étude d'une preuve informelle : GMP **71**
4.1 Présentation du Group Membership Protocol étudié 72
4.1.1 Contexte 72
4.1.2 Environnement de l'algorithme et modèle de fautes 73
4.1.3 L'algorithme du GMP 74
4.1.4 Propriétés attendues 76
4.2 Étape 1 : Analyse Préliminaire 76
4.3 Étape 2 : Restructuration de la preuve 79
4.3.1 Aperçu global de la preuve informelle 80
4.3.2 Preuve du Théorème 3 81
4.3.2.1 Notations et structure globale de l'arbre 82
4.3.2.2 Détail de la restructuration : racine 83
4.3.2.3 Détail de la restructuration : Branche 1 85
4.3.2.4 Détail de la restructuration : Branche 1.1 87
4.3.2.5 Détail de la restructuration : Branche 1.1.1.1 88
4.3.2.6 Détail de la restructuration : Branche 1.1.1.2 88
4.3.2.7 Détail de la restructuration : Branche 2 90
4.3.3 Preuve du Conjoint (5) 91
4.3.4 Bilan des faiblesses identifiées 92
4.4 Étape 3 : Test Guidé par la Preuve 93
4.4.1 Critères de sélection des entrées de test 93
4.4.2 Résultats expérimentaux 94
4.4.2.1 Test global du GMP 94
4.4.2.2 Test spécifique des faiblesses du Théorème 3 95
4.5 Étape 4 : Retour sur la preuve 96
4.5.1 Retour sur la preuve du Conjoint (5) 96
4.5.2 Retour sur la preuve du Théorème 3 97
4.6 Conclusion 99

5 Test guidé par une preuve formelle **103**
5.1 Application de l'approche à une preuve formelle 104
5.2 Injection d'une faute de conception 105
5.2.1 Partir d'un algorithme incorrect 106
5.2.2 Injecter une faute de conception dans une spécification et une preuve formelle complètes 107
5.3 Présentation du cas d'étude : GMP incorporé à la TTA 108
5.3.1 Différences avec le GMP informel 109
5.3.2 Spécification de l'algorithme de GMP du TTP/C 109
5.4 Analyse de haut niveau 112
5.4.1 Modèle de fautes et domaine d'entrée 113
5.4.1.1 Les axiomes du modèle de fautes 113
5.4.1.2 Résultat de l'analyse des axiomes 114
5.4.1.3 Définition du domaine d'entrée de test 114
5.4.2 Propriétés attendues et oracle de test 115
5.4.2.1 Propriétés de haut niveau devant être garanties par le GMP 115
5.4.2.2 Extension des propriétés 117
5.4.2.3 Définition de l'oracle de test 117
5.4.3 Test aléatoire aveugle de la deuxième version du GMP 118
5.5 Analyse détaillée 119
5.5.1 Technique des invariants disjonctifs 119
5.5.2 Diagramme des configurations du GMP 120
5.5.3 Présentation de la preuve formelle du GMP 122
5.5.4 Injection d'une faute de conception dans la spécification formelle de l'algorithme 123
5.5.4.1 Choix d'une expérience d'injection 124
5.5.4.2 Modifications du couple spécification et preuve formelles125
5.5.4.3 Analyse de la preuve partielle 126
5.6 Test guidé par la preuve 126
5.6.1 Premier profil de test, niveau 1 127
5.6.1.1 Conception du profil 127
5.6.1.2 Résultats expérimentaux 129
5.6.2 Deuxième profil de test, niveau 2 129

5.6.2.1 Résultats de l'analyse de niveau 2 130
5.6.2.2 Conception d'un profil de test 130
5.6.2.3 Résultats expérimentaux 132
5.7 Retour sur la preuve 133
5.7.1 De l'extraction d'informations à partir d'une preuve formelle . . 133
5.7.2 Modification d'un arbre de preuve en vue du test 135
5.8 Conclusion 136

Conclusion générale et Perspectives **139**

Annexe A : Preuve formelle PVS du diagnostic du GMP **145**

Bibliographie **149**

Table des figures

1.1 L'arbre de la sûreté de fonctionnement 6
1.2 Moyens de mise en oeuvre de la tolérance aux fautes 26

2.1 Vue générale du Test Guidé par la Preuve 35
2.2 Principales règles d'inférence utilisées 38
2.3 Exemple de restructuration sous forme d'arbres de preuve 40
2.4 Faiblesse identifiée dans un arbre de preuve 41

3.1 Ordonnançabilité d'un ensemble de tâches selon l'IBRMS 52
3.2 Ordonnancement FT-RMS, versions 1 et 2 53
3.3 Echec des versions 1 et 2 du FT-RMS 55
3.4 Ensemble de tâches sans échec 56
3.5 Racine de la preuve informelle du FT-RMS 59
3.6 Arbre de la preuve informelle de la condition $[S_1]$ 60
3.7 Arbre de la preuve informelle de la condition $[S_2]$ 60
3.8 Arbre de la preuve informelle de la condition $[S_3]$ 61
3.9 Entrée 9095 du jeu de test aléatoire aveugle 68

4.1 Scénarios de fonctionnement du GMP 75
4.2 Preuve informelle complète du Théorème 3 82
4.3 Arbre de preuve du Théorème 3 - vue globale 85
4.4 Arbre de preuve du Théorème 3 - Racine 85
4.5 Arbre de preuve du Théorème 3 - Branche 1 87
4.6 Arbre de preuve du Théorème 3 - Branche 1.1 88
4.7 Arbre de preuve du Théorème 3 - Branche 1.1.1 88

4.8 Arbre de preuve du Théorème 3 - Branche 1.1.1.1 88
4.9 Arbre de preuve du Théorème 3 - Branche 1.1.1.2 89
4.10 Arbre de preuve du Théorème 3 - Branche 2 90
4.11 Arbre de preuve du Théorème 3 - Branche 2.1 90
4.12 Automate du GMP 94
4.13 Arbre de preuve du Théorème 3 - Branche 1.1.1.2 suite 97
4.14 Arbre de preuve du Théorème 3 - Branche 1.1.1.2.1 98
4.15 Arbre de preuve du Théorème 3 - Branche 1.1.1.2.2 98

5.1 Cycle d'injection d'une faute dans le couple spécification-preuve 107
5.2 PVS code : définition du prédicat `no_cliques` dans la théorie membership_verification 112
5.3 PVS code : axiomes définissant une faute d'émission dans la théorie membership_fault_model 113
5.4 PVS code : corollaire all_or_none dans la théorie membership_faultmodel 113
5.5 PVS code : axiome définissant une faute de réception dans la théorie membership_fault_model 114
5.6 PVS code : lemme validity dans la théorie membership_verification . . 116
5.7 PVS code : lemme agreement dans la théorie membership_verification 116
5.8 PVS code : lemme self_diagnosis dans la théorie membership_verification 116
5.9 PVS code : lemme diagnosis ajouté à la théorie membership_verification 117
5.10 Diagramme des Configurations du Group Membership Protocol 121
5.11 PVS code: definition de l'état stable dans la théorie membership_verification 121
5.12 PVS code : axiome shutdown dans la théorie membership_fautmodel . 130
5.13 Séquent en échec (*les hypothèses liées à la configuration ne sont pas retranscrites*) 131
5.14 Séquents en échec équivalents par la négation 134
5.15 Migration d'une règle `cut` dans un arbre de preuve 136

Introduction générale

“ Software is notorious for being late, expensive, and wrong. Exasperated technical managers often ask “what’s so different about software engineering – why can’t we (or, less generously, you*) do it right ?” The unstated implication is that the traditional engineering disciplines – in which technical managers usually received their training – do things better. ”*

“ Le logiciel est notoirement connu pour être en retard, cher et faux. Des directeurs techniques exaspérés demandent souvent “qu’y a-t-il de si différent avec l’ingénierie logicielle – pourquoi nous (ou, moins généreusement, vous*) ne pouvons pas faire les choses correctement ?” L’implication sous-entendue est que les disciplines d’ingénierie traditionnelles – dans lesquelles les directeurs techniques ont généralement été formés – font mieux. ”*

— John Rushby, Formal Methods and their Role in the Certification of Critical Systems [Rushby 1995]

La *sûreté de fonctionnement* d’un système informatique est la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu’il leur délivre [Laprie et al. 1996]. La sûreté de fonctionnement est notamment un impératif des systèmes dits *critiques* c’est-à-dire des systèmes dont les défaillances peuvent être catastrophiques, que ce soit en termes de vies humaines ou en termes de coût.
Un des moyens de la sûreté de fonctionnement est la *tolérance aux fautes* qui vise à permettre à un système de remplir sa fonction en dépit des fautes.

Les mécanismes de tolérance aux fautes sont le plus souvent implémentés complètement ou en partie à base de logiciel. Les algorithmes sous-jacents sont des briques de base pour construire des architectures sûres de fonctionnement, et ce dans différents domaines applicatifs. Il est donc vivement souhaitable que la correction de ces algorithmes soit établie de façon rigoureuse.

La grande majorité des preuves construites dans ce but sont des preuves informelles, c’est-à-dire des démonstrations données en langage naturel et basées sur le raisonnement humain. La question se pose alors de savoir quelle confiance accorder à de telles preuves. Malheureusement, on trouve dans la littérature plusieurs exemples d’algorithmes publiés avec leur «preuve» et qui se sont révélés être incorrects par la suite.

Les preuves formelles, c'est-à-dire dont le développement est assisté par des systèmes logiciels vérifiant ses étapes de déduction, offrent une confiance bien plus importante dans la vérification. En contrepartie, l'effort nécessaire pour spécifier formellement l'algorithme et conduire sa preuve est généralement très important. Or une preuve formelle n'est jamais assurée d'aboutir, et on peut se retrouver confronté à un échec de preuve qui rend complètement inutile l'effort investi.

Pour pallier ces problèmes, notre proposition est d'utiliser des méthodes de test en complément de preuves, que ces preuves soient informelles (et donc sujettes à caution), ou formelles mais inachevées. Nos travaux s'inscrivent dans une démarche qui consiste à remonter le test le plus tôt possible dans le processus de développement : l'artefact testé pourra être un prototype, ou une spécification exécutable. On cherchera ainsi à vérifier que l'algorithme de base est correct (sous certaines hypothèses, il assure certaines propriétés), en dissociant ce problème de celui de la correction d'une implémentation particulière au sein d'une architecture cible.

Pour que le test complète utilement une preuve existante, il devrait idéalement cibler les lacunes éventuelles de cette preuve. Nous allons donc étudier la possibilité de guider la conception du test par l'analyse de la preuve, et l'identification de ses faiblesses.
Si l'idée d'un couplage entre le test et la preuve n'est pas nouvelle (elle a notamment été exprimée dès les années 70 par [Goodenough & Gerhart 1975]), on trouve cependant peu de contributions dans ce sens. La question suivante est donc ouverte : comment peut-on extraire, de l'analyse d'une preuve jugée non concluante, des informations pertinentes pour le test ?

Pour aborder cette question, nous nous sommes orientés vers une démarche expérimentale. Nous partirons d'exemples concrets d'algorithmes de tolérance aux fautes dont la preuve s'est avérée problématique, et chercherons à mettre en œuvre une méthode de test guidé par la preuve. Nous évaluerons ainsi la faisabilité et l'efficacité d'une telle méthode.

Ce mémoire comporte cinq chapitres.

Le **premier chapitre** dresse un état de l'art des techniques de vérification. Nous présentons notamment les travaux qui, comme dans notre cas, ont étudié des couplages possibles entre techniques différentes. La tolérance aux fautes est introduite, et nous montrons qu'elle constitue un champ d'application intéressant pour nos travaux. Nous concluons en précisant notre problématique : valider l'hypothèse que la preuve d'un algorithme peut servir de base à la détermination de critères pertinents de sélection de test.

Le **deuxième chapitre** présente la méthode que nous proposons pour mettre en œuvre le test à partir de preuves informelles. Le cœur de la méthode consiste en une reformulation du discours informel sous forme d'un arbre de preuve. Cet arbre offre une représentation de l'articulation logique de la démonstration, ainsi qu'un support pour son analyse et son évaluation pas à pas. L'objectif de cette restructuration de la preuve informelle est de faciliter l'identification de ses faiblesses, sur lesquelles le test sera ensuite focalisé.

Dans les **troisième** et **quatrième chapitres**, la faisabilité et l'efficacité de la méthode sont évaluées expérimentalement sur deux exemples d'algorithmes incorrects : un algorithme d'ordonnancement de tâches (*Fault-Tolerant Rate Monotonic Scheduling*, ou FT-RMS), et un algorithme d'appartenance de groupe (*Group Membersiph Protocol*, ou GMP), tous deux ayant fait l'objet d'une preuve informelle, et s'étant révélés incorrects après leur publication. Nous détaillons dans ces deux chapitres l'application des différentes étapes de la méthode que nous avons définie. Les résultats montrent que l'identification des faiblesses de la preuve peut s'avérer efficace pour guider le test, sous réserve que l'analyse de l'arbre ne mette pas en évidence un manque de rigueur trop important affectant l'ensemble de la démonstration.

Dans le **cinquième chapitre** nous reprenons le principe d'un test basé sur l'arbre de preuve pour l'étendre aux preuves formelles. Nous poursuivons nos expériences en étudiant l'applicabilité du test guidé par la preuve à un troisième cas concret, qui est une variante plus complexe de l'algorithme d'appartenance de groupe précédent. Pour cet exemple, nous disposons d'une modélisation PVS qui a été développée et prouvée par des chercheurs de l'Université de Ulm. A partir de la preuve complète d'origine, nous créons une preuve partielle, par un artifice expérimental qui consiste à injecter une faute de conception dans la modélisation. Pour une preuve formelle, l'établissement d'un lien entre les lemmes non prouvés et des sous-espaces d'entrée de test peut être plus problématique que pour une preuve informelle. Nos résultats montrent néanmoins que, lorsqu'un lien est possible, cette information peut s'avérer très pertinente pour guider le test.

Chapitre 1

Cadre des travaux et état de l'art

“ [. . .] il vaut mieux former ces esprits par des opinions de cette sorte, si manifestement incertaines soient-elles, puisqu'elles divisent les savants, plutôt que de les laisser libres et abandonnés à eux-mêmes. Peut-être en effet courraient-ils à des précipices s'ils n'avaient point de guide ; tant qu'ils mettront leurs pas dans les traces de leurs précepteurs, ils pourront parfois s'éloigner de la vérité, du moins suivront-ils un chemin plus sûr, à ce titre au moins qu'il aura été exploré par de plus prudents qu'eux-mêmes. ”

— René Descartes, Règles pour la direction de l'esprit, Règle II dans [Alquié]

Ce chapitre présente dans un premier temps (paragraphe 1.1) les concepts de base de la sûreté de fonctionnement, qui constituent le cadre de nos travaux.

Les travaux présentés dans ce manuscrit s'intéressent à la complémentarité du test et de la preuve appliqués à la vérification d'algorithmes de tolérance aux fautes. Pour les situer par rapport à l'état de l'art (paragraphe 1.2), nous commencerons par dresser un panorama des techniques de vérification du logiciel, en mettant plus particulièrement l'accent sur le test et la preuve. Nous terminerons par la présentation de travaux qui, comme dans notre cas, ont étudié des couplages possibles entre plusieurs techniques de vérification différentes.
La tolérance aux fautes sera introduite dans le paragraphe 1.3. Nous nous intéresserons ensuite, dans un quatrième paragraphe, au problème de la validation de la tolérance aux fautes, pour faire le lien avec les techniques précédemment présentées.

Ceci nous permettra de conclure en précisant notre problématique.

1.1 Concepts de base de la sûreté de fonctionnement

La *sûreté de fonctionnement* d'un système informatique est la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre [Laprie et al. 1996].

Les notions attachées à la sûreté de fonctionnement sont groupées en trois grandes classes : attributs, entraves et moyens, comme cela est illustré par la Figure 1.1 ci-dessous.

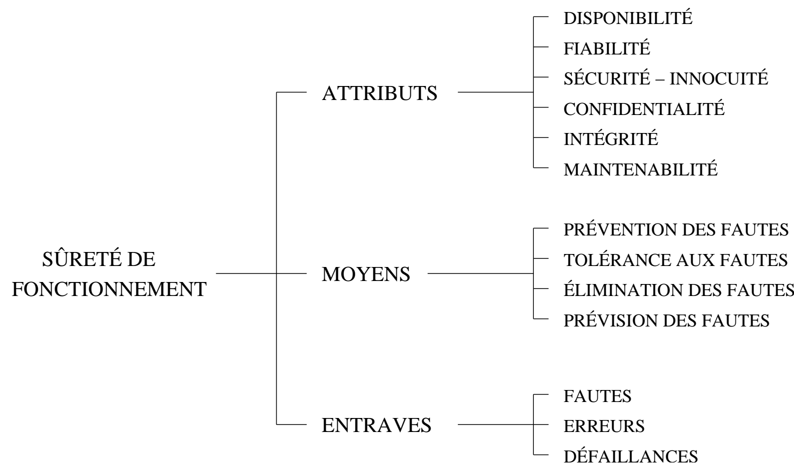


FIG. 1.1 – L'arbre de la sûreté de fonctionnement

Les *attributs* de la sûreté de fonctionnement permettent d'exprimer les propriétés qui sont attendues du système, et d'apprécier la qualité du service délivré. Par exemple, la disponibilité exprime le fait d'être prêt à l'utilisation : elle est requise pour tout système, bien qu'à des degrés variables. La sécurité-innocuité, qui exprime la non-occurrence de conséquences catastrophiques pour l'environnement, est requise pour des systèmes que l'on qualifie de *critiques*.

Les *entraves* à la sûreté de fonctionnement sont les circonstances indésirables, causes ou résultats de la non sûreté de fonctionnement. Trois concepts – faute, erreur, défaillance – sont nécessaires et suffisants pour les exprimer :

- une défaillance du système survient lorsque le service délivré par le système dévie de l'accomplissement de la fonction du système, c'est-à-dire dévie de ce à quoi le système est destiné ;
- une erreur est la partie de l'état du système qui est susceptible d'entraîner une défaillance ;
- une faute est la cause adjugée ou supposée d'une erreur.

Cette chaîne causale peut être résumée comme suit. Une faute est *active* lorsqu'elle produit une erreur : l'état interne d'un composant est corrompu. Une erreur peut disparaître ou, par propagation, créer de nouvelles erreurs. Une défaillance survient lorsque, par propagation, l'erreur affecte le service délivré par le système, et donc «passe à travers» l'interface système-utilisateur. Cette défaillance d'un système peut devenir une faute pour les systèmes qui interagissent avec lui, etc.

Les *moyens* de la sûreté de fonctionnement sont les méthodes et les techniques permettant de fournir au système l'aptitude à délivrer un service conforme à l'accomplissement de sa fonction, et de donner confiance dans cette aptitude. Le développement d'un système sûr de fonctionnement passe par l'utilisation *combinée* de l'ensemble de ces méthodes qui peuvent être classées en :

- prévention des fautes : comment empêcher l'occurrence ou l'introduction de fautes ;
- tolérance aux fautes : comment fournir un service à même de remplir la fonction du système en dépit de fautes ;
- élimination de fautes : comment réduire la présence (nombre, sévérité) des fautes ;
- prévision des fautes : comment estimer la présence, la création et les conséquences des fautes.

Dans le cadre de nos travaux, nous nous intéresserons plus particulièrement à l'élimination et à la tolérance aux fautes. Ces deux moyens sont complémentaires : l'élimination des fautes ne pouvant être parfaite, la mise en œuvre de mécanismes de tolérance aux fautes reste nécessaire. Ceci est d'autant plus crucial dans le cas de systèmes critiques, caractérisés par l'existence de modes de défaillances catastrophiques, que ce soit en termes de vies humaines ou en termes économiques.

1.2 Élimination des fautes

L'élimination des fautes est constituée de trois étapes : vérification, diagnostic et correction. La vérification consiste à déterminer si le système satisfait des propriétés, appelées conditions de vérification ; dans le cas contraire, les deux autres étapes sont entreprises : diagnostiquer la ou les fautes qui ont empêché les conditions de vérification d'être remplies, puis apporter les corrections nécessaires.
La vérification du logiciel a pour but de révéler les fautes de conception qui ont pu être introduites au cours de n'importe quelle phase du cycle de développement. Pour des raisons de coût et d'efficacité, il est important de les révéler au plus tôt, et des vérifications doivent être intégrées dès le début et tout au long du processus de développement.

On distingue quatre grandes classes de techniques de vérification : l'analyse statique, la vérification de modèle (ou *model-checking*), la preuve, et le test. Ces différentes techniques sont présentées dans les Paragraphes 1.2.1 à 1.2.4. Généralement, ces

techniques sont utilisées de manière indépendante : on choisit l'une ou l'autre technique selon l'étape de développement, et pour un problème de vérification donné. Certains travaux, présentés au paragraphe 1.2.5, ont cependant envisagé un couplage plus fort entre techniques, pour exploiter leur complémentarité au sein d'un même problème de vérification. Nous nous intéresserons en particulier aux travaux étudiant des collaborations entre le test et la preuve.

1.2.1 L'analyse statique

L'analyse statique vérifie des propriétés sur une description du logiciel, sans exécution de ce logiciel. Elle peut être manuelle, ou automatique.

L'*analyse statique manuelle* regroupe principalement les revues et les inspections, ces dernières suivant un processus plus formalisé [Strauss & Ebenau 1994]. Elle est applicable à toutes les étapes de développement : le document analysé peut être la spécification, le dossier de conception (générale ou détaillée), ou encore le code source du logiciel. L'analyse est faite lors de réunions impliquant un petit groupe de personnes, et donne lieu à des questions et remarques discutées avec l'auteur du document. Des listes de contrôle (*checklists*) sont utilisées pour guider l'analyse.

L'*analyse statique automatique* est généralement faite sur le code source. Les outils les plus utilisés sont sans doute les analyseurs syntaxiques, ou *parsers*, qui vérifient le code source par rapport à la grammaire du langage de programmation utilisé. Le résultat de cette analyse est généralement un arbre syntaxique, qui représente la décomposition de la structure du programme selon les règles de la grammaire. Cet arbre peut ensuite être utilisé pour d'autres vérifications, comme le contrôle de type. Des outils d'analyse syntaxique et de contrôle de type (plus ou moins puissants) sont notamment inclus dans tous les compilateurs.
L'analyse statique automatique a été appliquée à la vérification de différentes propriétés. Par exemple, l'outil PolySpace Verifier[1], commercialisé par société Polyspace technologies, analyse des programmes écrits en langage C, C++ ou ADA pour détecter des anomalies telles que le dépassement des bornes de tableaux, les débordements lors d'opérations arithmétiques, le code mort, les conflits d'accès aux variables partagées... L'outil FLUCTUAT, développé au CEA, étudie la stabilité numérique de programmes utilisant des flottants [Goubault 2001]. Des propriétés temps réels peuvent aussi être traitées, comme dans [Puschner & Koza 1989, Park & Shaw 1991, Theiling 2000, Colin & Puaut 2001, Ferdinand et al. 2001], où l'analyse du code source est utilisée pour la détermination de pires temps d'exécution (*Worst Case Execution Time, ou WCET*).
Pour ces différents types de propriétés, les techniques mises en œuvre utilisent souvent comme modèle le graphe de contrôle du programme[2]. Certaines se basent sur la théorie de l'*interprétation abstraite* [Cousot & Cousot 1977, Cousot 2001], qui formalise la notion d'approximation d'une sémantique concrète (qui décrit tous les comportements

[1] http ://www.polyspace.com/

[2] La notion de graphe de contrôle sera précisée dans notre présentation du test structurel, § 1.2.4.2.

possibles du programme) par une sémantique abstraite plus facile à vérifier. Cette théorie fournit un cadre pour concevoir des analyseurs corrects par construction. La technique de *slicing*, introduite par Weiser dans [Weiser 1984], permet d'extraire les parties d'un programme relatives à un certain critère de découpage. Le but de cette technique est d'obtenir un fragment de programme qui soit de taille minimale tout en conservant l'ensemble de son comportement vis à vis du critère donné. Les techniques de *slicing* ont largement évolué depuis [Weiser 1984]. Nous invitons les lecteurs à se référer à [IST 1998] pour une vue d'ensemble du domaine.

1.2.2 La vérification de modèle

Le principe du *model-checking*, ou vérification de modèle, est de vérifier de façon exhaustive des propriétés portant sur un modèle du système étudié. Cette vérification requiert d'une part une modélisation du système sous la forme d'un automate fini, et d'autre part l'expression de la propriété souhaitée par une formule dans une logique temporelle. L'outil de model-checking parcourt alors de façon exhaustive l'automate en vérifiant la formule pour chacun de ses états. On garantit ainsi la propriété pour tous les comportements du modèle. Si une violation de la propriété est trouvée, l'outil est généralement capable de fournir un contre-exemple illustratif.

Les deux principales logiques temporelles utilisées dans les outils sont PLTL (*Propositional Linear Temporal Logic*) [Pnueli 1981], et CTL (*Computation Tree Logic*) [Clarke & Emerson 1981, Emerson & Halpern 1982]. On pourra se reporter à [Bérard et al. 1999] pour une présentation des algorithmes de model-checking dans chaque cas.

La limitation principale du model-checking est le problème de l'explosion du nombre d'états. Lorsque la description du système est composée de n automates en parallèle, la taille de l'automate résultant croit exponentiellement avec n. De même, lorsque la description comporte des variables, la croissance est exponentielle avec le nombre de variables. Pour pallier ce problème, différentes approches ont été proposées (voir [Merz 2001] pour une introduction à ces approches). Le *model-checking symbolique* [Burch et al. 1992] utilise des représentations compactes d'ensembles d'états. On évite ainsi d'avoir à les énumérer. En particulier, la représentation d'ensembles d'états par des BDD (*Binary Decision Diagrams*) [Bryant 1986, Bryant 1992] est très utilisée dans ce cadre. Les techniques d'*ordre partiel* évitent, dans un système composé d'automates en parallèle, d'avoir à parcourir tous les chemins d'exécutions possibles (voir par exemple [Holzmann & Peled 1994, Ribet et al. 2002]). Le principe sous-jacent est que, pour vérifier la propriété, certains entrelacements d'événements sont équivalents. Enfin, les techniques d'*abstraction* [Clarke et al. 1994] consistent à construire, à partir de la description du système, un automate de taille réduite qui représente une abstraction des comportements initiaux. L'abstraction doit être adaptée à la propriété à vérifier : il ne faut pas que la propriété puisse être satisfaite par l'automate abstrait sans l'être par l'automate concret. La théorie de l'interprétation abstraite, initialement définie dans le cadre de l'analyse statique de programmes (voir paragraphe précédent), trouve ici un nouveau domaine d'application, encore relativement exploratoire.

L'existence d'outils performants, disponibles gratuitement sur Internet, a largement contribué à la diffusion du model-checking. Parmi ces outils, nous pouvons par exemple mentionner les suivants :

- SMV[3] [McMillan 1993] est un outil développé à l'université de Carnegie-Mellon (Pittsburgh, USA), et permet le model-checking symbolique de formules CTL, en se basant sur les BDDs. De par ce fait, SMV a pu être utilisé sur de très larges systèmes : [Clarke et al. 1994] présente une étude de cas d'un circuit logique à 10^{1300} états.
- SPIN[4] [Holzmann 1997] a principalement été développé aux Bell Labs (Murray Hill, USA). Il se base sur un langage d'entrée spécifique, Promela, et sur la logique temporelle PLTL. SPIN est un outil très efficace de par notamment ses techniques de model-checking à la volée et l'exploitation de techniques d'ordre partiel. On pourra se référer à [Loeffler & Serhouchni 1997] pour plus de détails sur le langage Promela et pour une illustration sur un cas d'étude bien connu dans le domaine des spécifications formelles : le contrôle de la chaudière d'une centrale nucléaire.
- Uppaal[5] [Larsen et al. 1997] est développé à Uppsala (Suède) et Ålborg (Danemark) et permet la simulation et la vérification d'automates temporisés, c'est à dire des automates étendus avec des horloges pour pouvoir manipuler la notion de temps. Un exemple de succès de Uppaal a été de permettre le diagnostic, et la correction, d'une faute de conception dans un protocole audio/vidéo [Havelund et al. 1997]. Ce protocole, utilisé depuis plusieurs années dans l'électronique grand public, manifestait parfois un comportement erroné mais il n'avait pas été possible jusque-là d'en déterminer la cause.
- Mec[6] [Arnold 1990, Arnold & Brlek 1995] a été développé au LaBRI (Bordeaux, France) et est utilisé pour la découverte de points de blocage, ou d'autres propriétés d'un système de transitions donné. Le formalisme utilisé pour décrire les systèmes est le modèle de Arnold-Nivat [Arnold & Nivat 1982], qui est suffisamment général pour s'adapter à de nombreux formalismes. Mec a été utilisé dans le cadre d'applications industrielles, telles que le compteur électrique d'une habitation [Alabau et al. 1997].
- KRONOS[7] [Yovine 1997, Bogza et al. 1998] a été développé par Verimag (Grenoble, France). Il permet de déterminer si un système temporisé, décrit comme la composition parallèle de plusieurs automates temporisés, satisfait une propriété exprimée par une formule dans la logique temporelle TCTL (*Timed Computation Tree Logic*, une version temporisée de la logique CTL). Un outil, TAXYS [Closse et al. 2001], a été développé intégrant KRONOS avec le langage ESTEREL [Berry & Gonthier 1992].

[3] `http ://www.cs.cmu.edu/ modelcheck/smv.html`

[4] `http ://netlib.bell-labs.com/netlib/spin/whatispin.html`

[5] `http ://www.docs.uu.se/docs/rtmv/uppaal`

[6] `http ://altarica.labri.fr/Tools/Mec5/`

[7] `http ://www-verimag.imag.fr/TEMPORISE/kronos/`

- HyTech[8] [Henzinger et al. 1997] est un outil de l'université de Berkeley (CA, USA). Il est conçu pour la vérification de systèmes hybrides linéaires (systèmes dont le comportement peut subir des changements aussi bien discrets que continus), synchronisés sur certaines de leurs transitions. HyTech calcule ensuite certaines parties de l'espace d'états, décrites par des expressions contenant des contraintes propositionnelles et des modes d'atteignabilité. Hytech a été utilisé sur plusieurs cas d'étude, principalement des systèmes de contrôle tels que [Stauner et al. 1997].

1.2.3 La preuve

Une preuve consiste à établir, par une suite finie d'étapes de raisonnement appelées inférences, qu'une certaine propriété est la conséquence logique d'axiomes, ou d'hypothèses, dont la vérité est admise.

Des preuves peuvent être utilisées à toutes les phases de développement du logiciel. On pourra par exemple chercher à prouver :

- qu'une spécification satisfait certaines bonnes propriétés ;
- qu'une étape de conception est correcte : une spécification détaillée satisfait une spécification plus abstraite (preuve de raffinement) ;
- qu'un programme est correct par rapport à une spécification détaillée (preuve de programme) ;
- qu'un programme termine toujours sous certaines conditions (preuve de terminaison).

Une preuve peut être informelle (§ 1.2.3.1) ou formelle (§ 1.2.3.2).

1.2.3.1 Preuves informelles

Une preuve informelle est une démonstration donnée en langage naturel, en utilisant éventuellement quelques notations mathématiques pour faciliter le discours. De telles preuves sont utilisées depuis toujours pour démontrer la validité de propriétés en mathématique et dans tous les domaines scientifiques. Le domaine informatique ne fait pas exception : dans le cadre de nos travaux, il n'est pas inutile de mentionner que la plupart des algorithmes de tolérance aux fautes sont publiés avec une preuve informelle.

Une preuve informelle est validée par consensus : elle est lue et acceptée par les pairs. Ceci la distingue d'une preuve formelle, qui peut être vérifiée automatiquement par des outils.

---

[8] `http ://www-cad.eecs.berkeley.edu/~tah/HyTech/`

1.2.3.2 Preuves formelles

Le prix à payer pour bénéficier de l'assistance d'outils de preuve est la nécessité d'une formalisation complète du problème de vérification : la description de l'algorithme, les hypothèses sur son environnement, et les propriétés attendues doivent être exprimées dans un langage ayant une syntaxe et une sémantique formelles, s'appuyant sur un cadre logique bien défini. Dans ce cadre, une preuve se ramène alors à des manipulations syntaxiques de formules. Les règles d'inférence indiquent comment produire des formules à partir de formules déjà produites, et une preuve peut être vue comme la construction d'un arbre de formules, dont les feuilles sont des axiomes et la racine la formule à prouver.

Le cadre classique des preuves est le calcul des prédicats du premier ordre : il permet de raisonner sur des formules qui comportent des variables. Deux exemples de systèmes de déduction pour ce calcul sont la déduction naturelle et le calcul des séquents (voir les définitions en encadré, et se reporter par exemple à [Monin 2000] pour une présentation détaillée). Pour pouvoir raisonner sur des développements de logiciel, ce calcul a dû être enrichi de diverses manières : par l'introduction de structures de données (types abstraits algébriques, constructions ensemblistes, ...), par l'introduction de nouveaux opérateurs permettant d'exprimer des propriétés sur des successions d'états (logiques temporelles), par la possibilité d'associer des propriétés à des morceaux de programmes (assertions de Floyd, logique de Hoare, plus faible précondition de Dijkstra). Le passage à une logique d'ordre supérieur donne la possibilité d'avoir des variables qui désignent des fonctions ou des prédicats.

Définition 1.1 — déduction naturelle

La déduction naturelle a été créée par Gentzen dans le but de formaliser plus fidèlement le raisonnement usuel. En déduction naturelle, il n'y a pas d'axiome logique ; mais on manipule des déductions *sous hypothèses*. Pour lever ces hypothèses, on utilise typiquement une règle d'introduction de $\Rightarrow$. Une *démonstration* est un cas particulier de déduction dans laquelle il n'y a plus d'hypothèses ; et un théorème est une formule pour laquelle existe une démonstration.

Définition 1.2 — calcul des séquents

Un séquent classique $\Gamma \vdash \Delta$ apparie deux suites de formules Γ et Δ. Intuitivement le séquent $\Gamma \vdash \Delta$ peut se lire : "la conjonction des hypothèses contenues dans Γ a pour conséquence la disjonction des formules contenues dans Δ." Par exemple $A, B \vdash C, D$ est analogue à $A \wedge B \Rightarrow C \vee D$. Le calcul des séquents est caractérisé par sa symétrie gauche/droite. La clef de voûte de cette symétrie est l'opérateur de négation, qui permet de faire passer des formules d'un côté ou de l'autre de $\vdash$.

Sauf à se limiter à un cadre restreint (le calcul des propositions, ou un fragment décidable de la logique du premier ordre), il n'existe pas de procédure de décision qui, pour toute formule, déterminerait s'il existe une preuve ou non. Les démonstrateurs de théorèmes doivent donc mettre en œuvre des heuristiques, sans garantie d'aboutissement. En pratique, l'intervention d'un opérateur humain est souvent nécessaire pour guider la construction de la preuve. Malheureusement, il y a toujours le risque de se retrouver dans un cas d'*échec de preuve* : on ne peut arriver à déterminer si la formule à prouver est valide ou non. L'investissement qu'a représenté la preuve a alors été inutile, ce qui constitue un sérieux inconvénient à l'utilisation de cette technique de vérification.

Nous discutons ci-dessous de facilités qui contribuent à rendre les environnements de preuves plus efficaces, ou plus faciles à utiliser. Nous présenterons ensuite des exemples d'environnements existants.

Facilités offertes par les environnements de preuve

La *réécriture* [Gorn 1967, Dershowitz & Jouannaud 1990] consiste à remplacer, dans une formule, une sous-formule par une autre qui a été démontrée égale. Par exemple, étant donné l'axiome $x+0=x$, la formule $(a+0)*2=4$ peut être réécrite en $a*2=4$. Cette technique est fondamentale pour la simplification des formules à prouver. Son automatisation est maintenant maîtrisée : la plupart des systèmes de preuve récents effectuent automatiquement ce type de simplifications.

Les *preuves par induction* sont une généralisation de la récurrence classique sur les entiers naturels où, pour prouver que $\forall n, P(n)$, on prouve d'abord $P(0)$, puis que $P(n) \Rightarrow P(n+1)$: ce raisonnement repose sur le fait que tous les entiers naturels sont engendrés par 0 et _+ 1. Des schémas de preuve par induction peuvent être définis pour d'autres types de données, lorsque les opérations permettant d'engendrer ces types sont connues. Les preuves par induction sont très utilisées en pratique, et il est donc souhaitable que l'environnement de preuve offre des facilités pour les réaliser. Typiquement, l'utilisateur doit rentrer le schéma d'induction à mettre en œuvre : en particulier, dans le cas où une formule comporte plusieurs variables, le choix de la variable sur laquelle porte l'induction doit être fait manuellement (bien que ce choix puisse être assisté). Les obligations de preuves associées à ce schéma peuvent ensuite être générées automatiquement, et l'outil tente de réaliser ces preuves.

Lorsque la preuve d'une propriété P n'aboutit pas, l'utilisateur est amené à inventer des propriétés intermédiaires, ou lemmes, supposées plus faciles à prouver que P, et à partir desquelles une preuve de P pourra être construite. Certains environnements de preuve ne permettent pas d'utiliser un lemme avant qu'il soit prouvé. Or, l'inverse est plus commode : l'utilisateur veut d'abord savoir si les lemmes sont réellement utiles pour prouver P, et n'effectuer leur preuve que si cette utilité a été établie. L'environnement de preuve doit alors mémoriser les dépendances entre lemmes, et signaler à l'utilisateur le statut de chaque preuve (terminée, terminée sous l'hypothèse de lemmes non prouvés, ou non terminée).

La plupart des environnements de preuve offrent à l'utilisateur un langage pour écrire des *tactiques* de preuves. Ceci permet de factoriser des étapes d'inférence, et de les réutiliser dans des parties de preuve similaires. Les langages de tactiques diffèrent dans la sophistication de leur structure de contrôle (séquences, itérations, structures conditionnelles) et dans la granularité des étapes de preuve considérées (étape élémentaire d'inférence, ou utilisation d'une autre tactique comme étape de la tactique courante).

Enfin, pour permettre à l'utilisateur de comprendre la structure de la preuve, et faciliter son analyse des parties en échec, il est souhaitable que l'environnement offre des facilités pour visualiser l'arbre de preuve. Ceci suppose que l'outil ne soit pas complètement boîte noire : il doit offrir une certaine visibilité sur les heuristiques qu'il a essayé de mettre en œuvre. L'arbre peut être montré à différents niveaux d'abstraction, par exemple en détaillant ou non les tactiques utilisées, et en détaillant ou non les preuves de certains lemmes intermédiaires. Nous verrons ultérieurement que la visualisation de l'arbre de preuve s'avérera utile dans le cadre de nos travaux, pour guider la conception du test.

Une discussion plus complète des caractéristiques souhaitables des environnements de preuve pourra être trouvée dans [Rushby 1993].

Quelques outils et environnements de preuve

Il existe de très nombreux systèmes de preuve. Selon le degré d'automatisation de la preuve, on parlera de vérificateur de preuve (qui se contente de vérifier une preuve déjà construite), d'outil d'assistance à la preuve, ou de démonstrateur de théorèmes. Nous avons essayé de répertorier les outils les plus connus.

- Le système dit de *Boyer-Moore* est un des ancêtres des démonstrateurs de théorèmes. Il a été utilisé dans le projet SIFT [Wensley et al. 1978] dans les années 80. Il est basé sur un système formel particulier, la logique de Boyer-Moore [Boyer & Moore 1979] : il faut donc exprimer la propriété à démontrer dans cette logique. C'est un vrai démonstrateur de théorèmes, qui comporte plusieurs heuristiques puissantes lui permettant de trouver des preuves non triviales, par exemple des preuves par introduction de lemmes.
- LP, le *Larch Prover* [Garland & Guttag 1991], est un outil de vérification de preuve pour une logique des prédicats du premier ordre typée. La spécification se fait en langage Larch [Guttag et al. 1985]. Le système est fondé sur un ensemble de fonctions déclarées, des propriétés et des axiomes exprimés sous forme d'équations, des règles de réécriture, des règles de déduction et des règles d'induction. TLP (*Temporal Logic Prover*) [Engberg 1995] est un système qui assiste la construction des preuves et offre une extension du langage Larch pour la logique temporelle TLA (*Temporal Logic of Actions*).
- *B* [Abrial 1996] est une méthode pour laquelle deux outils commerciaux : *Atelier B* et *B Toolkit* on été développés. Ils proposent chacun leur propre environnement de spécification et de preuve, ainsi que certaines autres fonctions comme

l'animation de spécification. L'intérêt principal de la méthode B est dans la possibilité de raffiner une spécification de haut niveau en une spécification suffisamment détaillée pour pouvoir être automatiquement traduite dans un langage de programmation. Chaque étape de raffinement donne lieu à des obligations de preuve. Le cadre mathématique est le calcul des prédicats du premier ordre et la théorie des ensembles. La sémantique du langage de spécification est basée sur la notion de plus faible précondition de Dijkstra.

- AUTOMATH [de Bruijn 1968], NUPRL [Constable et al. 1986], LEGO [Luo & Pollack 1992] et COQ [Barras et al. 1999] sont tous des démonstrateurs de théorèmes basés sur la théorie des types. Ils utilisent l'isomorphisme de Curry-Howard qui démontre la correspondance des propositions comme types, et des preuves comme termes, ce qui veut dire que les théorèmes peuvent être vus comme des types qui sont vrais s'il existe un élément de ce type. Ainsi la construction de la preuve revient à construire un élément de ce type. AUTOMATH est l'un des premiers démonstrateurs de théorèmes selon ce principe, il a été utilisé pour prouver un grand nombre de théorèmes mathématiques. NUPRL a été utilisé pour la vérification de plusieurs systèmes logiciels, LEGO est essentiellement un système théorique qui ne fournit pas de tactiques très puissantes. Le système COQ fournit bien plus de support à l'utilisateur. Il est basé sur le calcul des constructions inductives et le principe de l'extraction de programmes [Paulin-Mohring 1989]. COQ a été appliqué à de nombreux exemples non triviaux, comme la vérification de programmes JAVACARD [Barthe et al. 2000], la vérification matérielle [Coupet-Grimal & Jakubiec 1999], ou la modélisation géométrique [Puitg & Dufourd 1998].
- LCF (*Logic of Computable Functions*) [Gordon et al. 1979] est un des premiers démonstrateurs de théorèmes. Il a été programmé en ML et son langage sert également à écrire les stratégies de preuve. LCF a aussi introduit la terme de *tactique* et l'idée de preuve à l'envers : un utilisateur commence avec le but désiré et divise ce but en sous-buts plus simples en appliquant des tactiques. C'est l'inverse de la façon dont les preuves mathématiques sont traditionnellement écrites.
- HOL (*Higher Order Logic*) [Gordon & Melham 1993] a été développé principalement à l'université de Cambridge et est largement diffusé depuis 1988. C'est un démonstrateur de théorèmes dans la tradition de LCF. HOL a été utilisé essentiellement pour des preuves de conception de matériel notamment celle du microprocesseur tolérant aux fautes Viper pour le RSRE (*Royal Signals and Radars Establishment* du ministère de la défense britannique) [Cohn 1989]. Un des aspects les plus attrayants de ce système est la possibilité de définir des tactiques de démonstration adaptées à certains problèmes et à certains langages de spécification, des schémas d'induction, et de bâtir des extensions de la logique de départ appelées des "theories". L'une des plus importantes contributions de HOL est la formalisation des nombres réels et à virgule flottante [Harrison 1998]. Le système HOL a aussi été utilisé dans les domaines de la vérification matérielle

[Kropf 1999], de la sémantique de programme [Norrish 1998], et de la vérification de systèmes distribués [Prasetya 1995].

- ISABELLE [Paulson 1994, Nipkow et al. 2002] est un autre outil de preuve dans la lignée de LCF. ISABELLE a été développé à Cambridge et à Munich. La première version du système a été proposée en 1986. L'un des objectifs des concepteurs de ISABELLE était de développer un démonstrateur de théorème générique qui puisse supporter une grande diversité de logiques, et ayant un haut niveau d'automatisation. Cet outil est utilisé dans un large spectre d'applications : la formalisation des mathématiques, le développement de programmes, les langages de spécification et la vérification de programmes et systèmes [Paulson 2004].
- PVS [Owre et al. 1995], est le successeur de EDHM [Rushby et al. 1991]. C'est un système de preuve pragmatique, en ce sens qu'il favorise l'efficacité par rapport à la validité interne de ses procédures. Le démonstrateur de théorèmes de PVS incorpore de nombreuses et puissantes primitives qui sont utilisées pour construire les preuves. Ces primitives incluent des règles sur la quantification et les propositions, l'induction, la réécriture, et des procédures de décision pour l'arithmétique linéaire. Leurs implémentations sont optimisées pour des preuves de grande taille : par exemple, la simplification de propositions utilise des BDDs (*Binary Decision Diagrams*), et les règles de réécriture automatique sont en mémoire cache pour augmenter l'efficacité. PVS est développé au SRI International Computer Science Laboratory à Palo Alto (USA). Il a commencé a être développé en 1990 et sa première version a vu le jour en 1993. Le langage de base de PVS est le LISP. Ce système a été appliqué à de nombreux problèmes réels dont l'un des plus connus est la spécification et la modélisation des systèmes de contrôle de vol tolérants aux fautes de la navette spatiale [Crow & Vito 1996]. PVS a également été utilisé pour représenter les machines B [Muñoz 1999] à l'aide des types dépendants.

Nous verrons un exemple d'utilisation de PVS dans le cadre de nos travaux, au Chapitre 5.

1.2.4 Le test

1.2.4.1 Notions générales

Le *test* consiste à exécuter un programme avec des entrées valuées et à vérifier la conformité des sorties par rapport au comportement attendu. Sa mise en œuvre nécessite de résoudre deux problèmes : 1) le problème de la *sélection* d'entrées de test, et 2) le problème de l'*oracle* [Weyuker 1982] ou comment décider de l'exactitude des résultats observés, fournis par le programme en réponse aux entrées de test.

Sauf cas trivial, le test exhaustif d'un programme sur toutes ses entrées possibles n'est pas praticable. On est donc amené à sélectionner de manière pertinente un (petit) sous-ensemble du domaine d'entrée. Cette sélection s'effectue à l'aide de critères de

test, qui peuvent être liés à un modèle de la structure du programme ou à un modèle des fonctions que doit réaliser le programmes. Ces deux cas définissent respectivement le test structurel (§ 1.2.4.2) et le test fonctionnel (§ 1.2.4.3)
Malheureusement, l'imperfection des critères de test pour révéler les fautes de conception logicielles constitue une limitation majeure de ces approches. Ce constat a motivé des travaux sur la génération *probabiliste* d'entrées de test (par opposition, les autres approches sont alors qualifiées de *déterministes*). Nous présenterons ces travaux au paragraphe 1.2.4.4.
Enfin, l'analyse de mutation (§ 1.2.4.5) constitue un moyen pragmatique d'évaluer expérimentalement l'efficacité d'un jeu de test pour un programme donné.

En ce qui concerne le problème de l'oracle, un dépouillement automatisé des résultats de test est toujours souhaitable, et devient indispensable lorsqu'un grand nombre d'entrées ont été sélectionnées.
Les solutions les plus satisfaisantes sont basées sur l'existence d'une spécification formelle du programme sous test. La spécification est alors utilisée pour déterminer les résultats attendus, soit lors de la sélection des entrées de test, soit *a posteriori*. Dans le paragraphe relatif au test fonctionnel (§ 1.2.4.3), nous verrons plusieurs exemples d'approches de test à partir de spécifications formelles.
Le test dos-à-dos peut constituer une autre solution au problème de l'oracle, pour des systèmes critiques comportant de la redondance logicielle. Cette solution consiste à comparer les résultats du programme sous test avec ceux donnés par un autre programme développé indépendamment à partir de la même spécification (diversification fonctionnelle). En cas de discordance, la spécification joue le rôle de référence pour déterminer la (ou les) version(s) incorrecte(s). Le test dos-à-dos reste cependant une solution imparfaite, du fait de la présence éventuelle de fautes corrélées, conduisant les deux versions à produire des résultats identiques mais incorrects.
D'autres solutions d'oracle partiel peuvent être déterminées au cas par cas, en réalisant des contrôles de vraisemblance sur les résultats de test (contrôle de cohérence entre différentes données, contrôle d'appartenance à une plage de valeurs, ...). Ce principe de vérification partielle est par ailleurs couramment employé dans les mécanismes de détection d'erreur pour les systèmes tolérants aux fautes.

1.2.4.2 Test structurel

Les critères définis à partir de l'analyse structurelle de programmes sont nombreux [Beizer 1990]. Ils ont une vision *boîte de verre* du programme, et le modèle qu'ils utilisent est le graphe de contrôle, éventuellement enrichi par des informations sur le flot des données dans ce graphe.

Le *graphe de contrôle*, construit à partir du code source, fournit une vue compacte de la structure du programme. Il contient un nœud d'entrée et éventuellement plusieurs nœuds de sortie. Les nœuds de ce graphe sont des blocs d'instructions qui sont exécutées toujours dans le même ordre. Les arcs entre les nœuds correspondent aux branchements conditionnels ou inconditionnels dans le programme. Une exécution du

programme peut alors être vue comme un parcours complet entre le nœud d'entrée et un nœud de sortie, le chemin suivi étant déterminé par les valeurs des entrées de test.

Le critère de sélection le plus sévère demande d'activer au moins une fois chaque chemin exécutable entre le nœud d'entrée et un nœud de sortie. Un chemin est *exécutable* s'il existe des valeurs d'entrée qui l'activent.

En pratique, ce critère est rarement réalisable. D'une part, il n'y a pas d'algorithme général qui permette d'identifier automatiquement les chemins non-exécutables. D'autre part, dès lors qu'un programme contient une boucle, il peut comporter un nombre très grand, voire infini, de chemins. Dans ce cas, on est amené à se contenter de critères plus faibles, comme "toutes les instructions" ou "toutes les branches". Ces critères demandent respectivement que toutes les instructions (c'est-à-dire tous les nœuds du graphe), ou toutes les branches du graphe soient activées au moins une fois pendant le test.

Le graphe de contrôle peut être enrichi avec des informations relatives à la manipulations de variables. Les critères dits de *flots des données* [Rapps & Weyuker 1985] proposent alors de couvrir des sous-chemins entre les nœuds du graphe contenant des définitions des variables (elles reçoivent une valeur), et les nœuds (ou arcs) utilisant leurs valeurs dans des calculs (ou des prédicats). On peut citer, par exemple, le critère "toutes les utilisations" ou ses versions plus faibles "toutes les définitions", "toutes les C-utilisations" et "toutes les P-utilisations".

De façon générale, le test structurel est utilisé pour des composants logiciels de petite taille, car le graphe de contrôle devient très vite complexe au fur et à mesure que la taille du code source croît.

1.2.4.3 Test fonctionnel

Le test fonctionnel a une vision *boîte noire* du programme. Il repose sur un modèle décrivant le comportement attendu du programme. Contrairement à précédemment, il n'y a pas de modèle standard (comme pouvait l'être le graphe de contrôle). Le test fonctionnel recouvre donc un ensemble de méthodes diverses, selon le formalisme utilisé pour modéliser le comportement du programme. Nous présentons ici les méthodes les plus classiques.

Classes d'équivalence – Le domaine d'entrée du programme est partitionné en un nombre fini de sous-domaines, ou *classes d'équivalence*, en distinguant des plages de valeurs valides et invalides pour chacune des variables d'entrée [Meyers 1979]. Les classes d'équivalence se déduisent d'une analyse des spécifications informelles, chaque classe regroupant les valeurs d'entrée qui devraient activer de façon similaire les fonctionnalités du programme. Le critère retenu consiste alors à sélectionner un élément par classe. Ce test peut efficacement être complété par un test des valeurs aux limites, qui sélectionne des valeurs se trouvant aux frontières des différents sous-domaines.

Tables de décision – Une table de décision permet d'identifier les combinaisons d'entrées qui influent sur la logique du système, et peut être vue comme un langage de haut

niveau pour représenter des comportement combinatoires (sans mémoire). Elle comprend deux parties : une liste de conditions ou prédicats sur les entrées, et une liste d'actions à entreprendre (sorties). Chaque colonne de la table définit une règle, qui lie une combinaison de valeurs de vérité des conditions à une liste d'actions attendues. Le critère de test associé à la table consiste alors à activer au moins une fois chaque règle.

Machines à états finis – Les machines à états finis (MEFs) sont utilisées pour modéliser des comportements séquentiels. Une MEF est un graphe dont les nœuds représentent les états du système, et les arcs représentent les transitions entre états. Un alphabet fini d'événements d'entrée et de sortie permet d'étiqueter chaque transition avec l'entrée qui la provoque, et avec la sortie à produire lors du franchissement la transition. Les critères de sélection définis pour les MEFs sont des critères de couverture de la structure du graphe [Offutt et al. 2003] : passer par tous les états, par toutes les transitions [Huang 1975, Naito & Tsunoyama 1981], par toutes les paires de transitions [Pimont & Rault 1979].
Des versions plus sophistiquées du test de toutes les transitions ont été définies. Elles visent à l'exhaustivité vis-à-vis d'un modèle de faute simple, induisant des erreurs de transfert (la transition amène à un autre état que celui spécifié) ou de sortie (la transition ne génère pas l'événement attendu). La détection d'erreurs de transfert amène la nécessité de déterminer l'état d'arrivée. Cet état n'étant pas directement observable, on doit le caractériser par une ou plusieurs séquences d'événements en réponse auxquelles il fournit des sorties qui le distinguent des autres états. Les méthodes diffèrent selon la façon dont on construit ces séquences de caractérisation. La plus connue est la méthode W [Chow 1978], d'autres exemples sont la séquence de distinction et les séquences UIO (*Unique Input Output*) [Sabnani & Dahbura 1985, Ural 1992]. Ces méthodes ont également été étudiées pour des machines indéterministes, et des machines étendues avec des données [Phalippou & Groz 1990, Cavalli et al. 1996].

Systèmes de transitions étiquetés – Les systèmes de transitions étiquetés (*Labelled Transitions Systems*, ou LTS) sont un formalisme pour modéliser les systèmes communicants. En pratique, on ne spécifie pas directement selon ce formalisme : la spécification est donnée dans un langage de plus haut niveau (par exemple, SDL) dont la sémantique est définie par un LTS. La génération de tests (on parle dans ce cadre de *synthèse* de tests) s'effectue par composition du LTS avec des objectifs de test, qui spécifient des séquences d'événements à tester [Fernandez et al. 1996, Castanet et al. 1998]. Ces objectifs peuvent être déterminés de manière *ad hoc* ou générés automatiquement à partir de la spécification (par exemple, des objectifs relatifs à la couverture des transitions du modèle SDL d'origine).

Spécifications algébriques – Une spécification algébrique fournit une description axiomatique des fonctionnalités du programme. Dans ce cadre, tester un axiome consiste à instancier ses variables, et à vérifier que le programme satisfait la formule ainsi obtenue. Le choix des instances d'axiomes s'effectue à l'aide d'hypothèses de sélection [Gaudel 1995, Marre 1995] : les hypothèses d'uniformité définissent des sous-domaines d'entrée homogènes dans leur capacité à révéler une faute (c'est-à-dire des classes d'équivalence) ; les hypothèses de régularité limitent la taille des formules qui

peuvent être construites par "dépliage" des axiomes. Cette approche a notamment été étendue au traitement de descriptions LUSTRE [Marre & Arnould 2000] et aux algèbres de processus [Lestiennes & Gaudel 2002].

Langages ensemblistes – Les langages ensemblistes tels que VDM, Z et B permettent de modéliser l'espace d'état d'un logiciel, et les opérations faisant évoluer cet état. Les méthodes de test associées [Dick & Faivre 1993, Hierons 1997, Van Aertryck et al. 1997, Behnia & Waeselynck 1999] comportent deux aspects. D'abord, on effectue la sélection de cas de test pour chaque opération analysée individuellement (par exemple, en se basant sur des critères de couverture du prédicat avant/après de l'opération). Puis on cherche à construire des séquences d'appels aux opérations qui instancient ces cas (chaque opération dans la séquence amène le logiciel dans un état où l'opération suivante peut être activée).

1.2.4.4 Génération probabiliste d'entrées de test

Les critères structurels et fonctionnels définissent un ensemble d'éléments à activer au cours du test. Classiquement, la détermination de valeurs d'entrée selon un critère, qu'elle soit manuelle ou automatisée, cherche à minimiser la taille du test : les entrées de test sont alors sélectionnées de manière à ce que chaque élément ne soit activé qu'une seule fois. Or, les critères de test ne sont qu'imparfaitement liés aux fautes de conception réelles, ce qui constitue une limitation sérieuse pour l'efficacité de cette approche. Ceci a motivé des travaux sur la génération probabiliste d'échantillon éventuellement plus larges d'entrées de test. Dans ces travaux, les entrées sont générées aléatoirement selon une distribution des probabilités sur le domaine d'entrée, aussi appelée profil de test. Nous présentons ci-dessous deux approches différentes pour déterminer le profil de test.

Le *test opérationnel* met en œuvre une distribution des probabilités représentative du profil d'utilisation en vie opérationnelle [Musa et al. 1996, Whittaker 1997]. L'avantage est de concentrer l'effort de test sur l'élimination des fautes qui entravent le plus la fiabilité du logiciel : les fautes susceptibles d'induire un taux de défaillance élevé en opération sont celles qui ont la plus forte probabilité d'être révélées au cours du test. Cette approche pragmatique s'applique bien à des logiciels de criticité modérée : par exemple, [Musa et al. 1996] rapporte l'exemple d'un projet AT&T où elle a permis de diviser par deux l'effort de test système, et par dix les coûts de maintenance. Néanmoins, elle est inadéquate dans le cas de systèmes critiques, car ne permettant pas de révéler des fautes induisant de faibles taux de défaillance ($< 10^{-4}$ défaillances/heures).

L'approche du *test statistique* [Thévenod-Fosse et al. 1995] développée au LAAS peut, elle, s'appliquer aux logiciels critiques. Elle combine l'utilisation de critères de test (comme dans les approches déterministes) avec un procédé de génération aléatoire. Intuitivement, l'idée est que les critères de test fournissent une information certes imparfaite, mais néanmoins pertinente pour aider à révéler les fautes : activer plusieurs fois chaque élément du critère, avec des entrées aléatoires différentes, devrait permettre de compenser l'imperfection de ce critère.

L'approche repose sur la notion de *qualité de test* vis-à-vis d'un critère (voir l'encadré ci-après).

Définition 1.3 — Qualité du test statistique vis-à-vis d'un critère

Soient C un critère de test et E_C l'ensemble des éléments à activer selon ce critère. Le critère C est couvert avec une probabilité q_N si chaque élément de E_C a une probabilité au moins égale à q_N d'être activé pendant une séquence de test de longueur N. La valeur q_N est la qualité du test vis-à-vis du critère C. Elle est liée au nombre N d'entrées de test par la relation :

$$(1-\rho_{min})^N = 1 - q_N \qquad \text{avec} \quad \rho_{min} = min\{\rho_k, k \in E_C\} \tag{1.1}$$

ρ_k étant la probabilité qu'une entrée de test active l'élément k de E_C.
En pratique, un q_N voisin de 1.0 implique que chaque élément sera en moyenne activé plusieurs fois. Par exemple, pour l'élément le moins probable, le nombre moyen d'activations $n = N\rho_k$ peut être approximé par $-ln(1-q_N)$ lorsque ρ_k est petit, ce qui donne $n \simeq 3$ pour $q_N = 0.95$ et $n \simeq 9$ pour $q_N = 0.9999$.

La conception du test requiert alors deux étapes, la première étant la plus importante :

- la recherche d'un profil de test adéquat vis-à-vis du critère retenu, c'est-à-dire qui maximise la probabilité ρ_{min} de l'élément le moins probable ;
- l'évaluation de la longueur N du test selon ce profil, pour une qualité de test q_N désirée :

$$N \geq \frac{\ln(1-q_N)}{\ln(1-\rho_{min})} \tag{1.2}$$

La recherche du profil peut s'effectuer de façon analytique ou empirique, selon la complexité du problème d'optimisation sous-jacent.
L'approche analytique consiste à exprimer sous forme d'équations la probabilité d'activer chacun des éléments du critère en fonction de probabilités des valeurs d'entrée, et à résoudre ce système d'équations pour maximiser ρ_{min}.
L'approche empirique se base sur des mesures expérimentales de la couverture des éléments du critère, soit par instrumentation du programme (critère structurel), soit par instrumentation d'un modèle comportemental (critère fonctionnel). La recherche de la distribution de probabilités procède alors par essais successifs : on part d'une distribution simple (par exemple une distribution aléatoire uniforme), que l'on raffine jusqu'à obtenir une fréquence d'activation suffisamment importante pour chaque élément.

Des travaux récents, conduits au LRI [Gouraud et al. 2001, Gouraud 2004], ont proposé une mise en œuvre différente du test statistique. Ces travaux ont montré que certains problèmes de test, comme la sélection équiprobable de chemins dans un graphe, peuvent être exprimés en termes de génération de structures combinatoires. La mise en œuvre se décompose alors en deux étapes :

- une étape de tirage de chemins, qui utilise des outils combinatoires, et éventuellement des techniques de programmation linéaire pour optimiser la qualité de test vis-à-vis du critère retenu ;
- une étape de résolution de contraintes, avec des stratégies de résolution incluant un aspect aléatoire, pour construire des entrées de test qui activent les chemins tirés.

Ces travaux ont été appliqués au test statistique structurel de petites fonctions C, qui avaient été traitées dans [Waeselynck 1993] selon l'approche analytique. Les résultats expérimentaux par analyse de mutation montrent une efficacité de test comparable à celle qui avait été obtenue par l'approche analytique.

De façon générale, l'efficacité du test statistique, a été confirmée sur de nombreux exemples de logiciels [Thévenod-Fosse & Waeselynck 1998, Waeselynck & Thévenod-Fosse 1999, Chevalley & Thévenod-Fosse 2001], pour certains issus de domaines critiques tels que le nucléaire, l'avionique ou le spatial. Nous retiendrons donc le test statistique dans le cadre de nos travaux.

1.2.4.5 Analyse de mutation

Du fait de l'imperfection des critères de sélection, la validation des jeux de test – pour estimer leur aptitude à révéler les fautes – est un problème important. L'analyse de mutation [DeMillo et al. 1978] est une méthode expérimentale pour apporter des éléments de réponse à ce problème.

Une *mutation* est une modification syntaxique élémentaire introduite dans le code source d'un programme (par exemple, un signe '+' est remplacé par un signe '−'). On crée ainsi un grand nombre de mutants à partir du programme d'origine, et l'efficacité d'un jeu de test est mesurée par la proportion de mutants que ce jeu peut «tuer» (la faute injectée est révélée).

L'analyse de mutation soulève la question de la représentativité des mutations en tant que modèle de fautes. Bien que cette question reste posée, certains travaux [Daran & Thévenod-Fosse 1996] ont fourni des résultats encourageants : les mutations auraient la capacité de produire des erreurs similaires à celles produites par des fautes réelles (elles seraient donc aussi difficiles à révéler). Deux limitations pratiques concernent cependant le nombre de mutants à générer et l'identification des mutants sémantiquement équivalents au programme d'origine. On pourra se référer à [Offutt & Untch 2000] pour un état de l'art du domaine.

Le premier outil en date, Mothra [DeMillo et al. 1988], a été développé au *Georgia Institute of Technology*. Il est dédié à l'analyse de mutation de programmes écrits en Fortran. Parmi les outils plus récents, on peut notamment mentionner deux outils développés au LAAS : SESAME [Crouzet et al. 1998] pour des programmes en C, Pascal, Assembleur, ou Lustre ; et JavaMut [Chevalley & Thévenod-Fosse 2003] pour des programmes Java.

1.2.5 Couplage de techniques de vérification

Après cette présentation des quatre grandes classes de techniques de vérification, analyse statique, model-checking, preuve, et test, nous allons maintenant nous intéresser à des couplages possibles entre ces techniques. Plusieurs travaux ont cherché à exploiter leur complémentarité pour résoudre un problème de vérification.

Les couplages les plus étudiés se situent autour du model-checking (§ 1.2.5.1). En effet, l'émergence de model-checkers performants a conduit de nombreux chercheurs à s'intéresser à cette technique, et à envisager des applications possibles dans le cadre de leurs travaux. La combinaison du test et de la preuve a été moins étudiée (§ 1.2.5.2).

1.2.5.1 Couplages autour du model-checking

Un exemple réussi de couplage est l'intégration de techniques de model-checking dans l'environnement de preuve PVS. Le principe de l'approche décrite dans [Rushby 1999] est d'utiliser une interaction étroite entre : 1) la preuve de théorèmes, et 2) la génération d'abstractions pour le model-checking. Chaque étape consiste à appliquer alternativement l'une des deux techniques (preuve ou model-checking), en exploitant les informations obtenues lors des étapes précédentes. La preuve est utilisée pour justifier les abstractions réalisées pour le model-checking ; le model-checking aide à trouver des invariants à intégrer dans la preuve. Ce processus itératif, lorsqu'il termine, donne un contre-exemple indiquant comment la propriété a été violée, ou une preuve que cette propriété est satisfaite.
L'évolution récente des travaux autour de PVS porte maintenant sur le développement de SAL[9] (*Symbolic Analysis Laboratory*) [Bensalem et al. 2000, de Moura et al. 2004]. Il s'agit d'un cadre pour combiner différents outils de vérification pour les systèmes communicants. SAL intègre notamment le démonstrateur de théorèmes PVS, le model-checker SMV, un animateur de spécification, un outil de *slicing*, un vérificateur d'invariants utilisé pour calculer des abstractions et générer des contre-exemples (InVeSt [Bensalem et al. 1998]). Cet environnement a été utilisé pour prouver un cas d'étude comprenant plusieurs centaines de milliards d'états [Steiner et al. 2004].

En ce qui concerne le couplage test/model-checking, plusieurs méthodes et outils ont été développés pour automatiser la génération de jeux de test. Certains de ces travaux exploitent la capacité des model-checkers à générer des contre-exemples. Dans [Gargantini & Heitmeyer 1999] les auteurs forment la négation de propriétés de sûreté : les contre-exemples obtenus par model-checking leur donnent un jeu de test. Dans le même esprit, d'autres auteurs utilisent des modifications plus sophistiquées. Ils introduisent des mutations sur la spécification de propriétés et sur le modèle lui-même [Ammann et al. 1998, Ammann & Black 1999, Black et al. 2000]. Par rapport à l'analyse de mutation classique, l'avantage est ici que les modifications équivalentes sont automatiquement identifiées (la vérification montre que le modèle continue de satisfaire les propriétés). Pour les modifications non-équivalentes, les contre-exemples

[9] Site web de SAL : `http ://sal.csl.sri.com/`

trouvés correspondent à des traces de comportements différents entre le modèle originel et le modèle muté.
Une autre catégorie de travaux, menés dans le cadre du test à partir de LTS, a consisté à adapter les algorithmes de graphes utilisés par les model-checkers. Les algorithmes sont alors utilisés pour calculer efficacement le produit d'un LTS avec un objectif de test. Ces travaux ont amené la création, par exemple, de l'outil TGV [Fernandez et al. 1996], et ses évolutions plus récentes [Jéron & Morel 1999, Clarke et al. 2002].

1.2.5.2 Couplage entre le test et la preuve

L'analyse de la littérature montre qu'il existe peu de contributions proposant des solutions opératoires pour combiner la preuve et le test. L'idée a pourtant été exprimée dès les années 70 :

> *" A judicious combination of direct program proving and empirical judgment can reduce size of a complete test. "*
> *" Une combinaison judicieuse de preuve de programme et de jugement empirique peut réduire la taille d'un test complet. "*
>
> — J.B. Goodenough et S.L. Gerhart dans [Goodenough & Gerhart 1975]

Nous distinguons ci-dessous deux catégories de travaux, selon qu'ils souhaitent utiliser le test pour consolider la preuve, ou l'inverse.

Utilisation du test pour consolider la preuve

Des premiers travaux dans ce sens avaient été menés pour la vérification d'assertions de programmes [Geller 1978]. L'auteur proposait une approche inductive, qui consistait à tester des valeurs spécifiques du domaine d'entrée et à généraliser par la suite ces résultats à des classes d'équivalence, en utilisant des preuves mathématiques. Mais, dans toute approche inductive, le problème le plus délicat concerne la généralisation et non pas la vérification des cas de base : notre avis est que le test n'apporte donc pas d'aide significative à la preuve dans cette approche.
Une contribution originale est celle de Hayashi [Hayashi et al. 2002]. Il propose d'utiliser *l'isomorphisme de Curry-Howard* pour «tester» des preuves formelles, qu'elles soient partielles ou non. Cet isomorphisme établit une correspondance entre preuves et programmes, dans une logique constructive. Dans une telle logique, les preuves et les programmes sont équivalents selon les règles de cet isomorphisme : on peut donc extraire des programmes directement des preuves développées. La méthode proposée par Hayashi, appelée Animation de Preuve, revient à «exécuter» les preuves comme des programmes fonctionnels en utilisant des cas de test. Cette méthode a pour but de faciliter le développement de preuves formelles et un outil, ProofWorks, aide à trouver les «bogues» dans les preuves en cours de développement (preuves fausses) ou même dans les preuves terminées (erreurs de formalisation). Cependant l'utilisation de

l'isomorphisme de Curry-Howard restreint l'utilisation de cette méthode aux preuves en logiques constructives qui sont peu familières à la plupart des utilisateurs.
Dans [Rusu 2002], Rusu propose d'utiliser, pour faciliter la preuve, des techniques de génération de test basées sur des objectifs de test [Clarke et al. 2002]. Ces techniques de génération de test symbolique sont utilisées pour décomposer un système complexe en composants, les sous-graphes de composants obtenus sont ensuite spécifiés et prouvés à l'aide du système PVS. Sous l'hypothèse que le graphe d'un composant a un point d'entrée unique (toutes les transitions entrant dans le composant entrent par sa racine), ces vérifications propres à chaque composant sont ensuite étendues à l'ensemble du système.

Utilisation de la preuve pour consolider le test

Richardson et Clarke proposent dans [Richardson & Clarke 1985] une méthode combinant un partionnement fonctionnel et structurel en classes d'équivalence. Le but est à la fois de guider la sélection de données de test et de vérifier formellement la cohérence des deux partitions obtenues.
Dans [Cukic 1997], B. Cukic propose une approche basée sur la combinaison du test et de preuves partielles de programmes. Le principe est d'utiliser des techniques de *slicing* pour décomposer le programme étudié en parties indépendantes, qui regroupent les comportements liés à un sous-ensemble de ses variables d'entrée. Certaines de ces différentes parties du programme pourront alors être prouvées, réduisant donc l'espace des entrées devant être testées. Les parties ne pouvant être prouvées sont testées suivant un profil opérationnel associé aux entrées. Cette technique a été appliquée à plusieurs logiciels de contrôle/commande, notamment le programme de contrôle d'angle et de position d'un satellite.
Sinha et Suri [Sinha & Suri 1999a, Sinha & Suri 1999b] construisent un modèle d'un algorithme dans le langage de spécification de PVS, et interrogent ce modèle pour étudier la validité de la preuve informelle de l'algorithme. Les interrogations du modèle suivent un cycle d'itérations qui ciblent plus précisément au fur et à mesure les problèmes de la preuve. L'objectif est de pouvoir extraire des scénarios de test qui soient des contre-exemples de la preuve. Nous présenterons ces travaux plus en détail dans le Chapitre 3.
Plus récemment, les auteurs de [Castanet & Rouillard 2002] proposent une méthode de génération de test basée sur l'interfaçage entre un démonstrateur de théorèmes et un outil de model-checking. La méthode a été développée dans le cadre de l'environnement CClair [Castéran & Rouillard 1999], constitué d'un ensemble de théories Isabelle pour raisonner sur différents modèles d'automates. Dans ce cadre, la synthèse de cas de test peut être ramenée à la preuve constructive de l'existence d'une exécution du modèle qui satisfait l'objectif de test. L'aspect constructif de la preuve est résolu par l'utilisation de l'outil de model-checking Hytech (*c.f.* 1.2.2), qui produit une trace d'exécution candidate pour un objectif de test donné. Cette trace est ensuite prouvée à l'aide d'une tactique automatique de CClair, et retenue comme un cas de test valide.

Plus fondamentalement, H. Jouve envisage le lien entre stratégies de test et tactiques de preuve [Jouve 1999]. Dans le cadre de certaines spécifications (spécifications conditionnelles positives), elle montre qu'un critère de sélection des tests basé sur le dépliage des axiomes peut être comparé à la notion de tactique de preuve. L'application de ce critère amène à spécialiser le «but» de test de la même façon que l'application d'une tactique de preuve décompose le but de preuve en plusieurs sous-buts.
Cette idée nous semble très pertinente. Nous en retiendrons le principe général de l'utilisation d'un arbre de preuve en tant que guide pour le test.

1.3 Tolérance aux fautes

Malgré les efforts d'élimination des fautes, le concept des systèmes sans faute reste un idéal loin des réalités pratiques. Il reste donc nécessaire de mettre en œuvre de la tolérance aux fautes.
La présentation qui suit s'inspire largement de [Laprie et al. 1996].

1.3.1 Aspects généraux de la tolérance aux fautes

La tolérance aux fautes est mise en œuvre par le traitement d'erreur et le traitement de faute, comme présenté dans la Figure 1.2.

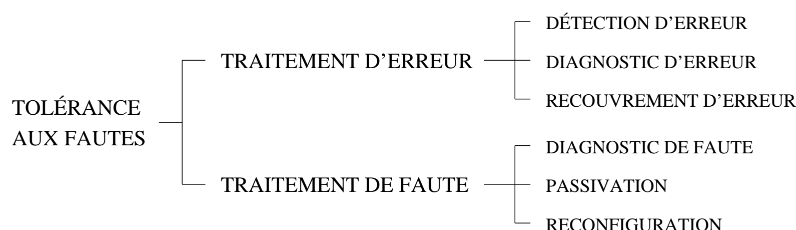


FIG. 1.2 – Moyens de mise en oeuvre de la tolérance aux fautes

Le *traitement d'erreur* est destiné à éliminer les erreurs, si possible avant qu'une défaillance ne survienne. Il fait appel à trois primitives :

- la *détection d'erreur*, qui permet d'identifier un état erroné ; les mécanismes correspondants vont de mécanismes génériques (par exemple des codes détecteurs d'erreur comme les CRC) à des mécanismes plus spécifiques basés sur des contrôles de vraisemblance ;
- le *diagnostic d'erreur*, qui permet d'estimer les dommages créés par l'erreur détectée, et par les erreurs éventuellement propagées avant la détection ;

- le *recouvrement d'erreur*, qui permet de substituer un état exempt d'erreur à l'état erroné.

Le *traitement de faute* est destiné à éviter qu'une, ou plusieurs fautes ne soient activées à nouveau. Il fait appel à trois primitives successives :

- le *diagnostic de faute*, qui consiste à déterminer les causes des erreurs, en termes de localisation et de nature ;
- la *passivation des fautes*, qui vise à empêcher une nouvelle activation des fautes, en retirant du processus d'exécution ultérieur les composants considérés comme fautifs ;
- la *reconfiguration*, qui consiste à modifier la structure du système, pour que les composants non-défaillants délivrent un service acceptable, bien qu'éventuellement dégradé.

Ces techniques de traitement d'erreur et de faute doivent être adaptées à la classe d'erreurs et de fautes visant à être tolérées. Ainsi, la tolérance aux fautes transitoires (par opposition à permanentes) ne nécessitera pas de traitement de faute, puisque le recouvrement d'erreur devrait suffire à éliminer les effets de la faute, qui a elle-même disparue. Une autre distinction importante concerne les fautes physiques et les fautes de conception, ces dernières étant beaucoup plus coûteuses à tolérer. Par exemple, la redondance à l'identique est envisageable dans le cas de fautes physiques, mais inadaptée dans le cas de fautes de conception : d'autres formes de redondance doivent être mises en œuvre, comme la diversification fonctionnelle où un même service est rendu via des conceptions et des réalisations séparées.

De façon générale, les redondances qui doivent être introduites pour tolérer les fautes conduisent à des architectures pouvant être vues, à un niveau ou à un autre, comme des architectures réparties. Nous présentons au paragraphe suivant le cadre conceptuel de la tolérance aux fautes dans les systèmes répartis.

1.3.2 La tolérance aux fautes dans les systèmes répartis

Un *système réparti*, ou *distribué*, est généralement défini comme un ensemble de *nœuds* ou *processeurs* (avec mémoire incorporée), possédant chacun sa propre horloge locale et reliés seulement par un réseau de communication par *messages*.

Le caractère réparti du système a les conséquences suivantes :

- le traitement d'erreur et de faute doit s'effectuer principalement par messages ;
- la granularité naturelle d'application des techniques de recouvrement, de masquage et de reconfiguration est celle des processus communicants ou bien celle des données réparties ;
- malgré les fautes et la concomitance de l'exécution, l'état global du système doit être maintenu cohérent, bien que non directement observable ni *a fortiori* manipulable.

Ce cadre conceptuel conduit à la définition des techniques de tolérance aux fautes sous la forme d'algorithmes répartis.

La maîtrise de la complexité de l'algorithmique répartie requiert une description claire des hypothèses de conception et la définition d'éventuelles abstractions simplificatrices.

Parmi ces différentes hypothèses, une catégorie importante porte sur l'aspect temporel et définit les approches dites *synchrone* et *asynchrone*. Ces deux approches se différencient selon la connaissance, lors de la conception de l'algorithme, de bornes minimales et maximales sur les délais de communication et de traitement. Lorsque ces délais ont des bornes connues l'approche est dite synchrone, et asynchrone lorsqu'elles ne le sont pas [Cristian 1991, Dilenno et al. 1991].
Une deuxième catégorie importante d'hypothèses concerne les modes de défaillance des processeurs et des moyens de communication. Ces hypothèses, qui ont une très grande influence sur les algorithmes que l'on peut mettre en œuvre, doivent être justifiées. Pour cela, elles doivent s'appuyer sur une mise en œuvre appropriée. Par exemple, une hypothèse de processeurs à *silence sur défaillance* (qui restent silencieux lorsqu'ils se sont détectés défaillants) peut se justifier si les processeurs possèdent des mécanismes internes performants pour détecter leurs propres erreurs.

Une classe d'abstractions concerne l'obtention d'un *temps global commun*. En effet, une des caractéristiques importantes de la répartition est l'absence d'une horloge physique globale accessible par tous les processeurs. Ce manque de référence temporelle unique a des conséquences négatives importantes. En particulier, il n'est pas facile de déterminer une relation d'ordre entre des événements ayant lieu sur des processeurs différents. Deux catégories de techniques permettent de reconstituer une référence de temps globale : les *horloges logiques* [Lamport 1978, Mattern 1988] et la *synchronisation d'horloges physiques* [Lamport 1978, Mishra & Schlichting 1992, Pfeifer et al. 1999].
Une autre classe d'abstractions traite de la *communication de groupe*, ou *communication multipoints*. Le problème de la communication de groupe en présence de fautes (ou *diffusion fiable*) et le problème sous-jacent de *consensus réparti* ont fait l'objet de très nombreux travaux, dont par exemple [Lamport et al. 1982, Fischer et al. 1985, Dolev et al. 1987, Hurfin et al. 2002]. Les protocoles diffèrent en termes d'hypothèses sous-jacentes et de buts à atteindre : selon les propriétés de consensus spécifiées, on peut distinguer les protocoles de diffusion générale ("*broadcast*"), les protocoles de diffusion sélective ("*multicast*"), et les protocoles d'appartenance ("*membership*").
Nous verrons notamment deux exemples de cette dernière catégorie de protocoles dans les Chapitres 4 et 5 de ce mémoire.

1.4 Validation de la tolérance aux fautes

La validation de la tolérance aux fautes et des mécanismes utilisés pour la mettre en œuvre implique une spécificité évidente : la nécessité de prendre en compte aussi bien l'activité normale du système due à sa fonction, que les fautes pouvant l'affecter.

Les objectifs de la validation peuvent être de deux types :

- le premier objectif, qui est lié à *la prévision des fautes*, est l'évaluation de la tolérance aux fautes, c'est-à-dire l'estimation de l'efficacité des mécanismes de tolérance aux fautes.
- le second objectif, qui est lié à *l'élimination des fautes*, est la vérification de la tolérance aux fautes, qui vise à révéler les fautes de conception présentes dans les mécanismes de tolérance aux fautes.

1.4.1 Évaluation de la tolérance aux fautes

Nous nous concentrons ici sur l'évaluation expérimentale par injection de fautes.

L'injection de fautes [Arlat et al. 1990] peut être vue comme une forme de test qui superpose deux types d'entrées : 1) les entrées liées à l'activité du système, et 2) les fautes. Les premiers travaux dans ce domaine ciblaient des fautes physiques, et utilisaient des techniques telles que l'injection de fautes sur les broches de circuits, ou le bombardement de circuits par des faisceaux d'ions lourds. Actuellement, l'injection de fautes tend à être de plus en plus implémentée par logiciel, y compris pour simuler les effets de fautes physiques.

Plusieurs mesures peuvent être utilisées pour évaluer la tolérance aux fautes. Les plus classiques sont la couverture et la latence.
La *couverture* [Bouricius et al. 1971, Arnold 1973] est définie comme la probabilité conditionnelle que, étant donnée une faute dans le système, le système parvient à la tolérer. On mesure expérimentalement cette couverture en classant l'ensemble des expériences d'une campagne d'évaluation par injection de fautes en deux catégories selon que le mécanisme étudié ait détecté une erreur ou non.
L'aspect dynamique du processus de traitement des fautes ou des erreurs ne peut pas non plus être ignoré et conduit à la latence. La *latence* est obtenue par la mesure de la durée entre un événement initial (instant d'injection ou d'activation de la faute) et un événement final (instant de détection d'une erreur).
Des exemples de travaux récents portent sur l'évaluation de mécanismes d'empaquetage[10] (*wrappers*) pour des logiciels exécutifs [Chevochot & Puaut 2001, Arlat et al. 2002].

Bien que l'injection de faute puisse être vue comme une forme de test, les travaux dans le cadre de l'évaluation expérimentale ne se réfèrent généralement pas aux méthodes de sélection de test présentées au paragraphe 1.2.4. La conception de campagnes d'injection de fautes soulève pourtant le problème de la combinaison judicieuse des deux types d'entrées considérées : fautes et entrées fonctionnelles du système. En effet, typiquement, on peut avoir jusqu'à 30% d'expériences non significatives (la faute injectée n'est jamais activée). Pour pallier ce problème, certains auteurs ont étudié la possibilité de mieux lier le choix des fautes à l'activité du système. Dans

[10] L'empaquetage consiste à rajouter une couche intermédiaire de tolérance aux fautes entre le support exécutif et la couche applicative

[Tsai et al. 1999] les auteurs proposent deux approches pour diminuer le nombre de fautes non activées. La première technique revient à *stresser* la charge du système sur lequel on injecte des fautes et d'injecter les fautes sur la partie du système la plus sollicitée à l'instant d'injection. La deuxième technique est basée sur une pré-analyse du système en termes de chemins d'exécution et de ressources utilisées sur ces chemins. L'objectif est de déterminer les entrées du système qui garantissent l'activation des fautes injectées. Dans [Fu et al. 2003], les auteurs se consacrent à l'évaluation des mécanismes de tolérance aux fautes par injection de fautes dans des services écrits en langage Java. L'injection est réalisée au niveau des blocs d'interceptions liés aux mécanismes d'exceptions. Ils mesurent ensuite la couverture structurelle des différents blocs lors des expériences d'injection de fautes. D'autres travaux, pouvant se rattacher à la problématique de la sélection d'entrées test, concernent l'identification de classes d'équivalence de fautes pour réduire la taille des expériences [Berrojo et al. 2002].

1.4.2 Vérification de la tolérance aux fautes

Le logiciel tient une place très importante dans les mécanismes de tolérance aux fautes, ce qui pose la question de la présence éventuelle de fautes de conception dans ces mécanismes.

Les différentes techniques de vérification que nous avons présentées peuvent alors s'appliquer dans ce cadre.

[Bernardeshi et al. 2001, Bernardeshi et al. 2002] ont utilisé le *model-checking* pour la vérification de plusieurs composants critiques de l'architecture GUARDS (*Generic Upgradable Architecture for Real-time Dependable Systems* [Powell et al. 1999]). Ces travaux se basent sur la notion d'*équivalence observationnelle* [Milner 1980] pour comparer le comportement normal du système en l'absence de fautes, avec celui du système incluant les mécanismes de tolérance aux fautes et en présence de fautes.
Un autre exemple récent [Steiner et al. 2004] est la vérification de l'algorithme de démarrage de l'architecture TTA (*Time Triggered Architecture* [Scheidler et al. 1997]). Cet exemple illustre bien la complexité induite par la considération de la tolérance aux fautes, qui augmente notablement la taille des modèles traités (de 10^9 à 10^{12} états atteignables). L'utilisation de techniques de model-checking symbolique a permis aux auteurs de conduire ce qu'ils appellent une «simulation exhaustive des fautes» et de vérifier entièrement l'algorithme.

Les *preuves* des algorithmes de tolérance aux fautes sont le plus souvent informelles, il y a cependant quelques exemples de mises en œuvre de preuves formelles.
Le système de preuve dit de *Boyer-Moore* a été utilisé dans le projet SIFT (*Software Implemented Fault Tolerant system*) [Wensley et al. 1978, Moser & Melliar-Smith 1990].
Plus récemment, plusieurs composants de l'architecture TTA ont été vérifiés à l'aide du système PVS : les protocoles de synchronisation d'horloge [Pfeifer et al. 1999] et d'appartenance de groupe (*group membership*) [Pfeifer 2000].

En ce qui concerne le *test*, on trouve des méthodes basées sur un modèle comportemental du mécanisme de tolérance aux fautes. L'objectif est de vérifier la conformité

par rapport à ce modèle. Le domaine d'entrée de test inclut les fautes que le mécanisme doit traiter.
Dans [Echtle & Leu 1995], les auteurs construisent un graphe d'accessibilité à partir d'une représentation en réseau de Petri, et génèrent ensuite des jeux de test pour couvrir les chemins de ce graphe.
[Avresky et al. 1996] construisent un arbre d'exécution à partir d'une spécification donnée sous formes d'assertions. Des séquences de test sont alors choisies selon un critère de couverture associé à cet arbre (feuilles, chemins, ...).
Les auteurs de [Arlat et al. 1999] utilisent une modélisation sous forme de chaînes de Markov continues. Les jeux de test sont générés selon l'approche test statistique, en suivant un critère de couverture des transitions du modèle.
[Arazo & Crouzet 2001] exploitent la capacité des model-checkers à générer des contre-exemples, pour tester les mécanismes de tolérance aux fautes à base de COTS.

Comme on peut le voir, la vérification des mécanismes et algorithmes de tolérance aux fautes se rattache à la problématique classique de la vérification du logiciel. La complexité de ces algorithmes, et la considération des fautes comme entrées additionnelles, rendent cependant cette vérification spécifiquement difficile.

1.5 Conclusion et approche proposée

La vérification de la tolérance aux fautes fournit un cadre intéressant pour étudier des couplages entre techniques complémentaires. Nos travaux traiteront plus particulièrement de l'utilisation de la preuve pour guider la conception du test, dans le cas d'algorithmes de tolérance aux fautes non triviaux. Ils viseront à compléter des preuves informelles, ou des preuves formelles inachevées.

Le test est une technique de vérification partielle. La sélection d'un ensemble fini d'entrées s'effectue à l'aide de critères de test, liés à des modèles structurels ou fonctionnels. Tous ces critères reposent sur des hypothèses. Par exemple, le test des transitions d'une machines à états suppose que les fautes de conception se manifestent comme des erreurs simples de transfert ou de sortie. Les objectifs de test utilisés pour les systèmes de transition représentent des comportements qu'il est jugé important de vérifier. Le test de classes d'équivalence fait des hypothèses d'uniformité au sein de chaque classe. Nous proposons d'étudier si une preuve, et plus particulièrement sa structure (c'est-à-dire l'arbre de preuve) est un vecteur d'information adéquat pour fonder de telles hypothèses. Dans la lignée des travaux sur le test statistique, nous considérerons ces hypothèses comme pertinentes (bien qu'éventuellement imparfaites) si elles permettent de concevoir des profils de test efficaces dans leur capacité à révéler des fautes.

Notre approche sera expérimentale, et s'appuiera sur des cas concrets d'algorithmes de tolérance aux fautes pour des systèmes critiques.
Le test à partir de preuves informelles sera étudié dans les Chapitres 2 à 4, en présentant d'abord la méthode que nous avons définie, puis en l'évaluant sur deux

exemples d'algorithmes. Le Chapitre 5 présentera l'extension de la méthode pour des preuves formelles et son application à un dernier cas d'étude.

Chapitre 2

Test guidé par une preuve informelle

> *“ La logique est donc la science des opérations intellectuelles, qui servent à* l’estimation *de la preuve, c’est à dire à la fois du procédé général consistant à aller du connu à l’inconnu, et des autres opérations en tant qu’auxiliaires de celui-ci. ”*
>
> *— Stuart Mill dans Système de logique, I. X.*

Pour introduire la motivation de nos travaux, plaçons nous dans la situation suivante : on vient de trouver dans la littérature un algorithme qui pourrait répondre aux besoins d’un système. Cet algorithme assure certaines propriétés de tolérance aux fautes, sous des hypothèses qu’il est possible de remplir dans le système. L’algorithme a été publié avec une preuve *informelle* dont la validité repose sur l’expertise de ses relecteurs. Or une telle preuve peut être jugée comme insuffisante. En effet, même dans le cas de revues ou colloques très sélectifs, il existe plusieurs exemples d’algorithmes qui se sont révélés incorrects après publication de leur «preuve». On souhaiterait donc avoir une plus grande confiance dans l’algorithme que celle apportée par sa seule preuve informelle.

Pour cela, il est possible d’envisager une réécriture formelle de l’algorithme et de ses spécifications, pour entreprendre une vérification à l’aide de méthodes formelles telles que celles décrites aux paragraphes 1.2.2 et 1.2.3. Cependant, cette solution représente un investissement important, certes envisageable dans le cas de systèmes critiques, mais peu rentable si l’algorithme s’avère incorrect et impropre à une utilisation dans le système cible. Il peut donc sembler pertinent de commencer par une méthode de vérification plus «légère», de façon à acquérir à moindre coût une certaine confiance dans la validité de l’algorithme publié.

Pour cela, une solution est la production d'un prototype de l'algorithme et sa soumission à une série de tests. Notons que, comme nous l'avons expliqué dans le paragraphe 1.2.4, le test exhaustif est impossible ; le test n'est donc qu'une méthode de vérification partielle. Le problème majeur sera alors de sélectionner des entrées de test pertinentes vis-à-vis d'éventuelles fautes de conception de l'algorithme.

Intuitivement, on peut penser que les fautes de conception résiduelles devraient être liées à des failles dans la preuve informelle. Il paraîtrait alors judicieux d'utiliser des informations extraites de l'analyse de cette preuve, pour guider la conception du test. Par exemple, une preuve par cas peut suggérer une décomposition du domaine d'entrée en classes d'équivalence. L'identification de parties complexes, ou obscures, de la preuve peut amener à tester certains sous-domaines d'entrée plus sévèrement que d'autres.

Bien qu'attrayante au premier abord, cette idée d'un *test guidé par la preuve* soulève un certain nombre de questions :

- comment extraire des informations d'une preuve informelle ?
- comment exploiter ces informations dans la conception du test ?
- Ces informations sont-elles réellement pertinentes pour guider le test vers des cas de défaillance de l'algorithme ?

Clairement, la faisabilité et l'efficacité du test guidé par la preuve devront être étudiées sur des exemples réalistes d'algorithmes publiés avec leur preuve informelle. En particulier, nous nous intéresserons à des algorithmes de tolérance aux fautes qui se sont révélés incorrects après publication.

Ce chapitre définit la méthode qui sera utilisée dans nos cas d'étude. Il précise comment nous allons mettre en pratique l'idée du test guidé par la preuve. Le premier paragraphe (§ 2.1) présente les grandes lignes de la méthode proposée. Les paragraphes suivants (§ 2.2 à 2.5) détaillent chacune des étapes de cette méthode, notamment la détermination du domaine d'entrée et de l'oracle de test, ainsi que la réécriture de la preuve informelle sous une forme facilitant son analyse en vue du test. La conclusion ouvre sur les deux cas d'étude traités dans les Chapitres 3 et 4.

2.1 Présentation générale de l'approche proposée

La méthode que nous proposons se décompose en quatre étapes successives (Figure 2.1), chaque étape définissant les bases et les informations nécessaires à la réalisation d'une ou plusieurs étapes suivantes.

Ces quatre étapes peuvent être décrites brièvement comme suit.

- L'*Analyse préliminaire* vise à obtenir une compréhension de l'algorithme et de ses spécifications : sous certaines hypothèses, certaines propriétés doivent être garanties. Dans le cas d'un algorithme de tolérance aux fautes, les hypothèses incluent le modèle des fautes qui doivent être tolérées et d'autres hypothèses

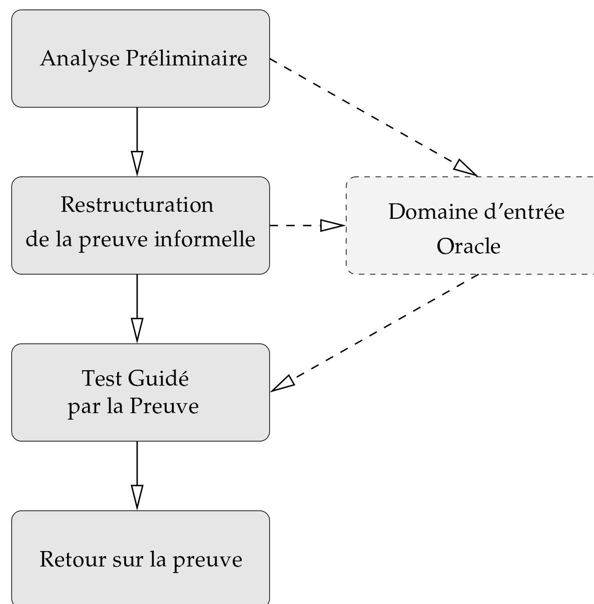


FIG. 2.1 – Vue générale du Test Guidé par la Preuve

liées à l'environnement. À partir de l'identification de ces hypothèses, on détermine le domaine d'entrée du test. Les propriétés qui doivent être garanties par l'algorithme fournissent la spécification d'un oracle de test, qui indiquera si le résultat d'un jeu de test correspond à un comportement correct ou à une défaillance. A ce stade, la compréhension acquise doit être suffisante pour permettre le développement d'un prototype de l'algorithme, et d'un environnement de test pour ce prototype. Ceci peut nécessiter une première lecture des arguments de la preuve, par exemple pour en clarifier les hypothèses.

- La *Restructuration de la preuve* forme le cœur de la méthode. Elle consiste à réécrire le discours informel sous la forme d'un arbre de preuve. Le but de cette réécriture est d'obtenir une vue claire et synthétique de la structure logique de la preuve. On interprète les étapes du raisonnement en termes d'étapes d'inférence d'un système de déduction tel que le calcul des séquents ou la déduction naturelle (§ 1.2.3.2). L'arbre de preuve ainsi obtenu est un support utile pour évaluer la validité de la preuve d'une manière systématique. Il est parcouru pas à pas, en examinant chaque application d'une règle d'inférence. Les différentes branches de preuve se voient alors attribuées une évaluation subjective de leur validité.
  La réécriture conserve un certain nombre de notations issues du discours informel, et reste ainsi beaucoup plus «légère» qu'une approche de formalisation complète. De ce fait, elle ne permet pas d'établir si l'algorithme est correct. Par contre, elle devrait être suffisante pour obtenir une évaluation rapide de la rigueur de la preuve, et pointer ses parties les moins convaincantes.

- Le *Test Guidé par la Preuve* exploite les résultats de cette analyse pour déterminer des critères de sélection de test, et éventuellement renforcer l'oracle de test pour observer la validité de lemmes intermédiaires. Les expériences de test sont ensuite conduites sur le prototype de l'algorithme. Nous utilisons une mise en œuvre par *test statistique* (§ 1.2.4.4). Selon cette approche, les cas identifiés par un critère de test doivent être activés plusieurs fois avec des entrées aléatoires différentes. L'objectif est de compenser l'imperfection du critère vis à vis des fautes de conception qu'il doit révéler. Si l'on considère le test guidé par la preuve, il semble raisonnable de ne pas espérer une parfaite adéquation entre les étapes de preuve douteuses et des entrées de test conduisant à défaillance. L'utilisation du test statistique nous permet de limiter ce problème, en faisant l'hypothèse que l'information extraite de la preuve, bien qu'imparfaite, permet de déterminer des sous-domaines ayant une probabilité significative de révéler une faute de conception résiduelle.
- Le *Retour sur la preuve* est la dernière étape de notre méthode. Cette étape se base sur l'idée que les informations obtenues par le test peuvent être utiles pour reprendre certaines étapes de raisonnement de la preuve informelle que nous avions jugées faibles lors de la restructuration de la preuve.

Après cette vue d'ensemble de la méthode, nous allons maintenant présenter plus en détails chacune de ces quatre étapes.

2.2 Étape 1 : Analyse préliminaire

L'analyse préliminaire vise à extraire les informations de haut niveau des présentations informelles de l'algorithme et de sa preuve.

Cet objectif peut être considéré comme atteint lorsque l'on est capable de réaliser l'environnement expérimental pour le test de l'algorithme :

- Le *prototype* est réalisé à partir de la description de l'algorithme, que les auteurs ont pu donner en langage naturel, ou sous une forme plus formalisée telle que du pseudo-code. Dans tous les cas, cette description doit être suffisamment claire et non ambiguë pour permettre une implémentation dans un langage de programmation.
- Le *domaine d'entrée du test* est déterminé par deux catégories d'hypothèses : (1) le modèle de fautes, qui regroupe toutes les hypothèses sur les fautes à tolérer, et (2) des hypothèses additionnelles sur l'environnement d'exécution de l'algorithme. A partir de l'identification de ces hypothèses, on doit être capable de donner une définition *opérationnelle* du domaine d'entrée, par exemple sous la forme d'une fonction de génération aléatoire qui engendre ce domaine. Notons que les informations extraites à ce stade ne permettent pas encore de définir un critère de sélection des entrées de test : la sélection aléatoire s'effectue selon un profil d'échantillonnage «en aveugle». Dans le cadre de nos travaux, un tel

profil aveugle sera systématiquement implémenté, à des fins de comparaison expérimentale avec des profils plus guidés.
Une fois le domaine d'entrée déterminé, il faut encore choisir un format pour encoder les données générées, et développer un moniteur de test qui lit un fichier d'entrée et pilote les expériences correspondantes sur le prototype.

- L'*oracle de test* est spécifié par les différentes propriétés $P_1, \ldots, P_n$ devant être garanties par l'algorithme. Les propriétés déterminent les sorties du prototype, et éventuellement les parties de son état interne qui devront être observées pour émettre un verdict. La violation d'au moins une de ces propriétés sera considérée par l'oracle comme une défaillance de l'algorithme. L'ensemble des propriétés identifiées définit ainsi l'ensemble des observateurs implémentés dans notre environnement de test.

A l'issue de l'étape d'analyse préliminaire, on peut déjà réaliser de premières expériences de test pour vérifier le comportement de l'algorithme sur des échantillons d'entrées aléatoires. Des comportements incorrects pourraient ainsi être mis en évidence à faible coût. Dans le cadre de nos travaux, ces premières expériences nous permettront d'évaluer la difficulté à révéler les fautes résiduelles des algorithmes étudiés.

2.3 Étape 2 : Restructuration de la preuve

La deuxième étape de notre méthode est la réécriture de la preuve informelle, ou plus précisément sa restructuration sous la forme d'un arbre de preuve. L'objectif de cette étape est :

1. d'obtenir une vue synthétique de la preuve informelle, qui offre une représentation explicite des étapes de raisonnement ;
2. de pointer des faiblesses potentielles dans les étapes de raisonnement, en se référant aux règles d'inférence du système de déduction logique considéré.

2.3.1 Représentation sous forme d'arbre

Les deux systèmes de déduction que nous avons considérés sont la déduction naturelle exprimée sous forme de séquents et le calcul des séquents (§ 1.2.3.2). Ces deux systèmes correspondent à des formulations différentes, mais équivalentes, des règles du raisonnement usuel.

Un *séquent* est une formule de la forme $\Gamma \vdash P$, où Γ est une conjonction d'hypothèses, et P une disjonction de buts à prouver sous ces hypothèses. Dans nos expressions de séquents, nous autorisons des notations synthétisant des éléments du discours informel. Par exemple, pour l'algorithme d'ordonnancement de tâches étudié au Chapitre 3,

le but P demandant qu'aucune tâche ne manque son échéance sera noté "ordonnancement valide", sans chercher à définir formellement cette notion d'ordonnancement valide. Notre but n'est pas de formaliser complètement le problème de vérification, mais de restructurer le discours sous une forme plus facile à analyser.

De façon générale, la restructuration de la preuve d'un algorithme démarre par l'expression d'un séquent :

$$\textit{algorithme, modèle de fautes, environnement} \vdash P_1 \land P_2 \land \ldots \land P_n$$

où *algorithme* renvoie à la description de l'algorithme, *modèle de fautes* et *environnement* aux hypothèses d'entrée, et $P_1, \ldots, P_n$ aux propriétés à prouver. Bien que ces différentes notions ne soient pas formellement définies, leur signification devrait être claire après le travail d'analyse préliminaire à l'étape précédente.

La preuve de l'algorithme peut alors être vue comme un arbre de séquents, le séquent initial à prouver correspondant à la racine de l'arbre. A partir de cette racine, on développe l'arbre vers le haut en appliquant des règles d'inférence de type :

$$\frac{\text{Séquent 2} \quad \text{Séquent 3} \quad \ldots \quad \text{Séquent n}}{\text{Séquent 1}} \text{ nom de la règle}$$

qui, à partir d'un séquent de forme *Séquent 1*, éclate la preuve en n branches.

Une partie des règles des deux systèmes logiques sont présentées Figure 2.2 (pour un inventaire complet, voir par exemple [Monin 1996]). Les règles choisies sont celles que le lecteur retrouvera dans les exemples d'arbres de preuve des Chapitres 3 et 4.

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \land B} \land_i \qquad \frac{\Gamma, B \vdash A \quad \Gamma, C \vdash A \quad \Gamma \vdash B \lor C}{\Gamma \vdash A} \lor_e$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \Rightarrow_i \qquad \frac{\Gamma \vdash B \quad \Gamma \vdash B \Rightarrow A}{\Gamma \vdash A} \Rightarrow_e$$

déduction naturelle

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \land B} \vdash\land \qquad \frac{\Gamma, B \vdash A \quad \Gamma, C \vdash A}{\Gamma, B \lor C \vdash A} \lor\vdash$$

$$\frac{\Gamma \vdash B \quad \Gamma, B \vdash A}{\Gamma \vdash A} \text{cut}$$

calcul des séquents

FIG. 2.2 – Principales règles d'inférence utilisées

Les règles $\wedge_i$ (pour la déduction naturelle) et $\vdash\ \wedge$ (pour le calcul des séquents) permettent de décomposer la preuve d'une propriété de la forme $A \wedge B$ en deux branches correspondant aux preuves de chacune des sous-propriétés A et B.
Les règles $\vee_e$ et $\vee\ \vdash$ correspondent à une décomposition des hypothèses. Le but A est prouvé sous l'hypothèse B et sous l'hypothèse C. Dans $\vee_e$, la complétude de la décomposition doit être prouvée ($\Gamma \vdash B \vee C$) ; alors que dans $\vee\ \vdash$, la règle ne peut s'appliquer que lorsque $B \vee C$ fait explicitement partie des hypothèses.
Les règles $\Rightarrow_e$ et cut servent à introduire un nouveau lemme : B dans les deux cas. C'est une pratique courante dans toute preuve. On essaie alors de prouver le but A avec une hypothèse additionnelle qui est B ; il est cependant nécessaire de prouver B sous les mêmes hypothèses Γ que le but originel A.

2.3.2 Exemple d'arbre de preuve

Pour illustrer la restructuration sous forme d'arbre, nous utilisons un fragment de preuve informelle tiré du Chapitre 4. Il porte sur une propriété notée *Théorème 3*, et s'effectue sous des hypothèses Γ que nous ne détaillons pas ici. Le discours informel est reproduit ci-dessous en anglais, tel qu'il s'est présenté à nous dans la publication d'origine [Katz et al. 1997]. Nous avons cependant tronqué les parties du texte qui n'étaient pas nécessaires pour illustrer notre propos.

> (*Théorème 3*)
> " If a processor *p* becomes send-faulty, ... Similarly, if *p* just became receive-faulty in the expected broadcast before its slot, ... [*preuve de ces deux cas*]
> If a processor *p* becomes receive-faulty in its transition to the next step, but *p* is not the next expected broadcaster, ... [*preuve de ce dernier cas*]."

On reconnaît là la forme d'une preuve par cas, ce qui s'exprime en déduction naturelle par l'application de la règle $\vee_e$ et en calcul des séquents par l'application successive de cut et de $\vee\ \vdash$.

Les deux possibilités sont montrées en Figure 2.3. Pour obtenir une représentation concise des séquents de ces arbres, nous avons choisi d'adopter les notations suivantes :

– Sfault pour " *a processor p becomes send-faulty* " ;
– Rfault_b pour " *p just became receive-faulty in the expected broadcast before its slot* " ;
– Rfault-not_b pour " *a processor p becomes receive-faulty in its transition to the next step, but p is not the next expected broadcaster* ".

Comme on peut le voir, nous avons décomposé la preuve en trois branches :

- preuve du Théorème 3 pour les cas (Sfault $\vee$ Rfault-b) (Branche 1) ;
- preuve du Théorème 3 pour le cas Rfault-not_b (Branche 2) ;
- preuve de la complétude de la décomposition en cas.

$$\dfrac{\dfrac{\text{(Branche 1)}}{\Gamma, \text{Sfault} \vee \text{Rfault-b} \vdash \text{Théorème 3}} \quad \dfrac{\text{(Branche 2)}}{\Gamma, \text{Rfault-not_b} \vdash \text{Théorème 3}} \quad \Gamma \vdash (\text{Sfault} \vee \text{Rfault-b}) \vee \text{Rfault-not_b}}{\Gamma \vdash \text{Théorème 3}} \vee_e$$

Arbre de preuve en déduction naturelle

$$\dfrac{\dfrac{\dfrac{\text{(Branche 1)}}{\Gamma, \text{Sfault} \vee \text{Rfault-b} \vdash \text{Théorème 3}} \quad \dfrac{\text{(Branche 2)}}{\Gamma, \text{Rfault-not_b} \vdash \text{Théorème 3}}}{\Gamma, (\text{Sfault} \vee \text{Rfault-b}) \vee \text{Rfault-not_b} \vdash \text{Théorème 3}} \vee\vdash \quad \Gamma \vdash (\text{Sfault} \vee \text{Rfault-b}) \vee \text{Rfault-not_b}}{\Gamma \vdash \text{Théorème 3}} \texttt{cut}$$

Arbre de preuve en calcul des séquents

FIG. 2.3 – Exemple de restructuration sous forme d'arbres de preuve

Cette dernière branche de preuve est implicite dans le discours informel : les auteurs l'ont sans doute considérée comme triviale et l'ont omise. En suivant les règles de la déduction naturelle ou du calcul des séquents, nous avons été obligés de l'expliciter. On voit ainsi l'intérêt de la restructuration pour guider l'analyse pas à pas de la preuve.

2.3.3 Identification des faiblesses de la preuve

L'analyse de la preuve consiste à donner une évaluation à chaque feuille de l'arbre, au fur et à mesure de sa construction. Dans l'exemple précédent, il est trop tôt pour juger de la validité des branches 1 et 2, qui n'ont pas encore été développées. Par contre, on a déjà un séquent terminal, exprimant l'exigence de complétude de la décomposition par cas : il nous faut porter un jugement sur ce séquent.

Nous utiliserons les notations suivantes pour exprimer notre jugement sur les feuilles d'un arbre de preuve :

- true dénote une branche de preuve terminée et que nous jugeons valide. Lorsque nous voulons expliciter le fait que notre jugement se base sur des hypothèses particulières dans la liste Γ, nous utiliserons une variante de cette notation, true : *hyp*, où *hyp* précise les hypothèses considérées. Dans l'exemple précédent, la décomposition en cas est jugée complète sous les hypothèses de fautes, ce que nous noterons dans le Chapitre 4 par true : *fault model*.
- false dénote une branche de preuve que nous jugeons fausse. Etant donné un séquent terminal $\Gamma \vdash P$, nous avons pu établir que $\Gamma \wedge \neg P$ est possible. Remarquons que le fait qu'un séquent terminal soit faux n'implique pas que le séquent racine soit faux. Par contre, il doit clairement être considéré comme non prouvé.
- $\perp$ dénote une branche que nous jugeons non-concluante, c'est-à-dire pour laquelle nous n'avons pu établir un jugement true ou false. Cette notation sera notamment utilisée pour des parties de raisonnement oubliées dans la preuve informelle, et que nous considérons comme non triviales. Elle sera aussi utilisée

lorsque l'expression d'un séquent comporte des notions dont la sémantique n'est pas assez clairement définie dans le discours informel.

- ? dénote une branche que nous ne parvenons pas à développer plus en détails. Le discours informel est trop confus pour nous permettre de l'interpréter en termes d'étapes d'inférence. On complétera éventuellement cette notation par un label (label ?) destiné à donner une idée intuitive de ce que les auteurs ont essayé de prouver.

A la fin de la restructuration, on dispose d'une vue synthétique de toutes les étapes du raisonnement, avec un jugement sur chaque branche. Il nous faut maintenant tirer partie de ces informations pour guider le test.

2.4 Étape 3 : Test Guidé par la Preuve

La difficulté va être d'exploiter les résultats de l'analyse de la preuve de façon constructive pour le test. Il s'agit de déterminer des critères de sélection de test, et éventuellement de renforcer l'oracle de test par des observateurs additionnels (§ 2.4.1). Les critères retenus seront utilisés pour la conception du test statistique (§ 2.4.2).

2.4.1 Des faiblesses de la preuve au test de l'algorithme

Considérons l'exemple d'un arbre d'une preuve non-concluante tel que celui schématisé Figure 2.4.

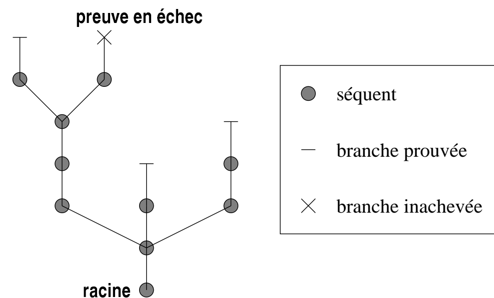


FIG. 2.4 – Faiblesse identifiée dans un arbre de preuve

Une défaillance de l'algorithme durant le test correspond à une falsification du séquent racine. Elle met en évidence non seulement le fait que l'algorithme est incorrect, mais également le fait que sa preuve l'est. La falsification du séquent racine est nécessairement due à la falsification d'une (ou plusieurs) des branches de l'arbre de preuve, et

si notre analyse de l'arbre est correcte, la (ou les) branches fausses feront parties de celles que nous aurons jugées douteuses.

Considérons donc le cas d'un séquent terminal, ou feuille, $\Gamma \vdash P$ jugé comme non-concluant (évaluation false, $\perp$ ou ?). Idéalement, le test devrait chercher à provoquer la falsification de ce séquent. Mais cet objectif n'est pas nécessairement exploitable en termes opérationnels : $\Gamma \wedge \neg P$ peut être non commandable, et non observable. Au pire, si l'analyse de la preuve ne fournit aucune information constructive pour le test, on se retrouve dans la même situation qu'à l'issue de l'Analyse Préliminaire c'est-à-dire :

- La falsification du séquent racine est observable (par l'implémentation de l'oracle de test).
- La falsification du séquent racine n'est pas spécifiquement commandable. Néanmoins, on sait exhiber une fonction de génération d'entrées de test telle que, si la falsification est possible, alors elle a une probabilité non nulle d'occurrence durant le test (implémentation de la distribution aléatoire aveugle sur tout le domaine d'entrée).

En pratique, pour obtenir des informations constructives, on partira du séquent feuille jugé non prouvé, et on remontera dans l'arbre jusqu'à arriver à une étape où l'on identifie : (1) une décomposition en cas de fonctionnement de l'algorithme, que l'on pourra chercher à couvrir, ou (2) l'introduction d'un lemme qui pourra être rendu observable. Nous donnons ci-dessous des exemples type de situations rencontrées dans les études de cas.

- En remontant vers la racine, on arrive à une étape de décomposition en cas de la preuve d'une propriété. Le séquent feuille appartient au sous-arbre de l'un des cas de fonctionnement introduits par les auteurs. Ce cas est pris comme un cas à couvrir durant le test.
- On arrive à une décomposition en cas $C_1 \ldots C_n$, et le séquent feuille affecte la preuve de complétude de la décomposition. Si l'on sait déterminer le sous-domaine d'entrée correspondant à $\neg(C_1 \vee \ldots \vee C_n)$ (le jugement était alors nécessairement false), on peut retenir cette configuration comme étant à couvrir. Si non (le jugement est indécis, $\perp$ ou ?), on ne sait pas si la configuration est faisable. On conserve donc une génération sur le domaine d'entrée complet, et on renforce l'oracle avec un observateur permettant de déterminer *a posteriori* si cette configuration a été couverte durant le test. Selon la définition des cas, ceci peut nécessiter une instrumentation du prototype, pour rendre des variables internes observables.
- On arrive à une étape d'introduction de lemme : on introduit une propriété B comme condition suffisante pour prouver la propriété d'origine A. Le séquent feuille douteux est dans la preuve de B (l'autre sous-arbre partant de cette étape correspond à la preuve de $B \Rightarrow A$). Généralement, on ne sait pas commander spécifiquement la falsification de B, mais B est éventuellement observable sans nécessiter d'effort important. On ajoute alors un observateur de B à l'oracle et le domaine d'entrée complet doit être retenu. Pour guider le test, on peut cependant utiliser des informations extraites de la preuve de $B \Rightarrow A$, lorsque

celle-ci fait apparaître des cas de fonctionnement. En effet, ultimement, c'est la violation de la propriété d'origine A qui nous intéresse (B n'est qu'un artefact de preuve pour établir A). Puisque la preuve de A s'appuie sur un lemme B douteux, A doit aussi être considérée comme douteuse. Les cas apparaissant dans la preuve de $B \Rightarrow A$, considérés par les auteurs comme pertinents pour établir la validité de A, peuvent alors être retenus pour affiner la distribution de probabilités sur le domaine d'entrée complet.

On effectue ainsi l'analyse pour chaque branche de preuve en échec. Le bilan des informations extraites peut ensuite donner lieu à la décision de concevoir une, ou plusieurs, expériences de test (par exemple, un test sur tout le domaine d'entrée selon un certain critère, complété par un test ciblant spécifiquement certains cas de fonctionnement).

2.4.2 Mise en œuvre du test statistique

Pour chaque expérience à réaliser, nous devons déterminer une distribution des probabilités sur le domaine d'entrée, adéquate vis-à-vis du critère retenu. L'approche mise en œuvre dépend de la facilité à commander les cas à couvrir.

La situation la plus favorable est lorsque la définition des cas permet l'identification directe des sous-domaines d'entrée correspondants : on sait alors déterminer les probabilités des cas en fonction des probabilités des valeurs d'entrée, ce qui facilite la conception du profil de test.

Parfois, l'établissement d'un lien entre cas à couvrir et sous-domaines d'entrée nécessite un effort (qui doit rester raisonnable) de modélisation : nous verrons au Chapitre 4 un exemple où la conception d'une distribution d'entrée s'est basée sur un automate à états, explicitant les informations nécessaires à l'activation des cas extraits de la preuve.

Dans des situations plus difficiles, on envisagera une recherche empirique de distribution de probabilités. Rappelons que cette approche consiste à partir d'une distribution simple (par exemple la distribution aléatoire aveugle), à mesurer la fréquence obtenue pour chaque cas à couvrir, et à affiner par essais successifs la distribution jusqu'à obtention d'une fréquence satisfaisante pour chaque cas. La mesure des fréquences des cas activés durant le test peut nécessiter une instrumentation du prototype de l'algorithme. Cette approche empirique de recherche de distribution sera utilisée dans l'étude de cas du Chapitre 3.

2.5 Étape 4 : Retour sur la preuve

Les expériences de test visent à révéler d'éventuelles fautes de conception de l'algorithme. Mais les résultats de test peuvent aussi apporter un autre bénéfice : celui de permettre un retour sur la preuve, qu'une faute ait été révélée ou non.

Rappelons qu'il existe deux types d'observateurs au niveau de l'oracle : 1) ceux qui observent les propriétés exprimées au niveau du séquent racine, et qui déterminent le verdict de défaillance ; 2) d'autres observateurs, éventuellement rajoutés à l'oracle pour déterminer la validité de lemmes intermédiaires dans la preuve. Tous ces observateurs sont susceptibles de fournir des contre-exemples pour la preuve, permettant ainsi d'identifier des parties fausses de l'arbre. Au contraire, la non observation de problèmes durant le test suggère que certains lemmes, jugés douteux, pourraient bien être valides.
Examinons l'intérêt d'une reprise de la preuve dans deux cas, faute révélée ou non.

Dans le cas où une défaillance est observée, on sait que l'algorithme est affecté par une faute de conception. La reprise de la preuve peut alors viser à déterminer la faille de raisonnement qui a laissé passer cette faute, dans un objectif de diagnostic. Cette analyse est guidée par la connaissance des scénarios de défaillance.

Dans le cas où aucune défaillance n'a été observée, il y a de fortes présomptions que l'algorithme soit correct. Ceci encourage à reprendre la preuve informelle, dans l'objectif de la consolider. Lorsqu'un lemme n'a jamais été mis en défaut au cours du test, on pourra tenter d'achever ses branches de preuves auparavant jugées comme faibles. Cela pourra nécessiter quelques modifications structurelles des sous-arbres correspondants, notamment si la restructuration avait conduit à mettre une évaluation false sur certaines feuilles. Lorsque des contre-exemples du lemme ont été trouvés par le test, alors des modifications plus importantes devront être envisagées : toutes les parties de l'arbre dépendant de ce lemme devront être retravaillées.

Au final, si l'algorithme n'a jamais été mis en défaillance durant le test, et si sa preuve a été reprise avec succès, notre confiance dans la validité de l'algorithme aura été significativement augmentée. Si toutefois on jugeait ce résultat encore insuffisant au vu de la criticité du système cible, nous pensons que le travail déjà effectué sur la preuve informelle devrait faciliter le développement d'une formalisation complète.

2.6 Conclusion

Nous avons proposé, dans ce chapitre, une méthode pour concevoir des expériences de test, en se basant sur l'analyse d'une preuve informelle. Cette méthode va maintenant être mise en œuvre sur des cas d'étude, pour une validation expérimentale de sa faisabilité et son efficacité.

On trouve dans la littérature plusieurs exemples d'algorithmes de tolérance aux fautes incorrects, et nous en avons considéré deux :

- un algorithme d'ordonnancement de tâches visant à tolérer les fautes transitoires, par ré-exécution des instances de tâches affectées. Cet algorithme a été publié avec une démonstration informelle, mais contient deux fautes de conception pouvant conduire à la violation d'échéances de tâches.

- un protocole d'appartenance de groupe permettant d'obtenir, dans un système réparti, un consensus sur l'identité des processeurs non défaillants. Par rapport à l'algorithme précédent, il offre un exemple de démonstration informelle plus aboutie, celle-ci ayant été consolidée par analyse d'instances de l'algorithme, en utilisant des techniques de model-checking. Il contient néanmoins une faute de conception pouvant conduire à la violation d'une propriété d'auto-diagnostic.

Ces deux cas d'étude sont développés en détail dans les deux chapitres suivants.

Chapitre 3

Premier cas d'étude d'une preuve informelle : FT-RMS

“ In order to bridge this gap between models and the world of real world entities, it is necessary to experiment. ”
“ Afin d'établir ce pont entre les modèles et le monde des entités de la réalité, il est nécessaire d'expérimenter. ”

— R.M. Hierons dans [Hierons 1997]

Les algorithmes d'ordonnancement de tâches sont au cœur des systèmes temps-réel critiques. Ils doivent permettre d'assurer des temps de réponses prévisibles, de sorte qu'aucune tâche ne manque son échéance. Bien qu'il existe un nombre important de travaux sur l'ordonnancement temps-réel, la plupart des résultats se basent sur des démonstrations informelles. Les techniques actuelles de model-checking restent insuffisantes pour une vérification automatique d'ordonnanceurs complexes, et la preuve de théorèmes peut s'avérer lourde à mettre en oeuvre. Par exemple, la vérification formelle de l'algorithme PCP (*Priority Ceiling Protocol*), réalisée au SRI dans l'environnement PVS, a nécessité de prouver 417 lemmes et théorèmes, pour environ 2500 lignes de spécification PVS [Dutertre 2000].

Dans le cas de systèmes tolérants aux fautes, l'analyse d'ordonnançabilité doit prendre en compte les effets des fautes, et les tâches réalisées par les mécanismes de tolérance. Par exemple, une stratégie classique pour tolérer les fautes transitoires consiste à relancer l'exécution des tâches ayant déclenché une détection d'erreur. Ceci suppose d'avoir prévu de la redondance temporelle, et d'utiliser cette redondance sans violer les échéances de tâches. La politique d'ordonnancement peut être complexe, avec des règles d'ajustement dynamique de la priorité de la tâche en réexécution. Dans ce cas, la détermination de scénarios de pire cas peut devenir problématique (on pourra se reporter à [de A. Lima & Burns 2001] pour une discussion de ce problème).

Le premier exemple que nous avons choisi comme cas d'étude relève de cette catégorie d'algorithmes d'ordonnancement : il inclut un schéma de recouvrement d'erreur par réexécution. Cet algorithme, prouvé par une démonstration informelle, s'est avéré incorrect après publication. Il constitue donc un exemple intéressant pour estimer la pertinence du Test Guidé par la Preuve. Les résultats que nous avons obtenus sur cet exemple, et que nous allons maintenant présenter, ont fait l'objet de deux publications, en français [Lussier & Waeselynck 2002a] et en anglais [Lussier & Waeselynck 2002b].

Ce chapitre est structuré en cinq paragraphes. Dans un premier paragraphe, nous introduisons le cas d'étude et son contexte. Après quelques rappels de base sur la théorie de l'ordonnancement temps-réel, nous donnons une description de l'algorithme lui-même. Nous présentons également les travaux qui ont conduit à montrer que cet algorithme est incorrect. Les quatre paragraphes suivants décrivent l'application des différentes étapes de notre méthode. Dans le deuxième paragraphe, nous menons l'*analyse préliminaire*, qui détermine notamment l'environnement de test d'un prototype de l'algorithme. Dans le troisième paragraphe, nous présentons la *restructuration* de la preuve informelle, dont l'objectif est de dégager les faiblesses éventuelles de la preuve, pour les utiliser comme guides du processus de test. Dans le quatrième paragraphe, nous donnons et commentons les résultats du *test guidé par la preuve informelle*. Les différents résultats obtenus sont notamment comparés à des techniques de test classiques, pour évaluer leur efficacité. Dans un dernier paragraphe, nous effectuons un *retour sur la preuve*, c'est à dire que nous utilisons les résultats de test pour affiner notre jugement sur la preuve informelle.

Nous rappelons que nos objectifs expérimentaux sont : 1) d'étudier la faisabilité de la méthode proposée, notamment de l'étape de restructuration de la preuve informelle sous forme d'arbre et de l'extraction d'informations pour guider le test ; et 2) d'évaluer l'intérêt de l'approche de Test Guidé par la Preuve sur un cas d'étude concret et réaliste.

3.1 Présentation du premier cas d'étude : le FT-RMS

L'algorithme FT-RMS (*Fault Tolerant Rate Monotonic Scheduling*) est un algorithme d'ordonnancement de tâches ayant pour but de tolérer des fautes transitoires par réexécution des instances de tâches fautives. Des variantes de cet algorithme ont été implantées dans deux systèmes d'exploitation temps-réel [Dong et al. 1999, Egan et al. 1999] : RT-Mach, un système académique, et DEOS (Digital Engineering Operating System), un système commercial pour les applications avioniques. La première définition du FT-RMS contenait deux fautes de conception [Ghosh et al. 1997], l'une d'entre elles ayant été corrigée dans une version révisée de l'algorithme [Ghosh et al. 1998]. Cet algorithme a ensuite été étudié par d'autres auteurs [Sinha & Suri 1999a] qui ont révélé la seconde faute. Notons que les implémentations de l'algorithme dans RT-Mach et DEOS correspondent à des variantes de l'algorithme qui peuvent tout à fait être correctes. Dans cette étude, nous nous focaliserons sur

les deux versions connues comme incorrectes, présentées dans [Ghosh et al. 1997, Ghosh et al. 1998].

Dans cette partie, nous introduisons brièvement les principes de l'algorithme RMS (*Rate Monotonic Scheduling*) sur lesquels le FT-RMS est basé, puis nous expliquons le fonctionnement du FT-RMS lui-même. Enfin nous présentons les travaux qui ont amené la seconde faute de conception à être révélée.

3.1.1 L'algorithme RMS

Le RMS est un algorithme d'ordonnancement statique de tâches périodiques, à priorités fixes. Chaque tâche τ_i est caractérisée par son temps d'exécution C_i et sa période d'activation T_i.

- *propriété à satisfaire :*
  Aucune instance de tâche ne doit manquer son échéance : la $k^{ième}$ instance de τ_i est activée (devient *ready*) à la date $t_{0i} + (k-1)T_i$ et doit être terminée avant l'activation de sa prochaine instance à la date $t_{0i} + kT_i$.
- *principe de l'algorithme :*
  La politique d'ordonnancement RM (*Rate Monotonic*) assigne des priorités aux tâches selon leur période d'activation : les tâches ayant des périodes plus courtes ont des priorités plus hautes. La tâche *ready* de plus haute priorité est toujours choisie pour être exécutée, et l'exécution d'une tâche est préemptée lorsqu'une tâche de plus haute priorité devient *ready*.

Les priorités étant attribuées de façon définitive, l'ordonnancement complet des tâches peut donc être déterminé hors-ligne avant leur exécution (ordonnancement statique).

Les schémas d'ordonnancement statique à priorités fixes ont été étudiés de façon approfondie. Nous rappelons ci-dessous un certain nombre de résultats théoriques.

Une condition suffisante pour qu'un ensemble de n tâches soit ordonnançable selon la politique RMS a été établie par Liu et Layland [Liu & Layland 1973] :

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \quad \leq \quad U_{LL} = n(2^{1/n} - 1) \tag{3.1}$$

U est le *taux d'utilisation*, c'est-à-dire la fraction du temps processeur passée à exécuter l'ensemble de tâches. La borne U_{LL} sur U est obtenue par une analyse de pire cas. Liu et Layland introduisent d'abord la notion d'*instant critique*, quand toutes les tâches deviennent *ready* en même temps : les tâches ont alors le temps de réponse le plus long. Ils considèrent ensuite les ensembles de tâches remplissant la condition d'*utilisation maximale* du processeur à l'instant critique : l'ordonnancement est faisable pour ces ensembles de tâches, mais toute augmentation du temps d'exécution C_i de l'une d'entre elles le rendrait impossible à réaliser. Ces ensembles de tâches sont de la forme (*c.f.* [Liu & Layland 1973]) :

$$\begin{cases} C_i = T_{i+1} - T_i & \forall\, i.\ 1 \leq i \leq n-1 \\ C_n = 2T_1 - T_n \end{cases} \tag{3.2}$$

les n tâches étant ordonnées selon leur priorité ($T_1 \leq T_2 \leq \ldots \leq T_n$). Liu et Layland ont démontré que le minimum des taux d'utilisation sur tous ces ensembles de tâches en *utilisation maximale* est : $U_{LL} = n(2^{1/n} - 1)$. Ainsi tout ensemble de tâches ayant un taux d'utilisation inférieur à cette valeur est ordonnançable en RMS.

La condition (3.1) correspond à une borne pessimiste. Elle autorise un taux d'utilisation U de 82,84% pour $n = 2$, et ce taux descend à 69,31% pour de grandes valeurs de n. Pour pallier ce problème, une analyse exacte a été proposée dans des travaux postérieurs : elle consiste à calculer le pire temps de réponse R_i de chaque tâche τ_i, et à vérifier que $R_i \leq T_i$. Soit $\{\tau_1, \ldots, \tau_{i-1}\}$ l'ensemble des tâches ayant une priorité plus forte que τ_i. Alors R_i est la plus petite solution de l'équation :

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil \times C_j \tag{3.3}$$

Une façon pratique de calculer ces R_i est de réécrire l'équation (3.3) comme une relation de récurrence et d'itérer le calcul jusqu'à l'obtention d'un point fixe ou le dépassement de T_i (*c.f.* [Joseph 1996]).

3.1.2 Deux versions de l'algorithme FT-RMS

L'algorithme FT-RMS étend le RMS en y incorporant de la tolérance aux fautes. Les fautes sont supposées être transitoires : la persistence de chaque faute est considérée suffisamment courte pour n'affecter qu'une seule instance de tâche. Le recouvrement est alors effectué par réexécution de cette instance. La détection d'erreur a lieu à la fin de l'exécution de chaque instance de tâche, et son coût est considéré comme étant inclus dans le temps d'exécution C_i. Des hypothèses supplémentaires concernent l'intervalle de temps entre deux fautes, mais nous les ignorons ici car le comportement erroné de l'algorithme peut être révélé en se limitant à une unique faute transitoire.

Le principe de l'algorithme est d'utiliser le RMS lorsqu'il n'y a pas de faute, et, lorsqu'une faute est détectée, de réexécuter la tâche concernée selon un certain *schéma de recouvrement* (Recovery Scheme). Il doit y avoir assez de redondance temporelle pour que ni la tâche réexécutée, ni aucune autre tâche ne manquent leurs échéances. Pour cela du *temps mort* (ou *slack*) est réservé pour la réexécution, et le schéma de recouvrement décrit comment ce *slack* doit être utilisé pour la réexécution dynamique. La réservation de *slack* est identique dans les deux versions de l'algorithme [Ghosh et al. 1997, Ghosh et al. 1998], mais le schéma de recouvrement n'est pas le même. La Définition 3.1 donne une vue d'ensemble de ces deux versions de l'algorithme, dont nous allons maintenant détailler les hypothèses et les principes de fonctionnement.

Définition 3.1 — Algorithme du FT-RMS

hypothèse : l'ensemble de tâches est ordonnançable en IBRMS

objectif : pouvoir réexécuter une instance de tâche fautive tout en assurant que toutes les tâches respectent leurs échéances

algorithme :

- en absence de fautes, on suit une politique RMS
- lorsqu'une faute survient, on suit un schéma de recouvrement :
  - version 1 : réexécution de l'instance à la même priorité,
  - version 2 : réexécution de l'instance à la même priorité, sauf vis-à-vis des tâches ayant une échéance plus tardive (celles-ci ne peuvent plus préempter la réexécution).

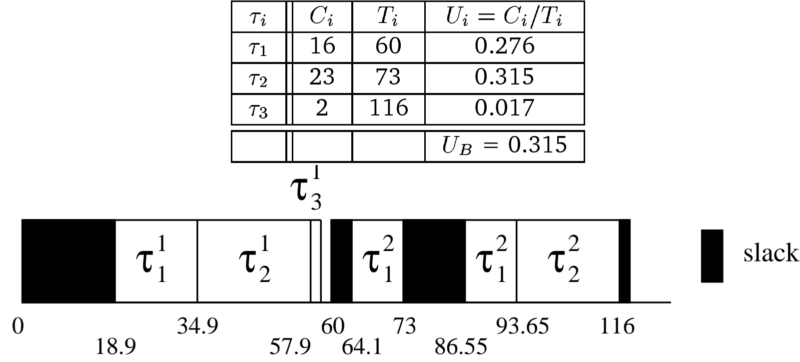
Hypothèse de l'algorithme : réservation de slack selon l'IBRMS

L'introduction de redondance temporelle se traduit par une condition sur l'ensemble de tâches considéré : il doit être ordonnançable en considérant une politique RMS avec *slack* inséré, l'IBRMS (*Inserted Backup RMS*). L'ordonnançabilité en IBRMS peut être vérifiée *a priori*, hors ligne. On construit un échéancier fictif, c'est-à-dire ne correspondant pas à une exécution réelle du système, et matérialisant l'insertion de *slack*.

Selon la définition de l'IBRMS, la quantité de *slack* insérée sur un intervalle de temps donné doit être proportionnelle à la longueur de cet intervalle. La proportion considérée, U_B, est constante dans le temps et égale au maximum des taux d'utilisation des tâches, c'est à dire : $U_B = max(C_i/T_i)$.

Pour matérialiser cet IBRMS, il faut cependant définir les intervalles sur lesquels il sera calculé. Le principe choisi par les auteurs consiste à insérer le *slack* entre les débuts de deux périodes consécutives ; par exemple entre la date d'activation de la $k^{ième}$ instance de τ_i et celle de la $l^{ième}$ instance de τ_j, sachant qu'aucune autre tâche n'est activée entre ces deux dates. Le *slack* est placé au début de l'intervalle de temps correspondant, et si cet intervalle est de L unités de temps, la longueur du *slack* est de $U_B \times L$. Il est à noter que, dans le schéma IBRMS, le slack est plus prioritaire que toutes les tâches.

Un exemple de schéma IBRMS pour un ensemble de trois tâches est donné Figure 3.1, où τ_i^k dénote la $k^{ième}$ instance de τ_i. Nous rappelons que la priorité des tâches est assignée inversement à leur période, ainsi τ_1 a la plus haute priorité et τ_3 la plus basse. Dans la construction de l'échéancier IBRMS, l'instant $t = 0$ est un *instant critique* (toutes les tâches sont *ready*).



τ_i	C_i	T_i	$U_i = C_i/T_i$
τ_1	16	60	0.276
τ_2	23	73	0.315
τ_3	2	116	0.017
			$U_B = 0.315$

FIG. 3.1 – Ordonnançabilité d'un ensemble de tâches selon l'IBRMS

Pour les auteurs, l'IBRMS peut être interprété comme l'ajout d'une tâche fictive de recouvrement (*backup*) τ_B ayant pour période $T_B = pgcd(T_i)$ et pour temps d'exécution $C_B = U_B \times T_B$ [1].

Algorithme FT-RMS et schémas de recouvrement

Considérons maintenant qu'un ensemble de tâches soit ordonnançable en IBRMS. Durant l'exécution effective des tâches, une politique RMS est suivie en absence de faute. Ceci revient à modifier l'échéancier, avec une permutation entre les créneaux temporels du *slack* et des tâches. Le *slack* peut donc être vu comme repoussé dans le temps par l'exécution des tâches. Lorsqu'une erreur est détectée, ce *slack* est récupéré en suivant le schéma de recouvrement.

Le **premier schéma de recouvrement** [Ghosh et al. 1997] est le plus simple. La tâche τ_r en faute est ré-exécutée à la même priorité. Mais ce principe est imparfait, comme cela est montré par la Figure 3.2.a, pour le même ensemble de tâches que précédemment : des tâches de priorité supérieure peuvent préempter la réexécution de τ_r, l'empêchant d'obtenir le *slack* théoriquement réservé. Pour résoudre ce problème, un deuxième schéma de recouvrement est proposé dans [Ghosh et al. 1998].

Avec ce **deuxième schéma de recouvrement**, la réexécution de la tâche τ_r ne peut plus être préemptée par des tâches ayant une échéance plus tardive, même si leur priorité est supérieure (*c.f.* Figure 3.2.b). On a donc, de fait, un réajustement dynamique de la priorité de τ_r. Cependant, comme nous le verrons dans le paragraphe 3.1.3, ce schéma contient encore une faute résiduelle (illustrée par la Figure 3.3, page 55).

[1] Nous avons cependant pu constater que cette interprétation n'est pas équivalente à la définition opératoire donnée précédemment, et illustrée par la Figure 3.1. La différence est significative lorsque l'on calcule les pires temps de réponse R_i des différentes tâches dans les deux cas : IBRMS ou ajout d'une tâche backup.

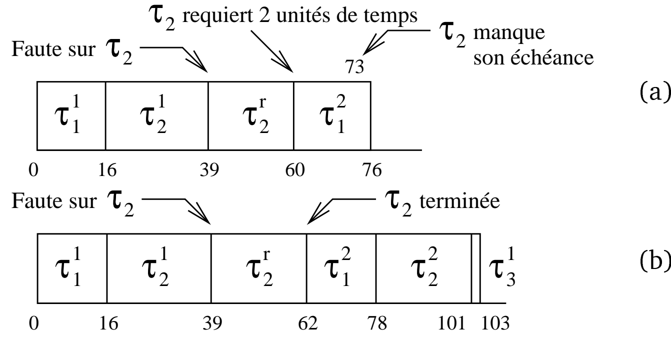


FIG. 3.2 – Ordonnancement FT-RMS, versions 1 et 2

Renforcement de la condition d'ordonnançabilité

Quel que soit le schéma de recouvrement, la preuve informelle de l'algorithme est basée sur l'hypothèse que l'ensemble de tâches est ordonnançable en IBRMS. Un utilisateur de l'algorithme peut cependant souhaiter avoir un critère plus facile à vérifier, qui ne nécessite pas de calcul d'ordonnançabilité. Pour cette raison, le auteurs proposent dans [Ghosh et al. 1997] et [Ghosh et al. 1998] une condition plus forte (la satisfaction de cette condition implique l'ordonnançabilité en IBRMS), basée sur un calcul simple du taux d'utilisation du temps processeur. La borne pour le taux d'utilisation en FT-RMS est liée au taux U_{LL} de Liu et Layland (*c.f.* équation (3.1)) comme suit :

$$\begin{aligned} U_{FT-RMS} &= U_{LL}\,(1 - U_B) \\ &= n\,(2^{1/n} - 1)(1 - U_B) \end{aligned} \tag{3.4}$$

Elle correspond à une variante de l'analyse pire cas de Liu et Layland (*c.f.* § 3.1.1, équations (3.1) et (3.2)), l'ensemble de tâches en *utilisation maximale* ayant été modifié pour tenir compte de la réservation de *slack* :

$$\begin{cases} C_i = (T_{i+1} - T_i)(1 - U_B) & \forall\, i.\ 1 \le i \le n-1 \\ C_n = (2T_1 - T_n)(1 - U_B) \end{cases} \tag{3.5}$$

Ce seuil U_{FT-RMS} a été pris en compte par les auteurs des travaux qui ont mis en évidence une faute de conception résiduelle dans le deuxième *schéma de recouvrement*, c'est-à-dire dans la seconde version de l'algorithme. Ces travaux [Sinha & Suri 1999a] sont présentés au paragraphe suivant.

3.1.3 Vérification du FT-RMS, résultats antérieurs

Les travaux de Sinha et Suri poursuivaient des objectifs similaires aux nôtres. Ils étudiaient l'utilisation de méthodes formelles pour guider le test, et avaient identifié

le FT-RMS comme un exemple pertinent pour aborder ce thème. La première version de l'algorithme étant connue comme incorrecte, ils désiraient étudier la capacité de leur approche à révéler la faute de conception correspondante. Il nous paraît donc intéressant de discuter de ces travaux, avant de détailler nos résultats.

L'approche proposée dans [Sinha & Suri 1999a] analyse le comportement de l'algorithme (première version [Ghosh et al. 1997]) en utilisant des exécutions symboliques de formules PVS. Le modèle PVS utilisé [Sinha & Suri 1999b] ne fournit pas une description opérationnelle de l'algorithme lui-même (la construction de l'échéancier n'y est pas décrite). Les auteurs proposent plutôt une interprétation de l'algorithme en termes de calcul de temps de réponse, en utilisant des formules telles que l'équation (3.3) du paragraphe 3.1.1. Les auteurs introduisent également un certain nombre de lemmes sur le modèle PVS, pour tester la valeur de vérité de certaines propriétés de l'algorithme. Ces propriétés peuvent être directement dérivées de lemmes de la preuve informelle, ou des résultats d'interrogations antérieures du modèle PVS avec des valeurs numériques.

Une première analyse du comportement du modèle a amené Sinha et Suri à établir que la réservation de *slack* pour le FT-RMS était erronée. Il existe une différence entre la quantité de *slack* qui est censée être disponible selon l'IBRMS, et le *slack* réellement accessible pour la réexécution dynamique. À partir de ces résultats, les auteurs ont affiné leurs recherches en choisissant un cas spécifique ciblant une *utilisation maximale*.

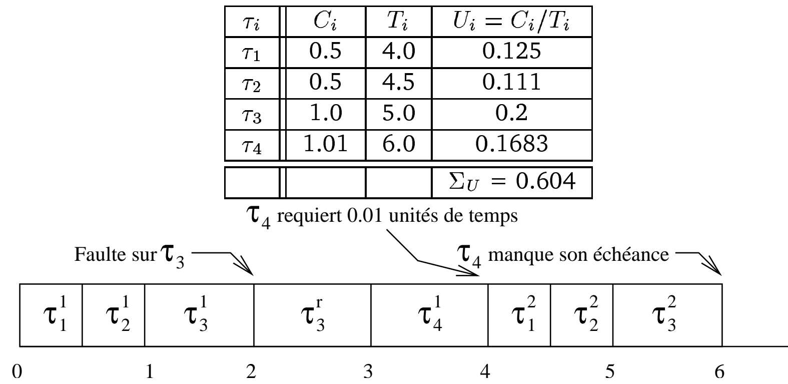
Les ensembles de tâches considérés comme un pire cas sont définis comme ci-dessous, où Δ est un nombre positif aussi petit que désiré :

$$\begin{aligned} C_i &= T_{i+1} - T_i, \quad \forall\, i.\ 1 \leq i \leq n-1, \\ C_n &= (2T_1 - T_n)/2 + \Delta, \\ &\quad avec \sum_i \frac{C_i}{T_i} \leq U_{FT-RMS} \end{aligned} \tag{3.6}$$

Dans cette définition, la condition d'ordonnançabilité en IBRMS est prise en compte par la limitation sur le taux d'utilisation global, qui doit être inférieur au seuil U_{FT-RMS} (équation (3.4)). Si on compare cet ensemble de tâches à la définition originale de Liu et Layland (équation (3.2)), le temps de calcul de la dernière tâche a été divisé par deux. Cette modification assure que l'ensemble de tâches se trouve en utilisation maximale du processeur si la tâche qui se ré-exécute est τ_n, mais cela réduit également la généralité de l'équation (3.6) pour étudier le comportement du FT-RMS, car la dernière tâche est traitée différemment des autres. La valeur Δ ajoutée à C_n empêche la réexécution de cette tâche dans la première version de l'algorithme, et cela bien que la condition de faisabilité de l'IBRMS soit respectée.

Les auteurs ont exhibé un ensemble de quatre tâches satisfaisant les équations ci-dessus, et l'ont incorporé au modèle PVS de l'algorithme. Ils ont ainsi vérifié que la première version de l'algorithme échoue si la quatrième tâche doit être ré-exécutée. Mais ils ont également observé un échec de l'algorithme en cas de réexécution de la troisième tâche (*c.f.* Figure 3.3). Ce défaut, contrairement au premier, n'est pas corrigé dans la deuxième version du FT-RMS.

Cette approche a donc permis d'obtenir deux classes de scénarios de test, qui satisfont toutes deux les contraintes données par l'équation (3.6). La première correspond à une faute sur la tâche de plus faible priorité τ_n et la deuxième sur τ_{n-1}. Ces classes de scénarios sont présentées comme des classes d'équivalence, c'est-à-dire que tous les membres d'une classe ont la même capacité à provoquer une défaillance de l'algorithme. Les scénarios révèlent, respectivement, la première et la deuxième faute de conception.



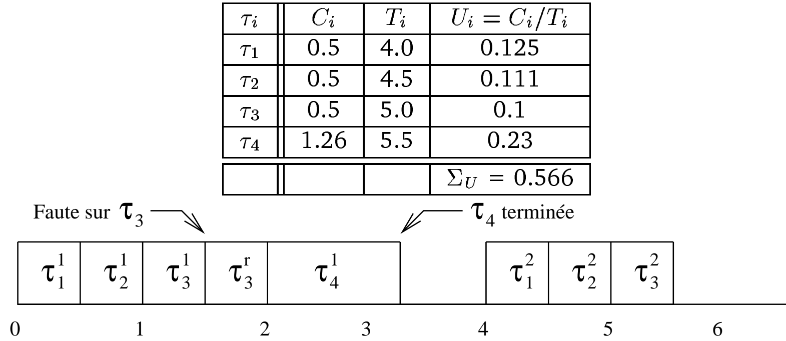
τ_i	C_i	T_i	$U_i = C_i/T_i$
τ_1	0.5	4.0	0.125
τ_2	0.5	4.5	0.111
τ_3	1.0	5.0	0.2
τ_4	1.01	6.0	0.1683
			Σ_U = 0.604

FIG. 3.3 – Echec des versions 1 et 2 du FT-RMS

Cependant, nous avons constaté que la deuxième classe d'équivalence n'est pas homogène dans sa capacité à révéler la deuxième faute de conception. En effet, un ensemble de tâches vérifiant les contraintes correspondantes (équation (3.6) et réexécution de τ_{n-1}) sera ordonnançable sous les deux versions de l'algorithme si $C_{n-1} \leq C_n - 2\Delta$, comme cela est illustré par la Figure 3.4. Dans cet exemple, $U_B = max(U_i) = 0.23$, et le seuil FT-RMS est $U_{FT-RMS} = 0.58$, selon l'équation (3.4). L'ensemble de tâches vérifie toutes les contraintes de la deuxième classe de scénarios mais aucune défaillance n'est constatée.

Les résultats présentés dans [Sinha & Suri 1999a] se sont donc avérés biaisés par le choix de valeurs spécifiques pour instancier leurs ensembles de tâches. Un autre choix aurait pu conduire à ne pas révéler la deuxième faute de conception.

La discussion que nous avons faite de ces travaux montre la difficulté d'identifier les pires cas d'algorithmes tels que le FT-RMS, par un raisonnement basé sur l'intuition. Pour notre part, nous allons maintenant étudier en détail la démonstration donnée par les auteurs de cet algorithme, dans sa deuxième version, et essayer d'en extraire des informations utiles au test.



τ_i	C_i	T_i	$U_i = C_i/T_i$
τ_1	0.5	4.0	0.125
τ_2	0.5	4.5	0.111
τ_3	0.5	5.0	0.1
τ_4	1.26	5.5	0.23
			$\Sigma_U = 0.566$

FIG. 3.4 – Ensemble de tâches sans échec

3.2 Étape 1 : Analyse Préliminaire

La première étape de la méthode de Test Guidé par la Preuve est l'analyse préliminaire. A la fin de cette étape, nous souhaitons avoir acquis une connaissance suffisante du FT-RMS et de son environnement pour pouvoir implémenter un prototype et un environnement de test de cet algorithme. Cela requiert trois éléments :

- *le prototype* :
  Pour pouvoir tester le FT-RMS, nous avons développé un prototype de cet algorithme en langage C pour chacune de ses deux versions [Ghosh et al. 1997, Ghosh et al. 1998]. Les deux prototypes se basent sur les deux *schémas de recouvrement* proposés pour construire un échéancier en présence d'une faute. Dans l'implantation, nous avons fait le choix de considérer un temps discret à valeurs entières. De plus, la date $t = 0$ est prise comme étant un *instant critique* : les premières requêtes de toutes les tâches arrivent à cet instant (toutes les tâches sont *ready* en 0).
- *le domaine d'entrée* :
  Chaque entrée de test est un ensemble de tâches défini par :
  - le nombre n de tâches, avec $n > 1$;
  - la période T_i et le temps d'exécution C_i de chaque tâche, avec $T_1 \leq \ldots \leq T_n$ (les tâches sont ordonnées selon leur priorité), et l'ensemble de tâches est ordonnançable en IBRMS ;
  - la description de la faute, avec l'identité de la tâche affectée τ_r et le numéro de l'instance devant être réexécutée ; le numéro de cet instance est tel que son échéance $D_r \leq PPCM(T_i)$; en effet, le plus petit commun multiple des T_i est le prochain instant critique, et on peut sans perte de généralité limiter la fenêtre temporelle de la faute à l'intervalle $[0 \ldots PPCM(T_i)[$.
- *l'oracle* :
  L'oracle de test a été intégré au prototype du FT-RMS, et effectue ses vérifications

en-ligne pendant le calcul de l'échéancier : à chaque échéance de tâche kT_i, il vérifie si l'instance correspondante est terminée. L'oracle de test reporte alors tout problème dû à un dépassement d'échéance. Notons que, en l'absence de problème, l'effet de la faute sur l'échéancier a disparu en $t = D_r + T_{n-1} + T_n$. Ceci permet, sans perte de généralité, de se limiter à des échéanciers partiels. En réponse à une entrée de test, le prototype calcule et vérifie l'ordonnancement des tâches entre 0 et $D_r + T_{n-1} + T_n$.

L'implantation de l'environnement de test complet, incluant une des deux versions du FT-RMS, le moniteur de test et l'oracle représente environ 900 lignes de code C.

Le domaine d'entrée précédemment défini détermine des fenêtres temporelles finies, mais pouvant être arbitrairement grandes. Dans le cadre d'expériences de test, nous devons introduire des bornes, et donc renforcer les hypothèses d'entrée. Nous donnons ci-dessous une définition opérationnelle du domaine d'entrée retenu pour nos expériences, sous la forme d'une fonction de génération aléatoire aveugle.
Chaque tâche τ_i générée est telle que :

$$5 \leq T_i \leq 130 \text{ , et } \quad 1 \leq C_i < T_i \text{ , et } \quad C_i/T_i < 0,4. \tag{3.7}$$

La construction d'un ensemble de tâches est réalisée en deux étapes : 1) on génère la tâche qui détermine U_B (c'est à dire la tâche ayant le plus grand taux d'utilisation C_i/T_i), puis 2) on ajoute de nouvelles tâches tant que l'ensemble reste ordonnançable en IBRMS. Tout ensemble doit contenir au moins deux tâches. Le test d'ordonnançabilité en IBRMS repose sur un calcul exact des temps de réponse après l'instant critique, plutôt que sur la borne conservatoire U_{FT-RMS}. Il nous a en effet paru important de rester au plus près des hypothèses considérées dans la preuve.
On génère alors uniformément l'identité de τ_r dans $[1 \ldots n]$, puis le numéro k de l'instance à ré-exécuter. Comme le PPCM des T_is peut encore être très grand, nous avons imposé que l'échéance $D_r = kT_r$ de cette instance soit inférieure à $t = 2^{20}$. On tire donc k tel que : $kT_r \leq Min(2^{20}, PPCM(T_i))$. Mis à part cette contrainte, la distribution aléatoire aveugle n'impose aucune limitation quant à la configuration des différentes tâches au moment de la faute.
Le Tableau 3.1 présente les résultats obtenus avec 10^4 ensembles de tâches générés selon cette distribution.

TAB. 3.1 – Résultats d'un test aléatoire aveugle

Nombre d'entrées	Taux de défaillance Version 1	Taux de défaillance Version 2
10^4	0,93%	0,04%

Comme on peut le constater, la défaillance de la deuxième version de l'algorithme est bien plus difficile à provoquer que celle de la première version. Nous allons maintenant

nous consacrer à l'étude de la preuve informelle de cette deuxième version, dans le but de déterminer si elle permet de mieux guider le test.

3.3 Étape 2 : Restructuration de la preuve

Dans cette étape, nous utilisons la *déduction naturelle* de Gentzen sous forme de séquent pour restructurer la preuve informelle du FT-RMS, et analyser l'arbre de preuve résultant (*c.f.* paragraphe 2.3).

Nos évaluations des différentes branches de preuve s'expriment à l'aide des notations introduites au chapitre précédent : true, false, $\bot$, ?. De plus, par souci de concision, nous avons ajouté une nouvelle notation : *complétude* est utilisée pour raccourcir la notation de la preuve de complétude d'une décomposition en cas, par l'application de la règle $\vee_e$.

Après une vue générale de la structure de la preuve informelle (§ 3.3.1), nous détaillons les sous-arbres correspondants (§ 3.3.2). Un bilan des faiblesses identifiées est ensuite réalisé (§ 3.3.3).

3.3.1 Structure générale de la preuve

La preuve informelle du FT-RMS est basée sur trois lemmes intermédiaires :

> $[S_1]$: Pour chaque tâche τ_i un *slack* d'au moins C_i doit être présent entre les instants kT_i et $(k+1)T_i$.
> $[S_2]$: Si une faute est détectée durant l'exécution de τ_r, alors le schéma de recouvrement doit permettre à τ_r de se réexécuter pendant une durée de C_r avant son échéance.
> $[S_3]$: Lorsqu'une tâche se réexécute, elle ne doit amener aucune autre tâche à manquer son échéance.

La Figure 3.5 montre comment ces lemmes sont utilisés dans la structure globale de la preuve de l'algorithme.

La liste Γ des hypothèses intègre plusieurs notions, que nous avons identifiées lors de l'analyse préliminaire :

- le classement des tâches selon leur priorité ($T_1 \leq T_2 \leq \ldots \leq T_n$),
- le fait que $t = 0$ est un instant critique,
- la condition d'ordonnançabilité de l'ensemble de tâches en IBRMS,
- l'occurrence d'une faute unique affectant une instance de τ_r,
- l'algorithme lui-même (stratégie RMS en l'absence de faute, et deuxième schéma de recouvrement après la faute).

$$\dfrac{\dfrac{\dfrac{\text{preuve } [S_1]}{\Gamma \vdash [S_1]} \quad \dfrac{\dfrac{\dfrac{\text{preuve } [S_2]}{\Gamma, [S_1] \vdash [S_2]} \quad \dfrac{\text{preuve } [S_3]}{\Gamma, [S_1] \vdash [S_3]}}{\Gamma, [S_1] \vdash [S_2] \wedge [S_3]}}{\Gamma \vdash [S_1] \Rightarrow [S_2] \wedge [S_3]}}{\Gamma \vdash [S_2] \wedge [S_3]} \qquad \dfrac{\bot}{\Gamma \vdash [S_2] \wedge [S_3] \Rightarrow \text{ordonnancement valide}}}{\Gamma \vdash \text{ordonnancement valide}} \Rightarrow_e$$

FIG. 3.5 – Racine de la preuve informelle du FT-RMS

Sous ces hypothèses, aucune tâche ne doit manquer son échéance, ce qui revient à dire que l'ordonnancement déterminé par le FT-RMS est valide.

Dans la preuve, la validité de la première règle $\Rightarrow_e$ introduisant les sous-buts $[S_2]$ et $[S_3]$ est considérée comme acquise par les auteurs, mais l'implication n'est pas si évidente à démontrer. En effet $[S_2]$ exprime une propriété sur l'instance de la tâche τ_r devant être réexécutée, mais rien n'est dit en ce qui concerne ses autres instances. Comme la réexécution peut, dans la deuxième version du FT-RMS, repousser dans le temps des tâches de plus haute priorité, et comme ces tâches peuvent elles-même préempter l'instance suivante de τ_r, on ne peut rien déduire trivialement. Ceci explique notre évaluation $\bot$. Plus généralement, le fait que les conditions $[S_2]$ et $[S_3]$ soient exprimées en termes de tâches, et non d'*instances* de tâches, peut être considéré comme un manque de précision. Les preuves de $[S_1]$, $[S_2]$ et $[S_3]$ sont présentées dans le paragraphe suivant.

3.3.2 Preuves des trois lemmes principaux

Preuve de $[S_1]$

La preuve de $[S_1]$, présentée Figure 3.6, est basée sur le critère de faisabilité de l'IBRMS. En IBRMS, une quantité de *slack* $U_B\ T_i$ est insérée sur toute période T_i, et comme $U_B \geq C_i \,/\, T_i$, alors la quantité de *slack* sur T_i est au moins C_i. Cette preuve est évidente si l'on considère que $[S_1]$ est une propriété liée à l'IBRMS. Mais les commentaires accompagnant la démonstration, et l'utilisation de $[S_1]$ pour prouver $[S_2]$ et $[S_3]$, nous amènent à penser que les auteurs en ont une interprétation plus large : $[S_1]$ est amenée à caractériser la quantité de *slack* présente dans un échéancier FT-RMS. En fait, le *slack* est considéré comme étant repoussé dans le temps par l'exécution des tâches. Mais la relation entre le *slack* inséré dans l'IBRMS et la distribution de temps morts en FT-RMS n'est jamais clairement posée. Nous considérons donc que l'utilisation des lemmes (1) et (2) n'est pas suffisante pour prouver la validité de $[S_1]$ dans le cadre du FT-RMS. Il est à noter que cette confusion entre le *slack* en IBRMS et en FT-RMS avait déjà été mise en évidence par Suri et Sinha [Sinha & Suri 1999a].

$$\dfrac{\dfrac{\dfrac{\text{true}: \mathit{slack}}{\Gamma \vdash \forall i.\ slack(T_i) = U_B T_i} \quad \dfrac{\text{true}: U_B}{\Gamma \vdash \forall i.\ U_B \geq C_i \ / \ T_i}}{\Gamma \vdash (1)\ \forall i.\ slack(T_i) = U_B T_i \ \wedge (2)\ \forall i.\ U_B \geq C_i \ / \ T_i} \quad \dfrac{\bot}{\Gamma \vdash (1) \wedge (2) \Rightarrow [S_1]}}{\Gamma \vdash [S_1]} \Rightarrow_e$$

FIG. 3.6 – Arbre de la preuve informelle de la condition $[S_1]$

$$\dfrac{\dfrac{\text{Cas 1 ?} \quad \text{Cas 2 ?} \quad \text{Cas 3 ?} \quad \dfrac{\bot}{\mathit{compl\acute{e}tude}}}{\Gamma, [S_1] \vdash \mathit{slack} \text{ réservé libre avant } D_r} \quad \dfrac{\bot}{\Gamma, [S_1] \vdash \mathit{slack} \text{ réservé libre avant } D_r \Rightarrow [S_2]}}{\Gamma, [S_1] \vdash [S_2]} \Rightarrow_e$$

FIG. 3.7 – Arbre de la preuve informelle de la condition $[S_2]$

Preuve de $[S_2]$

La preuve de S_2, présentée Figure 3.7, est construite en introduisant une nouvelle condition suffisante pour S_2, notée *slack réservé libre avant* D_r, et qui exprime que le *slack* réservé ne peut pas être repoussé au-delà de l'échéance D_r de l'instance fautive de τ_r. Justifier que cette condition est suffisante nécessite d'utiliser la définition de *slack* en S_1 comme applicable au FT-RMS, ce que nous avons critiqué ci-dessus. Nous avons donc mis une évaluation $\bot$ sur la branche la plus à droite. La preuve de la condition elle-même est décomposée en trois cas. Chacun d'entre eux considère une tâche τ_h de priorité supérieure à τ_r, et pouvant potentiellement la préempter (ces cas sont présentés Tableau 3.2, page 64). Dans le premier cas (Cas 1), τ_h a une échéance D_h inférieure ou égale à D_r, et préempte l'exécution ou la réexécution de τ_r. Les deux autres cas (Cas 2 et Cas 3)correspondent à une tâche τ_h ayant une échéance strictement postérieure à celle de τ_r, et dont la requête est reçue respectivement après et avant E_r, la date de fin d'exécution de l'instance en faute. La preuve de la complétude de ces trois cas n'est pas fournie.

Nous avons arrêté notre arbre de preuve au niveau des différents cas dégagés par les auteurs, à cause des limites de la formulation du discours informel. A titre d'exemple, nous donnons ci-dessous la démonstration du Cas 1 (τ_r est préemptée et $D_h \leq D_r$) :

> " Selon le schéma de recouvrement, τ_h va préempter τ_r et le *slack* réservé sera permuté avec l'exécution de τ_h. Ainsi le *slack* repoussé se trouve dans D_h. Et puisque $D_h \leq D_r$ on peut conclure que tout le *slack* repoussé par τ_h se trouve dans D_r. "

Une analyse fine de cette démonstration nécessiterait des définitions claires de notions comme la *permutation* de *slack*, le *slack réservé*, le *slack repoussé* et *se trouve dans* qui manquent dans la preuve informelle. De plus, l'enchaînement logique "*Ainsi*"au début de la deuxième phrase est peu clair. Nous n'avons pu interpréter ce discours

en termes de séquents et de règles d'inférence, d'où l'arrêt du développement de l'arbre avec une évaluation ?. Certaines remarques peuvent toutefois être faites sur la décomposition choisie : chacun des trois cas n'implique que deux tâches, τ_r et τ_h, ce qui limite fortement la généralité de telles démonstrations. Par exemple, des préemptions multiples par plusieurs tâches de priorités supérieures ne sont pas envisagées.

$$\dfrac{\dfrac{\dfrac{\text{Cas 4 ?} \quad \text{Cas 5 ?} \quad \text{Cas 6 ?} \quad \dfrac{\bot}{\textit{complétude}}}{\Gamma, [S_1], i > r \vdash \tau_i \text{ fini avant } D_i} \quad \dfrac{\dfrac{\text{Cas 7 ?}}{\tau_i \text{ retardée}} \quad \dfrac{\text{true}}{\neg\tau_i \text{ retardée}} \quad \dfrac{\text{true}}{\textit{complétude}}}{\Gamma, [S_1], i < r \vdash \tau_i \text{ fini avant } D_i} \quad \dfrac{\text{true}}{\textit{complétude}}}{\Gamma, [S_1], i \neq r \vdash \tau_i \text{ fini avant } D_i} \vee_e}{\Gamma, [S_1] \vdash [S_3]}$$

FIG. 3.8 – Arbre de la preuve informelle de la condition $[S_3]$

Preuve de $[S_3]$

La condition $[S_3]$, présentée Figure 3.8, est probablement la plus difficile à prouver, et nous savons qu'elle est fausse car un contre-exemple (Figure 3.3) a été mis en évidence dans [Sinha & Suri 1999a]. $[S_3]$ peut se reformuler de la façon suivante :

$\forall\, i.\ (\ i \neq r \Rightarrow \tau_i$ se termine avant son échéance)
ou encore :
$\forall\, i.\ (\ i \neq r \Rightarrow \tau_i$ fini avant D_i)

Pour raccourcir la présentation de l'arbre de preuve, nous utilisons la seconde notation. Après avoir passé $i \neq r$ en hypothèse, la preuve est divisée en deux sous-buts selon les priorités comparées de τ_i et τ_r. Le sous-but de gauche se consacre aux tâches de plus faible priorité ($i > r$). Elle est conduite en décomposant l'espace d'état en trois cas (4 à 6), selon le placement de la période de τ_i par rapport à celle de τ_r (voir Tableau 3.2, page 64). La complétude de cette décomposition n'est pas justifiée par les auteurs.
Le deuxième sous-but traite des tâches de priorité supérieure ($i < r$). Les auteurs indiquent que le seul cas pertinent est celui où une tâche de priorité supérieure est repoussée par la réexécution de τ_r (Cas 7).
Cette fois encore, nous avons dû stopper le développement de l'arbre au niveau des cas de preuve, car le discours informel est peu clair. De plus, on retrouve un problème majeur déjà identifié dans la preuve de $[S_2]$. L'analyse de chaque cas n'est faite que pour deux tâches : elle est centrée sur les interactions directes entre τ_r et une autre tâche, en termes d'utilisation du *slack* et de repoussement de tâche. En ignorant des cas plus complexes, les auteurs négligent des chaînes d'interactions indirectes : par exemple une tâche de priorité inférieure repoussée par la réexécution peut ensuite être préemptée par une tâche de priorité supérieure, et être donc encore plus retardée. C'est précisément le cas qui avait été mis en évidence dans la Figure 3.3.

3.3.3 Bilan des faiblesses identifiées

La restructuration du discours informel s'est avérée utile pour reprendre le raisonnement des auteurs pas à pas, et se poser des questions à chaque étape d'inférence. Une première lecture attentive de la preuve ne nous avait pas permis d'en obtenir une vue claire. Grâce à la réécriture en déduction naturelle, nous pouvons maintenant établir un jugement sur sa rigueur.

Comme nous l'avons vu, la plupart des feuilles de l'arbre ont une évaluation $\perp$ ou ?. Parmi celles-ci, on peut citer :

- des preuves de complétude liées à des décompositions en cas dans les preuves de $[S_2]$ et $[S_3]$;
- des preuves d'introduction de lemmes comme conditions suffisantes, dans la preuve générale du FT-RMS ainsi que dans celles de $[S_1]$ et $[S_2]$;
- des feuilles correspondant aux Cas 1 à 7 dont la formulation est trop imprécise pour que nous puissions les réécrire sous forme d'arbre de preuve.

Ceci nous amène à penser que la rigueur de la preuve n'est pas suffisante pour permettre d'assurer la validité de l'algorithme FT-RMS.

Il faut souligner que nous n'avons pas mis en évidence des problèmes triviaux de logique, tout comme nous n'avons pas déterminé de partie spécifiquement faible dans la preuve. Nous avons plutôt mis en évidence des problèmes de haut niveau, affectant tout le raisonnement, et donc l'ensemble de l'arbre de preuve :

- les auteurs n'ont pas su établir une distinction claire entre l'IBRMS (de nature entièrement statique) et le FT-RMS (ayant une nature dynamique, liée à l'utilisation du *slack*) ;
- l'ensemble des cas utilisés dans les preuves se limitent à étudier des paires d'instances de tâches, l'instance de τ_r affectée par la faute, et une autre instance de tâche qui interfère avec elle.

Ce dernier point a amené les auteurs à ne pas considérer des schémas d'interactions multiples, qui sont pourtant la principale difficulté de l'analyse d'ordonnançabilité.

De tels problèmes de haut niveau donnent malheureusement peu d'informations constructives pour le test. Il y a néanmoins une information que nous pouvons extraire de la preuve : la définition des cas fonctionnels (Cas 1 à Cas 7) considérés par les auteurs comme représentatifs du comportement de l'algorithme. Nous allons maintenant étudier la pertinence de ces cas pour guider le test.

3.4 Étape 3 : Test Guidé par la Preuve

D'après l'analyse précédente, on peut s'attendre à ce qu'un critère de test basé sur ces sept cas fonctionnels soit imparfait. Nous voulons cependant étudier dans quelle mesure ces cas peuvent être reliés au comportement incorrect de l'algorithme

(deuxième version), et si leur imperfection peut être compensée par le test statistique, en activant chacun de ces cas plusieurs fois avec des entrées de test différentes [Thévenod-Fosse et al. 1995]. La conception du test selon cette approche est présentée au paragraphe 3.4.1.
L'analyse de la preuve s'étant avérée donner peu d'information pour le test, nous avons jugé important de comparer cette approche avec d'autres approches plus classiques, structurelle et fonctionnelle (§ 3.4.2).
Les résultats expérimentaux de toutes ces approches sont commentés dans le paragraphe 3.4.3.

3.4.1 Critère de test basé sur les cas de preuve

La définition des cas de preuve introduits par les auteurs est donnée dans le Tableau 3.2. Les cas que nous avons finalement retenus pour être couverts lors des expériences de test sont présentés dans le Tableau 3.3. Il y a de légères différences entre les cas expérimentaux que nous retenons et les cas fonctionnels d'origine.
Le Cas 1 a été séparé en deux sous-cas pour étudier de façon séparée la préemption de l'exécution (Cas 1a) ou de la réexécution (Cas 1b) de τ_r par une tâche de priorité supérieure.
Certains cas se sont également avérés équivalents, ils ont donc été regroupés comme un cas unique à couvrir. L'équivalence est évidente pour les Cas 2 et 7. Dans le Cas 6, si la période d'une tâche chevauche celle de τ_r et est *ready* avant elle, alors sa prochaine instance chevauchera nécessairement τ_r et sera *ready* après elle, ce qui correspond au Cas 4. Les Cas 4 et 6 avaient été traités séparément dans la preuve informelle de $[S_3]$ car celle-ci ne considère qu'une seule instance pour chaque tâche, alors que ces deux cas représentent en fait une même tâche dont on considère deux instances successives.
D'après notre analyse, la complétude de la décomposition en cas est douteuse dans les preuves de $[S_2]$ et $[S_3]$. Nous avons donc ajouté deux cas supplémentaires : *autre sup* et *autre inf*. Ils correspondent à la présence éventuelle de tâches de priorité supérieure ou inférieure à τ_r, et qui n'appartiennent à aucun des cas de la démonstration informelle.

La commande de ces différents cas expérimentaux est problématique : nous ne savons pas exhiber *a priori* une fonction de génération exacte pour chacun des sous-domaines associés (pour *autre sup* et *autre inf*, nous ne savons d'ailleurs pas si ces sous-domaines sont non-vides). Nous retiendrons donc une approche empirique de recherche de distribution : partant d'une distribution initiale, par exemple la distribution aléatoire aveugle définie à l'issue de l'analyse préliminaire, nous mesurerons la couverture obtenue et essaierons de l'améliorer par essais successifs, si nécessaire (tout au moins en ce qui concerne la couverture des cas identifiés par les auteurs).

Pour obtenir des mesures de couverture des cas fonctionnels, nous avons dû instrumenter le prototype correspondant à la deuxième version du FT-RMS. L'oracle de test, initialement basé sur des vérifications d'échéances de tâches, se voit donc enrichi par l'observation des cas issus de la preuve informelle (Cas 1a à 5) ainsi que par la vérification de leur complétude (cas *autre sup* et *autre inf*).

TAB. 3.2 – Cas extraits de la démonstration informelle dans [Ghosh et al. 1998]

τ_r est la tâche en réexécution,
τ_h est une tâche de *priorité supérieure*, τ_l est une tâche de *priorité inférieure*
R_i et D_i sont l'instant d'arrivée de la requête (*ready time*) et la deadline de τ_i
E_r est l'instant correspondant à la fin de l'exécution de τ_r,
c'est à dire l'instant où une erreur est détectée.

Cas de preuve	**Sous-Buts**	**Priorité τ_i Vs τ_r**	**Description des Cas**	
1	$[S_2]$	sup	τ_h préempte τ_r	$D_h \leq D_r$ & $R_r \leq R_h <$ fin de réexécution
2	$[S_2]$	sup	τ_h est retardée par la réexécution de τ_r	$D_h > D_r$ & $E_r \leq R_h <$ fin de réexécution
3	$[S_2]$	sup	τ_h préempte l'exécution de τ_r	$D_h > D_r$ & $R_r \leq R_h < E_r$
4	$[S_3]$	inf	les périodes de τ_r et τ_l se chevauchent et τ_r est ready avant τ_l	$R_r < R_l$ & $D_r < D_l$
5	$[S_3]$	inf	la période de τ_r est entièrement contenue dans celle de τ_l	$R_r \geq R_l$ & $D_r \leq D_l$
6	$[S_3]$	inf	les périodes de τ_r et τ_l se chevauchent et τ_r est ready après τ_l	$R_r > R_l$ & $D_r > D_l$
7	$[S_3]$	sup	la réexécution de τ_r préempte τ_h	$D_h > D_r$ & $E_r \leq R_h <$ fin de réexécution

TAB. 3.3 – Cas expérimentaux

Cas expérimentaux	**Cas de la preuve**
1a	1
1b	1
2(7)	2 et 7
3	3
autre sup	ni 1(a ou b), ni 2(7), ni 3
4(6)	4 et 6
5	5
autre inf	ni 4(6), ni 5

3.4.2 Autres critères de test

Une approche fonctionnelle plus classique que la couverture des cas de preuve consiste à chercher à générer des pires cas de l'algorithme. En particulier, on peut biaiser la distribution d'entrée pour placer la faute après l'instant critique ($t = 0$) : c'est là que les tâches ont le temps de réponse le plus long en l'absence de faute, et on peut penser qu'une réexécution de tâche s'avérera plus problématique qu'à d'autres dates. Nous avons donc conçu deux variantes de la distribution aléatoire aveugle. Les ensembles de tâches sont générés comme précédemment, mais le placement de la faute est maintenant contraint :

- la distribution `d0` place toujours la faute sur la première instance de la tâche de C_i maximum,
- la distribution `d1` conserve la faute sur la première instance mais choisit la tâche aléatoirement.

Notons que ces distributions excluent de nombreux scénarios de faute, sans justification rigoureuse. En fait, [de A. Lima & Burns 2001] ont montré que la réexécution de la tâche la plus longue après l'instant critique (distribution `d0`) est insuffisante pour déterminer les pires cas, lorsque l'algorithme incorpore une règle d'ajustement dynamique des priorités (ce qui est le cas de la deuxième version du FT-RMS). Nous retenons néanmoins ces distributions à titre de comparaison, car elles nous paraissent représentatives d'un raisonnement basé sur l'intuition. Il sera intéressant de comparer leur efficacité avec celle du test guidé par la preuve, et d'étudier leurs liens éventuels avec les cas fonctionnels extraits de l'analyse de la preuve.

La dernière approche que nous envisageons est un test structurel des branches du prototype de l'algorithme (la couverture de tous les chemins est irréaliste du fait de boucles imbriquées, sur le temps et sur les indices de tâches). Comme pour le test guidé par la preuve, on partira de la distribution aléatoire aveugle, et on cherchera à améliorer la couverture structurelle si nécessaire. Nous utiliserons l'utilitaire Unix `tcov` pour évaluer la couverture des branches fournie par un jeu de test. A titre de comparaison, nous évaluerons également la couverture structurelle fournie par `d0` et `d1`.

3.4.3 Résultats expérimentaux

Le Tableau 3.4 compare l'efficacité des trois distributions retenues à ce stade, `d0`, `d1`, et aveugle, pour provoquer un comportement incorrect de la deuxième version du FT-RMS. Le nombre d'activations des différents cas de la preuve informelle pour les jeux de test correspondants est donné dans le Tableau 3.5. La couverture structurelle est satisfaisante pour les trois distributions : 100% des branches sont activées par chacun des jeux, avec un nombre minimum d'activations de 100 pour le jeu `d0`, 347 pour le jeu `d1`, et 2242 pour le jeu aléatoire aveugle (le plus long).

A la fin de l'analyse préliminaire, nous avions déjà remarqué que la distribution aléatoire aveugle s'avérait inefficace pour révéler la faute de conception dans la

TAB. 3.4 – Efficacité comparative des trois jeux de test générés

Distribution d'entrée	**Nombre d'entrées**	**Défaillances (FT-RMS, V2)**	**Taux de défaillance**
d0	100	5	5%
d1	1000	35	3,5%
aveugle	10000	4	0,04%

TAB. 3.5 – Nombre d'entrées de test activant les cas fonctionnels

Cas	**Jeu** `d0` (100 entrées)	**Jeu** `d1` (1000 entrées)	**Jeu** `aveugle` (10000 entrées)
1a	98	762	5284
1b	38	209	2563
2 (7)	79	251	1825
3	5	40	863
4 (6)	0	0	6089
5	38	761	4631
autre sup	0	0	3540
autre inf	0	0	0

deuxième version du FT-RMS. Ces mauvais résultats ne peuvent être expliqués ni par une couverture structurelle insuffisante (en termes de couverture de branches), ni par une couverture insuffisante des cas fonctionnels extraits de la preuve. Les deux critères de couverture précédents ne donnent donc pas d'information sur comment améliorer la distribution aveugle.

Les vérifications d'oracle que nous avons ajoutées pour observer la complétude des décompositions en cas sont plus instructives. La condition *autre sup* est activée 3540 fois par le jeu aléatoire aveugle : la décomposition en Cas 1,2,3 dans la preuve de $[S_2]$ n'est donc pas complète. La preuve a omis de considérer certaines configurations de paires de tâches (τ_r, τ_h).
Par contre, le cas *autre inf* n'est jamais activé quel que soit le jeu de test. Ceci nous encourage à penser que la décomposition en Cas 4,5,6 dans la preuve de $[S_3]$ pourrait être complète.

On pourra noter que la distribution aveugle couvre deux cas qui ne sont jamais activés par les autres distributions : *autre sup*, dont nous venons de discuter, et le Cas 4(6). Ces deux cas correspondent à des configurations qui ont une probabilité d'occurrence nulle sous `d0` et `d1`, du fait du placement de la faute à l'instant critique :

- Le cas *autre sup* correspond à une tâche de priorité supérieure dont aucune instance n'interagit directement avec l'exécution ou la réexécution τ_r. Or, à

l'instant critique, toutes les tâches de priorité supérieure préemptent l'exécution de τ_r.

- Le Cas 4(6) requiert un chevauchement des périodes, ce qui est impossible à l'instant critique. La période T_r est entièrement contenue dans celle de toute tâche de priorité inférieure, qui est donc dans le Cas 5.

Malgré ces lacunes, les distributions d0 et d1 s'avèrent significativement plus efficaces que l'aléatoire aveugle : le taux de défaillance observé est supérieur de plusieurs ordres de grandeur. La notion intuitive de faute à l'instant critique, bien que non fondée sur une analyse pire cas rigoureuse, a été plus pertinente que la considération des cas de preuve pour améliorer la distribution aveugle.

Dans le but d'affiner notre jugement sur la pertinence de ces cas de preuve, nous avons recherché d'éventuelles corrélations entre la couverture de cas spécifiques, et l'activation d'un comportement incorrect. A titre d'exemple, les Tableaux 3.6 et 3.7 présentent les combinaisons de cas couvertes lors des scénarios de défaillance trouvés sous l'aléatoire aveugle, et sous la distribution d0.

TAB. 3.6 – Couverture des cas pour les 4 défaillances de l'aléatoire aveugle

Jeu de test aveugle	Cas couverts
entrée 1083	1a, 4(6)
entrée 1733	4(6), *autre sup*
entrée 3762	1a, 5
entrée 9095	1a, 5

TAB. 3.7 – Couverture des cas pour les 5 défaillances de d0

Jeu de test d0	Cas couverts
entrée 20	1a, 1b, 5
entrée 60	1a, 1b, 5
entrée 71	1a, 2(7), 3, 5
entrée 73	1a, 1b, 5
entrée 96	1a, 1b, 2(7), 5

On observe que chaque cas se retrouve dans au moins un scénario de défaillance. La combinaison des Cas 1a et 5 est souvent observée mais elle n'est ni suffisante ni nécessaire, comme le montre l'entrée 1733 du jeu de test aléatoire aveugle. Cette configuration est favorisée par le placement de la faute à l'instant critique, mais

il est difficile, à ce stade, de conclure qu'elle constitue un facteur contribuant aux défaillances observées.

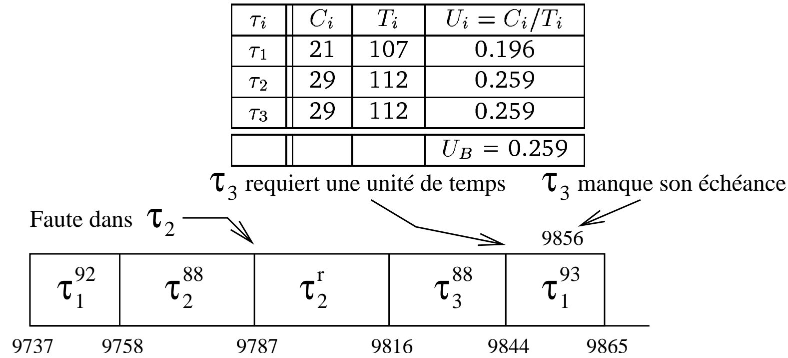
Après une analyse plus détaillée de chaque scénario de défaillance, notre conclusion est qu'il n'y a en fait aucune corrélation évidente entre des combinaisons de cas, et le comportement incorrect de l'algorithme. Le problème vient de ce que ces cas ont été issus d'un raisonnement sur des paires de tâches, alors que le comportement incorrect est lié à des schémas d'interaction plus complexes. Pour illustrer ce problème, nous détaillons ci-dessous un exemple issu de nos expériences. Il implique les Cas 1a et 5, mais la défaillance est due à l'existence d'une tâche qui ne joue aucun rôle dans la couverture de ces cas.

L'exemple est présenté dans la Figure 3.9. Une faute transitoire affecte l'instance 88 de τ_2. Deux des cas fonctionnels de la preuve informelle sont couverts par ce scénario :

- le Cas 1a, car τ_2^{88} est retardée par une tâche de priorité supérieure (τ_1^{92}) ;
- le Cas 5, car la période de τ_2^{88} est entièrement contenue dans celle d'une tâche de priorité inférieure (τ_3^{88}).

Les autres instances des tâches ne jouent aucun rôle dans la couverture des cas fonctionnels car ceux-ci se focalisent sur deux instances de tâches : l'instance à ré-exécuter et une instance d'une autre tâche interagissant avec celle-ci. En particulier, l'instance τ_1^{93} est ignorée car elle n'interagit pas directement avec l'exécution ou la réexécution de τ_2. Cependant, τ_1^{93} peut interférer avec l'exécution de tâches retardées par la réexécution de τ_2 (comme c'est le cas pour τ_3), et donc contribuer à la défaillance de l'algorithme.

Dans l'exemple présenté, une défaillance est observée car τ_3^{88} a été retardée par τ_1^{92}, puis par l'exécution et la réexécution de τ_2^{92}, et ne peut finalement plus tolérer d'être retardée encore de C_1 par τ_1^{93}.



τ_i	C_i	T_i	$U_i = C_i/T_i$
τ_1	21	107	0.196
τ_2	29	112	0.259
τ_3	29	112	0.259
			U_B = 0.259

FIG. 3.9 – Entrée 9095 du jeu de test aléatoire aveugle

Comme nos résultats de test le montrent, la probabilité d'obtenir un tel scénario est très faible si on se guide seulement sur les cas de preuve. Les interactions multiples

ayant été ignorées dans le raisonnement, les cas identifiés par les auteurs ne nous orientent pas vers elles.

3.5 Étape 4 : Retour sur la preuve

Le jeu de test aléatoire aveugle a permis de mettre en évidence que la décomposition en cas était incomplète dans la preuve de $[S_2]$. Cependant, il ne servirait à rien de reprendre cette preuve pour y ajouter le cas manquant. L'ensemble du raisonnement est erroné, du fait de la non considération des schémas d'interactions multiples, et de la mauvaise compréhension de la façon dont la récupération du *slack* modifie les interactions entre tâches. Toute la structure de la preuve serait à revoir. L'introduction des lemmes intermédiaires $[S_1]$, $[S_2]$, et $[S_3]$, ainsi que les décompositions en cas proposées, sont inadéquates pour construire une démonstration du FT-RMS. Nous avons vu que ces différents artefacts de preuve sont également inadéquats pour guider le test de cet algorithme.

Il faut noter que la diagnostic de la faute de conception, en vue d'une éventuelle correction dans une troisième version du FT-RMS, reste difficile. Ni l'analyse de la preuve, ni les résultats de test, ne nous permettent de caractériser précisément les configurations conduisant à défaillance. Une solution réside probablement dans une analyse rigoureuse du pire cas de temps de réponse. Ainsi que nous l'avons souligné à plusieurs reprises dans ce chapitre, une telle analyse est loin d'être évidente. On trouvera néanmoins dans [de A. Lima & Burns 2001] une approche qui nous semble pouvoir être adaptable au FT-RMS.

3.6 Conclusion

Cette première expérimentation du Test Guidé par la Preuve a fourni des résultats mitigés.
D'une part, la restructuration du discours informel sous forme d'arbre de preuve s'est avérée un support adéquat pour l'étude systématique de la démonstration. Nous avons utilisé la déduction naturelle de Gentzen sous forme de séquents, et avons obtenu une représentation claire de la structure de la preuve, permettant son analyse pas à pas. L'analyse a conduit à l'identification de défauts majeurs de raisonnement, se répercutant sur l'ensemble de l'arbre. Ainsi, bien que l'approche de restructuration soit beaucoup plus «légère» qu'une formalisation complète du problème, et ne puisse permettre de conclure sur la correction de l'algorithme, elle s'est avérée suffisante pour une évaluation rapide de la rigueur de la preuve informelle.
Si nous avons pu apprécier l'efficacité de la restructuration pour analyser la preuve, les informations que nous en avons retirées pour la conception du test se sont par contre avérées décevantes. Les défauts de raisonnement identifiés étaient de trop haut niveau pour permettre de guider le test. La seule information constructive que nous avons pu

extraire concernait des cas fonctionnels, considérés par les auteurs comme pertinents pour établir la correction de leur algorithme. Malheureusement, nos résultats de test ont montré qu'il n'y a pas de corrélation entre la couverture de ces cas et l'obtention de défaillances de l'algorithme.

Pour tempérer la remarque précédente, il convient de souligner que les algorithmes d'ordonnancement dynamiques sont connus pour être difficiles à tester. En particulier, le test structurel s'est lui aussi avéré inefficace pour le FT-RMS. Ceci ne suffit cependant pas à expliquer l'inadéquation des cas fonctionnels distingués par les auteurs. Les bons résultats des distributions `d0` et `d1` montrent qu'il n'y avait pas d'impossibilité à identifier des cas plus pertinents par un raisonnement intuitif.

Nous sommes donc amenés à la conclusion suivante : la preuve informelle du FT-RMS est simplement trop peu rigoureuse pour apporter des informations pertinentes pour le test. Lorsque l'évaluation de la preuve conduit à l'identification de problèmes majeurs, comme ici, nous conseillons de ne pas poursuivre la méthode de Test Guidé par la Preuve. La décision doit être prise à l'issue de l'étape de restructuration : une lecture attentive du discours informel s'avérera généralement insuffisante pour décider si une preuve existante peut être utile, ou non, pour guider le test.

D'après notre expérience, les critères pouvant déterminer l'arrêt de la méthode sont les suivants :

- La restructuration fait apparaître des problèmes de haut niveau, mettant en cause l'ensemble du raisonnement. Dans l'exemple du FT-RMS, tout le raisonnement est sous-tendu par l'idée incorrecte que l'analyse des interactions directes entre paires de tâches $(\tau_r, -)$ suffit pour démontrer l'ordonnançabilité de l'ensemble.
- Certaines notions fondamentales utilisées dans la preuve restent imprécises. Dans l'exemple traité, on a un glissement sémantique de la notion de *slack*, qui dénote à la fois le temps processeur réservé en IBRMS, et le temps processeur récupérable dynamiquement en FT-RMS.
- Le développement de l'arbre est bloqué sur des pans entiers de preuve. Dans l'exemple du FT-RMS, le fait que certaines parties de la preuve n'aient pas pu être réécrites sous la forme de règles de la déduction naturelle (Cas 1 à 7) est un bon indicateur de définitions imprécises et de raccourcis douteux dans le raisonnement informel.

Clairement, la pertinence du Test Guidé par la Preuve doit maintenant être évaluée sur des exemples de preuves informelles plus rigoureuses. L'étude de cas décrite au chapitre suivant fournit un tel exemple.

Chapitre 4

Deuxième cas d'étude d'une preuve informelle : GMP

Dans un système réparti, un protocole d'appartenance de groupe (*Group Membership Protocol*, ou *GMP*) permet d'obtenir un accord sur l'identité des processeurs non-défaillants ; les processeurs défaillants sont exclus du groupe. Un GMP est un service de base, sur lequel peuvent s'appuyer des politiques de tolérance aux fautes par réplication des données et des processus sur des processeurs différents.

L'appartenance de groupe s'apparente au problème de *consensus réparti*, dont le but est d'obtenir une décision commune à tous les processeurs. Dans le cas de systèmes asynchrones, pour lesquels on ne fait aucune hypothèse sur les temps de calcul et de communication, il a été établi que le problème du consensus est impossible à résoudre de façon déterministe en présence de fautes [Fischer et al. 1985]. Ce résultat d'impossibilité s'applique aussi à l'appartenance de groupe [Chandra et al. 1996]. Il faut alors soit affaiblir le problème, soit renforcer les hypothèses sur l'environnement (voir par exemple [Cristian & Schmuck 1995, Lin & Hadzilacos 1999, Babaoglu et al. 2001]).

Le GMP que nous avons choisi comme deuxième étude de cas est destiné à des systèmes distribués synchrones, avec des hypothèses additionnelles sur les fautes à tolérer. Ces différentes hypothèses sont justifiées par la considération d'une architecture particulière, la TTA (*Time-Triggered Architecture*), que nous présenterons brièvement par la suite. Sous ces hypothèses restrictives, le GMP proposé vise à offrir des propriétés très fortes : avec un minimum d'échanges d'information, tout processeur défaillant doit être rapidement exclu par les autres processeurs non-défaillants, et doit en outre être capable de s'exclure lui-même du groupe.

La preuve informelle de ce GMP est beaucoup plus aboutie que la preuve étudiée au chapitre précédent. Pour consolider leur démonstration «papier» du cas général, les auteurs ont analysé formellement une instance du problème à quatre processeurs,

par *model-checking*. Malgré cela, l'algorithme s'est avéré incorrect quelques mois après publication, et n'a jamais été intégré à l'architecture TTA.

Nos travaux sur cet exemple ont d'abord fait l'objet d'un rapport de recherche [Lussier & Waeselynck 2004c], puis d'une publication internationale [Lussier et al. 2004]. Pour les présenter, nous reprenons le même canevas que dans le chapitre précédent. Un premier paragraphe introduit le cas d'étude et son contexte (notamment la TTA). Puis les quatre paragraphes suivants décrivent l'application des étapes successives de notre méthode : analyse préliminaire, restructuration de la preuve informelle sous forme d'arbre, test guidé par les faiblesses identifiées, et enfin retour sur la preuve.

4.1 Présentation du Group Membership Protocol étudié

4.1.1 Contexte

La *Time-Triggered Architecture* (TTA) [Scheidler et al. 1997] est une architecture pour les systèmes distribués temps-réel [Kopetz 1995]. Cette architecture a été développée au cours des vingt dernières années à l'université de Vienne et est maintenant commercialisée par TTTech. Elle a fait l'objet de deux projets européens : "Time-Triggered Architecture" (Esprit OMI program) et "X-by-Wire" (Brite EuRam program). Un de ses composants clé est le *Time-Triggered Protocol* (TTP), un protocole devant gérer la couche de communication d'une architecture TTA [Kopetz & Grünsteidl 1994]. Le TTP intègre un certain nombre de services requis pour les systèmes temps-réel tolérants aux fautes, tels que la synchronisation d'horloges, la diffusion fiable de messages, et l'appartenance de groupe. Ses domaines d'applications sont typiquement les fonctions embarquées critiques dans l'automobile ou l'avionique. Par exemple, dans l'A380, le système de contrôle de la pression en cabine sera basé la technologie TTP.

L'algorithme étudié est une variante du protocole d'appartenance de groupe du TTP. Cette variante, proposée dans [Katz et al. 1997], vise des applications à des systèmes très contraints en termes de bande passante réseau. Elle cherche à minimiser l'impact de la tolérance aux fautes sur la taille des messages émis : le coût de ce GMP est seulement de un bit par trame émise sur le bus. Pour établir certaines propriétés fondamentales de l'algorithme, les auteurs ont développé une preuve informelle par induction sur le temps. Cette preuve a été consolidée par l'analyse formelle d'une instance à quatre processeurs. Des techniques de *model-checking* ont été utilisées pour mettre au point l'algorithme, et aider à établir des invariants pour la preuve informelle. Cependant, l'analyse a échoué à mettre en évidence une faute de conception résiduelle, qui se manifeste lorsqu'il ne reste plus que trois processeurs non-défaillants[1].

[1] voir la version corrigée de [Katz et al. 1997] sur le site des auteurs `http://www.csl.sri.com/papers/wdag97`. Cette faute a également été trouvée de façon indépendante par [Creese & Roscoe 1999].

4.1.2 Environnement de l'algorithme et modèle de fautes

L'intégration au sein du TTP permet des abstractions simplificatrices, sur lesquelles reposent les hypothèses de l'algorithme. On suppose ainsi que le système est composé d'un anneau de n processeurs, numérotés de *0* à *n-1*, et attachés à un bus d'émission commun. Physiquement, l'architecture comporte en fait plusieurs bus en redondance, mais ceci est transparent au niveau du GMP. De plus, le service de synchronisation d'horloges permet de faire l'hypothèse d'un temps global commun à tous les processeurs. On dispose d'un certain nombre de notions conceptuelles liées au temps et à l'exécution de l'algorithme dans le temps :

- Un pas d'horloge (*step*) correspond à une incrémentation du temps global t. Chaque pas d'horloge déclenche une exécution synchrone du GMP par tous les processeurs.
- L'accès au bus est géré par un schéma d'ordonnancement statique selon lequel sont assignées, à chaque processeur, des dates auxquelles il peut émettre (*slots*), les autres processeurs étant alors en mode réception. A la date t, seul le processeur t modulo n peut émettre[2]. Ce processeur reste cependant silencieux s'il s'est diagnostiqué comme défaillant (c'est-à-dire, s'il considère qu'il ne fait plus partie du groupe).
- Un *round* est un cycle de n pas d'horloge consécutifs pendant lesquels chaque processeur est passé par son *slot*.

L'objectif de l'algorithme est de maintenir des vues locales cohérentes du groupe des processeurs non-défaillants. Du fait de la présence d'autres mécanismes de tolérance aux fautes, les seules manifestation de fautes prises en compte au niveau du GMP se ramènent à deux types :

- faute d'émission (ou *send fault*). Un processeur échoue à émettre lors de son *slot* d'émission. Aucun autre processeur ne recevra de message à ce *slot*.
- faute de réception (ou *receive fault*). Un processeur échoue à recevoir un message émis.

Les fautes peuvent être intermittentes : à partir du moment où un processeur devient défaillant, il peut fonctionner correctement à certaines dates et manifester sa faute à d'autres. Il n'y a pas de réparation. Les processeurs restent défaillants et doivent être exclus du groupe pour ne plus interférer sur le fonctionnement du système.
On suppose que l'occurrence de fautes est suffisamment rare pour garantir les deux hypothèses supplémentaires suivantes :

- durant $n+1$ pas d'horloge consécutifs, un seul processeur non-défaillant peut devenir défaillant ;
- il reste toujours au moins deux processeurs non-défaillants dans le système.

[2]On suppose qu'il est physiquement impossible, pour un processeur dont ce n'est pas le slot, d'émettre sur le bus. Dans la TTA, ceci est garanti par des mécanismes bas-niveau, les *bus guardians*.

4.1.3 L'algorithme du GMP

Chaque processeur maintient deux variables locales :

- *membership*, qui représente sa liste des processeurs du groupe,
- *ack*, qui est une variable Booléenne représentant son avis sur l'émission la plus récente, d'un processeur qu'il estime valide, au sein du groupe.

L'algorithme indique comment mettre à jour ces variables à chaque pas d'horloge, selon que le processeur est en mode émission ou en mode réception.

L'émetteur b, s'il considère qu'il fait partie du groupe, diffuse un message contenant des données (liées à sa fonction) et un bit spécifique représentant la valeur courante de son *ack*. Il laisse alors son *membership* inchangé, et met à jour son *ack* à vrai : n'ayant aucun moyen direct de constater une faute d'émission, il considère par défaut que son émission est correcte.

Un processeur en réception p n'écoutera b que s'il considère que b fait partie du groupe, c'est à dire que b est dans le *membership* de p : on dit que b est un émetteur attendu par p. Dans ce cas, p mettra à jour ses variables locales selon qu'il ait reçu – ou pas – un message, et selon son accord avec le bit *ack* émis par b. Conceptuellement, le comportement de p peut donc être vu comme déterminé par deux entrées Booléennes :

- *arrived(b, p)*, qui est vraie si et seulement si p reçoit un message émis par b et que b est attendu à cette date.
- *ack(b)*, qui est vraie si et seulement si le bit émis par b est à un.

L'algorithme, tel que donné par les auteurs, est reproduit dans la Définition 4.1 . On constatera que le comportement en mode réception est plus complexe qu'en mode émission. En paraphrasant l'algorithme, p se retire de son *membership* si : (a) il ne reçoit pas une émission attendue alors qu'il n'a pas accepté la précédente , ou si (b) il n'a pas accepté l'émission précédente alors que b l'a acceptée. De plus, p retire b de son *membership* si : (c) aucune émission n'est reçue, ou si (d) p a accepté l'émission précédente alors que b l'a rejetée.
La variable locale *ack(p)* sera mise à vrai lorsque p reçoit l'émission attendue, et que soit l'émetteur a accepté l'émission précédente, soit p l'a rejetée.

Une idée plus intuitive du fonctionnement du GMP est donnée dans la Figure 4.1, qui illustre son principe par deux scénarios. Les deux scénarios commencent dans un état *stable* où tous les processeurs non-défaillants ont le même *membership* (noté *mem* en raccourci) qui les contient tous et uniquement eux. De plus, les processeurs ayant manifesté une faute dans le passé se sont déjà tous diagnostiqués comme défaillants. A ce moment, un nouveau processeur p devient défaillant, alors qu'au moins deux autres processeurs q_i restent non-défaillants.

Dans le premier scénario, p manifeste une faute d'émission. Aucun des q_i ne reçoit l'émission, p est retiré de leur *membership* selon la condition (c), et leur *ack* est mis à faux. Comme un émetteur considère toujours que son émission a été correcte, p conserve son *membership* inchangé et met son *ack* à vrai. Notons que p et les q_i sont toujours en accord sur l'identité du prochain émetteur attendu, soit q_1. Un *ack* faux

Définition 4.1 — Description du GMP (reproduite d'après [Katz et al. 1997])

Pour chaque processeur p, si l'émetteur b n'est pas dans le *membership* de p, aucune variable locale n'est modifiée.
Si p est l'émetteur b, et qu'il est dans son propre *membership*, alors il émet *ack(b)* et ensuite met à jour son *ack(b)* interne à vrai.
Sinon, si p n'est pas l'émetteur b, chaque variable de p est mise à jour comme suit (notons que *ack(p)* est une variable interne à p et que *ack(b)* est fourni par l'émission reçue de b) :

- *membership* : reste le même sauf éventuellement pour p et b
  - p est exclu dans deux cas :
    (a) $(\neg arrived(b,p)) \land \neg ack(p)$, ou
    (b) $arrived(b,p) \land ack(b) \land \neg ack(p)$.
  - b est exclu dans deux cas :
    (c) $\neg arrived(b,p)$, ou
    (d) $ack(p) \land \neg ack(b)$
- *ack* : prend la valeur de $arrived(b,p) \land (ack(b) \lor \neg ack(p))$.

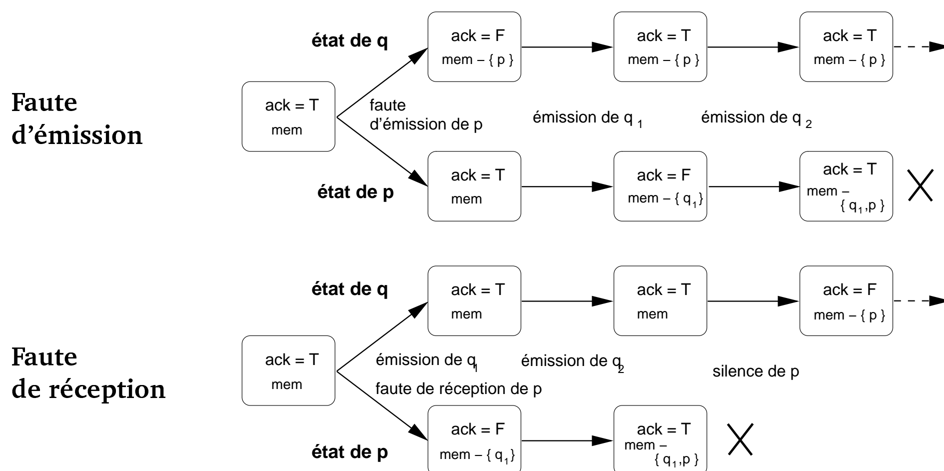


FIG. 4.1 – Scénarios de fonctionnement du GMP

est émis, amenant les autres processeurs non défaillants à constater qu'ils avaient précédemment pris la même décision que q_1 (exclure p). Leur *ack* est donc mis à jour à vrai. Le processeur p est en désaccord avec q_1 et l'exclut de son *membership*. Après un deuxième désaccord lors du *slot* de q_2, p diagnostiquera sa faute et s'éliminera de son *membership* selon la condition (b) de l'algorithme.

Dans le second scénario, p manifeste une faute de réception et retire q_1 de son *membership* selon (c). Au slot attendu suivant, il constate que l'émetteur q_2 n'a pas exclu q_1. Selon (b), p va alors diagnostiquer sa faute et se retirer de son *membership*.

Lorsque son propre *slot* sera atteint, il restera silencieux de façon à informer les autres processeurs du fait qu'il ne fait plus partie du groupe.

4.1.4 Propriétés attendues

L'algorithme précédent est censé assurer le respect de trois propriétés fondamentales, qui conduisent aux trois théorèmes de la preuve informelle du GMP :

- *Théorème 1* : Les *memberships* locaux de tous les processeurs non-défaillants sont toujours identiques et contiennent tous les processeurs non-défaillants.
- *Théorème 2* : Un processeur défaillant est retiré des *memberships* des processeurs non-défaillants dans le pas d'horloge suivant sa première émission en tant que processeur défaillant.
- *Théorème 3* : Un processeur devenu défaillant se retirera de son propre *membership* (et ainsi s'auto-diagnostiquera comme défaillant) une fois que les *slots* de, au plus, deux processeurs non-défaillants auront été passés.

Il a en fait été montré, quelques mois après la publication de l'algorithme, que le Théorème 3 pouvait être violé. Nous allons maintenant étudier la pertinence du Test Guidé par la Preuve pour révéler cette faute de conception du GMP.

4.2 Étape 1 : Analyse Préliminaire

L'analyse préliminaire se base sur la présentation de l'algorithme, ses hypothèses et les propriétés à satisfaire, telles que données par les auteurs. Cette présentation est examinée avec un point de vue opérationnel : il s'agit d'être capable d'implanter un prototype de l'algorithme, avec son environnement de test, et de pouvoir exhiber une fonction de génération aléatoire sur le domaine d'entrée. Comme nous allons le voir, ceci nous a amené à devoir interpréter certaines parties du discours informel.

- *le prototype* :
  Nous avons développé un prototype en langage C, qui implémente les différentes règles de mise à jour définies dans l'algorithme (Définition 4.1).
  Notons que, s'il est facile de reproduire l'algorithme dans un langage de programmation, comprendre le comportement induit est loin d'être évident. Nous avons trouvé utile de jouer pas à pas quelques scénarios, tels que ceux de la Figure 4.1. Ces scénarios partent d'un état *stable* où tous les processeurs défaillants dans le passé ont été diagnostiqués, et montrent comment le système revient à *stable* après occurrence d'une nouvelle faute. Nous avons ensuite cherché à généraliser ces scénarios sous forme d'un schéma de type machine à états, représentant le fonctionnement séquentiel global. Ce schéma distingue trois classes de chemins de *stable* à *stable*, et est représentatif de notre compréhension du GMP à ce stade. Il nous a paru intéressant de le mentionner ici car, comme

nous le verrons ultérieurement, notre compréhension aura évolué à l'issue de la restructuration de la preuve.
Dans notre prototype, l'état initial est un état *stable* : aucun processeur n'a encore manifesté de faute, et les vues locales du *membership* contiennent tous les processeurs du système. De plus les variables *ack* sont toutes initialisées à vrai.

- *le domaine d'entrée* :
Nous avons choisi de définir une séquence de test par les éléments suivants :
  - Le nombre n de processeurs, avec $n > 2$.
  - La liste des fautes devant être simulées, chaque faute étant caractérisée par sa date et le processeur affecté.

  Le type de faute peut être déduit de ces informations : une faute d'émission aura lieu lorsqu'un processeur est affecté à une date correspondant à son *slot*, sinon il s'agira d'une faute de réception.
  Les fautes d'émission et de réception peuvent en fait relever de deux catégories de fautes, qui ne sont pas explicitement différenciées dans le discours informel, mais correspondent à des contraintes d'entrée différentes :
  - Les *fautes initiales* sont celles qui affectent des processeurs jusque là non-défaillants. Il ne peut pas y avoir plus de *n-2* fautes de cette catégorie, puisqu'il doit toujours rester au moins deux processeurs non-défaillants. Deux de ces fautes doivent être séparées d'au moins *n+1* pas d'horloge. Leur date doit correspondre au *slot* d'un processeur non-défaillant jusque là : ce sera soit le *slot* du processeur devenant défaillant (faute d'émission), soit le *slot* d'un autre processeur non-défaillant (faute de réception).
  - Les *fautes ultérieures* affectent des processeurs ayant déjà manifesté une faute initiale. Elles sont liées à la simulation du comportement intermittent des processeurs défaillants, qui peuvent manifester leur faute à certaines dates et fonctionner correctement à d'autres. Il est à noter que, dans nos séquences de test, les fautes ultérieures ne sont que potentielles : par exemple, une faute ultérieure d'émission ne sera effective que si le processeur tente d'émettre, ce qui ne sera pas le cas s'il s'est déjà diagnostiqué comme défaillant.
  Le discours informel ne précise pas si une faute ultérieure peut être d'un type différent de celui de la faute initiale. Par exemple, un processeur ayant manifesté une faute initiale d'émission pourra-t-il manifester des fautes de réception par la suite ? Nous avons choisi d'adopter l'interprétation la moins restrictive : après une faute initiale, un processeur peut arbitrairement être affecté par tout type de faute.

  En combinant fautes initiales et ultérieures, on peut simuler différentes configurations de faute, selon leur persistence : faute permanente, intermittente ou transitoire.
- *l'oracle* :
Celui-ci est basé sur les Théorèmes 1 à 3 qui constituent les propriétés attendues de l'algorithme.

Nous avons eu à interpréter la formulation du Théorème 3 : " une fois que les *slots* de, au plus, deux processeurs non-défaillants auront été passés ". Pour une faute de réception initiale, la date de la faute doit-elle compter comme le premier *slot* d'un processeur non-défaillant ? Selon la preuve informelle elle-même, la réponse semble être négative. Cette ambiguïté résolue, la vérification des trois propriétés peut être implémentée. Elle requiert l'observation des *memberships* de tous les processeurs à chaque pas d'horloge.
D'après les propriétés attendues de l'algorithme, toute faute initiale a été diagnostiquée (par le processeur affecté, et par les autres processeurs non-défaillants) avant qu'une nouvelle faute initiale puisse survenir après *n+1* pas d'horloge. On peut donc, sans perte de généralité, limiter la fenêtre temporelle d'observation à $[0...t_{lastf}+n+1]$, où t_{lastf} est la date de la dernière faute initiale de la séquence de test.

L'environnement de test complet comprend le prototype de l'algorithme, un moniteur de test qui simule sur ce prototype l'effet des fautes (par exemple, une faute de réception sur p sera simulée en forçant son entrée $arrived$ à faux), et l'oracle qui vérifie en-ligne les propriétés attendues.

Bien que finie, la fenêtre temporelle de nos expériences de test peut encore être arbitrairement grande. Nous avons donc dû rajouter des contraintes supplémentaires. Elles concernent le nombre total de processeurs dans le système, et la date d'occurrence de la première faute. Nous donnons ci-après une fonction de génération aléatoire aveugle correspondant à ces choix.

Le nombre n de processeurs est tiré uniformément dans l'intervalle $[3 \ldots 20]$. D'après [Katz et al. 1997], cette borne supérieure est représentative des types de systèmes auxquels le GMP est destiné. Puis, on tire uniformément le nombre total de *fautes initiales* entre 1 et $n-2$. La génération de chaque faute initiale procède alors comme suit.
Soit t_{min} la date au plus tôt de la faute à générer (pour la première faute, $t_{min}=0$, et pour les autres t_{min} dépend de la date de la faute précédente). On tire aléatoirement une date t dans l'intervalle $[t_{min} \ \ldots \ t_{min}+n-1]$. Si cette date correspond au *slot* d'un processeur non-défaillant, elle est retenue. Sinon, on prend $t_{min}=t+1$, et on itère le processus jusqu'à ce qu'une date soit retenue. Notons que cette distribution force l'occurrence de la première faute au premier *round* d'exécution de l'algorithme. D'après notre compréhension du GMP, ceci s'effectue sans perte de généralité : tant qu'aucun processeur n'est défaillant, l'état du système à la date t est le même qu'en $t+n$ pour tout t. Pour les autres fautes initiales, nous n'excluons *a priori* aucune date. Néanmoins, les dates les plus tardives ont une probabilité décroissante, si bien qu'en pratique l'écart entre deux fautes reste modéré.
Une fois la date de la faute générée, on tire uniformément le processeur affecté parmi l'ensemble des processeurs encore non-défaillants. Après cette faute initiale, le processeur affecté a 50% de chance de manifester une faute à chaque pas d'horloge. De cette façon toutes les configurations de *fautes ultérieures* sont rendues équiprobables.

Nous avons généré un échantillon de 10^4 séquences aléatoires selon cette distribution, et les avons exécutées dans l'environnement de test. Les résultats obtenus sont présentés dans le Tableau 4.2.

TAB. 4.1 – Résultats d'un test aveugle du GMP

Nombre de séquences de test	Taux de défaillance du Théorème 1	Taux de défaillance du Théorème 2	Taux de défaillance du Théorème 3
10^4	0%	0%	6,15%

Aucune défaillance des Théorèmes 1 et 2 n'est constatée. Par contre le Théorème 3 présente un taux de défaillance significatif, correspondant à la faute de conception connue de l'algorithme.

Cette faute de conception n'est donc pas très difficile à révéler avec le profil de test aveugle issu de notre analyse préliminaire. Pourtant, elle avait échappé aux auteurs de la preuve, ce qui montre l'intérêt du test en tant que méthode de vérification complémentaire.

Pour cet exemple, on pourrait considérer que le test en aveugle n'a pas besoin d'être amélioré par des critères de sélection. Néanmoins, nous avons poursuivi la méthode par la restructuration de la preuve informelle, à des fins d'évaluation expérimentale. Il s'agit de déterminer si l'identification des faiblesses de la preuve s'avère pertinente pour davantage cibler le comportement erroné de l'algorithme.

4.3 Étape 2 : Restructuration de la preuve

Après cette analyse préliminaire, nous nous concentrons sur la preuve informelle des trois propriétés attendues de l'algorithme. Nous donnons tout d'abord une vue globale de la structure de cette preuve (§ 4.3.1). Contrairement à ce qui s'était passé pour l'étude de cas du FT-RMS (voir chapitre précédent), nous avons pu pousser la restructuration à un niveau de détail assez fin : nous n'avons pas été bloqués par des parties obscures du discours informel. Pour des raisons de place, nous ne pouvons pas reproduire ici la totalité des 12 arbres de preuve résultant. Nous avons donc choisi de nous concentrer sur deux parties jugées moins convaincantes.
Une de ces parties, correspondant à la preuve du Théorème 3, sera présentée de façon très détaillée (§ 4.3.2). Elle nous permettra d'illustrer le principe de restructuration d'une preuve informelle, en mettant en correspondance, pas à pas, les extraits de texte informel et leur interprétation en termes d'étapes d'inférence.
La deuxième partie (§ 4.3.3), correspondant à la preuve d'un lemme intermédiaire, sera présentée de façon plus succincte. Nous expliquerons pourquoi nous l'avons jugée faible, et comment cela est apparu dans la construction de l'arbre. Cependant, nous

ne donnerons pas les détails de cette partie de l'arbre.
Nous terminerons par un bilan de toutes les faiblesses identifiées (§ 4.3.4), de façon à préparer l'étape suivante de test guidé par la preuve.

4.3.1 Aperçu global de la preuve informelle

Les trois théorèmes qui définissent les propriétés attendues du GMP sont prouvés successivement dans le discours informel.

Le Théorème 1 est prouvé par induction sur le temps, c'est à dire que l'on prouve d'abord la propriété vraie en $t = 0$ (preuve du cas de base), puis que si elle est vraie jusqu'à t, alors elle est vraie en $t + 1$ (étape d'induction). Pour établir l'étape d'induction, il faut évidemment que la propriété à prouver constitue un invariant inductif. Or, ce n'est pas le cas du Théorème 1 : il a donc fallu le renforcer par l'adjonction de conjoints jusqu'à ce qu'un invariant inductif soit obtenu. Cette opération de recherche d'invariant, par renforcements répétés de la propriété à prouver, a été conduite en s'appuyant sur des techniques de *model-checking*. L'invariant final pour prouver le Théorème 1 contient 8 conjoints : $C(1) \wedge C(2) \wedge \ldots \wedge C(8)$. Les deux premiers permettent de déduire la propriété initiale : C(1) les *memberships* locaux de tous les processeurs non-défaillants sont identiques et C(2) tout processeur non-défaillant est dans son *membership* local. Nous ne détaillons pas ici les 6 autres conjoints destinés à obtenir une propriété inductive (seul le Conjoint (5) sera explicité par la suite, car nous avons trouvé des faiblesses dans sa preuve). Au final la preuve du Théorème 1 implique les étapes suivantes :

- preuve que les 8 conjoints sont vrais en $t = 0$.
- pour chaque conjoint C(i), preuve que si $C(1) \ldots \wedge C(8)$ a été vrai jusqu'à t, alors C(i) est vrai en $t + 1$

Les preuves des Théorèmes 2 et 3 sont construites différemment, sur un schéma de preuve par cas. Elle posent cependant l'invariance des 8 conjoints en hypothèse : la preuve précédente constitue donc le fondement de l'ensemble de la preuve du GMP.

Notre restructuration de la preuve du GMP comporte 12 arbres : les 3 théorèmes, les 8 conjoints de l'invariant du Théorème 1 et un lemme additionnel nommé "restricted change lemma". Elle a été menée en utilisant le calcul des séquents. Au total, la restructuration nous a demandé environ une semaine de travail, bien moins que ce qui aurait été nécessaire pour une formalisation complète de l'algorithme (telle que celle que nous étudierons dans le Chapitre 5). Rappelons que la restructuration n'a pas pour ambition d'établir si l'algorithme est correct ou incorrect. Elle vise seulement à obtenir un jugement rapide sur la rigueur de la preuve, et à identifier d'éventuelles faiblesses à cibler durant le test.

Dans cette optique d'évaluation de la preuve, la restructuration s'est avérée efficace et plusieurs problèmes ont été identifiés.

Un problème récurrent, que nous avons trouvé dans toutes les parties de la preuve, concerne la non-prise en compte explicite de la notion de *fautes ultérieures*. Par

exemple, après une faute initiale de réception, l'argumentation suppose que le processeur défaillant va toujours recevoir, ou émettre, les messages suivants s'il le souhaite (sauf dans l'étape d'induction du Conjoint (6) où la possibilité de ne pas recevoir deux émissions consécutives est mentionnée). Ceci revient à ne considérer que des fautes transitoires, et à ignorer les configurations de fautes permanentes ou intermittentes. Cette omission n'étant pas justifiée par les auteurs, nous la considérons comme un défaut de la preuve. Nous nous sommes rapidement rendu compte qu'elle amènerait un grand nombre de branches à être désignées comme non-concluantes. Aussi avons-nous décidé de laisser ce problème de côté dans un premier temps, et de réaliser la construction et l'évaluation des arbres de preuve sous l'hypothèse de fautes transitoires. Sous cette hypothèse restrictive, plusieurs problèmes ont encore été mis en évidence :

- L'étape d'induction du conjoint (5) contient trois branches que nous avons jugées fausses.
- Deux branches ont été jugées non concluantes dans la preuve du Théorème 3.

Nous reviendrons sur le problème des *fautes ultérieures* dans le bilan de la restructuration (§ 4.3.4), et allons maintenant nous concentrer sur les deux derniers problèmes identifiés. Ces deux problèmes illustrent la difficulté de raisonner sur le temps. Pour le Conjoint (5) comme pour le Théorème 3, la démonstration considère des fenêtres temporelles comportant plusieurs pas d'horloge. L'approche de restructuration s'est avérée très utile pour mettre en correspondance les fragments du discours avec des dates, et analyser la validité des arguments pour ces dates.

En ce qui concerne le Théorème 3, les auteurs de [Katz et al. 1997] ont fait le diagnostic suivant dans la version corrigée de l'article (*c.f.* leur note de bas de page 5) : "Cet argument est incorrect lorsqu'il n'y a que 3 processeurs". Dans la construction de l'arbre, que nous allons présenter de façon très détaillée au paragraphe suivant, nous nous sommes efforcés de ne pas tenir compte de notre connaissance de ce diagnostic. De fait, les faiblesses que nous avons identifiées ne semblent pas spécifiquement liées à des configurations à trois processeurs (le lien deviendra cependant apparent quand nous revisiterons la preuve après les résultats du test, au paragraphe 4.5).

4.3.2 Preuve du Théorème 3

Rappelons tout d'abord le Théorème 3, qui correspond à une propriété d'auto-diagnostic.

Théorème 3 :

> " Un processeur devenu défaillant se retirera de son propre *membership* (et ainsi s'auto-diagnostiquera comme défaillant) une fois que les *slots* de, au plus, deux processeurs non-défaillants auront été passés. "

L'argument informel pour établir ce théorème est reproduit dans son intégralité dans la Figure 4.2. Nous avons conservé la formulation originelle en anglais, afin de ne pas

risquer d'en altérer le sens par une traduction subjective. Le discours est très dense, et chaque mot a son importance.

Nous allons maintenant expliquer pas à pas notre construction d'un arbre de preuve à partir de ce texte de référence. Après une vue générale des notations employées et de la structure globale de l'arbre (§ 4.3.2.1), nous en détaillons chacune des parties (§ 4.3.2.2 à 4.3.2.7). Dans ces parties détaillées, nous montrons la correspondance entre 1) des extraits du discours en formel, et 2) leur représentation en termes de séquents et de règles d'inférence en calcul des séquents. On verra comment les faiblesses de la preuves se dégagent de cette représentation.

" If a processor p becomes send-faulty, all nonfaulty processors will set their **ack** bits to *false* in the step following that processor's slot, since the slot is expected and no message is received. Similarly, if p just became receive-faulty in the expected broadcast before its slot, it will broadcast **ack** as *false*, while the nonfaulty processors have **ack** = *true*, and thus will set their **ack** bits to *false*. In either case, p will set its **ack** bit to *true* in the step after it broadcasts. Until its own or the previous broadcast, p was nonfaulty, and thus its local membership set agreed with all other nonfaulty processors. By the invariant of Theorem 1, no nonfaulty processor will remove itself due to the new fault, thus the next expected slot of the nonfaulty processors is the same as the next expected slot of the faulty one. By the fault arrival assumption, the next expected slot must be nonfaulty, since it cannot be newly faulty, and by the invariant it cannot be an undiagnosed old fault. Thus the newly faulty processor p will receive **ack** = *false* in the message from the next expected slot, disagree with the broadcaster, and set its own **ack** bit to *false*. Since all nonfaulty processors receive that broadcast and agree with its **ack** bit, p will receive **ack** as *true* in the expected slot after that (using the fact that there are at least two nonfaulty processors within the group). At that point, the faulty processor p will remove itself from its local membership set, using (b).
If a processor p becomes receive-faulty in its transition to the next step, but p is not the next expected broadcaster, it will remove the broadcaster from its local membership set, but otherwise has the same local membership sets and next expected slot as the nonfaulty ones. It will also set **ack**(p) to *false*. Again by the fault model, the next expected broadcaster must be nonfaulty, will broadcast **ack** as *true*, and the receive-faulty processor p will remove itself using (b). "

FIG. 4.2 – Preuve informelle complète du Théorème 3

4.3.2.1 Notations et structure globale de l'arbre

Les notations employées dans l'arbre sont récapitulées dans la Définition 4.2 , page 84. Elles permettent d'exprimer de façon concise des notions reprises du discours informel.

Les hypothèses utilisées dans la preuve sont notamment exprimées à l'aide des notations suivantes :

- *fault model'*, pour les hypothèses liées au modèle de fautes du GMP (comme l'écart temporel entre deux fautes *initiales*), renforcées avec l'hypothèse qu'aucune faute *ultérieure* ne suit une faute *initiale* (c'est à dire qu'on se limite à des fautes transitoires) ;
- *algo*, pour les règles de mise à jour de *ack* et *membership* reproduites au paragraphe 4.1 ;
- *Inv(Th1)*, pour l'invariance des 8 conjoints définis dans la preuve par induction du Théorème 1 ;
- *C(i)*, pour l'invariance d'un conjoint particulier *i*.

Parmi les autres notations de la Définition 4.2 , nous pouvons mentionner celles relatives à des dates. La restructuration nous a amené à distinguer explicitement cinq instants temporels apparaissant dans le discours informel :

- t_f est la date où une faute initiale affecte *p*,
- t_{ok} est le premier *slot* d'un processeur non-défaillant après t_f ($t_{ok} > t_f$)
- t_p est le premier *slot* d'émission de *p* après sa faute ($t_p = t_f$ pour une faute d'émission, $t_p > t_f$ pour une faute de réception,
- t_{ok1}, t_{ok2} sont les premier et second *slots* de processeurs non-défaillants après t_p ($t_{ok2} > t_{ok1} > t_p \geq t_f$).

Lors de la construction de l'arbre de preuve, nous utilisons aussi des notations propres au calcul des séquents (comme cut, pour dénoter l'application de la règle de coupure). Les feuilles de l'arbres sont évaluées en utilisant les notations true, false, $\perp$, ?, déjà introduites dans les chapitres précédents.

Pour des raisons de clarté, la présentation de l'arbre de preuve du Théorème 3 est segmentée en plusieurs parties. La Figure 4.3, page 85, montre comment concaténer les sous-arbres correspondants pour obtenir l'arbre complet.
Dans les paragraphes suivants, nous détaillons la construction de chaque sous-arbre en adoptant un canevas de présentation commun : brève introduction, puis mise en correspondance avec le discours informel, et enfin commentaires justifiant les jugements portés sur les feuilles du sous-arbre.

4.3.2.2 Détail de la restructuration : racine

Pour construire la racine de l'arbre, nous cherchons à dégager l'articulation logique la plus externe dans le discours informel.

> "**If a processor *p* becomes send-faulty,** ... **Similarly, if *p* just became receive-faulty in the expected broadcast before its slot,** ... [argument commun à ces deux cas]
> **If a processor *p* becomes receive-faulty in its transition to the next step, but *p* is not the next expected broadcaster,** ... [argument pour ce dernier cas]

Définition 4.2 — Notations de l'arbre de preuve du Théorème 3

processeurs :

- p : processeur défaillant,
- q : tout autre processeur non-défaillant.

dates spécifiques :

- t_f : date à laquelle une faute initiale affecte p,
- t_{ok} : premier *slot* d'un processeur non-défaillant après t_f ($t_{ok} > t_f$).
- t_p : premier *slot* d'émission de p après sa faute ($t_p \geq t_f$),
- t_{ok1} : premier *slot* d'un processeur non-défaillant après t_p ($t_{ok1} > t_p$),
- t_{ok2} : second *slot* d'un processeur non-défaillant après t_p ($t_{ok2} > t_{ok1}$),

types de fautes :

- Sfault : p est affecté par une faute d'émission en t_f,
- Rfault_b : p est affecté par une faute de réception en t_f et est le prochain émetteur attendu,
- Rfault-not_b : p est affecté par une faute de réception en t_f et n'est pas le prochain émetteur attendu.

états des variables internes des processeurs :

- ack(i,t) : valeur de la variable ack du processeur i, après l'émission en t,
- ack$_{\neq}(t)$: les valeurs des ack de p et q sont différentes après l'émission en t,
- agree(t) : les *memberships* de p et de tous les q sont les mêmes au début du pas d'horloge t,
- remove$_q(t)$: le processeur q se retire de son *membership* en t.

propriétés du prochain émetteur attendu :

- same_next_b(t) : le prochain émetteur attendu à la fin de t est commun à p et tous les q,
- ok_next_b(t) : le prochain émetteur attendu commun est non défaillant durant son *slot* d'émission.

notations des hypothèses :

- *algo, $fault\ model'$, Inv(Th1), C(i)* : définis § 4.3.2.1 ;
- $\Gamma \equiv$ *algo, $fault\ model'$, Inv(Th1)* ;
- $\Gamma_2 \equiv \Gamma$, (Sfault $\vee$ Rfault-b) ;
- $\Gamma_3 \equiv \Gamma_2$, $\neg$ack(q,t_p), ack(p,t_p), same_next_b(t) ;
- $\Gamma_4 \equiv \Gamma$, Rfault-not_b.

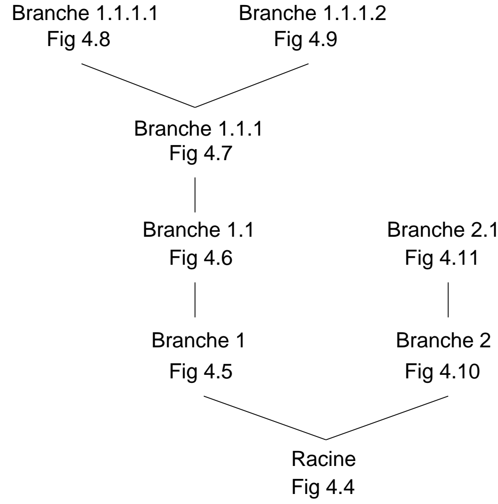


FIG. 4.3 – Arbre de preuve du Théorème 3 - vue globale

On reconnaît là une preuve par cas, représentée par le sous-arbre de la Figure 4.4. Les auteurs dissocient les différentes configurations de fonctionnement de l'algorithme en fonction de la faute initiale.
Le premier cas correspond aux configurations notées *Sfault* et *Rfault-b*, c'est-à-dire que l'on considère soit une faute d'émission, soit une faute de réception juste avant un slot d'émission. Ce cas est traité dans la Branche 1 (§ 4.3.2.3, Figure 4.5).
Le deuxième cas correspond à la configuration notée *Rfault-not_b*, où *p* est affecté par une faute de réception mais n'est pas le prochain émetteur attendu. Ce cas est traité dans la Branche 2 (§ 4.3.2.7, Figure 4.10).
La décomposition est jugée complète sous le modèle de faute considéré, d'où notre évaluation true.

$$\dfrac{\dfrac{\dfrac{(\text{Branche 1})}{\Gamma, (\text{Sfault} \vee \text{Rfault-b}) \vdash \text{Théorème 3}} \quad \dfrac{(\text{Branche 2})}{\Gamma, \text{Rfault-not_b} \vdash \text{Théorème 3}}}{\Gamma, (\text{Sfault} \vee \text{Rfault-b}) \vee \text{Rfault-not_b} \vdash \text{Théorème 3}}\ \vee\vdash \qquad \dfrac{\texttt{true}\ : \textit{fault model'}}{\Gamma \vdash (\text{Sfault} \vee \text{Rfault-b}) \vee \text{Rfault-not_b}}}{\Gamma \vdash \text{Théorème 3}}\ \texttt{cut}$$

FIG. 4.4 – Arbre de preuve du Théorème 3 - Racine

4.3.2.3 Détail de la restructuration : Branche 1

Regardons maintenant la structure de l'argumentation pour la Branche 1.

> " If a processor p becomes send-faulty, ... Similarly, if p just became receive-faulty in the expected broadcast before its slot, ... In either case **[argument montrant que p constatera un premier désaccord en t_{ok1}] [argument montrant que p constatera un deuxième désaccord en t_{ok2}] At that point, the faulty processor *p* will remove itself from its local membership set, using (b)**[3]."

Cette structure est rendue explicite dans le sous-arbre de la Figure 4.5. Notons que les cas de faute *Sfault* ou *Rfault-b* ont été incorporés aux hypothèses Γ_2. Leur point commun est que p sera émetteur avant d'avoir pu diagnostiquer sa faute, remettant ainsi son bit ack à vrai. Le raisonnement porte alors sur les deux *slots* suivants de processeurs non-défaillants, en t_{ok1} et t_{ok2}. Les auteurs veulent montrer que p sera en désaccord deux fois successives avec le bit ack émis, et qu'il s'exclura du groupe d'après la garde (b) de l'algorithme.

En effet, si on considère que p reçoit correctement les messages émis en t_{ok1} et t_{ok2}, il ne pourra pas se retirer de son *membership* en utilisant la garde (a) de l'algorithme. Les auteurs doivent donc prouver que *p* se retirera par la garde (b), et donc qu'il vérifiera le prédicat :

$$\text{garde (b)} : arrived(b, p) \wedge ack(b) \wedge \neg ack(p)$$

Selon cette garde, *p* doit être en désaccord avec la valeur du *ack* émis après avoir mis son propre *ack* à faux au slot précédemment écouté (désaccord avec l'émission précédente). Dans notre arbre de preuve, un désaccord sur le bit *ack* à la date *t* est noté $ack_{\neq}(t)$.
La structure de la Branche 1 se dégage ainsi : les auteurs démontrent $ack_{\neq}(t_{ok1})$ dans la Branche 1.1, puis utilisent ce lemme pour prouver $ack_{\neq}(t_{ok2})$. Le Théorème 3 suit directement de $ack_{\neq}(t_{ok1})$ et $ack_{\neq}(t_{ok2})$, car l'activation de la garde (b) de l'algorithme assure l'auto-diagnostic de *p*, comme nous venons de l'expliquer. On a donc une évaluation true dans la branche la plus à droite du sous-arbre de la Figure 4.5. Cette évaluation est faite sous réserve du modèle de faute restrictif considéré (p reçoit correctement après sa faute).

La preuve de $ack_{\neq}(t_{ok1})$ est assez détaillée, et sera développée dans les trois paragraphes suivants. Par contre, la preuve de $ack_{\neq}(t_{ok2})$ (branche du milieu) n'est qu'ébauchée. Elle correspond à une phrase du discours informel :

> " **Since all nonfaulty processors receive that broadcast and agree with its ack bit, *p* will receive ack as *true* in the expected slot after that (using the fact that there are at least two nonfaulty processors within the group).**"

Notons que le texte entre parenthèses (" using the fact that there are at least two nonfaulty processors within the group ") renvoie à plusieurs étapes implicites de raisonnement. Les auteurs considèrent sans doute pouvoir réutiliser le raisonnement développé dans la Branche 1.1, à la date t_{ok1}. Mais t_{ok2} et t_{ok1} correspondent à des états différents du système, et nous considérons que le raisonnement aurait dû être explicité en t_{ok2}. Ceci explique notre jugement $\perp$.

[3] Il s'agit ici de la condition (b) de l'algorithme page 75.

$$\dfrac{\dfrac{(\text{Branche 1.1})}{\Gamma_2 \vdash \text{ack}_{\neq}(t_{ok1})} \quad \dfrac{\dfrac{\bot}{\Gamma_2, \text{ack}_{\neq}(t_{ok1}) \vdash \text{ack}_{\neq}(t_{ok2})} \quad \dfrac{\text{true} \;:\; \textit{algo, fault model'}}{\Gamma_2, \text{ack}_{\neq}(t_{ok1}), \text{ack}_{\neq}(t_{ok2}) \vdash \text{Théorème 3}}}{\Gamma_2, \text{ack}_{\neq}(t_{ok1}) \vdash \text{Théorème 3}}\ \text{cut}}{\Gamma, (\text{Sfault} \vee \text{Rfault-b}) \vdash \text{Théorème 3}}\ \text{cut}$$

FIG. 4.5 – Arbre de preuve du Théorème 3 - Branche 1

4.3.2.4 Détail de la restructuration : Branche 1.1

Nous allons maintenant détailler la preuve du premier désaccord en t_{ok1}. L'examen du discours informel permet de dégager la structure suivante.

> “ If a processor p becomes send-faulty, **[argument montrant que tout processeur non-défaillant q a son ack à faux après l'émission de p en $t_p = t_f$]** Similarly, if p just became receive-faulty in the expected broadcast before its slot, **[argument montrant que tout processeur non-défaillant q a son ack à faux après l'émission de p en $t_p > t_f$]** In either case **[argument montrant que p a son ack à true après son émission en t_p] [argument montrant que p et tous les q sont d'accord sur l'identité du prochain émetteur attendu] [argument montrant que cet émetteur est bien un processeur non-défaillant] [argument montrant que p sera en désaccord sur le ack émis par ce processeur]** ”

Cette structure est explicitée dans la Figure 4.6. On voit apparaître les précisions suivantes : les valeurs des ack des processeurs après l'émission de p que nous notons $\neg ack(q, t_p) \wedge ack(p, t_p)$, l'accord sur l'identité du prochain émetteur attendu noté $same_next_b(t_p)$, et le fait que cet émetteur attendu est non défaillant noté $ok_next_b(t_p)$. Ces différents lemmes intermédiaires sont posés en hypothèse pour prouver $ack_{\neq}(t_{ok1})$, dans la branche la plus à droite de la Figure 4.6. Sous ces hypothèses additionnelles, nous sommes d'accord que $ack(p, t_{ok1})$ sera bien mis à *faux* selon l'algorithme du GMP présenté page 75 : $arrived(b, p)$ est assuré par $same_next_b(t_p) \wedge ok_next_b(t_p)$, et $\neg ack(b) \wedge ack(p)$ est immédiat. Nous avons donc un jugement true sur cette branche. Rappelons cependant que ce jugement est fait sous réserve que p reçoive correctement après sa faute (*fault model'*)

La preuve du lemme intermédiaire $ok_next_b(t_p)$ (branche du milieu) est elle-aussi jugée satisfaisante. Dans le discours informel, elle correspond à l'argument :

> “ **By the fault arrival assumption, the next expected slot must be non-faulty, since it cannot be newly faulty, and by the invariant it cannot be an undiagnosed old fault.** ”

Nous considérons cet argument comme valide, comme tenu du modèle de faute (écart temporel entre deux fautes initiales, nombre minimum de processeurs non-défaillants), et compte tenu de l'invariant du Théorème 1 (en particulier le Conjoint (8) : si b est attendu par un processeur non-défaillant, alors soit b est non-défaillant, soit il est défaillant depuis moins de n pas d'horloge).

Pour les autres lemmes intermédiaires (Branche 1.1.1), la preuve traite séparément $\neg ack(q, t_p) \wedge ack(p, t_p)$ d'une part, et $same_next_b(t_p)$ d'autre part. La décomposition

$$\dfrac{\dfrac{\text{(Branche 1.1.1)}}{\Gamma_2 \vdash \neg \text{ack}(q,t_p) \wedge \text{ack}(p,t_p) \wedge \text{same_next_b}(t_p)} \quad \dfrac{\dfrac{\text{true : } \mathit{fault\ model'}, \mathit{C(8)}}{\Gamma_3 \vdash \text{ok_next_b}(t_p)} \quad \dfrac{\text{true : } \mathit{algo}}{\Gamma_3, \text{ok_next_b}(t_p) \vdash \text{ack}_{\neq}(t_{ok1})}}{\Gamma_2, \neg \text{ack}(q,t_p), \text{ack}(p,t_p), \text{same_next_b}(t_p) \vdash \text{ack}_{\neq}(t_{ok1})}\ \text{cut}}{\Gamma_2 \vdash \text{ack}_{\neq}(t_{ok1})}\ \text{cut}$$

FIG. 4.6 – Arbre de preuve du Théorème 3 - Branche 1.1

$$\dfrac{\dfrac{\text{(Branche 1.1.1.1)}}{\Gamma_2 \vdash \neg \text{ack}(q,t_p) \wedge \text{ack}(p,t_p)} \quad \dfrac{\text{(Branche 1.1.1.2)}}{\Gamma_2 \vdash \text{same_next_b}(t_p)}}{\Gamma_2 \vdash \neg \text{ack}(q,t_p) \wedge \text{ack}(p,t_p) \wedge \text{same_next_b}(t_p)}\ \vdash \wedge$$

FIG. 4.7 – Arbre de preuve du Théorème 3 - Branche 1.1.1

est reproduite Figure 4.7. Nous allons détailler les deux branches correspondantes dans les deux paragraphes suivants.

4.3.2.5 Détail de la restructuration : Branche 1.1.1.1

La branche 1.1.1.1 établit $\neg ack(q, t_p) \wedge ack(p, t_p)$. Elle correspond à la partie suivante du discours :

> "If a processor *p* becomes send-faulty, **all nonfaulty processors will set their ack bits to *false* in the step following that processor's slot, since the slot is expected and no message is received.** Similarly, if *p* just became receive-faulty in the expected broadcast before its slot, **it will broadcast ack as *false*, while the nonfaulty processors have ack = *true*, and thus will set their ack bits to *false*.** In either case, ***p* will set its ack bit to *true* in the step after it broadcasts.** "

La reformulation en sous-arbre de preuve est donnée Figure 4.8. La preuve de $\neg ack(q, t_p)$ est décomposée sur les cas *Sfault* et *Rfault-b*, alors que $ack(p, t_p)$ est prouvée en une seule étape. Les trois feuilles résultantes sont jugées satisfaisantes.

$$\dfrac{\dfrac{\dfrac{\text{true : } \mathit{fault\ model'}, \mathit{algo, Inv(Th1)}}{\Gamma, \text{Sfault} \vdash \neg \text{ack}(q,t_p)} \quad \dfrac{\text{true : } \mathit{fault\ model'}, \mathit{algo, Inv(Th1)}}{\Gamma, \text{Rfault-b} \vdash \neg \text{ack}(q,t_p)}}{\Gamma_2 \vdash \neg \text{ack}(q,t_p)}\ \vee\vdash \quad \dfrac{\text{true : } \mathit{algo}, \mathit{fault\ model'}}{\Gamma_2 \vdash \text{ack}(p,t_p)}}{\Gamma_2 \vdash \neg \text{ack}(q,t_p) \wedge \text{ack}(p,t_p)}\ \vdash \wedge$$

FIG. 4.8 – Arbre de preuve du Théorème 3 - Branche 1.1.1.1

4.3.2.6 Détail de la restructuration : Branche 1.1.1.2

La branche 1.1.1.2 établit $same_next_b(t_p)$: après t_p, p et tous les q sont d'accord sur l'identité du prochain émetteur attendu. L'argument informel est reproduit ci-dessous.

> **“Until its own or the previous broadcast, *p* was nonfaulty, and thus its local membership set agreed with all other nonfaulty processors. By the invariant of Theorem 1, no nonfaulty processor will remove itself due to the new fault, thus the next expected slot of the nonfaulty processors is the same as the next expected slot of the faulty one.** ”

En reformulant cet argument (Figure 4.9), on s’aperçoit que la notion de *“next expected slot”* dans la dernière phrase est confuse : on ne sait pas s’il s’agit du slot après t_f, ou après t_p. Examinons cet argument pas à pas. La première phrase exprime le fait que avant la faute, p avait la même vue locale du groupe que les autres processeurs non-défaillants, ce que nous avons traduit par la notation $agree(t_f)$. Le début de la deuxième phrase exprime le fait qu’aucun processeur non-défaillant ne s’est retiré du groupe du fait de la faute, ce que nous avons traduit par $\neg remove_q(t_f)$. Il semblerait donc, que, à la fin de la deuxième phrase, les auteurs soient en train d’établir $same_next_b(t_f)$, alors qu’il faudrait établir $same_next_b(t_p)$. Dans le cas d’une *Sfault*, ces deux dates sont les mêmes. Mais pour une *Rfault-b*, il manque une étape de raisonnement montrant que le slot d’émission de p après la faute (t_p) ne peut conduire à un désaccord sur l’identité du prochain émetteur attendu. C’est pourquoi nous avons une évaluation $\perp$ dans la branche de droite : de notre point de vue, la preuve de $same_next_b(t_p)$ est douteuse dans le cas *Rfault-b*.

$$\dfrac{\dfrac{\texttt{true}:\ \textit{Inv(Th1)}}{\Gamma_2 \vdash \text{agree}(t_f) \wedge \neg\text{remove}_q(t_f)} \qquad \dfrac{\perp :\ \textit{confusion entre } t_f \textit{ et } t_p}{\Gamma_2, \text{agree}(t_f), \neg\text{remove}_q(t_f) \vdash \text{same_next_b}(t_p)}}{\Gamma_2 \vdash \text{same_next_b}(t_p)}\ \texttt{cut}$$

FIG. 4.9 – Arbre de preuve du Théorème 3 - Branche 1.1.1.2

Il est à noter que dans la preuve de $ack_{\neq}(t_{ok2})$, que nous avions jugée trop sommaire (§ 4.3.2.3, Figure 4.5), le fragment de texte :

> **“... (using the fact that there are at least two non-faulty processors within the group)** ”

était la seule justification fournie pour établir que p et les q sont toujours d’accord sur le prochain émetteur attendu après t_{ok1}, et que ce processeur est non-défaillant. On comprend bien que nous ayons jugé cet argument insuffisant.

Ceci conclut notre reformulation de l’ensemble de la Branche 1 (cas *Sfault* et *Rfault-b*). On a donc identifié deux problèmes :

- $ack_{\neq}(t_{ok1})$ est douteux dans le cas *Rfault-b*, du fait de la confusion temporelle entre t_f et t_p lors de la preuve du lemme intermédiaire $same_next_b(t_p)$.
- $ack_{\neq}(t_{ok2})$ est douteux dans les deux cas, *Sfault* et *Rfault-b*, car la preuve est à peine ébauchée là où un raisonnement non-trivial serait requis.

4.3.2.7 Détail de la restructuration : Branche 2

Il nous reste à traiter le cas *Rfault-not_b* que nous avions laissé en suspens en Figure 4.4 (racine de l'arbre). Rappelons qu'il s'agit du cas où p est affecté par une faute de réception mais n'est pas le prochain émetteur attendu. Dans ce cas, le diagnostic se fait en un *slot* de processeur valide après la faute : p met son ack à faux lors de sa faute de réception en t_f, et ne va pas perdre cette information par une ré-initialisation à vrai lors de son *slot* d'émission (puisque p n'est pas le prochain émetteur attendu). Il suffit alors de prouver un seul désaccord sur le *slot* attendu suivant (que nous noterons t_{ok}). L'argument correspondant est reproduit ci-dessous.

> “ If a processor p becomes receive-faulty in its transition to the next step, but p is not the next expected broadcaster, **it will remove the broadcaster from its local membership set, but otherwise has the same local membership sets and next expected slot as the nonfaulty ones. It will also set ack(p) to *false*. Again by the fault model, the next expected broadcaster must be nonfaulty, will broadcast ack as *true*, and the receive-faulty processor p will remove itself using (b).** ”

Comme on le voit sur les Figures 4.10 et 4.11, on retrouve une structure de preuve similaire à précédemment, mais plus simple : preuve que p et les q sont d'accord sur l'identité du prochain émetteur attendu, que cet émetteur est bien non-défaillant, et que p constatera un désaccord sur le ack émis, entraînant l'auto-diagnostic par la commande (b) de l'algorithme. Nous ne donnons pas ici le développement complet de ce sous-arbre, que nous avons évalué comme satisfaisant. Nous voulons juste faire apparaître qu'ici il n'y a plus de confusion temporelle pour prouver $same_next_b$ (branche la plus à gauche de la Figure 4.11). La date considérée pour cette propriété est sans ambiguïté t_f, et le raisonnement utilisé dans la branche 1.1.1.2 peut être repris ici (justifiant l'allégation un peu rapide que : “ [p] has the same local membership sets and next expected slot as the nonfaulty ones ”).

Nous considérons donc cette partie de l'arbre de preuve comme valide, bien qu'elle contienne quelques renvois implicites au raisonnement précédent.

$$\dfrac{\dfrac{\text{(Branche 2.1)}}{\Gamma_4 \vdash \text{ack}_{\neq}(t_{ok})} \qquad \dfrac{\texttt{true} \ : algo,\ fault\ model'}{\Gamma_4, \text{ack}_{\neq}(t_{ok}) \vdash \text{Théorème 3}}}{\Gamma, \text{Rfault-not_b} \vdash \text{Théorème 3}}\ \texttt{cut}$$

FIG. 4.10 – Arbre de preuve du Théorème 3 - Branche 2

$$\dfrac{\dfrac{\text{(non détaillé, évaluation } \texttt{true}\text{)}}{\Gamma_4 \vdash \neg\text{ack}(p,t_f) \wedge \text{ack}(q,t_f) \wedge \text{same_next_b}(t_f)} \qquad \dfrac{\dfrac{\texttt{true} \ : fault\ model',\ C(8)}{\Gamma_3 \vdash \text{ok_next_b}(t_f)} \quad \dfrac{\texttt{true} \ : algo,\ fault\ model'}{\Gamma_3, \text{ok_next_b}(t_f) \vdash \text{ack}_{\neq}(t_{ok})}}{\Gamma_4, \neg\text{ack}(p,t_f), \text{ack}(q,t_f), \text{same_next_b}(t_f) \vdash \text{ack}_{\neq}(t_{ok})}\ \texttt{cut}}{\Gamma_4 \vdash \text{ack}_{\neq}(t_{ok})}\ \texttt{cut}$$

FIG. 4.11 – Arbre de preuve du Théorème 3 - Branche 2.1

4.3.3 Preuve du Conjoint (5)

Après cette présentation détaillée de la restructuration de la preuve du Théorème 3, nous allons passer plus vite sur l'autre problème identifié, qui affecte le Conjoint (5).

Le Conjoint (5) fait partie des 8 conjoints de l'invariant développé pour prouver le Théorème 1. Sa validité est essentielle non seulement pour établir le Théorème 1, mais aussi pour établir les Théorèmes 2 et 3 dont les preuves mettent l'invariant en hypothèse. Le Conjoint (5) correspond à la propriété ci-dessous.

Conjoint (5) :

> Si un processeur p est devenu défaillant moins de n pas d'horloge auparavant, et si q est un processeur non-défaillant, alors soit p est l'émetteur courant, soit l'émetteur courant est dans le *membership* de p ssi il est dans celui de q.

Rappelons que la preuve informelle des différents conjoints de l'invariant est faite par induction sur le temps. Dans le cas du Conjoint (5), la validité à l'instant initial est évidente, tous les processeurs étant non-défaillants. Par contre, l'étape d'induction est complexe, avec un raisonnement impliquant plusieurs pas d'horloge. Dans le discours, des termes tels que : “ *n steps ago* ”, “ *since then and up until the next step* ” doivent être interprétés. Ils définissent des fenêtre temporelles dont les bornes dépendent du fait que la date de référence soit t ou $t+1$ dans la partie du raisonnement considérée. Il n'est en fait pas étonnant que nous ayons trouvé des confusions temporelles à ces bornes.

Trois des branches de preuve du Conjoint (5) ont été jugées false. Elles correspondent toutes au même problème, lié à un lemme intermédiaire de la preuve : “ *In all steps since then and up until the next step, r is not the broadcaster* ... ”. Ce lemme doit établir qu'un processeur r n'est jamais émetteur dans l'intervalle $[t-(n-1), t]$, sous l'hypothèse que r sera émetteur en $t+1$. Ce lemme est faux car si r est émetteur en $t+1$, alors il a nécessairement émis en $(t+1)-n = t-(n-1)$. Notons que le fait que ce lemme intermédiaire soit faux n'implique pas que le Conjoint (5) lui-même soit faux. Cependant, il doit être considéré comme non prouvé.

Outre ces trois branches fausses, nous avons identifié un autre point sensible. Dans une partie du raisonnement, les auteurs utilisent implicitement l'hypothèse que le Conjoint (2)[4] est vrai en *t+1* (alors que le schéma général de la preuve est de supposer que les conjoints ne sont vrais que jusqu'à t). Nous avons cependant jugé que cela ne posait pas de problème, car l'hypothèse supplémentaire n'introduit pas de circularité dans la preuve : l'étape d'induction du Conjoint (2) ne nécessite pas que le Conjoint (5) soit valide en $t+1$.

[4]Ce conjoint énonce que tous les processeurs non-défaillants appartiennent à leur propre *membership*.

4.3.4 Bilan des faiblesses identifiées

Bien qu'un certain nombre de problèmes aient été identifiés au cours de la restructuration, cette preuve informelle est bien plus rigoureuse que celle que nous avons étudiée dans le Chapitre 3. Nous jugeons donc qu'elle peut fournir des informations utiles pour guider la conception du test.

La restructuration a mis en évidence deux problèmes à portée générale, et un problème plus localisé.

Un premier problème général affecte toute la preuve : les auteurs n'ont pas pris en compte toutes les configurations de fautes possibles. Sauf dans le cas du Conjoint (6), où la possibilité de ne pas recevoir deux messages consécutifs est mentionnée, l'occurrence de fautes ultérieures est ignorée.
Il nous sera donc nécessaire de tester le comportement du GMP pour des cas de fautes intermittentes ou permanentes, c'est à dire pour des configurations comprenant des fautes ultérieures.

Un second problème général concerne le lemme intermédiaire que nous avons jugé faux dans la preuve du Conjoint (5). Ce lemme est utilisé dans plusieurs branches de cette preuve et notre jugement final est que l'invariance du Conjoint (5) n'est prouvée pour aucune configuration d'entrée.
Ce problème est de portée générale car les autres conjoints du Théorème 1 utilisent l'hypothèse que le Conjoint (5) est valide jusqu'en t dans leur étape d'induction. De plus, les Théorèmes 2 et 3 sont prouvés sous l'hypothèse d'invariance des huit conjoints. Ainsi, l'ensemble de la preuve du GMP peut être invalidée par propagation. Tous les cas fonctionnels qui peuvent être extraits des sous-arbres de preuve sont donc potentiellement pertinents pour le test du GMP. De plus, il paraît souhaitable de renforcer l'oracle de test pour y incorporer un observateur spécifique de la validité du Conjoint (5). Ceci peut se faire sans instrumentation supplémentaire du prototype : comme dans le cas des trois théorèmes déjà incorporés à l'oracle, l'observation des *membership* des différents processeurs à chaque date est suffisante pour établir un verdict de défaillance.

Le dernier problème que nous avons identifié est localisé dans la preuve du Théorème 3. Les deux branches évaluées $\perp$ impliquent que la validité de ce théorème est douteuse pour des configurations de fautes transitoires de type *Sfault* ou *Rfault-b*.

On voit donc se dessiner deux types de test :

- un test global, selon un profil d'échantillonnage qui n'exclut aucun point du domaine d'entrée. Ce profil devra combiner des configurations de fautes permanentes, intermittentes et transitoires avec tous les cas fonctionnels extraits de l'analyse de la preuve.
- un test spécifique pour une vérification plus sévère des cas *Sfault* et *Rfault-b* sans faute ultérieure.

Nous allons maintenant préciser la mise en œuvre de ces tests.

4.4 Étape 3 : Test Guidé par la Preuve

Tous les tests seront conduits dans l'environnement développé à l'issue de l'analyse préliminaire, en renforçant l'oracle par un observateur de la validité du Conjoint (5).

4.4.1 Critères de sélection des entrées de test

Nous devons concevoir un profil de test global permettant de couvrir tous les cas fonctionnels extraits de la preuve :

- les conditions d'activation des différentes règles de mise à jour de l'algorithme, qui apparaissent dans les arbres de preuve (notation *algo*) ;
- les cas de faute initiale *SFault*, *RFault-b*, et *Rfault-not_b* issus de l'arbre du Théorème 3 ;
- des fautes initiales non suivies de fautes ultérieures (tous les arbres) ;
- deux fautes de réception consécutives (arbre du Conjoint (6))

ainsi que les cas de fautes permanentes ou intermittentes ignorés par la preuve.

Pour préciser comment commander ces différents cas, et comment les combiner, une analyse du comportement séquentiel global de l'algorithme est nécessaire. Nous avons décidé de reprendre le schéma d'automate réalisé lors de l'analyse préliminaire, pour le raffiner en lui incorporant les informations identifiées ci-dessus. L'automate résultant est donné dans la Figure 4.12. Il permet d'identifier 20 classes de chemins de *stable* à *stable* (au lieu de 3 dans l'automate initial), correspondant à 20 classes de scénarios de faute. Les états de cet automate sont caractérisés par les valeurs de *ack* et *membership* locales aux processeurs, en distinguant p (le processeur nouvellement défaillant) et les q (processeurs non-défaillants), et en faisant abstraction du nombre et de l'identité de ces processeurs. Chaque transition correspond à un *slot* d'émission de p ou d'un q, p pouvant manifester sa faute (ou non) à ce *slot*. On ne représente pas explicitement les *slots* d'émission des processeurs déjà diagnostiqués comme défaillants avant la faute de p. Ces processeurs restent silencieux, ne sont plus écoutés, et ne font donc pas évoluer l'état du système : implicitement, leurs *slots* correspondent à une boucle réflexive sur chacun des états.
En plus du diagramme de la Figure 4.12, une analyse a été menée pour identifier les règles de l'algorithme activées par p et les q lorsqu'une transition est tirée. Notons que cette analyse ne fait pas apparaître de problème particulier lié à l'occurrence de fautes ultérieures : de façon générale, l'état d'arrivée est le même que p manifeste sa faute ou non, mais les règles activées sont différentes dans les deux cas.

Le critère que nous retenons pour le test global du GMP est la couverture des 20 classes de chemins de l'automate. Sous le profil de test correspondant, le nombre de processeurs et de fautes initiales sont générés comme sous le profil aveugle. La génération des fautes est cependant modifiée pour rendre les 20 classes équiprobables.

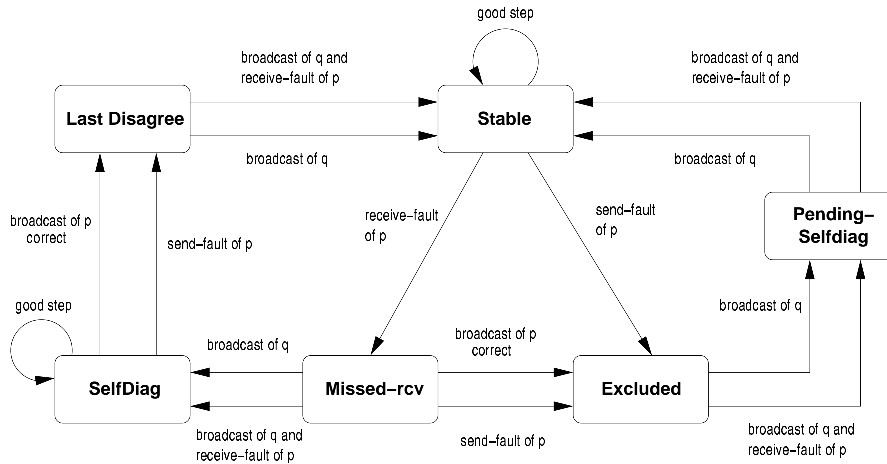


FIG. 4.12 – Automate du GMP

Le test global devra être complété par un test spécifique, ciblant les deux configurations de fautes identifiées lors de l'analyse de la preuve du Théorème 3. Les fonctions de génération correspondantes sont telles que :

- la faute initiale est fixée à *SFault* ou *RFault-b* ;
- il n'y a pas de fautes ultérieures.

4.4.2 Résultats expérimentaux

Nous présentons dans cette section les résultats des tests du GMP selon ces différents profils.

4.4.2.1 Test global du GMP

Un échantillon de 45 séquences de test a été généré selon le profil global. Considérant que chaque classe de chemin a une probabilité d'au moins 1/20 d'être sélectionnée dans une séquence, cela donne une qualité de test d'au moins 0,9 (*c.f.* § 1.2.4.4). Notons que cette estimation est très pessimiste car une séquence de test peut contenir plusieurs fautes initiales (5,25 en moyenne, mais l'écart type est élevé du fait de la dépendance vis-à-vis du nombre total n de processeurs). En pratique l'échantillon active plusieurs fois chaque classe avec des séquences de fautes aléatoires différentes.

Les résultats de ce test global sont donnés dans le Tableau 4.2. Le Conjoint (5) a été ajouté au trois théorèmes dans l'oracle.

Les 45 séquences de test provoquent 4 défaillances du Théorème 3, et aucune autre propriété n'est mise en défaut. Pour comparer ces résultats à ceux du test aléatoire

TAB. 4.2 – Résultats du test global du GMP

Nombre de séquences de test	Défaillances du Th1	Défaillances du Th2	Défaillances du Th3	Défaillances du C(5)
45	0	0	4	0
10^4	0%	0%	7,65%	0%

aveugle, nous avons conduit un test avec un échantillon plus large de 10^4 séquences de test, et constaté un taux de défaillance de 7,65% (contre 6,15% pour le test aveugle). Toutes les défaillances sont dues à la faute de conception connue de l'algorithme.

Le profil de test global, assurant une couverture équiprobable des 20 classes de chemins de l'automate, n'est donc pas beaucoup plus stressant que le profil aveugle pour révéler cette faute. Mais ce profil était destiné à révéler des problèmes liés aux faiblesses identifiées dans la preuve du Conjoint (5), et aux configurations de fautes intermittentes ou permanentes ignorées dans l'ensemble de la preuve. Le résultat le plus probant de ce profil de test est donc que ces faiblesses n'ont pas spécifiquement conduit à des défaillances de l'algorithme. En particulier, le fait que le Conjoint (5) ne soit jamais mis en défaut suggère qu'il pourrait être valide malgré les trois branches identifiées comme fausses dans sa preuve informelle.

4.4.2.2 Test spécifique des faiblesses du Théorème 3

Nous avons conduit des expériences de test pour les deux configurations *SFault* et *RFault-b*, en utilisant 10 séquences d'entrées ciblant chacune d'entre elles. Pour chacune des deux configurations, nous avons également répété les expériences avec un échantillon plus large de séquences pour pouvoir évaluer le taux de défaillances résultant.

La première configuration *SFault*, correspondant à une faute d'émission initiale sans faute ultérieure, ne conduit jamais à aucune défaillance. Ces résultats suggèrent que le Théorème 3 est sans doute valide pour cette configuration d'entrée.

Les résultats des tests avec la deuxième configuration *RFault-b*, correspondant à une faute transitoire de réception sur le prochain émetteur attendu, sont présentés dans la Table 4.3.

Avec seulement 10 séquences de test, nous constatons 2 défaillances du Théorème 3, ce résultat étant confirmé par un taux de défaillance de 19% sur un jeu de 10^4 séquences de test. Notons que les autres vérifications de l'oracle de test (Théorème 1, Théorème 2 et Conjoint (5)) n'ont jamais été mises en défaut. Ce taux de défaillance est trois fois supérieur à celui obtenu par un test aléatoire aveugle de l'algorithme. L'identification du cas *RFault-b*, pour lequel la preuve du Théorème 3 avait été jugée douteuse, s'est donc avérée une information pertinente pour guider le test.

Tab. 4.3 – Résultats du test spécifique de la configuration *Rfault-b*

Nombre de séquences de test	Défaillances du Théorème 3
10	2
10^4	19%

En inspectant plus en détail les séquences de test conduisant à défaillance, nous avons observé que les fautes qui mettent l'algorithme en échec correspondent toujours à des fautes de type *RFault-b* dans un groupe ne contenant plus que 3 processeurs non-défaillants, quel que soit le nombre total de processeurs à l'origine (rappelons que nos séquences de test contiennent éventuellement plusieurs fautes initiales, conduisant à une restriction progressive de la taille du groupe). Ces observations vont être confirmées dans l'étape suivante de retour sur la preuve.

4.5 Étape 4 : Retour sur la preuve

Dans un premier temps, les information extraites de l'arbre de preuve nous ont permis de guider le test du GMP. Les résultats de ces tests vont maintenant nous servir à reprendre la preuve, et à affiner notre analyse des branches jugées douteuses.

Nous allons discuter des problèmes identifiés aux paragraphes 4.3.2 et 4.3.3, relatifs aux preuves du Conjoint (5) et du Théorème 3.

4.5.1 Retour sur la preuve du Conjoint (5)

Le Conjoint (5) n'ayant jamais été mis en défaut lors de nos tests, nous pensons qu'il pourrait être valide, malgré les trois branches de preuve jugées fausses.

Rappelons que dans l'étape d'induction, les auteurs utilisent un lemme intermédiaire censé être vrai sur un intervalle de temps $[t-(n-1), t]$, alors qu'il est manifestement faux pour la limite inférieure de cet intervalle. Nous avons donc décidé de reprendre la preuve en traitant séparément la limite qui pose problème. Sur l'intervalle restreint $[t-(n-2), t]$, l'argumentation des auteurs est reprise et jugée satisfaisante. Pour le cas de la date $t-(n-1)$, nous avons développé une autre démonstration, qui ne se base plus sur le lemme en question. Plus précisément, notre preuve utilise l'algorithme, et un des conjoints de l'invariant supposé vrai en $t-(n-1)$.

Le Conjoint (5) est donc maintenant considéré comme prouvé, selon notre analyse.

4.5.2 Retour sur la preuve du Théorème 3

Deux branches de l'arbre de la preuve informelle du Théorème 3 avaient été estimées faibles dans la restructuration de la preuve informelle. Nous allons revenir sur ces deux branches, pour essayer d'établir un lien avec les scénarios de défaillance trouvés par le test.

Tout d'abord, nous avons développé la branche notée $\perp$ du sous-arbre présenté Figure 4.9, et qui cherchait à prouver qu'après t_p, le processeur p et tous les q sont d'accord sur l'identité du prochain émetteur attendu. La Définition 4.3 introduit quelques notations supplémentaires pour l'arbre de preuve obtenu (on se reportera à la page 84 pour les autres notations).

Définition 4.3 — Notations complémentaires de l'arbre du Théorème 3

propriétés du prochain émetteur attendu :

- same_b(t) $\equiv$ à la date t, l'émetteur b est attendu par p ssi il est attendu par q
- C'(5) $\equiv t_{ok1} - t_f < n \Rightarrow$ same_b(t_{ok1})

notations des hypothèses :

- $\Gamma_5 \equiv \Gamma, \text{agree}(t_f), \neg\text{remove}_q(t_f)$.

Nous avons commencé par décomposer l'hypothèse *Sfault* $\vee$ *Rfault-b*, présente dans Γ_2, pour différencier le traitement de ces deux configurations de faute (Figure 4.13). Rappelons que leur traitement simultané dans la preuve d'origine avait amené une confusion entre les dates t_f et t_p. Les deux sous-arbres de preuve obtenus sont présentés Figures 4.14 (cas *Sfault*) et 4.15 (cas *Rfault-b*).

$$\dfrac{\dfrac{\dfrac{\text{(Branche 1.1.1.2.1)}}{\Gamma_5, \text{Sfault} \vdash \text{same_next_b}(t_p)} \quad \dfrac{\text{(Branche 1.1.1.2.2)}}{\Gamma_5, \text{Rfault-b} \vdash \text{same_next_b}(t_p)}}{\Gamma, (\text{Sfault} \vee \text{Rfault-b}), \text{agree}(t_f), \neg\text{remove}_q(t_f) \vdash \text{same_next_b}(t_p)} \vee\vdash}{\Gamma_2, \text{agree}(t_f), \neg\text{remove}_q(t_f) \vdash \text{same_next_b}(t_p)}$$

FIG. 4.13 – Arbre de preuve du Théorème 3 - Branche 1.1.1.2 suite

Pour prouver chacun de ces cas, il semble judicieux d'utiliser le Conjoint (5), qui exprime justement une propriété sur l'émetteur attendu par p et les q.

> *Conjoint (5)* :
> Si un processeur p est devenu défaillant moins de n pas d'horloge auparavant, et si q est un processeur non-défaillant, alors soit p est l'émetteur courant, soit l'émetteur courant est dans le *membership* de p ssi il est dans celui de q.

Le Conjoint (5), qui est invariant, peut en particulier être instancié à la date t_{ok1} (premier *slot* de processeur non-défaillant après t_p). Le processeur p vient d'être émetteur, et il y a encore au moins deux processeurs non-défaillants, dont l'émetteur courant en t_{ok1}. Nous pouvons éliminer la conclusion “ *p est l'émetteur courant* ”, pour ne garder que celle qui nous intéresse : “ *l'émetteur courant est dans le membership de p ssi il est dans celui de q* ”.

L'instance *C'(5)* que nous introduisons dans les sous-arbres des figures 4.14 et 4.15, est définie comme suit :

$$t_{ok1} - t_f < n \Rightarrow \text{same_b}(t_{ok1})$$

Pour chacun de ces sous-arbres, la branche la plus à gauche exprime que C'(5) se déduit de l'invariance du Conjoint (5). L'autre partie décompose C'(5) sur l'opérateur d'implication. La branche la plus à droite demande alors de prouver que same_b(t_{ok1}) implique le but désiré. Cela peut se faire à l'aide de l'invariant, qui garantit qu'aucun processeur ne peut être attendu ni par p ni par q entre t_p et t_{ok1}, et que l'émetteur en t_{ok1} est nécessairement attendu par q. L'hypothèse same_b(t_{ok1}) nous garantit alors qu'il est aussi attendu par p. Pour utiliser cette hypothèse, il faut néanmoins prouver dans la branche du milieu que le prémisse $t_{ok1} - t_f < n$ est vrai. Là, le raisonnement est légèrement différent selon que l'on considère une *Sfault* (avec $t_f = t_p$) ou une *Rfault-b* (avec $t_f < t_p$). Outre p, il y a potentiellement entre 0 et $n-3$ autres processeurs déjà défaillants dans le système, et leurs *slots* peuvent s'intercaler entre t_f et t_{ok1}. On a alors une borne supérieure de $n-2$ ou $n-1$ pas d'horloge selon le cas.

$$\dfrac{\dfrac{\text{true}: C(5)}{\Gamma_5, \text{Sfault} \vdash \text{C'(5)}} \quad \dfrac{\dfrac{\text{true}:\ t_{ok1} - t_f \leq n-2}{\Gamma_5, \text{Sfault} \vdash t_{ok1} - t_f < n} \quad \dfrac{\text{true}: Inv(Th1)}{\Gamma_5, \text{Sfault}, \text{same_b}(t_{ok1}) \vdash \text{same_next_b}(t_p)}}{\Gamma_5, \text{Sfault}, \text{C'(5)} \vdash \text{same_next_b}(t_p)} \Rightarrow\vdash}{\Gamma_5, \text{Sfault} \vdash \text{same_next_b}(t_p)}\ \text{cut}$$

FIG. 4.14 – Arbre de preuve du Théorème 3 - Branche 1.1.1.2.1

$$\dfrac{\dfrac{\text{true}: C(5)}{\Gamma_5, \text{Rfault-b} \vdash \text{C'(5)}} \quad \dfrac{\dfrac{\text{true}:\ t_{ok1} - t_f \leq n-1}{\Gamma_5, \text{Rfault-b} \vdash t_{ok1} - t_f < n} \quad \dfrac{\text{true}: Inv(Th1)}{\Gamma_5, \text{Rfault-b}, \text{same_b}(t_{ok1}) \vdash \text{same_next_b}(t_p)}}{\Gamma_5, \text{Rfault-b}, \text{C'(5)} \vdash \text{same_next_b}(t_p)} \Rightarrow\vdash}{\Gamma_5, \text{Rfault-b} \vdash \text{same_next_b}(t_p)}$$

FIG. 4.15 – Arbre de preuve du Théorème 3 - Branche 1.1.1.2.2

Ceci clôt notre développement de la branche 1.1.1.2, que nous considérons maintenant comme achevée.

Il reste cependant à étudier l'autre branche initialement évaluée à $\perp$ dans la Figure 4.5. Cette branche correspond à la preuve de $ack_{\neq}(t_{ok2})$, que nous avions jugée trop

sommaire. Pour la développer, nous nous sommes calqués sur le même raisonnement que pour $ack_{\neq}(t_{ok1})$. En particulier, ce raisonnement nous a amenés à essayer d'établir que le prochain émetteur attendu après t_{ok1} doit encore être le même pour p et les q (*same_next_b(t_{ok1})*) pour chacun des cas *Sfault* et *Rfault-b*.
Malheureusement, l'utilisation d'une instance du Conjoint (5) en t_{ok2} (le prochain *slot* de processeur non-défaillant) nécessite de prouver $t_{ok2} - tf < n$, et puisque nous avons avancé dans le temps, cela donne :

- $n - 1 < n$ pour une *Sfault*, ce qui est encore vrai, et
- $n < n$ pour une *Rfault-b*, ce qui est faux. La borne n est atteinte dans le pire cas où, en plus de p, il y a $n - 3$ autres processeurs déjà défaillants dans le système[5].

Il ne servirait à rien d'essayer de prouver différemment le cas qui pose problème. On peut vérifier que, dans ce cas, l'émetteur en t_{ok2} n'est plus dans le $membership$ de p : il ne sera pas écouté, p laissera son $membership$ inchangé, et la propriété d'auto-diagnostic en deux *slots* de processeurs non-défaillants sera violée[6]. La reprise de la preuve a donc permis de retrouver les résultats de test : on a une défaillance du Théorème 3 lorsqu'une faute de type *RFault-b* arrive dans un groupe ne contenant plus que trois processeurs valides.

A l'issue cette dernière étape de notre méthode, nous sommes capables de pointer précisément sur la faille de raisonnement qui a laissé passer la faute de conception du GMP. Dans le discours informel, la faille correspond au fragment de texte : "... *(using the fact that there are at least two non-faulty processors within the group)* ", qui ne permet en fait pas d'établir que p écoute l'émetteur en t_{ok2}. Cette faille a été rendue possible par une confusion entre les dates t_f et t_p, déjà apparente dans le raisonnement sur l'émetteur attendu en t_{ok1}, et qui s'est avérée invalider la preuve pour t_{ok2}.

4.6 Conclusion

Le Test Guidé par la Preuve s'est avéré efficace dans le cas de l'algorithme d'appartenance de groupe étudié dans ce chapitre. Contrairement à ce qui s'était passé pour le cas du FT-RMS, notre restructuration de la preuve n'a, cette fois, pas été bloquée par des parties obscures du discours informel. Nous avons donc pu conduire une analyse beaucoup plus précise, l'arbre de preuve du Théorème 3 à lui seul étant nettement plus détaillé que celui que nous avions pu extraire de l'ensemble de la preuve du FT-RMS dans le Chapitre 3. Plusieurs faiblesses de la preuve ont ainsi été mises en évidence, l'une d'entre elles étant liée à la faute de conception dans l'algorithme. Elle a permis d'identifier un sous-ensemble du domaine d'entrée pour lequel le taux de détection lors des tests s'est avéré multiplié par trois par rapport à un test aléatoire en aveugle. Ce résultat encourageant montre que l'identification

[5] Dans un système à trois processeurs, $n = 3$, une *Rfault-b* provoque donc toujours une défaillance.

[6] Pire, p ne pourra en fait jamais se diagnostiquer comme défaillant, et on aboutira à une partition du système en deux cliques : le groupe des deux processeurs non-défaillants, et un groupe constitué du seul p qui considère être le seul processeur restant.

des lacunes de la preuve peut s'avérer efficace pour guider le test, sous réserve que l'analyse de l'arbre ne mette pas en évidence un manque de rigueur trop important affectant l'ensemble de la démonstration.

L'information extraite de la preuve du GMP, bien que pertinente, reste cependant imparfaite pour révéler la faute de conception. En effet, si nous avons bien été guidés vers le cas *Rfault-b*, l'identification précise de la configuration à trois processeurs n'a pas été faite à ce stade. Nous pensons que cette identification *a priori* aurait demandé une reprise très fouillée de la preuve. Il parait moins coûteux de commencer par une analyse plus «légère», et d'utiliser le test statistique pour compenser d'éventuelles imperfections des critères de sélection issus de cette analyse. Une fois que les scénarios de défaillance sont trouvés par le test, il devient plus facile de revisiter la preuve pour mettre en relation ces scénarios avec des étapes de raisonnement erronées.

Nous pouvons maintenant tirer quelques conclusions sur la validation expérimentale de notre approche, au vu des résultats sur les deux études de cas. Le FT-RMS et le GMP constituent deux exemples non-triviaux, représentatifs d'algorithmes de tolérance aux fautes dont la vérification s'avère problématique. L'application des étapes successives de notre méthode nous conduit aux remarques suivantes.

- L'*analyse préliminaire* n'a pas posé de problème particulier, bien qu'il ait parfois fallu interpréter quelques éléments du discours informel. Nous avons toujours réussi à développer un prototype de l'algorithme considéré, et à déterminer un environnement de test pour ce prototype, incluant un moniteur et un oracle de test. A chaque fois, nous avons pu proposer une fonction de génération aléatoire pour échantillonner sur le domaine d'entrée. Cette fonction dite aléatoire *aveugle* permet de s'assurer qu'il est possible d'extraire, des hypothèses de l'algorithme, une définition constructive de son domaine d'entrée. De même, l'oracle de test interprète les propriétés requises en termes d'observations du comportement opérationnel du prototype. Notons que, dans certains cas, l'analyse préliminaire pourrait suffire à révéler une faute de conception à faible coût : l'exemple du GMP montre qu'une erreur subtile de raisonnement ne se traduit pas nécessairement par une faute très difficile à révéler par le test, même si elle avait échappé aux auteurs de la preuve. On a donc toujours intérêt, à l'issue de l'étape d'analyse préliminaire, à mener de premières expériences de test permettant une vérification «grossière» de l'algorithme.
- La *restructuration* du discours informel sous forme d'arbre de preuve s'est avérée un support très utile pour identifier des faiblesses dans le raisonnement, et ce pour les deux exemples traités. L'arbre offre une vision synthétique de la preuve, guidant son analyse pas à pas. Bien que l'on conserve des notations informelles dans l'expression des séquents, on est amené à préciser notre compréhension du discours, et à se poser les bonnes questions à chaque étape de raisonnement. Cette approche a permis de mettre en évidence des problèmes de nature diverse, aussi bien de haut niveau (glissement sémantique de certaines notions, oublis majeurs se répercutant sur tout l'arbre) que spécifiques à une étape (décomposition en cas incomplète, confusion entre deux dates, raccourcis abusifs car considérés comme «similaires au raisonnement précédent»).

La restructuration se veut assez «légère» pour pouvoir être conduite rapidement. Dans ce but, nous recommandons d'arrêter et d'utiliser une notation $\perp$ ou ? lorsqu'une branche résiste à l'analyse : il ne s'agit pas de retravailler la preuve pour l'améliorer, mais plutôt d'évaluer sa rigueur en l'état.
Enfin, comme le montre l'exemple du FT-RMS, une évaluation trop négative pourra amener à conclure que la preuve n'est pas exploitable pour le test. D'une part la preuve fournit alors peu d'information (arbre peu développé, avec la plupart des feuilles évaluées à $\perp$ ou ?), et d'autre part on peut penser que les cas distingués par les auteurs ne sont pas nécessairement pertinents pour étudier le comportement de l'algorithme (ce que nous avons pu constater expérimentalement pour le FT-RMS). Pour des preuves plus rigoureuses, notre hypothèse est que les informations extraites sont pertinentes pour guider le test.

- La difficulté de l'étape de *Test Guidé par la Preuve* est de réussir à utiliser de façon constructive la connaissance des faiblesses de la preuve. Sur les deux exemples traités, nous avons pu voir comment nous avons, selon les cas, renforcé les observations réalisées par l'oracle, ou précisé la fonction de génération des entrées en retenant un critère de sélection. Pour le GMP, la conception d'une fonction de génération globale a nécessité un petit travail de modélisation, pour prendre en compte le comportement séquentiel de l'algorithme. Le niveau d'abstraction du modèle, et les informations à incorporer, ont été guidés par les résultats de l'analyse de la preuve.
  Notons qu'on ne s'attend pas à ce que l'analyse de la preuve ait permis l'identification précise des sous-domaines d'entrée conduisant à défaillance. Les critères retenus sont supposés pertinents, mais imparfaits. Cette hypothèse, qui motive l'utilisation du test statistique, s'est avérée justifiée dans le cas du GMP.
- Les résultats de test permettent enfin un *retour sur la preuve*, en se basant sur l'arbre développé lors de l'analyse préliminaire. La non-observation de défaillance peut être un encouragement à consolider des branches précédemment jugées faibles, comme nous avons pu le faire pour le Conjoint (5) du GMP, ou pour le cas *Sfault* du Théorème 3. La connaissance de scénarios de défaillance peut également permettre d'affiner l'analyse en pointant sur la faille précise qui a laissé passer la faute de conception, ainsi que nous l'avons fait pour le cas *Rfault-b*.
  Remarquons que, si une modification corrective de l'algorithme est envisagée, la preuve devra encore être reprise pour prendre en compte cette modification. L'arbre de preuve pourrait alors rester un support très utile pour identifier les branches affectées.

Ceci conclut nos travaux sur les preuves informelles. Nous allons maintenant envisager l'extension de ces travaux au cas de preuves formelles, mais partielles. Nous y gagnerons les avantages inhérents à la formalisation : pas de notions à la sémantique imprécise, pas d'étapes implicites de raisonnement, et l'identification des branches non prouvées ne dépendra plus d'une évaluation manuelle. Cependant, il reste à déterminer si l'extraction d'informations constructives pour le test s'avérera réalisable en pratique à partir d'une preuve formelle.

Chapitre 5

Test guidé par une preuve formelle

Nous allons maintenant nous pencher sur les preuves formelles, et étudier la pertinence de l'approche de Test Guidé par la Preuve pour un algorithme dont la preuve formelle serait inachevée. Les objectifs de notre approche dans le cas d'une preuve formelle sont multiples :

- éviter la perte complète de l'effort de preuve lorsque celle-ci ne peut aboutir ;
- exhiber des contre-exemples éventuels pour guider la correction de l'algorithme et de sa preuve ;
- encourager la poursuite de l'effort de preuve si les tests ne révèlent aucune faute.

Cette étude a fait l'objet d'un article en français [Lussier & Waeselynck 2004a] et d'une publication internationale en anglais [Lussier & Waeselynck 2004b].

Ce chapitre est divisé en sept parties.

Dans le premier paragraphe, nous envisageons comment adapter la méthode, précédemment définie pour des preuves informelles, à une preuve formelle partielle (c'est-à-dire inachevée).
Dans le deuxième paragraphe, nous expliquons le procédé que nous avons retenu pour permettre l'évaluation expérimentale de cette méthode. S'il était relativement aisé de trouver dans la littérature des exemples de «preuves» informelles d'algorithmes incorrects, les exemples de preuves formelles inachevées sont plus difficiles à obtenir : seuls les succès de preuve sont publiés et disponibles dans le domaine public. Nous avons donc recouru à un artifice, qui consiste à injecter une faute de conception dans une spécification dont la preuve formelle est achevée. Ce procédé d'injection nous permet de rendre inachevée une preuve formelle complète, et ainsi d'utiliser une telle preuve comme cas d'étude pour le test guidé par la preuve.
Le troisième paragraphe est la présentation de notre cas d'étude : nous nous concentrons dans ce chapitre sur la version finale du GMP implanté dans l'architecture TTA.

Cette version présente des similarités avec celle étudiée dans le Chapitre 4 mais est cependant plus complexe. Pour étudier cet algorithme, nous avons pu disposer de sa formalisation et de sa preuve complète sous PVS.
Dans le quatrième paragraphe, nous détaillons l'analyse de haut niveau faite sur la preuve formelle du GMP. Cette analyse vise à extraire les informations nécessaires à la conception d'un environnement de test, et à réaliser un premier test aléatoire aveugle de l'algorithme étudié.
Le cinquième paragraphe présente l'analyse détaillée de la preuve du GMP. Nous décrivons d'abord l'approche de preuve suivie par ses auteurs, basée sur les *invariants disjonctifs*. Nous abordons ensuite l'injection d'une faute de conception dans le couple spécification – preuve formelles, et l'étude de la preuve partielle résultante.
Dans un sixième paragraphe, nous détaillons les résultats des expériences de Test Guidé par la Preuve formelle.
Un dernière paragraphe traitera du retour sur la preuve c'est-à-dire de l'impact potentiel des résultats du test sur la preuve formelle. De plus, nous discuterons de l'impact de la structure de la preuve formelle sur l'efficacité de notre méthode.

5.1 Application de l'approche à une preuve formelle

Notre objectif est d'étudier si l'approche de Test Guidé par la Preuve peut être efficace lorsqu'elle se base sur une preuve formelle, tout comme nous avons pu voir dans le Chapitre 4 qu'elle pouvait l'être lorsqu'elle se base sur une preuve informelle. Nous allons donc définir une méthode pour pouvoir appliquer cette approche sur une preuve formelle. Nous nous basons sur une démarche similaire à celle définie pour une preuve informelle dans le paragraphe 2.1.

La méthode que nous proposons s'articule autour des mêmes quatre étapes, la principale différence se situant dans l'*analyse détaillée* de la preuve formelle, qui remplace la *restructuration de la preuve* qui était appliquée à une preuve informelle. Les quatre étapes sont présentées ci-dessous :

- L'*Analyse de haut niveau* remplace l'analyse préliminaire. Son but est comme précédemment d'obtenir une compréhension globale de l'algorithme et de ses exigences (sous certaines hypothèses, l'algorithme doit garantir la satisfaction de certaines propriétés). A l'issue de cette étape, notre compréhension doit être suffisante pour nous permettre de réaliser l'environnement de test de l'algorithme. L'analyse d'une preuve formelle est cependant différente de celle d'une preuve informelle par plusieurs aspects. Les hypothèses déterminant le domaine d'entrée seront les axiomes de la preuve formelle, et les propriétés attendues (oracle) correspondront aux lemmes de plus haut niveau que cette preuve doit vérifier. Les ambiguïtés du discours informel ne sont donc plus un problème, mais la complexité d'une spécification formelle et sa taille peuvent rendre son analyse complexe. En particulier les axiomes peuvent être répartis dans différents fichiers de spécification.

- L'*Analyse détaillée* remplace la restructuration de la preuve informelle. Il s'agit ici d'obtenir une compréhension fine de la preuve formelle, et de la structure de son arbre de preuve à la fois global (lien entre les différents lemmes de la preuve) et local (arbre de preuve détaillé de chaque lemme). Contrairement au cas d'une preuve informelle, notre objectif ne sera pas d'établir des verdicts de validité de certaines branches de la preuve : la preuve étant formelle, les branches prouvées et non-prouvées sont déjà identifiées. Cette étape doit cependant permettre d'extraire des informations pour guider le test de l'algorithme à partir de l'analyse des branches en suspens. Nous distinguons deux niveaux d'analyse de la preuve formelle :
  - le niveau 1, ou *niveau des lemmes*, correspond à l'identification de la structure générale de la preuve et de la hiérarchie des lemmes et théories, ainsi que des lemmes de la preuve formelle qui restent en suspens ;
  - le niveau 2, ou *niveau des séquents*, correspond à l'identification précise des séquents en échec dans les arbres de preuve des lemmes non prouvés.

  L'investissement nécessaire, et la compétence requise en terme d'utilisation de l'outil de preuve considéré, ne sont pas du tout les mêmes selon que l'on se situe à l'un ou l'autre niveau d'analyse. L'approche que nous conseillons est de conduire tout d'abord une analyse de niveau 1, d'exploiter les informations correspondantes pour la conception du test, et de ne conduire l'analyse de niveau 2 que si les résultats sont insuffisants pour révéler une faute éventuelle.
- Le *Test guidé par la preuve* a des objectifs similaires, que notre méthode soit appliquée à une preuve formelle ou informelle. Les lacunes de la preuve sont analysées pour en déduire un ou plusieurs critères de test, et d'éventuelles extensions de l'oracle défini dans l'analyse de haut niveau. Comme précédemment, la mise en œuvre du test s'effectue selon l'approche test statistique, pour compenser l'imperfection des critères retenus. Selon les résultats de premières expériences de test (niveau 1), on pourra avoir à revenir sur l'étape d'analyse détaillée (niveau 2) pour extraire des informations plus précises de la preuve formelle, et conduire ensuite de nouvelles expériences de test.
- Le *Retour sur la preuve* a pour objectif de tenter de corriger les branches inachevées de la preuve formelle. Selon les résultats du test, faute révélée ou non, cette correction pourra nécessiter une modification de l'algorithme. Les contre-exemples éventuellement trouvés par test, pour les propriétés de haut niveau et/ou pour les lemmes intermédiaires, devraient aider à cette reprise.
  Dans le cadre de nos travaux, cette étape sera également l'occasion de discuter de l'impact de la structure de la preuve sur la faisabilité de notre méthode.

5.2 Injection d'une faute de conception

Une preuve informelle peut être publiée alors qu'elle est fausse. Son expression en langage naturel autorise des failles de raisonnement qui ont pu échapper à la fois

aux auteurs de la preuve et à ses relecteurs avant publication. Au contraire, dans le cas d'une preuve formelle, les étapes du raisonnement sont toutes conduites avec l'assistance d'un outil de preuve, qui ne laisse pas de place aux démonstrations implicites et aux hypothèses imprécises, principales causes d'erreurs dans les preuves classiques. Il n'est pas possible aux auteurs de la preuve de croire que leur preuve a abouti si tel n'est pas le cas. Comme les échecs de preuve ne donnent généralement pas lieu à publication, on trouvera difficilement dans la littérature des cas d'étude pour l'évaluation expérimentale de notre méthode.

Pour obtenir un exemple de preuve formelle inachevée, nous avons considéré deux possibilités :

- la construction d'une preuve formelle d'un algorithme contenant une faute de conception (application au FT-RMS),
- l'insertion d'une faute de conception dans un couple spécification formelle – preuve formelle complète (application au GMP).

Ces deux possibilités sont discutées aux Paragraphes 5.2.1 et 5.2.2. L'injection d'une faute de conception est la solution que nous avons finalement retenue.

5.2.1 Partir d'un algorithme incorrect

Notre première approche a été de tenter de développer une spécification et une preuve formelle d'un algorithme contenant une faute de conception connue. Pour cela, nous avons débuté une formalisation complète du FT-RMS présenté dans le Chapitre 3. Notre objectif était d'obtenir une preuve formelle partielle du FT-RMS qui soit plus aboutie que la preuve informelle, dont les faiblesses de trop haut niveau ne nous avaient pas permis de guider efficacement le test de l'algorithme. Rappelons que, d'après nos conclusions, la structure de la preuve informelle était inadéquate pour construire une démonstration de l'algorithme. Il nous a donc fallu réfléchir à une nouvelle structure de preuve, dans le cadre d'une formalisation du problème.

Comme nous l'avons mentionné dans le paragraphe 3.1.3, il existe un modèle PVS du FT-RMS (Version 1), développé par d'autres auteurs [Sinha & Suri 1999b]. Malheureusement, ce modèle ne peut servir de base à nos travaux : il est orienté vers le calcul symbolique (et non la preuve) de formules censées représenter le comportement de l'algorithme. Il ne comporte pas de description de l'algorithme lui-même. Pour notre part, nous aurions besoin d'une description de l'algorithme, de lemmes établissant une relation explicite entre cette description et le calcul de temps de réponse des tâches, et de théories mathématiques pour raisonner sur ces calculs (comme le calcul de points fixes).

Ces exigences sont par exemple couvertes par la spécification et la preuve en PVS du PCP (*Priority Ceiling Protocol*) présentée dans les travaux de B. Dutertre [Dutertre 2000]. Nous avons donc étudié si le développement du PCP pouvait être adapté au FT-RMS.

En nous calquant sur sa structure, nous avons pu développer un modèle des notions de bas niveau de l'algorithme, telles que le temps, et la génération de l'échéancier

des taches selon la première version du FT-RMS (la plus simple). Nous nous sommes ensuite heurtés à une difficulté majeure : l'établissement d'un lien entre les propriétés prouvées à ce niveau, et les propriétés de haut niveau attendues de l'algorithme. Nous avons réalisé que nous ne pourrions pas facilement utiliser les approches de preuve mises en œuvre par B. Dutertre, qui portent sur des propriétés différentes. Construire entièrement une nouvelle structure de preuve, avec le risque d'aboutir à un grand nombre de lemmes non pertinents en échec, nous a paru un effort démesuré par rapport aux enjeux de notre étude. Nous avons donc décidé de nous orienter vers une autre solution : l'insertion d'une faute de conception dans une preuve formelle aboutie.

5.2.2 Injecter une faute de conception dans une spécification et une preuve formelle complètes

Nous avons retenu le système de preuve PVS comme support à nos travaux[1]. Dans ce cadre, la modélisation formelle est structurée en *théories*, avec des lemmes utilisateurs à prouver. Un processus d'injection de faute dans un couple spécification – preuve formelles peut alors être décrit comme suit (Figure 5.1).

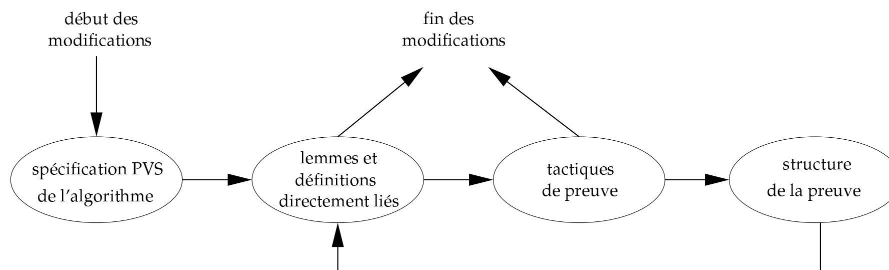


FIG. 5.1 – Cycle d'injection d'une faute dans le couple spécification-preuve

La faute introduite correspond à une modification de la description de l'algorithme. Cette modification peut être élémentaire (mutation) ou plus sophistiquée comme nous le verrons dans le paragraphe 5.5.4. Un certain nombre de lemmes deviennent alors non prouvés, ou mal définis. Il faut donc propager la modification à d'autres parties de la modélisation PVS.

L'ampleur de la propagation peut être plus ou moins importante, selon la faute injectée :

[1]Des environnements proches tels que HOL, ou Isabelle, pourraient faire l'objet d'extensions de nos travaux.

- Dans un premier temps, on cherche à faire correspondre les différents *lemmes* avec la version modifiée de l'algorithme. Il s'agit donc de répercuter la modification insérée dans tous les lemmes lui étant directement liés.
- Ensuite on souhaite prouver tous les lemmes qui peuvent l'être sans modification supplémentaire. On peut être pour cela amené à adapter les *tactiques* de preuves associées aux lemmes, et leurs points d'utilisation dans les preuves des lemmes.
- La dernière étape impacte plus profondément la *structure* de la preuve. La décomposition globale de la preuve en lemmes est revue, avec éventuellement l'invention de nouveaux lemmes. On entre alors dans un cycle de modifications qui nous renvoie à la première étape.

Le processus termine lorsque l'on obtient :

- une *preuve complète*, indiquant que la modification injectée ne modifie pas le comportement de l'algorithme vis à vis des propriétés qu'il doit assurer ; nous sommes alors devant le cas bien connu dans le cadre de l'analyse de mutation d'une *mutation équivalente* ;
- une *preuve incomplète* (une partie des lemmes restent non prouvés) que nous jugeons représentative d'une tentative de preuve de l'algorithme modifié ; en particulier, la preuve doit être «suffisamment achevée», c'est-à-dire que le nombre de lemmes en suspens doit être relativement réduit.

En pratique, nous arrêterons également le processus lorsque nous jugerons que l'obtention d'une preuve partielle réaliste nécessiterait une modification trop importante de la structure de la preuve.

5.3 Présentation du cas d'étude : GMP incorporé à la TTA

Au cours du développement de la TTA (*Time-Triggered Architecture*), son protocole de communication, le *Time-Triggered Protocol* (*c.f.* § 4.1) a évolué d'un seul protocole de communication à deux protocoles dissociés :

- le TTP/A, un protocole à faible bande passante optimisé pour les composants de faible coût, et
- le TTP/C, un protocole tolérant aux fautes et à haute sûreté de fonctionnement qui assure des services comme une synchronisation d'horloge, un protocole d'appartenance de groupe, et une détection rapide d'erreurs.

Plusieurs composants de la version actuelle du TTP/C [TTTech 2000] ont été vérifiés formellement : par exemple, les protocoles de synchronisation d'horloge [Pfeifer et al. 1999], et d'appartenance de groupe (*group membership*) [Pfeifer 2000] ont été prouvés à l'aide du système PVS.
C'est à cette modélisation du *Group Membership Protocol* que nous allons nous intéresser.

La criticité et la complexité du GMP, ainsi que sa délicate interaction avec les autres mécanismes du TTP/C (notamment la synchronisation d'horloges), ont amené le développement de plusieurs variantes successives. L'une d'entre elles [Katz et al. 1997] a fait l'objet du Chapitre 4. Mais cette version s'est avérée incorrecte, et n'a pas été intégrée dans TTP/C. L'algorithme que nous étudions dans ce chapitre est plus complexe et a lui-même connu des changements notables. Plusieurs versions sont présentées dans [Pfeifer 2000, Pfeifer & von Henke 2001] et [Pfeifer 2003]. C'est cette dernière version que nous avons étudiée. Sa spécification et sa preuve formelle complète sous PVS nous ont été fournies par son auteur : Holger Pfeifer de l'université de Ulm. Avant de présenter la description de l'algorithme, nous le comparons brièvement à celui étudié au chapitre précédent.

5.3.1 Différences avec le GMP informel

On retrouve, pour ce GMP, les hypothèses liées à l'architecture TTA. Le système est composé d'un anneau de `n` processeurs, numérotés de `0` à `n-1`, rattachés à un bus. On fait l'hypothèse d'une horloge globale, qui déclenche l'exécution synchrone de l'algorithme sur tous les processeurs. Chaque processeur accède au bus à tour de rôle, lorsque c'est son *slot* d'émission. Il reste cependant silencieux s'il s'est diagnostiqué comme défaillant. Au moins deux processeurs doivent être non-défaillants dans le système. Enfin, les fautes considérées sont toujours de deux types, faute d'émission et faute de réception, avec un comportement arbitraire après la manifestation d'une première faute.

Par rapport au GMP précédemment étudié, il y a cependant quelques différences sur les hypothèses de fautes. Tout d'abord, les fautes affectant des processeurs jusque-là non-défaillants doivent être espacées de $2n$ pas d'horloge (au lieu de $n + 1$). Ensuite, la modélisation des fautes d'émission et de réception est plus riche. Par exemple, pour une faute d'émission, on distingue le cas où 1) rien n'est émis (aucune activité n'est générée sur le bus), du cas où 2) quelque chose est émis qui ne sera pas interprété comme un message valide par les autres processeurs.

Le fonctionnement de l'algorithme lui-même est également plus complexe que précédemment. L'accord dans l'algorithme du Chapitre 4 était géré par un seul bit de *ack* ajouté aux émissions : dans ce nouvel algorithme, l'émetteur ajoute plusieurs bits de contrôle CRC calculés à partir des données du message et de sa vue interne du *membership*. Ces bits permettent aux récepteurs de comparer (dans une certaine mesure) leur *membership* à celui de l'émetteur. Outre ce mécanisme de contrôle, un deuxième mécanisme d'auto-diagnostic a été ajouté, basé sur des compteurs de messages acceptés ou rejetés.

5.3.2 Spécification de l'algorithme de GMP du TTP/C

Une explication très détaillée du fonctionnement du GMP peut être trouvée dans [Pfeifer 2003]. Nous nous bornerons ici à reproduire une description de l'algorithme

sous forme de commandes gardées ($garde \longrightarrow action$), et à en présenter les grandes lignes. La description de l'algorithme présentée reproduit fidèlement la spécification formelle du GMP dans le système PVS[2].

La Définition 5.1 présente les 14 commandes gardées extraites de la spécification formelle, et qui définissent le comportement d'un processeur p à la date t, selon son mode à ce pas d'horloge (émetteur, récepteur). Lorsque p est récepteur (gardes 3 à 14), l'émetteur du slot courant est le processeur b. Les gardes sont évaluées dans l'ordre, et le processeur p exécute uniquement l'action correspondant à la première garde vraie. Le *membership* de p à la date t est noté mem_p^t.

En mode réception, l'arrivée (ou non) d'un message détermine les variables d'entrée suivantes :

- $arrives_p^t$ est une variable Booléenne vraie si le processeur p reçoit un message valide à la date t.
- $null_p^t$ est une variable Booléenne vraie si p ne reçoit rien à la date t.
- mem_b^t est le *membership* émis par b et reçu par p (lorsque $arrives_p^t$ est vrai).

Notons que cette modélisation fait l'hypothèse conceptuelle qu'un message contient le *membership* de l'émetteur, alors qu'il contient en réalité un *checksum* CRC. Cette hypothèse est justifiée par le fait que le receveur peut calculer un *checksum* à partir des données émises et de sa propre vue du groupe, et comparer le résultat avec le *checksum* émis. Si les deux *checksums* sont égaux, alors le récepteur p peut conclure (avec une certaine probabilité, considérée comme suffisamment proche de 1) que les deux vues du groupe sont identiques, comme dans la garde (12). Dans le cas où une différence est observée, p ne peut pas directement déterminer l'identité des processeurs sur lesquels p et b sont en désaccord. Par contre, il peut essayer de reconstruire le *membership* de b, en faisant des calculs de CRC à partir de variantes de son propre *membership*, comme dans les gardes (4), (5), (8), (9).

Les commandes en mode réception peuvent être classées en 4 catégories, selon les variables internes considérées dans les gardes :

- commande (3), gardée par le fait que p ne soit plus dans son *membership*. Son état est alors figé.
- commandes (4) à (7), gardées par la variable Booléenne $prev_p^t$, qui vaut vrai si p considère qu'il est le dernier processeur du groupe à avoir émis un message valide sur le bus. $prev_p^t$ est positionnée à vrai par la commande (1) du mode émetteur, et repasse à faux par les commandes (4) et (5).
- commandes (8) à (11), gardées par la variable Booléenne $doubt_p^t$, qui vaut vrai si p considère qu'il a peut-être manifesté une faute d'émission. $doubt_p^t$ est positionnée à vrai par la commande (5) et repasse à faux lorsque que p est capable de décider qu'il n'a pas manifesté de faute d'émission (commande (8)), ou qu'il en a manifesté une et doit se retirer de son *membership* (commande (9)).

[2] Il y a de légères différences entre le source PVS que nous reproduisons, et sa présentation dans [Pfeifer 2003].

Définition 5.1 — Commandes gardées de l'algorithme de Group Membership du TTP/C

Émetteur :

(1)	$acc_p^t > rej_p^t \wedge$ $acc_p^t \geq 2$	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \wedge$ $prev_p^{t+1} = T \wedge$ $acc_p^{t+1} = 1 \wedge$ $rej_p^{t+1} = 0$
(2)	autre cas	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \setminus \{p\}$

Récepteur :

(3)	$p \notin mem_p^t$	$\longrightarrow$	pas de changement
(4)	$prev_p^t \wedge arrives_p^t \wedge$ $mem_b^t = mem_p^t \cup \{p, b\}$	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \wedge$ $prev_p^{t+1} = F \wedge$ $acc_p^{t+1} = acc_p^t + 1$
(5)	$prev_p^t \wedge arrives_p^t \wedge$ $mem_b^t = mem_p^t \cup \{b\} \setminus \{p\}$	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \setminus \{b\} \wedge$ $prev_p^{t+1} = F \wedge$ $doubt_p^{t+1} = T \wedge$ $rej_p^{t+1} = rej_p^t + 1 \wedge$ $succ_p^{t+1} = b$
(6)	$prev_p^t \wedge null_p^t$	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \setminus \{b\}$
(7)	$prev_p^t$	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \setminus \{b\} \wedge$ $rej_p^{t+1} = rej_p^t + 1$
(8)	$doubt_p^t \wedge arrives_p^t \wedge$ $mem_b^t = mem_p^t \cup \{p, b\} \setminus \{succ_p^t\}$	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \wedge$ $doubt_p^{t+1} = F \wedge$ $acc_p^{t+1} = acc_p^t + 1$
(9)	$doubt_p^t \wedge arrives_p^t \wedge$ $mem_b^t = mem_p^t \cup \{succ_p^t, b\} \setminus \{p\}$	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \cup \{succ_p^t\} \setminus \{p\} \wedge$ $doubt_p^{t+1} = F \wedge$ $acc_p^{t+1} = acc_p^t + 1$
(10)	$doubt_p^t \wedge null_p^t$	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \setminus \{b\}$
(11)	$doubt_p^t$	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \setminus \{b\} \wedge$ $rej_p^{t+1} = rej_p^t + 1$
(12)	$arrives_p^t \wedge (mem_p^t = mem_b^t)$	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \wedge$ $acc_p^{t+1} = acc_p^t + 1$
(13)	$null_p^t$	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \setminus \{b\}$
(14)	autre cas	$\longrightarrow$	$mem_p^{t+1} = mem_p^t \setminus \{b\} \wedge$ $rej_p^{t+1} = rej_p^t + 1$

- commandes (12) à (14), correspondant au cas standard où p est dans son *membership*, ne vient pas d'émettre, et n'a pas de doute sur sa dernière émission.

Un processeur émetteur exécutant la commande (1) considère toujours son émission comme ayant été valide jusqu'à ce que deux autres processeurs valides successifs aient manifesté, par l'émission de leur propre *membership*, leur désaccord avec son émission (par l'activation des commandes gardées (5) puis (9)).
L'accusé de réception des messages se fait donc implicitement par la diffusion du *membership*, et un émetteur ne peut pas savoir directement s'il a émis correctement : ce diagnostic se fait à l'aide des commandes gardées par $prev_p^t$ et $doubt_p^t$.

Par exemple, supposons que p manifeste une faute d'émission à la date t. En $t+1$, il reçoit un message valide du processeur b, et constate qu'il n'est plus dans le *membership* de b (garde (5)). Il en déduit alors que b n'a pas reçu son message, mais ne sait pas si cela provient d'une faute de réception de b, ou d'une faute d'émission de sa part. Par défaut, il décide alors d'exclure b de son *membership*, mais positionne sa variable *doubt* à vrai. Lorsqu'il recevra un deuxième message, il testera l'accord de ce deuxième émetteur sur les deux émissions précédentes (la sienne et celle de l'émetteur *succ* après lui). Si ce message lui confirme que son émission était incorrecte, il se retirera alors de son propre *membership* en exécutant la commande (9), sinon il sera assuré de sa propre validité et restera dans son *membership* (commande (8)).

Ce mécanisme n'est cependant pas suffisant pour garantir l'auto-diagnostic dans tous les cas de fautes. C'est pourquoi p gère aussi deux compteurs, *acc* et rej, représentant le nombre de messages acceptés ou rejetés depuis sa dernière émission. De façon générale, *acc* est incrémenté lorsque p reçoit un message valide, et qu'il est en accord avec le *membership* de l'émetteur, et rej est incrémenté lorsque p reçoit un message invalide, ou un message valide avec lequel il est en désaccord sur le *membership*.
Les valeurs des compteurs sont comparées périodiquement, à chaque émission (gardes des commandes (1) et (2)). Elles permettent à p de se diagnostiquer comme défaillant s'il a accepté moins (ou autant) de messages qu'il n'en a rejetés, ou s'il n'en a accepté aucun (c'est-à-dire si $acc_p^t < 2$, *acc* ayant été remis à 1 au *slot* de sa précédente émission). Les gardes des commandes (1) et (2) ont été intégrées dans un prédicat particulier de la spécification formelle donné ci-dessous :

```
no_cliques(t, p): bool =
    accepted(t, p) > rejected(t, p) ∧ accepted(t, p) ≥ 2
```

FIG. 5.2 – PVS code : définition du prédicat `no_cliques` dans la théorie membership_verification

Nous allons maintenant aborder notre méthode elle-même en commençant par l'analyse de haut niveau de ce GMP.

5.4 Analyse de haut niveau

Nous devons extraire, de la spécification formelle du GMP, les informations nécessaires au développement d'un environnement complet de test : prototype, domaine d'entrée, oracle.

L'implémentation d'un prototype du GMP à partir de la description détaillée de l'algorithme ne présente pas de difficulté. Le code que nous avons développé est une

transcription quasi littérale, en langage C, de la spécification PVS correspondant à l'algorithme tel qu'il est définit par les 14 gardes de la Définition 5.1 , page 111. Par contre, la détermination du domaine d'entrée et de l'oracle a nécessité plus de travail.

5.4.1 Modèle de fautes et domaine d'entrée

5.4.1.1 Les axiomes du modèle de fautes

L'identification des hypothèses d'entrée, et en particulier du modèle de fautes, est faite en extrayant tous les axiomes des différentes théories de la modélisation formelle. Nous ne reproduisons pas ici l'intégralité de ces axiomes, nous nous limitons à ceux essentiels à la compréhension de notre discours. Le comportement des processeurs lors d'une faute d'émission est défini par les deux axiomes `faulty_broadcaster` et `nonarrival` reproduits ci-après. Dans l'expression de `nonarrival`, *arrivesOK(t,p)* et *null(t,p)* correspondent respectivement aux entrées notées $arrives_p^t$ et $null_p^t$ dans la description précédente de l'algorithme. Les axiomes définissent une propriété de cohérence des fautes d'émission : aucun processeur ne peut recevoir de message valide sur une faute d'émission ; si l'émetteur n'a généré aucune activité sur le bus (*nomsg*), alors tous les processeurs non défaillants ont leur entrée *null* à vraie.

```
faulty_broadcaster: AXIOM
  LET b = broadcaster(t + 1) IN
    ¬ faulty(t, b) ∧ faulty(t + 1, b) ⊃ ¬ sendsOK(t + 1, b)

nonarrival: AXIOM
  LET b = broadcaster(t) IN
    (¬ sendsOK(t, b) ⊃ ¬ arrivesOK(t, p)) ∧
     (nomsg(t, b) ∧ ¬ faulty(t, p) ⊃ null(t, p))
```

FIG. 5.3 – PVS code : axiomes définissant une faute d'émission dans la théorie membership_fault_model

Un corollaire de ces axiomes (et d'autres axiomes définissant la réception d'une émission correcte) est que tous les processeurs p non-défaillants ont toujours des entrées $arrives_p^t$ cohérentes (corollaire `all_or_none`).

```
all_or_none: COROLLARY
  ¬ faulty(t, p) ∧ ¬ faulty(t, q) ⊃ arrivesOK(t, p) = arrivesOK(t, q)
```

FIG. 5.4 – PVS code : corollaire all_or_none dans la théorie membership_faultmodel

Une faute de réception est (notamment) définie par l'axiome `faulty_receiver`: le processeur qui manifeste cette faute a son entrée $arrives_p^t$ à faux.

```
faulty_receiver: AXIOM
  LET b = broadcaster(t + 1) IN
    ¬ faulty(t, p) ∧ faulty(t + 1, p) ∧ p ≠ b ⊃
     ¬ faulty(t + 1, b) ∧ ¬ arrivesOK(t + 1, p)
```

FIG. 5.5 – PVS code : axiome définissant une faute de réception dans la théorie membership_fault_model

5.4.1.2 Résultat de l'analyse des axiomes

En analysant l'ensemble des axiomes extraits de la modélisation PVS, nous nous sommes aperçus d'un problème. Selon ces axiomes, la situation où un processeur reçoit à la fois $null_p^t$ et $arrives_p^t$ n'est pas exclue. Une telle situation voudrait dire que le bus a été à la fois vide et occupé par un message valide, ce qui est physiquement incohérent. Ces axiomes, s'ils sont suffisants à la preuve formelle, sous-spécifient notre domaine d'entrée en terme de comportement des processeurs par rapport au bus de données.

Nous avons été amenés à faire des choix de spécification vis à vis de ce problème. Dans notre domaine d'entrée, les processeurs non-défaillants (ou les processeurs défaillants qui ne manifestent pas de faute à cette date) ne peuvent recevoir $null$ que si rien n'a été émis sur le bus. Une telle émission nulle peut être due soit à une faute d'émission, soit au fait qu'un processeur reste volontairement silencieux (car il s'est déjà diagnostiqué comme défaillant). Par contre, nous autorisons que le comportement d'un processeur lors d'une faute de réception soit indépendant de l'émission à cette date. Il peut recevoir $null$ alors que l'émission a été valide, tout comme il peut croire détecter de l'activité sur le bus ($null$ est faux) lors du slot d'un processeur silencieux (il croit alors recevoir une trame invalide). Tous ces comportements sont possibles d'après les axiomes spécifiés.

Les choix réalisés nous amènent à distinguer quatre types de fautes, présentées dans le Tableau 5.1. Dans cette table, p représente un processeur qui manifeste une faute et q tout autre processeur ne manifestant pas de faute à cette date.

5.4.1.3 Définition du domaine d'entrée de test

A partir de l'analyse précédente, nous pouvons maintenant définir le *domaine d'entrée du test* pour le GMP.
Une séquence d'entrée de test est définie par le nombre `n` de processeurs dans le

TAB. 5.1 – Fautes générées par l'environnement de test

type de faute	nom	caractéristiques du point de vue de l'algorithme
émission	*no_msg*	les q reçoivent : $\neg arrives_q^t \wedge null_q^t$
	not_no_msg	les q reçoivent : $\neg arrives_q^t \wedge \neg null_q^t$
réception	*null*	p reçoit : $\neg arrives_p^t \wedge null_p^t$
	not_null	p reçoit : $\neg arrives_p^t \wedge \neg null_p^t$

système (avec `n>2`), et par la liste complète des fautes affectant le système. Chaque faute est caractérisée par sa date d'occurrence, le processeur affecté, et le type de faute (parmi les 4 définis dans le Tableau 5.1). Comme dans le cas du GMP informel, on distingue les fautes initiales (affectant des processeurs jusque-là non-défaillants) des fautes ultérieures (manifestées après une faute initiale). La liste de fautes doit vérifier les hypothèses de fautes suivantes :

- elle comporte uniquement des fautes d'émission ou de réception telles que décrites dans le Tableau 5.1 ; le type de faute doit être conforme à sa date d'occurrence et au processeur affecté (ex : une faute d'émission affecte le processeur émetteur à ce slot) ;
- il y a toujours au moins 2 processeurs non-défaillants, c'est-à-dire que le nombre de fautes initiales dans la séquence est inférieur ou égal à $n-2$;
- il y a au moins `2n` pas d'horloges entre deux fautes initiales ;
- une faute initiale a toujours lieu lors du slot d'un processeur non-défaillant jusque-là ; ce sera soit le *slot* du processeur devenant défaillant (faute d'émission), soit le *slot* d'un autre processeur non-défaillant (faute de réception) ;
- après une faute initiale, le processeur peut être arbitrairement affecté par tout type de faute ultérieure, ou se comporter correctement à certaines dates.

5.4.2 Propriétés attendues et oracle de test

5.4.2.1 Propriétés de haut niveau devant être garanties par le GMP

Sous les hypothèses présentées précédemment, et avec l'algorithme décrit au paragraphe 5.3.2, le GMP doit garantir les propriétés suivantes à tout instant.

Validité des vues locales du groupe (Figure 5.6) :

Un processeur non-défaillant doit avoir tous les processeurs non-défaillants dans son *membership*, et au plus un processeur défaillant (pour prendre en compte le fait que le diagnostic n'est pas instantané). Un processeur défaillant doit s'être retiré de son *membership*, ou s'il en fait encore partie, son *membership* ne doit contenir que lui-même et un sous-ensemble des processeurs non-défaillants.

```
validity(t): bool =
    (∀ (p: (faulty?(t))):
        ¬ membership(t, p)(p) ∨ (membership(t,p) ⊆ (the_mem(t) ∪ {p})))
      ∧
    (∀ (p: (λ (x: Proc): ¬ faulty?(t)(x))):
        membership(t, p) = the_mem(t) ∨
          (∃ x: faulty(t, x) ∧ membership(t, p) = (the_mem(t) ∪ {x})))

validity: THEOREM validity(t)
```

FIG. 5.6 – PVS code : lemme validity dans la théorie membership_verification

Dans la définition PVS (Figure 5.6), *the_mem* est un artefact de la modélisation correspondant à une vue globale (et exacte) du groupe des processeurs non-défaillants ; $membership(t,p)$ correspond à la variable interne mem_p^t dans la description de l'algorithme ; $membership(t,p)(p)$ correspond au prédicat $p \in mem_p^t$.

Accord sur les membres du groupe (Figure 5.7) :

Tous les processeurs non-défaillants doivent avoir le même *membership*.

```
agreement(t): bool =
    ∀ (p, q: (λ (x: Proc): ¬ faulty?(t)(x))):
      membership(t, p) = membership(t, q)

agreement: THEOREM agreement(t)
```

FIG. 5.7 – PVS code : lemme agreement dans la théorie membership_verification

Auto-diagnostic en temps borné (Figure 5.8) :

Un processeur qui devient défaillant doit détecter sa faute et se retirer de son *membership* en moins de 2n pas d'horloge.

```
self_diagnosis(t): bool =
    ∀ x:
      ¬ faulty(t, x) ∧ faulty(t + 1, x) ⇒
        (∃ s:
           0 < s ∧ s ≤ 2 × n ∧
             ¬ membership(t + s, x)(x))

self_diagnosis: COROLLARY self_diagnosis(t)
```

FIG. 5.8 – PVS code : lemme self_diagnosis dans la théorie membership_verification

5.4.2.2 Extension des propriétés

L'analyse de ces trois propriétés nous amène à constater un manque : si l'*auto-diagnostic* est requis en temps borné ($< 2n$ pas d'horloge), il n'est pas explicitement requis que le *diagnostic* par des processeurs non-défaillants soit également fait en temps borné. Notons que, dans le GMP informel, le diagnostic était garanti par le Théorème 2. En nous basant sur l'expression de la propriété d'auto-diagnostic du GMP formel, nous proposons d'ajouter la propriété suivante.

Diagnostic en temps borné :

Tous les processeurs non-défaillants doivent retirer de leur *membership* un processeur défaillant en moins de 2n pas d'horloge.

```
diagnosis(t): bool =
    ∀ x:
      ¬ faulty(t, x) ∧ faulty(t + 1, x) ⇒
       (∃ s:
          0 < s ∧ s ≤ 2 × n ∧
            (∀ (p: (λ (x: Proc): ¬ faulty?(t + s)(x))):
               ¬ membership(t + s, p)(x)))

diagnosis: COROLLARY diagnosis(t)
```

FIG. 5.9 – PVS code : lemme diagnosis ajouté à la théorie membership_verification

En fait, l'analyse de la preuve nous amène à penser qu'un lemme intermédiaire, utilisé pour prouver l'auto-diagnostic, devrait aussi permettre de prouver le diagnostic. Nous avons alors inclus cette propriété dans le modèle PVS : effectivement, nous avons réussi à terminer sa preuve (*c.f.* Annexe).

5.4.2.3 Définition de l'oracle de test

Notre *oracle* de test intègre les trois propriétés de validité, accord et auto-diagnostic définies par l'auteur de la preuve, plus la propriété de diagnostic que nous venons de définir.

Son implémentation s'est strictement basée sur les définitions PVS de ces quatre propriétés. Elle nécessite l'observation ou le calcul de plusieurs paramètres :

- l'observation des *memberships* de tous les processeurs à chaque pas d'horloge ;
- la reconstitution de *the_mem* à partir de la lecture du fichier d'entrée ;
- l'identification de la date à partir de laquelle un processeur devient défaillant, pour calculer les temps de diagnostic et d'auto-diagnostic.

Notons qu'on peut, sans perte de généralité, limiter la fenêtre temporelle d'observation à $[0...t_{lastf} + 2n - 1]$, où t_{lastf} est la date de la dernière faute initiale de la séquence de test.

5.4.3 Test aléatoire aveugle de la deuxième version du GMP

A l'issue de cette étape d'analyse, nous avons défini un environnement de test pour le GMP. Nous allons maintenant conduire de premières expériences de test selon un profil aléatoire aveugle sur le domaine d'entrée.

Par rapport au domaine d'entrée complet, le profil que nous avons implémenté rajoute des hypothèses, de façon à limiter la longueur des expériences.

Notamment, les systèmes que l'on simule comprennent de 3 à 20 processeurs, cette borne haute étant représentative des systèmes auxquels le GMP est destiné. De plus, après une faute initiale, les fautes ultérieures ne sont plus générées après $2n$ pas d'horloge : si aucune violation de propriété n'est observée, le processeur concerné aura été diagnostiqué par tous les autres processeurs (y compris lui-même) et n'influera plus sur le comportement du système.

L'algorithme correspondant est présenté ci-après.

Définition 5.2 — Algorithme de génération des séquences d'entrées aléatoires aveugles

1. tirer au hasard le nombre n de processeurs (entre 3 et 20)
2. tirer au hasard le nombre de fautes initiales (entre 1 et n-2)
3. tant que l'on doit générer des fautes :
   - tirer au hasard l'instant de la faute initiale t_f (slot d'un processeur non défaillant entre t et t+n)
   - tirer au hasard le processeur affecté parmi les non-défaillants
   - tirer au hasard le type de faute, selon le mode du processeur affecté (émetteur, ou récepteur) à la date tirée
   - tant que $t < t_f + 2n$ générer le comportement ultérieur :
     - 50% de chance de manifester une faute en t,
     - si manifestation de faute : tirer au hasard le type de faute selon le mode du processeur affecté.

Par précaution, nous avons également introduit, au niveau de l'oracle, une vérification systématique des axiomes PVS définissant le modèle de faute, de façon à pouvoir détecter des séquences de test qui seraient invalides selon les hypothèses.

Nous n'avons pas constaté de défaillance du GMP selon notre profil aveugle. Ce résultat n'est pas surprenant, puisqu'en conformité avec le fait que la preuve formelle de l'algorithme ait pu aboutir. Ces premiers tests nous ont cependant permis une première validation de notre prototype du GMP.

5.5 Analyse détaillée

La preuve formelle du GMP *Group Membership Protocol* vise à démontrer que, à tout instant, les trois propriétés de Validité, Accord et Auto-diagnostic sont assurées par l'algorithme. Elle a été développée à l'aide du système de preuve PVS [Owre et al. 1995]. Cette preuve contient 28 théories pour 348 obligations de preuve, et environ 2500 lignes de code PVS. Nous allons nous concentrer dans cette étude sur les théories de plus haut niveau dans la hiérarchie de la preuve formelle. Ces théories regroupent notamment les propriétés invariantes devant être assurées par l'algorithme. La preuve du GMP a été développée selon une approche originale, dite des *invariants disjonctifs*, que nous présenterons au paragraphe 5.5.1. La structure de preuve résultante sera expliquée aux paragraphes 5.5.2 et 5.5.3. Nous mettons ensuite en œuvre le processus d'injection de faute de conception, pour obtenir une preuve partielle comportant quelques lemmes en échec (§ 5.5.4).

5.5.1 Technique des invariants disjonctifs

Les approches classiques de preuve d'invariant consistent à renforcer cet invariant par l'adjonction d'invariants auxiliaires, de façon itérative, jusqu'à obtenir un invariant qui soit inductif. Nous avons vu un exemple d'une telle preuve pour la version de GMP proposée dans [Katz et al. 1997] et étudiée dans le Chapitre 4. Dans [Rushby 2000], J. Rushby reconnaît que l'invariant de la preuve informelle s'est avéré difficile à développer :

> “ The informal proof of inductiveness of the conjoined invariants is long and arduous. ”

Le fait que cette preuve ait laissé passer une faute de conception a poussé J. Rushby à en développer une formalisation complète. Mais le nombre et la complexité des invariants auxiliaires, ainsi que l'explosion des cas, n'ont pas permis d'achever la preuve.

Après plusieurs tentatives infructueuses pour établir une preuve inductive, l'auteur propose dans [Rushby 2000] une approche différente, nommée technique des invariants disjonctifs (*Disjunctive Invariants*), qui s'est révélée efficace pour construire une preuve formelle complète d'une version simplifiée de l'algorithme (sans la propriété d'auto-diagnostic). Cette méthode partage des points communs avec les *Diagrammes de Vérification* de Manna et Pnueli [Manna & Pnueli 1994]. Le principe est de renforcer une propriété de sûreté à prouver en la décomposant en une disjonction de “configurations” pour lesquelles la propriété pourra facilement être démontrée inductive.

Les transitions entre les configurations forment naturellement une représentation sous la forme d'un diagramme qui donne un aperçu du fonctionnement du programme : le *Diagramme des Configurations*.

Cette technique dite des *Invariants Disjonctifs* a été réutilisée avec succès pour prouver le GMP du TTP/C que nous avons étudié [Pfeifer 2000, Pfeifer 2003], ainsi que la partie concernant la synchronisation d'horloge du TTP/C (utilisée en hypothèse du protocole d'appartenance de groupe) [Pfeifer et al. 1999].

5.5.2 Diagramme des configurations du GMP

La preuve formelle développée se base donc sur cette méthode des invariants disjonctifs. Les auteurs ont développé une représentation du comportement de l'algorithme sous la forme d'un *Diagramme des Configurations*. Ce diagramme pour le GMP est présenté graphiquement Figure 5.10, une description formelle en est donnée dans la théorie `membership_spec` de la spécification PVS. On pourra se reporter à [Pfeifer 2003] pour une analyse détaillée de ce graphique et de sa formalisation PVS.
Le lecteur pourra reconnaître dans ce diagramme un certain nombre d'états de l'automate du GMP présenté dans le Chapitre 4 (Figure 4.12). La version originale de l'automate du GMP étudié dans le Chapitre 4 présentait des noms d'états différents lorsque nous l'avons développée ; mais nous avons modifié ces notations pour faciliter la comparaison entre les automates des deux variantes de l'algorithme.

Le diagramme est un automate symbolique censé représenter une abstraction du comportement global du système, conséquence de l'exécution locale du GMP par un nombre `n` non borné de processeurs. Un état, ou configuration, est défini par un prédicat sur les variables internes des différents processeurs, en distinguant le dernier processeur devenu défaillant (noté `x`), le dernier émetteur valide (noté `z`), et l'émetteur courant (noté `b`). Toute transition correspond à un pas d'horloge, et est elle aussi associée à un prédicat (la condition de transition).
La construction du diagramme est itérative. On commence par définir une configuration initiale (ici, stable). Partant d'une configuration existante, on propose alors un ensemble de conditions de transition, non nécessairement exclusives, mais dont la disjonction est complète. Pour chaque condition, on simule une exécution de l'algorithme par réécriture et simplification. Le résultat est analysé manuellement pour déterminer s'il faut définir une nouvelle configuration d'arrivée, ou si la transition amène dans une configuration déjà existante (ou éventuellement sa généralisation). Le processus termine lorsque le diagramme est clos.
Il est à noter que dans l'automate présenté Figure 5.10, nous avons fait abstraction des transitions réflexives sur les états représentant les pas d'horloge correspondant aux *slots* de processeurs déjà diagnostiqués comme défaillants (*dead steps*).

Dans l'état stable (*c.f.* Figure 5.11 pour une définition formelle), tous les processeurs défaillants ont été diagnostiqués : ils se sont exclus de leur propre *membership* ; ils ont

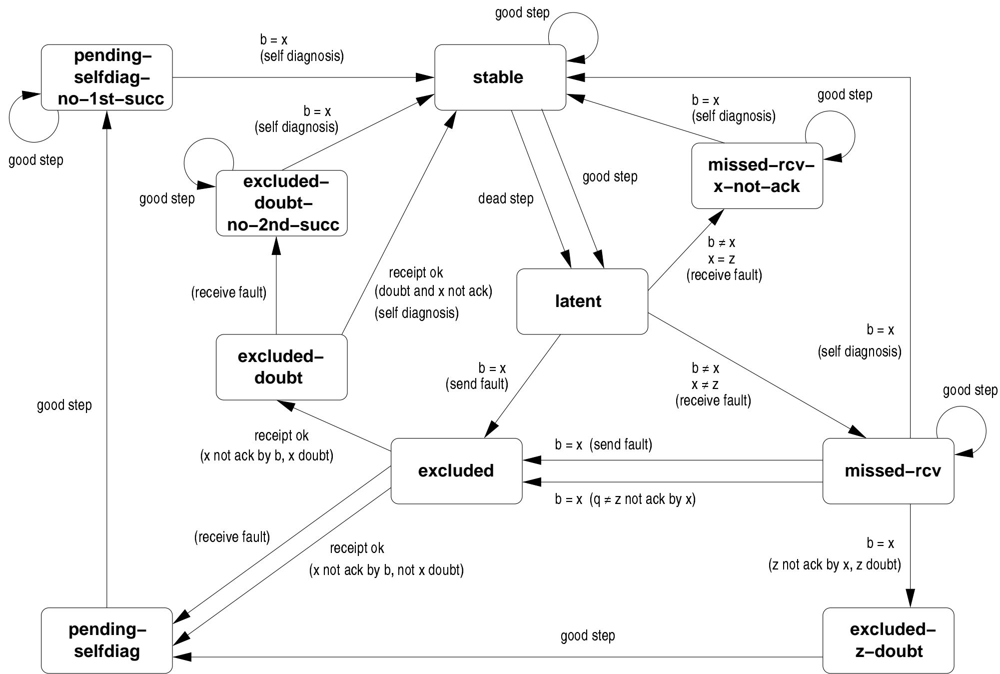


FIG. 5.10 – Diagramme des Configurations du Group Membership Protocol

été exclus des *memberships* des processeurs non-défaillants. Les *memberships* des processeurs non-défaillants contiennent tous les processeurs non-défaillants, et seulement eux.

```
stable(t, z): bool =
    recent(t, z) ∧
     (∀ p:
        IF faulty(t, p)
          THEN ¬ membership(t, p)(p)
        ELSE membership(t, p) = the_mem(t) ∧
             IF p = z
               THEN accepted(t, p) = 1 ∧ rejected(t, p) = 0
             ELSE accepted(t, p) > rejected(t, p) + 1
             ENDIF
             ∧ prev(t, p) = (p = z)
             ∧ doubt(t, p) = FALSE
        ENDIF)
```

FIG. 5.11 – PVS code : definition de l'état stable dans la théorie membership_verification

L'état `latent` correspond à l'occurrence d'une faute qui va se manifester au prochain pas d'horloge (*c.f.* transitions sortantes) et va affecter un processeur (`x`) jusque-là non défaillant.

Un chemin de type stable $\longrightarrow$ latent $\longrightarrow \ldots \longrightarrow$ stable caractérise donc un scénario de diagnostic de faute.

5.5.3 Présentation de la preuve formelle du GMP

De façon générale, la preuve formelle des propriétés de haut niveau d'un algorithme n'est jamais conduite d'un bloc. Des propriétés intermédiaires (lemmes) sont introduites pour décomposer les preuves.
Dans le cas du GMP, la décomposition de la preuve est déterminée par la modélisation sous forme de diagramme des configurations. Cette modélisation amène à définir des propriétés sur chaque configuration (état) du diagramme, et à introduire des lemmes propres à un tel modèle : lemmes de transition entre deux états du diagramme, lemmes de complétude du diagramme, et lemmes de vivacité de chaque état (qui assurent qu'il n'y a pas de blocage). A ces lemmes de «haut-niveau» s'ajoutent également toutes les propriétés de plus bas niveau utilisées elles-mêmes pour les prouver (telles que, par exemple, la définition des relations sur les ensembles nécessaires pour définir les gardes des commandes (4), (5), (8) et (9) de l'algorithme).

Comme nous l'avons déjà indiqué, la preuve formelle complète du GMP comprend en tout 28 théories différentes, générant 348 obligations de preuve (lemmes et preuves de typage).
Ces théories peuvent être regroupées selon ce qu'elles spécifient. Par exemple, il y a plusieurs théories définissant des notions mathématiques pour pouvoir traiter les preuves par induction sur le temps dans un système synchrone.
Nous ne présenterons ici que les théories de haut niveau, théories servant principalement à l'introduction des hypothèses de la preuve formelle et à la spécification de l'algorithme et de ses propriétés :

- La principale théorie est `membership_verification`, elle regroupe les trois propriétés attendues, mais surtout l'ensemble des définitions et lemmes liés aux preuves de haut niveau du diagramme des configurations du GMP.
- La théorie `membership_spec` comprend la spécification complète des 14 commandes gardées définissant le comportement de l'algorithme, la définition de l'état initial du système, et la définition de la relation de transition temporelle entre deux pas d'horloge qui fait évoluer le système.
- Les théories `membership_faultmodel`, `membership_faultmodel_defs`, et `membership_faultmodel_props` définissent l'ensemble des hypothèses de fautes ainsi que le lien entre les processeurs et le bus de communication.

Technique des invariants disjonctifs

La preuve complète du GMP est alors décomposée selon le principe des *invariants disjonctifs*.

Les auteurs prouvent à l'aide de PVS l'invariance des propriétés de validité et d'accord dans chacune des configurations générées. Ces preuves sont de taille raisonnable, l'explosion combinatoire étant fortement limitée par l'ensemble des variables fixées par la configuration considérée.
Chaque transition entre deux états du diagramme est ensuite prouvée. Cette étape consiste à prouver que, partant de l'état initial de la transition, le système va dans l'état d'arrivée, sous les hypothèses de franchissement de la transition.
Il faut ensuite prouver la complétude du diagramme des configurations. Cette étape consiste à prouver que, partant de l'état stable, le système ne peut pas arriver dans un état non défini dans la spécification formelle.
Ces trois étapes permettent d'assurer que les invariants définis sur chaque état du diagramme sont vrais de façon globale pour l'algorithme. Ces invariants correspondent aux deux premières propriétés attendues de l'algorithme, soit la *Validité* et l'*Accord*.

Il reste encore à prouver la propriété d'*Auto-diagnostic*. Cela se fait par la preuve que le système ne peut rester en dehors de l'état stable pendant plus de `2n` pas d'horloge, on peut noter que cela correspond à une propriété de *vivacité* du diagramme et la définition d'une borne supérieure sur son temps de parcours.

Ces différentes étapes ont été suivies et achevées avec succès pour la version du GMP présentée dans [Pfeifer 2003].

Remarques sur la preuve formelle

Lors de l'analyse de haut niveau, nous avons mentionné l'hypothèse qu'il y a toujours au moins `2n` pas d'horloge entre deux fautes initiales (affectant des processeurs jusqu'alors non-défaillants). Cette hypothèse n'est en fait pas donnée en tant que telle. Elle est retranscrite par l'axiome `no_faults_when_not_stable`, qui doit être associé à la propriété de vivacité et à la définition de l'état stable (*c.f.* § 5.5.2) pour exprimer l'hypothèse liée au temps `2n` entre deux fautes. On voit là que, dès l'analyse de haut niveau, une certaine compréhension de la structure de la preuve était nécessaire.

Notons que la propriété de *Diagnostic en temps borné* que nous avons ajoutée à l'oracle de test (et à la modélisation PVS) est une conséquence directe de la propriété de vivacité, tout comme l'*Auto-diagnostic*.

5.5.4 Injection d'une faute de conception dans la spécification formelle de l'algorithme

Pour pouvoir étudier notre approche de Test Guidé par la Preuve, il nous est nécessaire d'injecter une faute de conception dans la spécification formelle du GMP, dans le but d'obtenir une preuve formelle partielle.

Nous expliquons d'abord comment nous avons effectué le choix de la faute de conception à injecter, puis nous présentons la preuve formelle partielle résultante.

5.5.4.1 Choix d'une expérience d'injection

Nous avons envisagé plusieurs possibilités pour définir la faute de conception que nous allions injecter dans l'algorithme, et donc la modification que nous devions apporter à sa spécification formelle, contenue dans le fichier de théorie `membership_spec` :

- une mutation élémentaire de l'algorithme ;
- une modification correspondant à une différence entre cet algorithme et celui présenté dans le Chapitre 4 ;
- les différences entre la spécification formelle PVS et la présentation informelle de l'algorithme dans [Pfeifer 2003] ;
- une modification apportée entre la version finale et des versions antérieures, et décrite dans [Pfeifer 2003].

Nous avons évalué ces différentes fautes de conceptions selon plusieurs critères, notre objectif étant d'obtenir un algorithme et une preuve formelle présentant certaines caractéristiques : 1) d'une part la faute de conception insérée dans l'algorithme ne doit pas être trop facile à révéler par un test aléatoire aveugle (le taux de défaillance sous ce profil doit être faible), 2) d'autre part la nouvelle preuve formelle liée à notre algorithme modifié doit être partielle et représentative d'un échec de preuve.
Le choix de la faute a procédé en plusieurs temps :

- Nous avons d'abord injecté les différentes fautes de conception envisagées dans notre prototype, et avons éliminé celles qui faisaient défaillir l'algorithme trop fréquemment lors du test aléatoire aveugle.
- Ensuite, nous avons vérifié si les fautes de conception qui n'étaient pas révélées par le test aveugle correspondaient à des versions équivalentes de l'algorithme, vis-à-vis des propriétés attendues (problème bien connu des mutations équivalentes). Pour cela, nous avons modifié la spécification formelle et réalisé les étapes présentées dans le paragraphe 5.2.2. Toute modification de l'algorithme résultant en une preuve formelle complète était alors écartée, car présentant le même comportement vis à vis des propriétés devant être garanties par l'algorithme.
- Nous avons également écarté les fautes qui nécessitaient une reprise importante de la structure de la preuve (en particulier, des modifications majeures du diagramme des configurations).

Nous citons trois exemples de telles modifications de l'algorithme (se reporter à la description de l'algorithme par ses commandes gardées page 111) :

- La modification de la garde (12) de l'algorithme, en supprimant la condition $(mem_p^t = mem_b^t)$ nous donne un taux de défaillance de 75% avec un profil de test aléatoire aveugle ; cette faute de conception est donc trop grossière pour être d'une étude pertinente.

- La suppression de l'incrémentation du compteur *rej* dans la commande (7) de l'algorithme résulte en un taux de défaillance nul par le test aveugle, et la preuve formelle correspondant à la spécification modifiée peut être complètement achevée. C'est donc une mutation équivalente et elle a été écartée.
- La suppression des commandes gardées (4) à (11), c'est-à-dire de l'auto-diagnostic par deux désaccords successifs (limitant l'auto-diagnostic aux compteurs *acc* et *rej*) résulte également en un taux de défaillance nul par le test aléatoire uniforme. Cependant, elle entraîne une modification majeure du diagramme des configurations, et impliquerait de ce fait une modification également majeure de la structure de la preuve formelle. Cette modification a donc aussi été écartée.

Après un certain nombre d'expériences (environ une dizaine), notre choix s'est finalement porté sur la modification décrite dans [Pfeifer 2003]. Elle ne correspond pas à une modification équivalente (preuve formelle inachevée) et donne un taux de défaillance de 0,6% avec un test aléatoire aveugle. De plus, elle représente une étape des évolutions de l'algorithme vers sa version finale et, en tant que telle, est représentative d'un algorithme incorrect pouvant résulter en une preuve formelle partielle.

5.5.4.2 Modifications du couple spécification et preuve formelles

La faute injectée consiste à modifier le prédicat `no_cliques` (*c.f.* Figure 5.2, § 5.3.2) qui détermine le comportement associé aux gardes (1) et (2) de l'algorithme. La modification correspond à un affaiblissement du prédicat :

$$\begin{array}{c} acc_p^t > rej_p^t \wedge \\ acc_p^t \geq 2 \end{array} \quad \text{devient} \quad acc_p^t > rej_p^t \tag{5.1}$$

Cette faute spécifique correspond à une des modifications apportées lors du passage des premières versions PVS de l'algorithme [Pfeifer 2000, Pfeifer & von Henke 2001] à sa dernière version [Pfeifer 2003]. La deuxième partie de la garde ($acc_p^t \geq 2$) a été identifiée par les auteurs comme nécessaire pour éviter une défaillance dans un scénario particulier de fonctionnement.
Ce scénario correspond à une activation spécifique du chemin stable → latent → missed-rcv-x-not-ack → stable, c'est-à-dire à une faute de réception sur le dernier émetteur valide à l'instant de la faute. Le processeur fautif x doit également manifester un comportement particulier après sa première faute : x doit faire des fautes de réception de type $null$ à chaque pas d'horloge.
Dans la version correcte, la dernière transition du chemin correspond à l'activation par le processeur x de la commande (2), la garde de (1) étant fausse. Dans la version que nous avons modifiée, c'est la commande (1) qui est activée, le processeur x ne se diagnostique pas comme défaillant, et le comportement du système sort du diagramme des configurations. La propriété d'auto-diagnostic en temps borné est alors violée.

Pour obtenir une preuve partielle réaliste, il ne suffit pas d'injecter la faute au niveau de la description PVS de l'algorithme (dans la théorie membership_spec), il faut aussi propager la modification à d'autres parties de la spécification formelle, ainsi qu'aux preuves réalisées à partir de cette spécification (*c.f.* § 5.2.2). Une première adaptation des différents lemmes impactés par la modification de l'algorithme, ainsi que des tactiques de preuve associées à leurs preuves, nous a permis de reconstruire l'ensemble de l'arbre de preuve, à l'exception de trois lemmes restant non-prouvés. Ces trois lemmes correspondent aux preuves des transitions suivantes :

- pending-selfdiag-no-1st-succ → stable,
- excluded-doubt-no-2nd-succ → stable,
- missed-rcv-x-not-ack → stable.

Après une analyse plus poussée, la deuxième transition a cependant pu être prouvée. Pour cela nous avons modifié les invariants de trois états du diagramme des configurations : excluded, excluded-doubt, et excluded-doubt-no-2nd-succ, puis tous les lemmes impactés par ces modifications. L'ensemble des preuves liées à ces états et lemmes (notamment toutes les transitions) ont du être modifiées pour être en accord avec les modifications apportées. Après ces modifications, il a été possible de prouver la transition excluded-doubt-no-2nd-succ → stable.

Il reste donc deux transitions en suspens dans la preuve du GMP. Nous considérons cette preuve partielle comme pouvant représenter une tentative de preuve réaliste de l'algorithme modifié.

5.5.4.3 Analyse de la preuve partielle

La preuve partielle ainsi obtenue va donc servir de base pour guider la conception du test.
Pour cela, rappelons que deux niveaux d'analyse sont possibles.

Le premier niveau, ou niveau des lemmes, correspond à l'identification des deux transitions non prouvées. Il exploite les résultats bruts de la preuve (deux lemmes sont en échec) et ne nécessite que la compréhension de la structure globale de la preuve (ces lemmes correspondent à deux transitions particulières du diagramme de configurations).
Le deuxième niveau, ou niveau des séquents, requiert une analyse plus poussée des arbres de preuve de chacune des transitions en échec. Cet effort ne se justifie que si l'analyse précédente s'avère insuffisante. Nous ne présentons pas pour l'instant l'analyse de deuxième niveau. Nous y reviendrons après les résultats de test donnés par le niveau des lemmes.

5.6 Test guidé par la preuve

Dans un premier temps, nous allons concevoir un profil de test statistique qui cible l'activation des deux transitions non prouvées (§ 5.6.1).

Nous raffinerons ensuite ce profil en prenant en compte les détails des arbres de preuve associés aux deux transitions (§ 5.6.2).

Les résultats obtenus dans chaque cas seront comparés à ceux fournis par le test aléatoire aveugle.

5.6.1 Premier profil de test, niveau 1

5.6.1.1 Conception du profil

Le critère de sélection de test retenu est la couverture de tous les chemins stable → ... → stable qui incluent les deux transitions non prouvées.
Nous avons fait abstraction des transitions *dead steps* correspondant aux *slots* de processeurs déjà diagnostiqués comme défaillants. Ainsi, sur le diagramme des configurations de la Figure 5.10 on agrège les chemins :

- entrant dans l'état latent par l'une ou l'autre des deux transitions possibles. La transition étiquetée good step est toujours couverte par la première faute de la séquence de test. La transition dead step est couverte lorsque la séquence comporte plusieurs fautes initiales, certaines se manifestant après le *slot* d'un processeur précédemment diagnostiqué.
- avec ou sans activation des boucles réflexives *dead steps* sur les états autres que latent. Ces boucles sont là encore couvertes par des séquences comportant plusieurs fautes initiales.

En faisant abstraction des transitions *dead steps*, il y a six chemins qui activent la transition pending-selfdiag-no-1st-succ → stable et un chemin qui active la transition missed-rcv-x-not-ack → stable, sur l'ensemble des dix chemins du diagramme.

Les trois chemins n'activant aucune des deux transitions ciblées n'ont pas été retenus dans notre critère, ce sont :

- stable → latent → missed-rcv → stable
- stable → latent → excluded → excluded-doubt → stable
- stable → latent → excluded → excluded-doubt → excluded-doubt-no-second-succ → stable

Les chemins retenus sont les suivants :

- stable → latent → missed-rcv-x-not-ack → stable
- un des 2 chemins passant par les états stable → latent → excluded → pending-selfdiag → pending-selfdiag-no-1st-succ → stable (l'autre chemin est infaisable)
- les 4 chemins passant par les états stable → latent → missed-rcv → excluded → pending-selfdiag → pending-selfdiag-no-1st-succ → stable
- stable → latent → missed-rcv → excluded-z-doubt → pending-selfdiag → pending-selfdiag-no-1st-succ → stable

Notons que le chemin identifié comme infaisable correspond à l'activation de la transition excluded → pending-selfdiag avec receipt ok (après une faute d'émission, une réception correcte active obligatoirement la transition excluded → excluded-doubt selon la garde (5) de l'algorithme).

Nous présentons dans la Définition 5.3 l'algorithme de génération pour couvrir un des 7 chemins identifiés du diagramme des configurations.

Définition 5.3 — Profil 1 – Algorithme de génération pour le chemin : stable → latent → excluded → pending-selfdiag → pending-selfdiag-no-1st-succ → stable

Ce chemin correspond à une faute d'émission suivie d'une faute de réception.

1. génération du préfixe (avant la faute)
   - tirage aléatoire d'un des processeurs non-fautifs,
   - date de la faute injectée = prochain *slot* de ce processeur.
2. faute initiale et contrôle de la configuration
   - générer une faute d'émission du processeur choisi (équiprobabilité entre les deux types de fautes possibles : *no_msg* et *not_no_msg*),
   - réception incorrecte au prochain *slot* d'un processeur non-fautif (sinon on active la transition excluded → excluded-doubt).
3. génération du suffixe (après le contrôle)
   - comportement aléatoire uniforme par la suite : tant que $t < t_f + 2n$ générer le comportement ultérieur :
     - 50% de chance d'injecter une faute à chaque pas d'horloge,
     - si injection de faute : tirer de façon équiprobable le type de faute selon le mode du processeur à cette date (selon le modèle de la Tableau 5.1).

Une remarque générale est que le profil de test que nous avons conçu ne constitue qu'une approximation d'un profil optimal qui rendrait les 7 chemins équiprobables. En effet, la conception d'un tel profil n'est pas évidente. Dans la spécification PVS, les conditions de franchissement des transitions dépendent notamment de prédicats sur des variables internes du modèle. Pour pouvoir contrôler parfaitement (si cela est possible) le profil de test, il nous aurait fallu, sans que cela soit nécessairement suffisant, implanter dans l'environnement de test une observation fine de l'ensemble des variables internes considérées dans le modèle PVS, de façon à pouvoir déterminer l'état abstrait dans lequel on se trouve à tout instant. De plus, pour un état donné, les conditions de franchissement des transitions sortantes ne sont pas forcément exclusives.

Nous exerçons donc un contrôle imparfait sur la couverture de certains chemins dont l'activation est complexe ; il s'agit des chemins passant par l'état missed-rcv. Notons cependant qu'aucune séquence de test pouvant activer les chemins ciblés n'est exclue

du profil retenu ; par contre, il reste possible de générer certaines séquences n'activant aucune des deux transitions ciblées (correspondant à une activation de la transition missed-rcv → stable, c'est-à-dire à un auto-diagnostic de x à son premier slot d'émission après sa faute).

5.6.1.2 Résultats expérimentaux

Selon ce profil, nous avons généré 5×10^4 séquences de test. Ces différents jeux de test ont été soumis au prototype du GMP, le code de ce prototype ayant été modifié pour reproduire la faute de l'algorithme.

Les résultats que nous avons obtenus sont les suivants :

- le taux de défaillance est de 0,88% pour les jeux de test générés selon le profil guidé ;
- toutes les défaillances observées portent sur la propriété d'auto-diagnostic en temps borné (au bout de 2n slots, on observe que le processeur défaillant ne s'est pas retiré de son *membership*).

Si l'on dissocie les résultats pour chacune des deux transitions, on obtient :

- 1,02% pour la transition excluded-doubt-no-2nd-succ → stable, et
- 0,1% pour la transition missed-rcv-x-not-ack → stable.

L'amélioration obtenue par rapport au profil aveugle (0,6%) n'est donc pas très significative. Notons que, à proprement parler, le critère retenu *est* pertinent : aucun des trois chemins écartés ne peut provoquer une défaillance de l'algorithme. Néanmoins, il est clair que l'on cible encore imparfaitement les conditions précises de défaillance. Une approche de test déterministe pour couvrir les chemins retenus s'avérerait ici totalement inefficace. Avec l'approche test statistique, le taux de défaillance obtenu approche les 1%, et les dépasse pour l'une des transitions. Un échantillon de quelques centaines de séquences de tests suffirait pour révéler la faute avec une probabilité élevée, ce qui n'est pas excessif.

Ces résultats ont été obtenus à partir d'une analyse relativement peu fouillée de la preuve, ne tenant compte que de l'existence de lemmes non prouvés, et sans considérer d'éventuelles informations contenues dans les arbres de preuve de ces lemmes. Nous allons maintenant considérer ce type d'information pour affiner le profil de test.

5.6.2 Deuxième profil de test, niveau 2

L'analyse au niveau des séquents demande beaucoup plus d'investissement que le premier niveau d'analyse qui, lui, se contente d'une utilisation directe des lemmes en échec. Elle requiert notamment une certaine expertise dans l'outil de preuve, pour être capable d'identifier, dans les arbres de preuve des lemmes, les parties qui n'aboutissent pas.

Il est à noter que ceci suppose que l'outil offre des facilités pour visualiser les arbres

de preuve. Obtenir uniquement les séquents en échec, sans l'arbre de preuve complet, rendrait difficile – voire impossible – leur exploitation. Le système PVS, par exemple, offre une vue pas à pas de la démonstration ainsi qu'une interface graphique permettant d'étudier l'arbre de preuve dans son ensemble et chacun de ses séquents.
Nous donnons ci-dessous les résultats de notre analyse détaillée des arbres de preuve des deux transitions en suspens. Nous en déduirons un nouveau profil de test, dont nous évaluerons expérimentalement l'efficacité.

5.6.2.1 Résultats de l'analyse de niveau 2

Chacun des deux lemmes de transition comprend un nombre important de branches de preuve. La première décomposition de l'arbre se fait notamment sur la définition de l'état d'arrivée (stable) qui crée 7 branches de preuve pour traiter les 7 cas composant cet état (voir sa définition donnée Figure 5.11, § 5.5.2). Chacune de ces branches de preuve peut ensuite donner naissance à plusieurs autres décompositions.

Les sous-arbres des 7 cas utilisent notamment l'axiome `shutdown`. Celui-ci assure que l'émetteur se diagnostique comme défaillant, puis reste silencieux lors de son émission.

```
shutdown: AXIOM
  LET b = broadcaster(t) IN
    ¬ no_cliques(t, b) ⊃ nomsg(t, b) ∧ ¬ sendsOK(t, b)
```

FIG. 5.12 – PVS code : axiome shutdown dans la théorie membership_fautmodel

Pour pouvoir utiliser cet axiome, la précondition $\neg no_cliques$[3] doit être satisfaite sous les hypothèses correspondant à l'état initial (pending-selfdiag-no-1st-succ ou missed-rcv-x-not-ack selon la transition considérée) et sous la condition d'activation de la transition (x est émetteur). C'est cette démonstration qui est non-concluante dans les différentes branches de preuve inachevées. Dans tous les cas, l'échec de preuve est que l'on ne parvient pas à prouver le but $acc_x^t \leq rej_x^t$, sous l'hypothèse $acc_x^t = 1$ (Figure 5.13).

5.6.2.2 Conception d'un profil de test

L'hypothèse $acc_x^t = 1$ est assurée par les définitions des états pending-selfdiag-no-1st-succ et missed-rcv-x-not-ack. Pour tenter de falsifier le séquent en échec, et ainsi trouver un contre-exemple expérimental, nous avons donc cherché à générer des séquences de test activant les transitions avec $rej_x^t = 0$.

[3]Rappelons que $no_cliques$ correspond à la garde (1) de l'algorithme, qui a été modifiée par la faute injectée : elle exprime maintenant la condition $acc_x^t > rej_x^t$.

$$
\begin{array}{l}
x \in \mathrm{mem}_x^t \\
\mathrm{faulty}_x^t \\
\mathrm{faulty}_x^{t+1} \\
\neg\ \mathrm{doubt}_x^t \\
\mathrm{acc}_x^t = 1 \\
\hline
\neg\ (\mathrm{acc}_x^t > \mathrm{rej}_x^t)
\end{array}
$$

FIG. 5.13 – Séquent en échec (*les hypothèses liées à la configuration ne sont pas retranscrites*)

Le compteur rej a été mis à zéro lors de la dernière émission de *x*. Il faut donc que, depuis sa dernière émission, *x* n'ait jamais exécuté de commande incrémentant ce compteur. L'analyse des commandes de l'algorithme montre que :

- *x* ne doit pas avoir exécuté les commandes (5), (7), (11), ou (14) depuis sa dernière émission,
- *acc* étant mis à 1 après sa dernière émission, et *acc* étant toujours égal à 1 dans les états pending-selfdiag-no-1st-succ et missed-rcv-x-not-ack, *x* n'a donc pas exécuté les commandes (4), (8), (9), ou (12),
- *x* n'a pas non plus exécuté la commande (3) car dans les configurations voulues, *x* ne s'est pas encore retiré de son *membership*.

Les seules commandes que *x* est autorisé à exécuter selon cette condition sont donc : (6), (10) et (13). Selon les gardes de ces commandes, cela revient à dire que *x* n'a pas détecté d'activité sur le bus depuis sa première émission. Le processeur fautif *x* a donc toujours reçu l'entrée *null* à chaque pas temporel depuis son émission. Cela peut être le cas pour une réception valide d'un émetteur silencieux (car s'étant diagnostiqué comme défaillant), ou bien pour une faute de réception de type *null*.

Le profil de test précédent est donc raffiné selon ces conclusions :

- en éliminant certains chemins incompatibles avec cette exigence. Par exemple la transition excluded $\rightarrow$ pending-selfdiag sur une réception correcte n'est plus activable (car elle causerait une augmentation du compteur *rej*) ;
- en restreignant le sous-espace d'entrée associé aux chemins restants. On couvrira ces chemins avec des séquences de test dont le suffixe ne comporte que des fautes de réception de type *null* (la longueur du suffixe dépend du chemin sélectionné).

Nous présentons dans la Définition 5.4 l'algorithme de génération d'une faute couvrant le même chemin du diagramme des configurations que celui présenté dans le paragraphe 5.6.1. Notons que la seule partie modifiée concerne le suffixe de génération.

Définition 5.4 — Profil 2 – Algorithme de génération pour le chemin :
stable → latent → excluded → pending-selfdiag → pending-selfdiag-no-1st-succ → stable
Ce chemin correspond à une faute d'émission suivie d'une faute de réception.

1. génération du préfixe (avant la faute)
   - tirage aléatoire d'un des processeurs non-fautifs,
   - date de la faute injectée = prochain *slot* de ce processeur.
2. faute initiale et contrôle de la configuration
   - générer une faute d'émission du processeur choisi (équiprobabilité entre les deux types de fautes possibles : *no_msg* et *not_no_msg*),
   - réception incorrecte au prochain *slot* d'un processeur non-défaillant (sinon on active la transition excluded → excluded-doubt).
3. génération du suffixe (après le contrôle)
   - comportement spécifique par la suite (uniquement des fautes *null*) : tant que $t < t_f + 2n$ générer le comportement ultérieur :
     - 100% de chance d'injecter une faute sur un *slot* non-fautif, faute obligatoirement de type *null*,
     - 50% de chance d'injecter une faute sur un *slot* fautif (silencieux), si on injecte une faute, elle sera aussi de type *null*.

5.6.2.3 Résultats expérimentaux

Les résultats obtenus sous ce profil, pour 5 jeux de 10^4 séquences de test, sont les suivants :

- le taux de défaillance est maintenant de 98,7% ;
- comme précédemment, toutes les défaillances constatées portent sur la propriété d'auto-diagnostic en temps borné.

Nous avons vérifié expérimentalement que toutes les séquences ne conduisant pas à défaillance activaient la transition missed-rcv → stable, et correspondaient donc au contrôle imparfait que nous exerçons sur la génération des chemins passant par l'état missed-rcv.

L'analyse détaillée de la preuve formelle a donc permis de focaliser le test sur un sous-espace d'entrée pertinent vis-à-vis de la faute introduite. La connexion entre le critère de test et la faute de conception de l'algorithme est maintenant parfaite (par contre notre contrôle de ce critère reste imparfait).
Notons également que, les scénarios de défaillance liés à la transition missed-rcv-x-not-ack → stable sont similaires au scénario donné dans [Pfeifer 2003], et rappelé dans paragraphe 5.5.4. Par contre, les scénarios liés à la transition pending-selfdiag-no-1st-succ → stable sont nouveaux : ils correspondent à des séquences de test plus longues, et n'avaient pas été identifiés manuellement par l'auteur de la preuve.
Dans [Pfeifer 2003] cet auteur mentionne que l'algorithme du GMP assure la propriété

d'auto-diagnostic plus rapidement que la borne de `2n-1` pas temporels. En se basant sur sa compréhension du diagramme des configurations, il émet la conjecture que le seuil réel devrait être d'un tour et demi dans l'anneau des processeurs (`n+n/2` pas d'horloge). Or pour certaines de nos séquences expérimentales correspondant au nouveau scénario de défaillance, l'auto-diagnostic peut nécessiter jusqu'à `2n-2` pas d'horloge sur la version correcte de l'algorithme.

5.7 Retour sur la preuve

Dans les chapitres précédents, la partie de la méthode dédiée au *retour sur la preuve* présentait les modifications qui pourraient être apportées sur la preuve informelle, au vu des résultats du test.

La preuve formelle que nous avons utilisée comme cas d'étude de ce dernier chapitre était, elle, complète : ce n'est qu'après avoir inséré une modification dans l'algorithme originel que nous avons pu obtenir une preuve inachevée, sujet d'intérêt pour la méthode que nous proposons. Ce cas d'étude se prête donc assez mal à établir des diagnostics sur la preuve à partir de nos résultats de test. Il serait en effet difficile de ne pas être biaisé par notre connaissance exacte de la faute et de sa correction.

Nous allons donc nous pencher sur un autre aspect de notre étude expérimentale qui est le lien entre les séquents de preuve en échec (étudié au deuxième niveau d'analyse de la preuve formelle) et leur utilisation pour définir des critères de test. L'extraction d'informations constructives pour le test a en effet été la principale difficulté que nous avons rencontrée lors de notre étude.

5.7.1 De l'extraction d'informations à partir d'une preuve formelle

Deux remarques principales sont à faire suite à notre étude de la preuve formelle du GMP.

La première est que la technique de preuve des *invariants disjonctifs* (§ 5.5.1) nous a été d'une aide précieuse lors de notre analyse. Tout d'abord, le diagramme des configurations est un guide très utile pour la compréhension du fonctionnement de l'algorithme. De plus, de par le fait que la preuve se base sur ce diagramme, la majorité des preuves de haut niveau sont liées au diagramme lui-même. Dans nos expériences, les preuves de haut niveau en échec se sont avérées être des transitions du diagramme. Le lien entre les lemmes en échec et le comportement opérationnel du prototype de l'algorithme est alors relativement facile à établir au niveau 1 d'analyse.

La deuxième remarque porte sur la difficulté de l'analyse du niveau 2, lorsque l'on cherche à établir un lien entre un séquent terminal en échec et des cas de fonctionnement de l'algorithme. Notre expérience est que l'établissement de ce lien est beaucoup plus problématique dans le cas d'un arbre de preuve formel qu'il ne l'était dans le cas d'un arbre informel. Cela est notamment dû au fait que l'expression détaillée du

séquent en échec ne permet généralement pas d'obtenir une compréhension intuitive de ce qu'il cherche à prouver.

Au problème du nombre des formules concernées s'ajoute celui des manipulations syntaxiques effectuées par l'outil, qui exploite la symétrie du calcul des séquents pour transférer les formules entre parties gauche et droite du séquent. En effet, dans PVS, dès qu'une formule se trouve en négation d'un côté du séquent, elle est automatiquement transférée de l'autre côté :

$$A, \neg B \vdash C, \neg D \text{ devient } A, D \vdash C, B \tag{5.2}$$

On perd alors l'information de quels étaient les hypothèses et buts originels de la branche de preuve, ce qui ne facilite pas la compréhension.

Prenons un exemple extrait de la preuve d'une des deux transitions analysées : celle partant de l'état `pending-selfdiag-no-first-succ` pour aller vers `stable`. Une version très simplifiée (en élaguant les hypothèses) d'un de ses séquents terminaux en échec est donné Figure 5.14.a. Même sous cette forme simplifiée, il est bien difficile de comprendre intuitivement que ce séquent est en fait lié à la garde (2) ($\neg no_cliques$) de l'algorithme, que l'on cherche à prouver. Par contre, sous la forme de la Figure 5.14.b, on identifierait plus facilement la définition de l'état `pending-selfdiag-no-first-succ` en hypothèse (avec notamment $acc_x^t = 1$), et le but à prouver $acc_x^t \leq rej_x^t$. La compréhension de ce séquent nous amène alors à chercher à falsifier $acc_x^t \leq rej_x^t$ avec $rej_x^t = 0$, et nous avons vu tout le bénéfice que nous avons pu en retirer pour la conception du test.

(a)		(b)
$1 > rej_x^t$		$acc_x^t = 1$
$acc_x^t = 1$		$x \in mem_x^t$
——————-	$\equiv$	$\neg\, doubt_x^t$
$x \in C(mem_x^t)$		——————-
$doubt_x^t$		$acc_x^t \leq rej_x^t$

FIG. 5.14 – Séquents en échec équivalents par la négation

En pratique, l'analyse d'un séquent en échec s'effectue par une remontée dans l'arbre de preuve, jusqu'à atteindre un séquent dont nous pouvons comprendre les hypothèses et le but. Par exemple, c'est en remontant jusqu'à l'étape d'introduction de l'axiome `shutdown` (par une règle `cut`) que nous avons pu comprendre le séquent de la Figure 5.14.a, et en déduire la forme équivalente – mais plus intuitive – de la Figure 5.14.b.

Expérimentalement, nous avons pu observer qu'il nous a toujours été nécessaire de remonter au-delà des tactiques de preuve développées spécifiquement pour la preuve

du GMP, car le nombre d'opérations élémentaires qu'elles appellent est trop important, et car ces opérations sont conduites de façon «boîte noire ».

Cette remontée de l'arbre formel est fastidieuse, et doit être réalisée pour chaque séquent en échec. Par exemple, il y en avait 3 pour la transition pending-selfdiag-no-1st-succ → stable, et 7 pour l'autre transition. Tous correspondaient en fait à un problème similaire, mais l'analyse a dû être faite pour chaque cas.

La question que nous nous sommes alors posés est de savoir si, en modifiant la structure de l'arbre en une autre structure équivalente, nous pourrions obtenir moins de séquents en échec, ce qui faciliterait grandement notre analyse pour le test. Le paragraphe suivant explore cette idée pour les deux arbres de preuve étudiés dans le cadre du GMP.

5.7.2 Modification d'un arbre de preuve en vue du test

Une formule peut être prouvée par plusieurs arbres de preuve, qui peuvent différer par les règles dont ils sont composés ou par l'ordre d'application de ces règles. C'est à ce dernier point que nous allons nous intéresser.

Dans notre cas d'étude, et en général dans le cadre de l'utilisation de PVS, la simplification du séquent racine est toujours conduite en premier dans un arbre de preuve. Cette simplification utilise des règles de réécriture, ou des règles de déduction décomposant les séquents en séquent plus simples, telles que des règles de décomposition du but. Après l'application de ces simplifications, on pourra utiliser des tactiques plus complexes, ou bien utiliser la coupure (équivalente au modus ponens de la déduction naturelle) pour introduire un lemme additionnel. Or, cet ordre d'application des règles n'est pas forcément adéquat dans le cadre de notre méthode.

Prenons l'exemple de l'arbre de preuve de la transition `missed-rvc-not-ack` → `stable`, dont nous donnons une vue schématique Figure 5.15.a. Il commence par une décomposition du but en 7 sous-buts (nous n'en représentons que 5 pour ne pas alourdir la figure). Plus tard, le même lemme (`shutdown`) sera introduit dans chacun des sous-arbres résultants par l'utilisation de la règle `cut`, et donnera lieu à 7 séquents en échec ($\times$).

Le lemme introduit étant le même partout, la migration de son introduction avant la décomposition en sous-buts devrait conserver un arbre équivalent vis-à-vis des règles utilisées, bien que leur ordre d'utilisation soit modifié. L'avantage est alors de n'avoir plus qu'un seul séquent en échec, comme schématisé dans la Figure 5.15.b.

Nous avons pu expérimenter cette idée sur les deux arbres de preuves analysés. Dans les deux cas, le résultat obtenu est une preuve n'ayant plus qu'une seule branche en échec, branche facilement identifiable comme étant liée à la preuve de l'hypothèse d'introduction du lemme `shutdown` par un `cut`.

L'idée de migration des règles semble donc pouvoir être très efficace, tout au moins dans les cas que nous avons étudiés. Une étude plus systématique s'imposerait cependant pour établir des schémas génériques de migrations. Notons que cette piste a

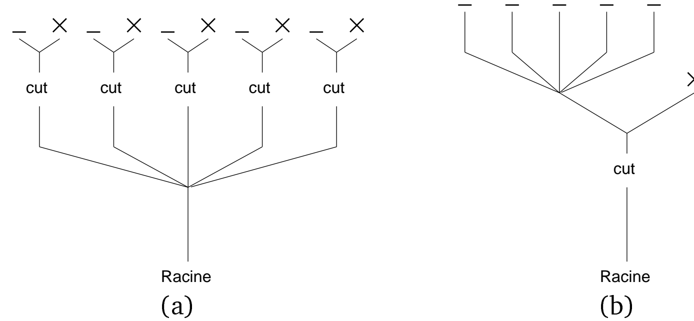


FIG. 5.15 – Migration d'une règle `cut` dans un arbre de preuve

déjà été explorée dans [Jouve 1999], qui étudiait les liens entre stratégies de test et tactiques de preuve, dans le cadre de spécifications axiomatiques et d'une sémantique basée sur la logique générale conditionnelle positive avec constructeurs. Son objectif était d'utiliser le dépliage des axiomes de preuve pour la génération de tests.

5.8 Conclusion

Nos résultats ont montré que le Test Guidé par la Preuve pouvait être très efficace dans le cadre d'une preuve formelle partielle. Une telle approche pourrait alors être utilisée pour obtenir des contre-exemples, notamment dans les cas où des techniques de model-checking s'avéreraient difficiles à appliquer (comme dans le cas du GMP étudié, *c.f.* la discussion conduite dans [Rushby 2000]). De cette façon, l'effort important investi dans le processus de preuve formelle n'est pas entièrement perdu et le test est dirigé vers les fautes de conception qui n'ont pas été révélées par la preuve partielle.

L'analyse de la preuve, pour guider la conception du test, peut être réalisée à deux niveaux.
Le niveau le plus facile, ou niveau des lemmes, ne requiert que la connaissance du langage de spécification, et la compréhension de la structure générale de la preuve. On exploite alors les résultats bruts de la preuve (lemmes prouvés ou non) pour déterminer des critères de test. Nos résultats montrent que ce niveau d'analyse peut déjà donner des informations pertinentes, bien qu'imparfaitement liées aux fautes résiduelles. La génération de larges échantillons de données de test, selon l'approche test statistique, pourrait alors permettre de compenser cette imperfection.
Le deuxième niveau d'analyse, ou niveau des séquents, n'est recommandé que si l'analyse précédente n'a pas permis de révéler de faute. Il nécessite une certaine maîtrise de l'outil de preuve, et l'effort à investir est important. De plus, il est à notre avis indispensable de disposer d'une documentation détaillée de la preuve formelle (telle que celle que nous avions dans [Pfeifer 2003]), ou bien de travailler en collaboration avec la personne ayant développé la preuve formelle. Si l'analyse est menée à bien, nos

résultats montrent cependant qu'elle peut parfois permettre de cibler très précisément des cas de défaillance.

De façon générale, une conclusion que nous pouvons tirer de cette première étude est que la structure de la preuve a un fort impact sur la faisabilité de notre méthode, et ce aux deux niveaux d'analyse.
Au niveau des lemmes, l'application de notre méthode a été grandement facilitée par la technique de preuve formelle utilisée : les invariants disjonctifs et le diagramme des configurations. Comme mentionné dans [Pfeifer 2003], le diagramme des configurations s'est avéré un outil très utile lors de la construction de la preuve, pour comprendre le fonctionnement de l'algorithme et analyser l'impact de changements mineurs dans sa spécification formelle. Ce diagramme s'est également avéré très utile dans le cadre de nos travaux sur le test. La structure de la preuve étant basée sur une vue opérationnelle du comportement de l'algorithme, il nous a été possible d'établir un lien entre les lemmes non-prouvés et des cas fonctionnels de l'algorithmes. Dans le cas de preuves plus classiques, ce lien pourrait être beaucoup plus difficile à établir.
Au niveau des séquents, nous avons montré que l'ordre d'application des règles d'inférence avait une influence sur la difficulté de l'analyse. Il reste cependant à déterminer comment exploiter cette idée pour transformer une preuve existante en une preuve plus facile à analyser.

Conclusion générale et Perspectives

“ Tous les bons esprits répètent, depuis Bacon qu'il n'y a de connaissances réelles que celles qui reposent sur des faits observés. ”

— Comte dans Cours de philosophie positive. Première leçon

Les travaux qui ont été présentés dans ce mémoire s'inscrivent dans une tendance générale qui vise à renforcer la vérification des étapes amont du processus de développement, de façon à révéler au plus tôt d'éventuelles fautes de conception. En particulier, notre objectif est d'encourager au prototypage et au test des algorithmes de tolérance aux fautes, pour vérifier leur correction avant leur implémentation dans une architecture cible. Pour cela, nous proposons de baser le test sur les hypothèses et les preuves de validité qui sont généralement développées avec ces algorithmes. Les preuves que nous considérons sont des preuves informelles, ou des preuves formelles inachevées. Nos travaux ont cherché à valider expérimentalement le principe d'une conception du test guidée par l'identification des faiblesses de ces preuves, dans un objectif de complémentarité.

Une première partie de cette thèse s'est consacrée aux preuves informelles. Ces preuves sont couramment utilisées lors de la conception et de la publication d'algorithmes, même dans le cas d'algorithmes destinés à des systèmes critiques. Comme les exemples que nous avons étudiés le montrent, les preuves informelles sont généralement insuffisantes pour garantir le comportement de l'algorithme considéré. Dans le cas de systèmes critiques, une formalisation complète serait envisageable, mais c'est un investissement coûteux et qui sera perdu si l'algorithme s'avère incorrect. Nous proposons donc une solution plus «légère» basée sur le prototypage de l'algorithme et le test de ce prototype guidé par des informations extraites de la preuve informelle.

Pour mettre en œuvre ce principe d'un *test guidé par la preuve*, le Chapitre 2 propose une méthode, qui est ensuite appliquée à deux cas d'étude dans les Chapitres 3 et 4. Cette méthode est constituée de quatre étapes successives que nous résumons ci-dessous :

- L'*analyse préliminaire* a pour objectif de nous permettre de réaliser l'environnement de test de l'algorithme. Pour cela, nous étudions la spécification et la preuve informelle de l'algorithme pour : 1) réaliser le prototype, et 2) en extraire les hypothèses (détermination du domaine d'entrée de test), et les propriétés devant être assurées par l'algorithme (détermination de l'oracle de test). A l'issue de cette étape, des premières expériences de test «en aveugle» sont réalisées, pour tenter de révéler des fautes de conception à faible coût.
- La *restructuration de la preuve* est une reformulation du discours informel sous la forme d'un arbre de preuve en déduction naturelle ou en calcul des séquents. Le but de cette réécriture est de nous permettre une analyse fine des arguments de la preuve, analyse que n'aurait pas permis une relecture attentive de la preuve (comme le montrent les algorithmes qui ont été publiés bien qu'incorrects). Cette restructuration conserve un certain nombre de notations du discours informel pour conserver la «légèreté» de la méthode. On ne cherche pas à compléter la preuve dans cette étape, mais seulement à évaluer ses faiblesses potentielles. Dans le cas où une preuve est jugée trop peu rigoureuse, la méthode est arrêtée.
- Le *test guidé par la preuve* exploite les résultats de l'analyse précédente de façon constructive pour le test. Les faiblesses identifiées dans la preuve sont utilisées pour définir des critères de sélection et pour renforcer l'oracle de test. La génération de tests s'effectue alors selon l'approche test statistique, pour compenser l'imperfection des critères de test extraits de la preuve. L'hypothèse est que les éventuelles fautes de conception résiduelles sont bien liées à des lacunes dans la preuve, mais que ce lien est imparfaitement identifié par notre analyse.
- Le *retour sur la preuve* est la dernière étape de notre méthode. Il a pour but de nous amener à prendre en compte les résultats du test précédent pour éventuellement améliorer la preuve informelle. Si aucune défaillance n'a été constatée, on pourra tenter de compléter ou de corriger les parties que nous avions notées comme faibles. Si une faute a été révélée, il faudra corriger l'algorithme. Les contre-exemples obtenus par le test, et l'analyse de l'arbre de preuve obtenu à l'étape de restructuration, peuvent alors aider à diagnostiquer la faute.

Nous avons étudié l'efficacité de cette méthode sur deux algorithmes de tolérance aux fautes destinés à des systèmes critique : le FT-RMS et le GMP.
L'étude du FT-RMS nous a amené à introduire un point d'arrêt de la méthode à l'issue de l'étape de restructuration. Lorsque la restructuration du discours met en évidence des manques de rigueur flagrants, il est probable que la preuve ne pourra pas fournir d'informations pertinentes pour la conception du test. Le retour sur la preuve est alors qu'elle doit être entièrement reprise, en se basant sur une autre structure.
L'étude du GMP a permis des retours plus constructifs sur la preuve : d'une part certaines parties ont pu être consolidées ; et d'autre part les résultats de test ont permis de pointer précisément sur la faille de raisonnement qui a laissé passer une faute de conception. Cette étude a aussi montré que l'identification des lacunes d'une preuve pouvait s'avérer efficace pour guider le test. L'hypothèse que les critères de test extraits de la preuve sont certes imparfaits, mais néanmoins pertinents, s'est avérée justifiée dans ce cas.

Nous avons ensuite, dans le Chapitre 5, étudié l'adaptation aux preuves formelles de la méthode en quatre étapes définie pour les preuves informelles. Une adaptation évidente concerne le remplacement de la restructuration du discours par une analyse de la structure de la preuve formelle.

Dans un premier temps, nous avons été amenés à définir une technique d'injection d'une faute de conception dans un couple spécification et preuve formelles. Cet artifice expérimental vise à obtenir une preuve formelle partielle qui puisse être un cas d'étude pour notre méthode.
Le choix de recourir à l'injection d'une faute à été motivé par deux raisons :

- Il est difficile de trouver dans la littérature des exemples de preuve formelle inachevée. En effet, une telle preuve n'a aucune valeur en tant que vérification et n'est donc pas publiée.
- Notre technique permet de définir une preuve partielle à partir d'une preuve complète. Dans nos expériences, nous avons envisagé plusieurs injections pour finalement en choisir une. Cela permet donc d'obtenir plusieurs cas d'étude proches à partir d'une seule spécification et preuve formelle. Cela nous offre donc la possibilité de poursuivre nos expériences avec d'autres fautes, sans avoir à repartir de zéro sur un nouvel exemple d'algorithme.

Les expériences de test conduites à partir de cette preuve formelle modifiée se sont avérées très encourageantes. Nous avons déterminé deux niveaux d'analyse de la preuve.
Le premier niveau, nommé niveau des lemmes, correspond à l'analyse directe des lemmes en suspens dans la preuve formelle. Il ne requiert pas d'investissement important dans la compréhension de la preuve, et peut déjà donner des informations pertinentes pour le test.
Si ce premier niveau de test ne révèle pas de faute, cela peut être dû au fait qu'il ne cible pas assez précisément les faiblesses de la preuve formelle. On est alors amené à analyser la preuve formelle plus en détail, au niveau des séquents en échec dans les arbres de preuve des lemmes.
Lors de nos expériences, ce niveau des séquents a été très efficace, puisqu'il nous a permis de cibler précisément les scénarios amenant à une défaillance de l'algorithme. Néanmoins, il nécessite un effort d'analyse très important, et l'extraction d'informations constructives pour le test reste problématique.

> *“ Il n'y a pas de problèmes résolus, il y a seulement des problèmes plus ou moins résolus. ”*
>
> *— Henri Poincaré*

Ces travaux ne prétendent pas avoir résolu le problème du test guidé par la preuve. Mais le principe a été exposé et nous avons pu montrer expérimentalement qu'il

pouvait être efficace et méritait que l'on s'y intéresse. La principale question qui s'ouvre à nous est celle de la systématisation, voire de l'automatisation de cette approche.

Dans le cas des preuves informelles, il est peu probable que l'on puisse aller beaucoup plus loin, de par le fait que l'on se base sur un discours informel, et le souhait de conserver une approche plus «légère» qu'une formalisation complète. Notamment, si nous voulions consolider l'étape de la restructuration pour permettre une évaluation moins subjective de la preuve, il est peu probable que cette «légèreté» puisse être gardée.
Comme nous avons pu le constater, le problème majeur reste la rigueur de la spécification (et de la preuve) informelle. Cette constatation va dans le sens de travaux visant à l'établissement de spécifications plus rigoureuses, bien que restant écrites en langage naturel. On pourra notamment se reporter aux travaux sur les langages naturels contrôlés (*Controlled Natural Languages*) tels que [Fuchs et al. 1999, Fuchs et al. 2000].
Cependant, dans le cadre de la méthode que nous proposons, certaines questions restent ouvertes et intéressantes à étudier sur d'autres cas d'étude. Notamment le lien entre les différent éléments que nous cherchons à identifier dans notre approche :

- les faiblesses de la preuve informelle ;
- l'extraction d'informations constructives pour le test ;
- les éventuelles fautes résiduelles de l'algorithme.

Pour les preuves formelles, les perspectives sont plus ouvertes.

Dans la continuation directe de nos travaux, nous avons déjà démarré l'étude de nouvelles injections de faute, pour étudier plusieurs exemples de preuves partielles obtenues à partir de la preuve du GMP. Cela a notamment pour but d'évaluer l'efficacité des deux niveaux d'analyse de la preuve que nous avons définis, et d'étudier plus avant le problème de l'établissement d'un lien entre des branches de preuve inachevées et des cas de test.
La structure générale de la preuve formelle du GMP s'est avérée adéquate pour une utilisation dans le cadre de la conception du test. Cela est dû à la technique des invariants disjonctifs, qui se base sur un modèle opérationnel du comportement de l'algorithme (le diagramme de configurations) pour construire la preuve. Notre travail en a été grandement facilité. Nous pensons que ce type de preuve, facilitant la collaboration entre la preuve et des techniques comme le test ou la vérification de modèle, doit être encouragé et est sans doute amené à se développer dans le futur. Cependant notre méthode devrait aussi être étudiée dans le cadre de preuves formelles plus «conventionnelles», qui se prêteront sans doute moins facilement à l'analyse en vue du test.

La formalisation de la preuve offre également des perspectives pour une éventuelle semi-automatisation de notre méthode.
La piste que nous avons ouverte est celle de modifications structurelles des arbres de preuve formels, pour réduire le nombre de branches en échec et faciliter leur analyse. L'approche envisagée est de modifier l'ordre d'application des règles d'inférence, par

migration de ces règles. Il devrait être possible de définir des schémas de migration pour pouvoir automatiser cette réduction du nombre de branches en échec.

Pour aller plus loin, l'analyse des branches devrait elle-même être automatisée. Mais c'est un objectif bien plus ambitieux. Pour extraire des critères de test, nous ne voyons pas pour l'instant comment s'affranchir de la nécessité d'obtenir une compréhension intuitive, en termes de comportement de l'algorithme, de ce que le séquent cherche à prouver.

Enfin, des liens avec des techniques comme le dépliage d'axiomes pourraient aussi être envisagés. Cela n'a pas été considéré dans cette thèse mais est certainement une piste intéressante.

Annnexe A : Preuve formelle PVS du diagnostic du GMP

Nous donnons ci-dessous la preuve formelle sous PVS de la propriété de *Diagnostic* du GMP formel, propriété que nous avons rajoutée à la spécification.

Partie 1/3

```
(|disgnosis| "" (EXPAND "diagnosis")
 ((""(SKOSIMP*)
   ((""(USE "liveness")
     ((""(EXPAND "liveness")
       ((""(INST - 0)
         ((""(GROUND)
           (("1" (SKOSIMP* -)
             (("1" (INST?)
               (("1" (SPLIT 2)
                 (("1" (PROPAX) NIL NIL) ("2" (PROPAX) NIL NIL)
                  ("3" (SKOSIMP*)
                   (("3" (TYPEPRED "p!1")
                     (("3" (EXPAND "faulty?")
                       (("3" (EXPAND "stable")
                         (("3" (FLATTEN)
                           (("3" (INST - "p!1")
                             (("3"
                               (ASSERT)
                               (("3"
                                 (FLATTEN -8)
```

Partie 2/3

```
                    (("3"
                      (HIDE -1 -3 -4 -5 -6 -7 -9 -10 2)
                      (("3"
                        (HIDE 1 2)
                        (("3"
                          (EXPAND "the_mem")
                          (("3"
                            (USE
                             "faults_latch_always"
                             :SUBST
                             ("p"
                              "x!1"
                              "t"
                              "t!1+1"
                              "i"
                              "s!1-1"))
                            (("3"
                              (ASSERT)
                              (("3"
                                (HIDE -4)
                                (("3" (GRIND) NIL NIL))
                                NIL))
                              NIL))
                            NIL))
                          NIL))
                        NIL))
                      NIL))
                    NIL))
                  NIL))
                NIL))
              NIL))
            NIL))
          NIL))
        NIL))
      NIL))
    NIL))
  NIL))
NIL)
```

Partie 3/3

```
             ("2" (USE "no_faults_when_not_stable")
              (("2" (GROUND)
                (("2" (CASE "the_mem(t!1)(x!1)")
                  (("1" (REPLACE -2 :DIR RL)
                    (("1" (EXPAND "the_mem") (("1" (PROPAX) NIL NIL))
                      NIL))
                    NIL)
                   ("2" (EXPAND "the_mem") (("2" (PROPAX) NIL NIL))
                    NIL))
                  NIL))
                NIL))
              NIL))
            NIL))
          NIL))
        NIL))
      NIL))
    NIL))
  NIL)
```

Cette preuve peut être incluse directement dans le fichier de preuve de la théorie `membership_verification`, théorie de plus haut niveau de la spécification PVS formelle du GMP.
Elle complète la description formelle de la propriété de Diagnostic donnée dans le paragraphe 5.4.2.2.

Annexe A

Bibliographie

[Abrial 1996] J.-R. Abrial, *The B-Book : Assigning Programs to Meanings*, Cambridge University Press, 1996.

[Alabau et al. 1997] M. Alabau, D. Begay, & J.-P. Radoux, "The embedded software on an electricity meter : An experience in using formal methods in an industrial project.", *Science of Computer Programming*, vol. 28 :pp. 93–110, 1997.

[Alquié] F. Alquié, editeur, "Règles pour la direction de l'esprit, Règle II", *Descartes, Œuvres philosophiques*, volume I de *classiques*, page 82, Garnier.

[Ammann et al. 1998] P. Ammann, P. Black, & W. Majurski, "Using model checking to generate tests from specifications", dans *2nd IEEE Int. Conf. on Formal Engineering Methods (ICFEM'98)*, pages 46–54, IEEE Computer Society, 1998, Brisbane, Australia.

[Ammann & Black 1999] P. Ammann & P. E. Black, "Abstracting formal specifications to generate software tests via model checking", dans *18th Digital Avionics Systems Conference (DASC'99)*, volume 2, pages 10.A.6.1–10, St. Louis, MIS, USA, IEEE Computer Society, 1999, NIST-IR 6405 (extended version).

[Arazo & Crouzet 2001] A. Arazo & Y. Crouzet, "Formal Guides for Experimentally Verifying Complex Software-Implemented Fault Tolerance Mechanisms", dans *7th Int. Conf. on Engineering of Complex Computer Systems (ICECCS'01)*, pages 69–79, Skövde, Sweden, IEEE, 2001.

[Arlat et al. 1990] J. Arlat, M. Aguéra, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, & D. Powell, "Fault Injection for Dependability Validation : A Methodology and some Applications", *IEEE Transactions on Software Engineering*, 16(2) :166–182, 1990.

[Arlat et al. 1999] J. Arlat, J. Boué, & Y. Crouzet, "Validation-based Development of Dependable Systems", *IEEE Micro*, 19(4) :66–79, 1999.

[Arlat et al. 2002] J. Arlat, J.-C. Fabre, M. Rodríguez, & F. Salles, "Dependability of COTS Microkernel-Based Systems", *IEEE Transactions on Computers*, 51(2) :138–163, 2002.

[Arnold 1990] A. Arnold, "MEC : A System for Constructing and Analysing Transition Systems", dans Joseph Sifakis, editeur, *International Workshop on Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science, pages 117–132, Grenoble, France, Springer-Verlag, 1990, LNCS 407.

[Arnold & Brlek 1995] A. Arnold & S. Brlek, "Automatic Verification of Properties in Transition Systems", *Software – Practice and Experience*, 25(6) :579–596, 1995.

[Arnold & Nivat 1982] A. Arnold & M. Nivat, "Comportements de Processus", dans *Colloque AFCET 'Les Mathématiques de l'Informatique'*, pages 35–68, 1982.

[Arnold 1973] T. Arnold, “The Concept of Coverage and its Effect on the Reliability Model of Repairable Systems”, *IEEE Transactions on Computers*, C-22(3) :251–254, 1973.

[Avresky et al. 1996] D. Avresky, J. Arlat, J.-C. Laprie, & Y. Crouzet, “Fault Injection for the Formal Testing of Fault Tolerance”, *IEEE Transactions on Reliability*, 45(3) :443–455, 1996.

[Babaoglu et al. 2001] O. Babaoglu, R. Davoli, & A. Montresor, “Group Communication in Partitionable Systems : Specification and Algorithms”, *IEEE Transactions on Software Engineering*, 27(4) :308–336, 2001.

[Barras et al. 1999] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Lahlhére, C. Muñoz, C. Murthy, C. Parent-Vigouroux, P. Loiseleur, C. Paulin-Mohring, A. Saïbi, & B. Werner, *The Coq Proof Assistant Reference Manual, Version 6.3*, 1999.

[Barthe et al. 2000] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, S. de Sousa, & S. Yu, “Formalisation of the Java Card Virtual Machine in Coq”, dans *Formal Techniques for Java Programs (FTfJP'00) – ECOOP Workshop on Fomal Techniques for Java Programs*, pages 50–56, Sophia-Antipolis, France, 2000.

[Behnia & Waeselynck 1999] S. Behnia & H. Waeselynck, “Test Criteria Definition for B Models”, dans *World Congress on Formal Methods (FM'99)*, pages 509–529, Toulouse, France, Springer-Verlag, 1999, LNCS 1708.

[Beizer 1990] B. Beizer, *Software Testing Techniques*, Van Nostran Reinhold, 1990, 2nd edition, New York.

[Bensalem et al. 2000] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Muñoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, & A. Tiwari, “An Overview of SAL”, dans *5th Langley Formal Methods Workshop (LFM'00)*, pages 187–196, NASA Langley Research Center, Hampton, VA, USA, 2000.

[Bensalem et al. 1998] S. Bensalem, Y. Lakhnech, & S. Owre, “InVeSt : A tool for the verification of invariants”, dans *Computer Aided Verification (CAV'98)*, pages 505–510, Springer-Verlag, 1998, LNCS 1427.

[Bérard et al. 1999] B. Bérard, M. Bidoit, F. Laroussinie, A. Petit, & P. Schnoebelen, *Vérification de logiciels, Techniques et outils du model-checking*, Vuibert, 1999, Ouvrage Collectif, Coordination Philippe Schnoebelen.

[Bernardeshi et al. 2001] C. Bernardeshi, A. Fantechi, & S. Gnesi, “Formal Validation of Fault Tolerance Mechanisms inside GUARDS”, *Journal of Reliability Engineering and System Safety*, 71(3) :261–270, 2001.

[Bernardeshi et al. 2002] C. Bernardeshi, A. Fantechi, & S. Gnesi, “Model Checking Fault Tolerant Systems”, *Software Testing, Verification and Reliability*, 12 :251–275, 2002.

[Berrojo et al. 2002] L. Berrojo, F. Corno, L. Entrena, I. González, C. Lopez, M. Sonza Reorda, & G. Squillero, “An Industrial Environement for High-Level Fault-Tolerant Structures Insertion and Validation”, dans *20th VLSI Test Symposium (VTS'02)*, pages 229–236, Monterey, CA, USA, IEEE, 2002.

[Berry & Gonthier 1992] G. Berry & G. Gonthier, “The ESTEREL synchronous programming language : Design, semantics, implementation ”, *Science of Computer Programming*, 19(2) :87–152, 1992.

[Black et al. 2000] P. E. Black, V. Okun, & Y. Yesha, “Mutation of model checker specifications for test generation and evaluation”, dans *Mutation Testing for the New Century (MUTATION 2000)*, pages 14–20, Kluwer Academic, 2000.

[Bogza et al. 1998] M. Bogza, C. Dawns, O. Maler, A. Olivero, S. Tripakis, & S. Yovine, "KRONOS : a model-checking tool for real-time systems", dans *10th International Conference on Computer Aided Verification (CAV'98)*, Lecture Notes in Computer Science, pages 546–550, Vancouver, Canada, Springer-Verlag, 1998, LNCS 1427.

[Bouricius et al. 1971] W. Bouricius, W. Carter, D. Jessep, P. Schneider, & A. Wadia, "Reliability Modeling for Fault-Tolerant Computers", *IEEE Transactions on Computers*, C-20(11) :1306–1311, 1971.

[Boyer & Moore 1979] R. Boyer & J. Moore, *A Computational Logic*, Academic Press, 1979.

[Bryant 1986] R. Bryant, "Graph-based algorithms for boolean function manipulation", *IEEE Transactions on Computers*, 35(8) :677–691, 1986.

[Bryant 1992] R. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams", *ACM Computer Surveys*, 24(3) :293–318, 1992.

[Burch et al. 1992] J. Burch, E. Clarke, K. McMillan, D. Dill, & L. Hwang, "Symbolic model checking : 10^{20} states and beyond", *Information and Computation*, 98(2) :142–170, 1992.

[Castanet et al. 1998] R. Castanet, O. Kone, & P. Laurencot, "On the Fly Test Generaion for Real Time Protocols", dans *International Conference on Computer Communications and Networks (ICCCN'98)*, pages 378–387, Louisiane, USA, 1998.

[Castanet & Rouillard 2002] R. Castanet & D. Rouillard, "Generate certified test cases by combining theorem proving and reachability analysis", dans *TESTCOM'02*, pages 249–266, Berlin, Germany, Kluwer Academic, 2002.

[Castéran & Rouillard 1999] P. Castéran & D. Rouillard, "Introduction to CClair", Rapport technique, LaBRI, Université de Bordeaux I, 1999, `http ://www.dept-info.labri.u-bordeaux.fr/~casteran/CClair/Tutorial.ps`.

[Cavalli et al. 1996] A. Cavalli, B. M. Chin, & K. Chon, "Testing methods for SDL systems", *Computer Networks and ISDN Systems*, 28(12) :1669–1683, 1996.

[Chandra et al. 1996] T. Chandra, V. Hadzilacos, S.Toueg, & B. Charron-Bost, "On the Impossibility of Group Membership", dans *15th Symposium on Principles of Distributed Computing*, pages 322–330, Philadelphia, PA, USA, ACM Press, 1996.

[Chevalley & Thévenod-Fosse 2001] P. Chevalley & P. Thévenod-Fosse, "An empirical evaluation of statistical testing designed from UML state diagrams : the flight guidance system case study", dans *12th International Symposium on Software Reliability Engineering (ISSRE'2001)*, pages 254–263, Hong Kong, 2001.

[Chevalley & Thévenod-Fosse 2003] P. Chevalley & P. Thévenod-Fosse, "A mutation analysis tool for Java programs", *International Journal on Software Tools for Technology Transfer (STTT)*, 5(1) :90–103, 2003.

[Chevochot & Puaut 2001] P. Chevochot & I. Puaut, "Experimental Evaluation of the Fail-Silent Behavior of a Distributed Real-Time Run-Time Support Built from COTS Components", dans *International Conference on Dependable Systems and Networks (DSN'01)*, pages 304–313, Göteborg, Sweden, 2001.

[Chow 1978] T. Chow, "Testing software design modeled by finite-state machines", *IEEE Transactions on Software Engineering*, 4(3) :178–187, 1978.

[Clarke et al. 2002] D. Clarke, T. Jéron, V. Rusu, & E. Zinovieva, "STG : a Symbolic Test Generation tool", dans *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, pages 470–475, Springer-Verlag, 2002, LNCS 2280.

[Clarke & Emerson 1981] E. Clarke & E. Emerson, “Design and synthesis of synchronization skeletons using branching time temporal logic”, dans *Logics of Programs Workshop*, pages 52–71, Yorktown Heights, NY, USA, Springer-Verlag, 1981, LNCS 131.

[Clarke et al. 1994] E. M. Clarke, O. Grumberg, & D. E. Long, “Model Checking and Abstraction”, *ACM Transactions on Programming Languages and Systsems (TOPLAS)*, 16(5) :1512–1542, 1994.

[Closse et al. 2001] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, & S. Yovine, “TAXYS : a Tool for the Development and Verification of Real-Time Embedded Systems”, dans *13th International Conference on Computer Aided Verification (CAV'01)*, pages 391–395, Paris, France, springer-Verlag, 2001, LNCS 2102.

[Cohn 1989] A. Cohn, “The notion of Proof in Hardware Verification”, *Journal of Automated Reasoning*, 5 :127–139, 1989.

[Colin & Puaut 2001] A. Colin & I. Puaut, “Worst-Case Execution Time analysis of the RTEMS Real-Time Operating System”, dans *13th Euromicro Conference on Real-time Systems*, pages 191–198, Delft, The Netherlands, IEEE Computer Society, 2001.

[Constable et al. 1986] R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, & S. Smith, *Implementing Mathematics with the Nuprl Proof Development System*, Prentice Hall, 1986.

[Coupet-Grimal & Jakubiec 1999] S. Coupet-Grimal & L. Jakubiec, “Hardware verification using co-induction in COQ”, dans *Theorem Proving in Higher Order Logics : 12th International Conference (TPHOLs'99)*, pages 91–108, Springer-Verlag, 1999, LNCS 1690.

[Cousot 2001] P. Cousot, “Abstract Interpretation Based Formal Methods and Future Challenges”, *Informatics, 10 Years Back - 10 Years Ahead*, pages 138–156, Springer Verlag, R. Wilhelm (ed), 2001, LNCS 2000.

[Cousot & Cousot 1977] P. Cousot & R. Cousot, “Abstract Interpretation : A Unifed Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”, dans *4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, USA, 1977.

[Creese & Roscoe 1999] S. Creese & A. Roscoe, “TTP : A case study in combining induction and data independence”, Rapport technique PRG-TR-1-99, Oxford University Computing Laboratory, Oxford, England, 1999.

[Cristian 1991] F. Cristian, “Understanding Fault-Tolerant Distributed Systems”, *Communications of the ACM*, 34(2) :56–78, 1991.

[Cristian & Schmuck 1995] F. Cristian & F. Schmuck, “Agreeing on Processor Group Membership in Timed Asynchronous Distributed Systems”, Rapport technique CSE95-428, UCSD, 1995.

[Crouzet et al. 1998] Y. Crouzet, P. Thévenod-Fosse, & H. Waeselynck, “Validation du test logiciel par injection de fautes : l'outil SESAME”, dans *11ème Colloque National de Fiabilité et Maintenabilité*, pages 551–559, Arcachon, 1998.

[Crow & Vito 1996] J. Crow & B. D. Vito, “Formalizing Space Shuttle software requirements”, dans *1st Workshop on Formal Methods in Software Practice (FMSP'96)*, pages 40–48, ACM, 1996.

[Cukic 1997] B. Cukic, “Combining testing and correctness verification in software reliability assessment”, dans *2nd IEEE High-Assurance Systems Engineering Workshop (HASE'97)*, pages 182–187, IEEE Computer Society, 1997, Washington, DC.

[Daran & Thévenod-Fosse 1996] M. Daran & P. Thévenod-Fosse, "Software error analysis : a real case study involving real faults and mutations", *Software Engineering Notes (SIGSOFT)*, 21(3) :158–171, 1996.

[de A. Lima & Burns 2001] G. M. de A. Lima & A. Burns, "An effective schedulability analysis for fault-tolerant hard real-time systems", dans *13th Euromicro Conf. on Real-Time Systems (ECRTS'01)*, pages 126–135, 2001, Delft, The Netherlands.

[de Bruijn 1968] N. de Bruijn, "The mathematical language AUTOMATH, its usage, and some of its extensions", dans *Symposium on Automatic Demonstration*, volume 125 de *Lecture Notes on Mathematics*, pages 29–61, Versailles, France, Springer-Verlag, 1968, LNM 125.

[de Moura et al. 2004] L. de Moura, S. Owre, H. Ruess, J. Rushby, N. Shankar, M. Sorea, & A. Tiwari, "SAL 2", dans *Computer-Aided Verification (CAV'04)*, pages 496–500, Boston, MA, USA, Springer-Verlag, 2004, LNCS 3114.

[DeMillo et al. 1988] R. DeMillo, D. Guindi, W. McCracken, A. Offut, & K. King, "An Extented Overview of the Mothra Software Testing Environment", dans *2nd Workshop on Software Testing, Verification and Analysis*, pages 142–151, Banff, Canada, IEEE Computer Society Press, 1988.

[DeMillo et al. 1978] R. DeMillo, R. Lipton, & F. Sayward, "Hints on Test Data Selection : Help for the Practicing Programmer", *Computer*, 11(4) :34–41, 1978.

[Dershowitz & Jouannaud 1990] N. Dershowitz & J. Jouannaud, "6 : Rewrite Systems", *Handbook of theoretical computer science*, volume B : Formal models and semantics, Elsevier Science Publishers, 1990.

[Dick & Faivre 1993] J. Dick & A. Faivre, "Automating the Generation and Sequencing of Test Cases from Model-Based Specifications", dans *Formal Methods Europe (FME'93)*, pages 268–284, Springer-Verlag, 1993, LNCS 670.

[Dilenno et al. 1991] T. Dilenno, D. Yaskin, & J. Barton, "Fault Tolerance Testing in the Advanced Automation System", dans *21st International Symposium on Fault Tolerant Computing (FTCS-21)*, pages 18–25, Montréal, Quebec, Canada, IEEE Computer Society Press, 1991.

[Dolev et al. 1987] D. Dolev, C. Dwork, & L. Stockmeyer, "On the Minimal Synchronisation needed for Distributed Consensus", *Journal of the ACM*, 34(1) :77–97, 1987.

[Dong et al. 1999] L. Dong, R. Melhem, D. Mossé, S. Ghosh, W. Heimerdinger, & A. Larson, "Implementation of a transient-fault-tolerance scheme on DEOS", dans *5th Real-Time Technology and Application Symposium (RTAS'99)*, pages 56–66, IEEE Computer Society, 1999, Vancouver, Canada.

[Dutertre 2000] B. Dutertre, "Formal analysis of the priority ceiling protocol", dans *IEEE Real-Time Systems Symposium (RTSS'00)*, pages 151–160, 2000, Orlando, FL.

[Echtle & Leu 1995] K. Echtle & M. Leu, "Test of Fault Tolerant Distributed Systems by Fault Injection", dans *Fault-Tolerant Parallel and Distributed Systems*, pages 244–251, Los Alamitos, CA, USA, IEEE, 1995.

[Egan et al. 1999] E. Egan, D. Kutz, D. Mikulin, R. Melhem, & D. Mossé, "Fault-tolerant RT-Mach (FTRT-Mach) and an application to real-time train control", *Software : Practice and Experience*, 29(3) :1–17, 1999.

[Emerson & Halpern 1982] E. Emerson & J. Halpern, "Decision procedures and expressiveness in the temporal logic of branching time", dans *14th ACM Symposium on Theory of Computing (STOC'82)*, pages 169–180, San Francisco, CS, USA, 1982.

[Engberg 1995] U. Engberg, *Reasoning in the Temporal Logic of Actions*, Thèse de doctorat, University of Aarhus, Department of Computer Science, Denmark, 1995.

[Ferdinand et al. 2001] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, & R. Wilhelm, “Reliable and Precise WCET Determination for a Real-Life Processor”, dans *First Workshop on Embedded Software (EMSOFT 2001)*, Lecture notes in Computer Science, pages 469–485, Springer-Verlag, 2001, LNCS 2211.

[Fernandez et al. 1996] J. C. Fernandez, C. Jard, T. Jéron, L. Nedelka, & C. Viho, “Using On-the-Fly Verification Techniques for the Generation of Test Suites”, dans *8th Int. Conf. on Computer-Aided Verification (CAV’96)*, pages 348–359, Springer-Verlag, 1996, LNCS 1102.

[Fischer et al. 1985] M. Fischer, N. Lynch, & M. Paterson, “Impossibility of Distributed Consensus with One Faulty Process”, *Journal of the ACM*, 32(2) :374–382, 1985.

[Fu et al. 2003] C. Fu, R. P. Martin, K. Nagaraja, & D. Wonnacott, “Compiler-directed Program-fault Coverage for Highly Available Java Internet Services”, dans *International Conference on Dependable Systems and Networks (DSN’03)*, pages 595–604, San Francisco, CA, USA, 2003.

[Fuchs et al. 1999] N. E. Fuchs, U. Schwertel, & S. Torge, “Controlled Natural Language Can Replace First-Order Logic”, dans *14th IEEE International Conference on Automated Software Engineering (ASE’99)*, pages 295–298, Cocoa Beach, FL, USA, IEEE Computer Society, 1999.

[Fuchs et al. 2000] N. E. Fuchs, U. Schwertel, & S. Torge, “A Natural Language Front-End to Model Generation”, *Journal of Language and Computation*, 1(2) :1999–214, 2000.

[Gargantini & Heitmeyer 1999] A. Gargantini & C. Heitmeyer, “Using Model Checking to Generate tests from Requirements Specifications”, dans *Joint 7th Eur. Software Engineering Conf. and 7th ACM SIGSOFT Int. Symp. on Foundations of Software Eng. (ESEC/FSE’99)*, pages 146–162, Toulouse, France, Springer-Verlag, 1999, LNCS 1687.

[Garland & Guttag 1991] S. Garland & J. Guttag, “A guide to LP, the Larch Prover”, Rapport technique 82, Digital Equipment Corporation, Systems Research Center, 1991.

[Gaudel 1995] M.-C. Gaudel, “Testing can be formal, too”, dans *6th International Conference on Theory and Practice of Software Development (TAPSOFT’95)*, pages 82–96, Springer-Verlag, 1995, LNCS 915.

[Geller 1978] M. Geller, “Test Data as an Aid in Proving Program Correctness”, *Communications of the ACM*, 21(5) :368–375, 1978.

[Ghosh et al. 1997] S. Ghosh, R. Melhem, & D. Mossé, “Fault-tolerant rate monotonic scheduling”, dans *6th IFIP Conf. on Dependable Computing for Critical Applications (DCCA’97)*, pages 121–145, IEEE Computer Society, 1997, Garmish-Partenkirchen, Germany.

[Ghosh et al. 1998] S. Ghosh, R. Melhem, D. Mossé, & J. S. Sarma, “Fault-tolerant rate monotonic scheduling”, *Real-Time Systems*, 15(2) :149–181, 1998.

[Goodenough & Gerhart 1975] J. B. Goodenough & S. L. Gerhart, “Toward a theory of test data selection”, *IEEE Trans. on Software Engineering*, SE-1(2) :156–173, 1975.

[Gordon & Melham 1993] M. Gordon & T. Melham, *Introduction to HOL : A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, 1993.

[Gordon et al. 1979] M. Gordon, R. Milner, & C. Wadsworth, *Edinbourgh LCF : A Mechanised Logic of Computation*, Springer-Verlag, 1979, LNCS 78.

[Gorn 1967] S. Gorn, “Explicit definitions and linguistic dominoes”, *Systems and Computer Science*, pages 77–1967, 1967, University of Toronto Press.

[Goubault 2001] E. Goubault, "Static analyses of the precision of floating-point operations", dans *86th International Symposium SAS'01*, pages 234–259, Paris, France, Springer Verlag, 2001, LNCS 2126.

[Gouraud 2004] S. D. Gouraud, *Utilisation des structures combinatoires pour le test statistique*, Thèse de doctorat, Université de Paris-Sud-Orsay, L.R.I., 2004.

[Gouraud et al. 2001] S.-D. Gouraud, A. Denise, M.-C. Gaudel, & B. Marre, "A new way of automating statistical testing methods", dans *International Conference on Automated Software Engineering (ASE'01)*, pages 5–12, IEEE Computer Society, 2001.

[Guttag et al. 1985] J. Guttag, J. Horning, & J. Wing, "The LARCH family of specification languages", *IEEE Sotware*, 2(5) :24–36, 1985.

[Harrison 1998] J. Harrison, *Theorem Proving with the Real Numbers*, Springer-Verlag, 1998.

[Havelund et al. 1997] K. Havelund, A. Skou, K. Larsen, & K. Lund, "Formal modelling and analysis of an audio/video protocol : An industrial case study using UPPAAL", dans *18th IEEE Real-Time Systems Symposium (RTSS'97)*, pages 2–13, San Francisco, CA, USA, IEEE Computer Society Press, 1997.

[Hayashi et al. 2002] S. Hayashi, R. Sumitomo, & K. Shii, "Towards the animation of proofs – testing proofs by examples", *Theoretical Computer Science*, 272(1–2) :177–195, 2002.

[Henzinger et al. 1997] T. A. Henzinger, P.-H. Ho, & H. Wong-Toi, "HyTech : A Model Checker for Hybrids Systems", *International Journal on Software Tools for Technology Transfer*, 1(1) :110–122, 1997, Springer-Verlag.

[Hierons 1997] R. M. Hierons, "Testing from a Z specification", *Software Testing, Verification and Reliability*, 7 :19–33, 1997.

[Holzmann 1997] G. J. Holzmann, "The model checker SPIN", *IEEE Transcripts on Software Engineering*, 23(5) :279–295, 1997.

[Holzmann & Peled 1994] G. J. Holzmann & D. Peled, "An Improvement in Formal Verification", dans *7th IFIP WG6.1 International Conference on Formal Description Techniques (FORTE 1994)*, pages 197–211, Bern, Suisse, 1994.

[Huang 1975] J. C. Huang, "An approach to program testing", *ACM Computing Surveys*, 7(3) :113–128, 1975.

[Hurfin et al. 2002] M. Hurfin, A. Mostefaoui, & M. Raynal, "A Versatile Family of Consensus Protocols Based on Chandra-Toueg's Unreliable Failure Detectors", *IEEE Transactions on Computers*, 51(4) :395–408, 2002.

[IST 1998] IST, "Special Issue on Program Slicing", *Information and Software Technology*, 40(11–12), 1998.

[Jéron & Morel 1999] T. Jéron & P. Morel, "Test Generation Derived from Model-Checking", dans *11th Int. Conf. on Computer-Aided Verification (CAV'99)*, pages 108–122, Trento, Italy, Springer Verlag, 1999, LNCS 1633.

[Joseph 1996] M. Joseph, *Real-Time Systems : Specification, Verification and Analysis*, Prentice Hall, 1996, London.

[Jouve 1999] H. Jouve, "Collaboration entre test et preuve", 1999, Rapport de DEA, La.M.I. (Laboratoire de Mathématiques et d'Informatique), Université d'Evry.

[Katz et al. 1997] S. Katz, P. Lincoln, & J. Rushby, "Low-Overhead Time-Triggered Group Membership", dans M. Mavronicolas & P. Tsigas, editeurs, *11th Int. Workshop on Distributed Algorithms (WDAG'97)*, pages 155–169, Saarbrücken Germany, Springer-Verlag, 1997, LNCS 1320.

[Kopetz 1995] H. Kopetz, “The time-triggered approach to real-time system design”, dans B. Randell, J.-C. Laprie, H. Kopetz, & B. Littlewood, editeurs, *Predictably Dependable Computing Systems*, ESPRIT Basic Research Series, pages 53–66, Springer-Verlag, 1995.

[Kopetz & Grünsteidl 1994] H. Kopetz & G. Grünsteidl, “TTP – A Time-Triggered Protocol for Fault-Tolerant Real-Time Systems”, *IEEE Computer*, 27(1) :14–23, 1994.

[Kropf 1999] T. Kropf, “Recent advancements in hardware verification - how to make theorem proving fit for an industrial usage”, dans *12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99)*, pages 1–4, Springer-Verlag, 1999, LNCS 1690.

[Lamport 1978] L. Lamport, “Time, Clocks and the Ordering of Events in a Distributed System”, *Communications of the ACM*, 21(7) :558–565, 1978.

[Lamport et al. 1982] L. Lamport, R. Shostak, & M. Pease, “The Byzantine Generals Problem”, *ACM Transactions on Programming Languages and Systems*, 4(3) :382–401, 1982.

[Laprie et al. 1996] J.-C. Laprie, J. Arlat, J. Blanquart, A. Costes, Y. Crouzet, Y. Deswarte, J.-C. Fabre, H. Guillermain, M. Kaâniche, K. Kanoun, C. Mazet, D. Powell, C. Rabéjac, & P. Thévenod, *Guide de la Sûreté de Fonctionnement*, Laboratoire d'Ingénierie de la Sûreté de Fonctionnement, Cepadues-Éditions, 1996, seconde édition.

[Larsen et al. 1997] K. Larsen, P. Petterson, & W. Yi, “UPPAAL in a nutshell”, *International Journal of Software Tools for Technology Transfer*, 1(1–2) :134–152, 1997, Springer-Verlag.

[Lestiennes & Gaudel 2002] G. Lestiennes & M.-C. Gaudel, “Testing Processes from Formal Specifications with Inputs, Outputs and Data Types”, dans *13th Int. Symposium on Software Reliability Engineering (ISSRE'02)*, pages 3–14, IEEE Computer Society, 2002, Annapolis, MD, USA.

[Lin & Hadzilacos 1999] K. Lin & V. Hadzilacos, “Asynchronous group membership with oracles”, dans *13th International Symposium on Distributed Computing (DISC'99)*, pages 79–93, Bratislava, Slovakia, Spinger Verlag, 1999, LNCS 1693.

[Liu & Layland 1973] C. L. Liu & J. Layland, “Scheduling algorithm for multiprogramming in a hard real-time environment”, *Journal of the ACM*, 20(1) :46–61, 1973.

[Loeffler & Serhouchni 1997] S. Loeffler & A. Serhouchni, “Creating a validated implementation of the Steam boiler control”, dans *3rd SPIN Workshop*, Enschede, The Netherlands, 1997.

[Luo & Pollack 1992] Z. Luo & R. Pollack, *LEGO Proof Development System : User's Manual*, Department of Computer Science, University of Edimburgh, 1992.

[Lussier & Waeselynck 2002a] G. Lussier & H. Waeselynck, “Analyse d'une preuve informelle pour guider le test”, dans *10èmes Journées Formalisation des Activités Concurrentes (FAC'02)*, Toulouse, France, LAAS rapport de recherche n° 02043, 2002, `http ://www.laas.fr/`sim`francois/SVF/FAC02/inputs/soum/guillaume.ps`.

[Lussier & Waeselynck 2002b] G. Lussier & H. Waeselynck, “Informal Proof Analysis Towards Testing Enhancement”, dans *13th Int. Symposium on Software Reliability Engineering (ISSRE'02)*, pages 27–38, Annapolis, MD, USA, IEEE Computer Society, 2002.

[Lussier & Waeselynck 2004a] G. Lussier & H. Waeselynck, “Collaboration entre test et preuve formelle : Test Guidé par la Preuve. Etude expérimentale sur l'algorithme d'appartenance de groupe du TTP/C”, dans *12èmes Journées Formalisation des Activités Concurrentes (FAC'04)*, Toulouse, France, LAAS rapport de recherche n° 04208, 2004, `http ://www.cert.fr/francais/deri/michel/FAC/2004/Papiers/L3.pdf`.

[Lussier & Waeselynck 2004b] G. Lussier & H. Waeselynck, “Deriving test sets from partial proofs”, dans 15th *Int. Symposium on Software Reliability Engineering (ISSRE'04)*, pages 14–24, Saint Malo, France, IEEE Computer Society, 2004.

[Lussier & Waeselynck 2004c] G. Lussier & H. Waeselynck, “Proof Guided Testing : Towards Complementarity of Verification Techniques”, Rapport technique, LAAS rapport de recherche n° 03198, 2004.

[Lussier et al. 2004] G. Lussier, H. Waeselynck, & K. Guennoun, “Proof-Guided Testing : an Experimental Study”, dans 28th *International Computer Software and Applications Conference (COMPSAC'04)*, pages 528–533, Hong-Kong, Chine, IEEE Computer Society, 2004.

[Manna & Pnueli 1994] Z. Manna & A. Pnueli, “Temporal Verification Diagrams”, dans M. Hagiya and J.C. Mitchell, editeur, *International Symposium on Theoretical Aspects of Computer Software (TACAS'94)*, pages 726–765, Sendai, Japan, Springer-Verlag, 1994, LNCS 789.

[Marre 1995] B. Marre, “LOFT : A Tool for Assisting Selection of Test Data Sets from Algebraic Specificatons”, dans *TAPSOFT : Theory and Practice of Software Development*, Lecture Notes in Computer Science, pages 799–800, Springer Verlag, 1995, LNCS 915.

[Marre & Arnould 2000] B. Marre & A. Arnould, “Test Sequences Generation from LUSTRE Descriptions : GATEL”, dans 15th *IEEE ACM International Conference on Automated Software Engineering (ASE'00)*, pages 229–238, Grenoble, France, IEEE Computer Society Press, 2000.

[Mattern 1988] F. Mattern, “Virtual Time and Global States in Distributed Systems”, dans *International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Gers, North-Holland, 1988.

[McMillan 1993] K. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, Boston, 1993.

[Merz 2001] S. Merz, “Model Checking : A Tutorial Overview”, *Modeling and Verification of Parallel Processes*, pages 3–38, Springer-Verlag, F. Cassez et al. (ed), 2001, LNCS 2067.

[Meyers 1979] G. Meyers, *The Art of Software Testing*, Wiley, New York, USA, 1979.

[Milner 1980] R. Milner, *A calculus of communicating systems*, Springer Verlag, 1980, LNCS 92.

[Mishra & Schlichting 1992] S. Mishra & R. Schlichting, “Abstractions for Constructing Dependable Distributed Systems”, Rapport technique TR 92 -12, Department of Computer Science, University of Arizona, AZ, USA, 1992.

[Monin 1996] J.-F. Monin, *Comprendre les méthodes formelles, Panorama et outils logiques*, Collection technique et scientifique des télécommunications, Masson & CNET-ENST, 1996, Paris.

[Monin 2000] J.-F. Monin, *Comprendre les méthodes formelles, Panorama et outils logiques*, Masson & CNET-ENST, 2000, 2nd édition.

[Moser & Melliar-Smith 1990] L. Moser & P. Melliar-Smith, “Formal Verification of Safety-critical Systems”, *Software Practice and Experience*, 20(8) :799–821, 1990.

[Muñoz 1999] C. Muñoz, “PBS : Support for the B-method in PVS”, Rapport technique SRI-CSL-99-01, SRI International, 1999.

[Musa et al. 1996] J. Musa, G. Fuoco, N. Irving, D. Kropfl, & B. Juhlin, “The Operational Profile”, *Handbook of Software Reliability Engineering*, pages 167–216, McGraw-Hill/IEEE Computer Society Press, M. Lyu (ed), 1996.

[Naito & Tsunoyama 1981] S. Naito & M. Tsunoyama, “Fault detection for sequential machines by transition tours”, dans 11th *IEEE Fault Tolerant Computing Symposium (FTCS 1981)*, pages 238–243, 1981.

[Nipkow et al. 2002] T. Nipkow, L. Paulson, & M. Wenzel, *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, Springer-Verlag, 2002, LNCS 2283.

[Norrish 1998] M. Norrish, *C formalised in HOL*, Thèse de doctorat, University of Cambridge, 1998, Rapport technique n° 453.

[Offutt et al. 2003] J. Offutt, S. Liu, A. Abdurazik, & P. Ammann, "Generating Test Data from State-Based Specifications", *Journal of Software Testing, Verification and Reliability*, 13 :25–53, 2003.

[Offutt & Untch 2000] J. Offutt & R. Untch, "Mutation 2000 : Uniting the Orthogonal", dans *Mutation 2000 : Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 45–55, San Jose, CA, 2000.

[Owre et al. 1995] S. Owre, J. Rushby, N. Shankar, & F. von Henke, "Formal Verification for Fault-Tolerant Architectures : Prolegomena to the Design of PVS", *IEEE Trans. on Software Engineering*, 21(2) :107–125, 1995.

[Park & Shaw 1991] C. Y. Park & A. C. Shaw, "Experiments with a Program Timing Tool Based on Source-Level Timing Schema", *IEEE Computer*, 24(5) :48–57, 1991.

[Paulin-Mohring 1989] C. Paulin-Mohring, *Extraction de programmes dans le calcul des constructions*, Thèse de doctorat, Université de Paris VII, 1989.

[Paulson 1994] L. Paulson, *Isabelle : A Generic Theorem Prover*, Springer-Verlag, 1994, LNCS 828.

[Paulson 2004] L. Paulson, "Isabelle bibliography", 2004, `http ://www.cl.cam.ac.uk/users/lcp/paper/isabelle.html`.

[Pfeifer 2000] H. Pfeifer, "Formal Verification of the TTP Group Membership Algorithm", dans T. Bolognesi & D. Latella, editeurs, *Formal Methods for Distributed System Development Proceedings of FORTE XIII / PSTV XX 2000*, pages 3–18, Pisa, Italy, Kluwer Academic Publishers, 2000.

[Pfeifer 2003] H. Pfeifer, *Formal Analysis of Fault-Tolerant Algorithms in the Time-Triggered Architecture*, Thèse de doctorat, Universität Ulm, Allemagne, 2003.

[Pfeifer et al. 1999] H. Pfeifer, D. Schwier, & F. W. von Henke, "Formal Verification for Time-Triggered Clock Synchronization", dans C. B. Weinstock & J. R. (eds.), editeurs, *7th Dependable Computing for Critical Applications (DCCA'99)*, volume 12 de *Dependable Computing and Fault-Tolerant Systems*, pages 207–226, IEEE Computer Society, 1999.

[Pfeifer & von Henke 2001] H. Pfeifer & F. W. von Henke, "Formal Analysis for Dependability Properties : the Time-Triggered Architecture Example", dans *8th IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA'01)*, pages 343–352, Antibes Juan-les-Pins, France, IEEE, 2001.

[Phalippou & Groz 1990] M. Phalippou & R. Groz, "Evaluation of an empirical approach for computer-aided test case generation", dans *3rd International Workshop on Protocol Test Systems*, pages 131–147, Washington, USA, 1990.

[Pimont & Rault 1979] S. Pimont & J. C. Rault, "An approach towards reliable software", dans *4th International Conference on Software Engineering*, pages 220–230, Munich, Germany, 1979.

[Pnueli 1981] A. Pnueli, "The temporal semantics of concurrent programs", *Theoretical Computer Science*, 13 :45–60, 1981.

[Powell et al. 1999] D. Powell, J. Arlat, L. Beus-Dukic, A. Bondavalli, P. Coppola, A. Fantechi, E. Jenn, C. Rabéjac, & A. Wellings, "GUARDS : a Generic Upgradable Architecture for Real-time Dependable Systems", *IEEE Transactions on Parallel and Distributed Systems*, 10(6) :580–599, 1999.

[Prasetya 1995] W. Prasetya, *Mechanically Supported Design of Self-stabilizing Algorithms*, Thèse de doctorat, Institute for Programming research and Algorithmics (IPA), Utrecht University, 1995.

[Puitg & Dufourd 1998] F. Puitg & J.-F. Dufourd, "Formal specification and theorem proving breakthroughs in geometric modeling", dans *11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'98)*, pages 401–422, Springer-Verlag, 1998, LNCS 1479.

[Puschner & Koza 1989] P. Puschner & C. Koza, "Calculating the Maximum Execution Time of Real-Time Programs", *Real-Time Systems*, 1(2) :159–176, 1989.

[Rapps & Weyuker 1985] S. Rapps & E. Weyuker, "Selecting Software Test Data Using Data Flow Information", *IEEE Transactions on Software Engineering*, 11(4) :367–375, 1985.

[Ribet et al. 2002] P. Ribet, F. Vernadat, & B. Berthomieu, "On Combining Persistent Sets Method with the Covering Steps Graph Method", dans *22th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'2002)*, pages 344–359, Springer-Verlag, 2002, LNCS 2529.

[Richardson & Clarke 1985] D. J. Richardson & L. A. Clarke, "Partition analysis : a method combining testing and verification", *IEEE Trans. on Software Engineering*, 11(12) :1477–1490, 1985.

[Rushby 1993] J. Rushby, "Formal Methods and the Certification of Critical Systems", Rapport technique, Computer Science Laboratory, SRI Int., 1993, CSL-93-7.

[Rushby 1995] J. Rushby, "Formal Methods and their Role in the Certification of Critical Systems", Rapport technique, Computer Science Laboratory, SRI Int., 1995, CSL-95-1.

[Rushby 1999] J. Rushby, "Integrated formal verification : Using model checking with automated abstraction, invariant generation and theorem proving", dans *Theoretical and Practical Aspects of SPIN Model Checking : 5th and 6th Int. SPIN Workshops*, pages 1–11, Springer-Verlag, 1999, LNCS 1680.

[Rushby 2000] J. Rushby, "Verification diagrams revisited : Disjunctive invariants for easy verification", dans E. A. Emerson & A. P. Sistla, editeurs, *Computer-Aided Verification (CAV'00)*, pages 508–520, Chicago, IL, Springer-Verlag, 2000, LNCS 1855.

[Rushby et al. 1991] J. Rushby, F. von Henke, & S. Owre, "An Introduction to Formal Specification and Verification Using EHDM", Rapport technique SRI-CSL-91-02, Computer Science Laboratory, SRI International, CA, USA, 1991.

[Rusu 2002] V. Rusu, "Verification using test generation techniques", dans *Formal Methods Europe (FME'02)*, pages 252–271, Copenhagen, Denmark, Springer-Verlag, 2002, LNCS 2391.

[Sabnani & Dahbura 1985] K. K. Sabnani & A. Dahbura, "A new technique for generating generating protocol tests", dans *9th Data Communications Symposium*, pages 36–43, Whistler Mountain, BC, Canada, ACM, 1985.

[Scheidler et al. 1997] C. Scheidler, G. Heiner, R. Sasse, E. Fuchs, H. Kopetz, & C. temple, "Time-triggered architecture (tta)", dans J.-Y. Roger, B. Stanford-Smith, & P. Kidd, editeurs, *Advances in Information Technologies : The Business challendge. (EMMSEC'97)*, pages 758–765, Florence, Italie, IOS Press, 1997.

[Sinha & Suri 1999a] P. Sinha & N. Suri, “Identification of test cases using a formal approach”, dans *29th Symposium on Fault-Tolerant Computing (FTCS’99)*, pages 314–321, IEEE Computer Society, 1999, Madison, WI.

[Sinha & Suri 1999b] P. Sinha & N. Suri, “On the use of formal techniques for analysing dependable real-time protocols”, dans *21st IEEE Real-Time Systems Symposium (RTSS’00)*, pages 126–135, IEEE Computer Society, 1999, Phoenix, AZ.

[Stauner et al. 1997] T. Stauner, O. Müller, & M. Fuchs, “Using HYTECH to verify an automotive control system”, dans O. Maler, editeur, *International Workshop on Hybrid and Real-Time Systems (HART 97)*, pages 139–153, Springer-Verlag, 1997, LNCS 1201.

[Steiner et al. 2004] W. Steiner, J. Rushby, M. Sorea, & H. Pfeifer, “Model Checking a Fault-Tolerant Startup Algorithm : From Design Exploration To Exhaustive Fault Simulation”, dans *International Conference on Dependable Systems and Networks (DSN’04)*, pages 189–198, Florence, Italie, IEEE Computer Society, 2004.

[Strauss & Ebenau 1994] S. Strauss & R. Ebenau, *Software Inspection Process*, McGraw-Hill, Inc., 1994.

[Theiling 2000] H. Theiling, “Extracting Safe and Precise Control Flow from Binaries”, dans *7th International Conference on Real-time Computing Systems and Applications (RTCSA)*, pages 23–30, Cheju Island, South Korea, IEEE Computer Society, 2000.

[Thévenod-Fosse & Waeselynck 1998] P. Thévenod-Fosse & H. Waeselynck, “Software statistical testing based on structural and functional criteria”, dans *11th International Software Quality Week (QW’98)*, volume 2, San Francisco, USA, 1998.

[Thévenod-Fosse et al. 1995] P. Thévenod-Fosse, H. Waeselynck, & Y. Crouzet, “Software Statistical Testing”, dans B. Randell, J.-C. Laprie, H. Kopetz, & B. Littlewood, editeurs, *Predictably Dependable Computing Systems*, pages 253–272, Springer Verlag, 1995.

[Tsai et al. 1999] T. Tsai, M.-C. Hsueh, H. Zhao, Z. Kalbarczyk, & R. Iyer, “Stress-Based and Path-Based Fault Injection”, *IEEE Transactions on Computers*, 48(11) :1183–1201, 1999.

[TTTech 2000] TTTech, “Specification of the TTP/C Protocol”, disponible sur demande à TT-Tech, 2000, `http ://www.tttech.com`.

[Ural 1992] H. Ural, “Formal methods for test sequence generation”, *Computer Communications*, 15(5) :311–325, 1992.

[Van Aertryck et al. 1997] L. Van Aertryck, M. Benveniste, & D. L. Metayer, “CASTING : a Formally Based Software Test Generation Method”, dans *IEEE International Conference on Formal Engineering Methods (ICFEM’97)*, pages 101–111, Hiroshima, Japon, 1997.

[Waeselynck 1993] H. Waeselynck, *Vérification des logiciels critiques par le test statistique*, Thèse de doctorat, Institut National Polytechnique de Toulouse, 1993.

[Waeselynck & Thévenod-Fosse 1999] H. Waeselynck & P. Thévenod-Fosse, “A case study in statistical testing of reusable concurrent objects”, dans *3rd European Dependable Computing Conference (EDDC-3)*, pages 401–418, Prague, République Tchèque, Springer Verlag, 1999, LNCS 1667.

[Weiser 1984] M. Weiser, “Program slicing”, *IEEE Transactions on Software Engineering*, 10(4) :352–357, 1984.

[Wensley et al. 1978] J. Wensley, L. Lamport, J. Goldberg, M. Green, K. Levitt, P. Melliar-Smith, R. Shostack, & C. Weinstock, “SIFT : The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control”, *Proceedings of the IEEE*, 66(10) :1240–1255, 1978.

[Weyuker 1982] E. Weyuker, “On Testing Non-Testable Programs”, *The Computer Journal*, 25(4) :465–470, 1982.

[Whittaker 1997] J. Whittaker, “Stochastic software testing”, *Annals of Software Engineering*, 4 :115–131, 1997.

[Yovine 1997] S. Yovine, “KRONOS : a verification tool for real-time systems”, *International Journal of Software Tools for Technology Transfer*, 1(1/2) :123–133, 1997.

Test guidé par la preuve –
Application à la vérification d'algorithmes de tolérance aux fautes

Résumé
Nos travaux étudient la conception du test en complément de preuves : l'objectif est de définir des critères de sélection de test qui ciblent les lacunes de ces preuves. Le champ d'application proposé est la vérification d'algorithmes de tolérance aux fautes. Les preuves considérées peuvent être des démonstrations informelles, publiées dans la littérature, ou des preuves formelles inachevées. Dans le premier cas, nous définissons une méthode basée sur une reformulation du discours informel sous forme d'un arbre de preuve. L'arbre offre une représentation de l'articulation logique de la démonstration, ainsi qu'un support pour son analyse pas à pas. La faisabilité et l'efficacité du test guidé par la preuve sont évaluées expérimentalement sur deux exemples d'algorithmes incorrects : un algorithme d'ordonnancement de tâches, et un algorithme d'appartenance de groupe. Les résultats montrent que l'identification des lacunes de la preuve peut s'avérer efficace pour guider le test, sous réserve que l'analyse de l'arbre ne mette pas en évidence un manque de rigueur affectant l'ensemble de la démonstration. Dans le cas de preuves formelles, nous reprenons le principe d'un test basé sur l'arbre de preuve. L'établissement d'un lien entre les lemmes non prouvés et des sous-espaces d'entrée de test peut alors être plus problématique que précédemment. L'étude expérimentale d'un autre algorithme d'appartenance de groupe, partiellement prouvé avec le système PVS, montre néanmoins que, lorsqu'un lien est possible, cette information peut s'avérer pertinente pour guider le test.

Mots-clefs : Test, Preuve mathématique, Algorithmes de tolérance aux fautes, Vérification logicielle, Collaboration de techniques de vérification.

Proof Guided Testing –
Application to the verification of fault-tolerance algorithms

Abstract
Our work studies the design of testing to supplement correctness proofs : the goal is to define test selection criteria which focus on the weak parts of the proof. The proposed field of application is the verification of fault-tolerance algorithms. The target proofs can be informal demonstrations, published in the literature, or partial formal proofs. In the first case, we define a method based on the reformulation of the informal discourse as a proof tree. This tree offers a representation of the logical structure of the demonstration, and a support for its step by step analysis. The feasibility and efficiency of proof guided testing are experimentally assessed using two examples of flawed algorithms : a task scheduling algorithm, and a group membership algorithm. The results show that identification of the weak parts of the proof can be effective to guide testing, provided that the tree analysis does not reveal a lack of rigor affecting the whole demonstration. In the case of formal proofs, we retain the principle of testing based on the proof tree. Establishing a link between unproved lemmas and subspaces of the test input domain can be more difficult than previously. Still, the experimental study of another group membership algorithm, partially proved with the PVS system, shows that, if a link can be established, this information can be relevant to guide testing.

Keywords : Dependability, Fault Tolerance Algorithms, Collaboration of Verification Techniques, Software Verification, Testing, Mathematical Proof.