Génération automatique de cas de test guidée par les propriétés de sûreté

Jérôme Vassy

▶ **To cite this version:**

Jérôme Vassy. Génération automatique de cas de test guidée par les propriétés de sûreté. Génie logiciel [cs.SE]. Université Joseph-Fourier - Grenoble I, 2004. Français. NNT : . tel-00007373

HAL Id: tel-00007373
https://theses.hal.science/tel-00007373

Submitted on 10 Nov 2004

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université Joseph Fourier - Grenoble I
UFR Informatique et Mathématiques Appliquées

THESE

pour obtenir le titre de

Docteur de l'Université Joseph Fourier

Discipline : Informatique

présentée et soutenue publiquement

par

Jérôme Vassy

le 19 Octobre 2004

Génération automatique de cas de test guidée par les propriétés de sûreté

Composition du Jury :

R. Groz	**Président**
P. Legall	**Rapporteurs**
R. Castanet	
B. Marre	**Examinateurs**
F. Ouabdesselam	
I. Parissis	

Remerciements

Le travail de recherche que je présente dans cette thèse a été effectué au sein de l'équipe VASCO du Laboratoire Logiciels, Systèmes et Réseaux. Je remercie M. Paul Jaquet et M. Farid Ouabdesselam, les deux directeurs successifs du Laboratoire LSR de m'avoir permis de mener ces travaux.

Je remercie aussi M. Roland Groz qui a bien voulu présider mon Jury ainsi que Mme Pascale Le Gall et M. Richard Castanet qui ont accepté de rapporter cette thèse.

Je remercie également:

- M. Bruno Marre qui a accepté de participer au Jury.
- M. Ioannis Parissis et M. Farid Ouabdesselam, mes deux directeurs de thèse qui se sont toujours montrés présents pour m'aider et me conseiller dans mon travail de recherche; ainsi que pour leurs relectures attentives de ce manuscrit.
- M. Jean-Luc Richier pour toutes ses remarques qui m'ont aidé à améliorer la qualité de ce travail et toute l'aide qu'il a pu m'apporter face aux difficultés techniques que j'ai rencontrées.
- L'ensemble de l'équipe administrative du LSR, Liliane, Pascale, Martine et Solange qui nous rendent la vie administrative plus facile.
- Francis, Karim, Olivier, Pierre, Rémy et Tanguy pour toute l'amitié qu'ils m'ont accordée au cours de ces années de thèse.
- Mes parents, Gilbert et Annie Vassy, pour leur soutien continue au cours de ces années de thèse.

Enfin je remercie les deux stagiaires, Cédric Messoumian et Angiolini Indiamaniry qui ont participé activement à l'élaboration de ce travail.

Table des matières

1 Introduction 9

I Contexte du travail: Test à partir de spécifications et test des logiciels réactifs synchrones 15

2 Validation à partir de spécifications 17
2.1 Test à partir de spécifications . . . 17
2.1.1 Spécifications et logiciels . . . 17
2.1.2 Génération automatique de données de test . . . 18
2.1.3 Construction de l'oracle . . . 24
2.1.4 Environnement du logiciel . . . 25
2.2 Problèmes liés à l'évaluation des techniques de test . . . 25
2.3 Validation des propriétés de sûreté . . . 25

3 Introduction à LUSTRE et son utilisation pour le test 29
3.1 Logiciels réactifs synchrones . . . 29
3.1.1 Logiciels réactifs . . . 29
3.1.2 Modèle synchrone . . . 30
3.2 Un langage dédié aux logiciels synchrones: LUSTRE . . . 31
3.2.1 Introduction au langage . . . 31
3.2.2 Tableaux et nœuds récursifs en LUSTRE . . . 33
3.2.3 Propriétés de logique temporelle . . . 34
3.3 Automate associé à un programme LUSTRE . . . 35
3.3.1 Arbre des exécutions . . . 35
3.3.2 Automate de contrôle . . . 36
3.4 Validation des logiciels basés sur LUSTRE . . . 36
3.4.1 LESAR . . . 37
3.4.2 Test structurel en LUSTRE . . . 37
3.4.3 LUTESS . . . 38
3.4.4 LURETTE . . . 38
3.4.5 GATeL . . . 39

4 L'environnement lutess **41**
4.1 L'outil LUTESS . . . 41
4.1.1 Introduction à LUTESS . . . 41
4.1.2 Les techniques de sélection des vecteurs d'entrées . . . 43
4.1.3 Structure syntaxique d'un nœud de test . . . 45
4.1.4 Exemple d'un climatiseur . . . 46
4.2 Fondements théoriques de LUTESS . . . 48
4.2.1 Sémantique des traces d'un nœud de test . . . 48
4.2.2 Simulateur d'environnement . . . 50
4.3 Technique d'implantation du simulateur d'environnement . . . 54
4.3.1 Les bdd . . . 54
4.3.2 Principes d'implantation du simulateur d'environnement . . . 55

II Test guidé par les propriétés de sûreté **61**

5 Test guidé par les propriétés de sûreté **63**
5.1 Motivations . . . 63
5.2 Expression des propriétés de sûreté dans le nœud de test . . . 64
5.2.1 Structure syntaxique . . . 64
5.2.2 Sémantique . . . 65
5.3 Simulateur guidé par les propriétés de sûreté . . . 65
5.3.1 Problématique . . . 65
5.3.2 Chemin dans l'environnement . . . 66
5.3.3 Propriétés de sûreté . . . 67
5.3.4 Simulateur d'environnement guidé par les propriétés de sûreté . . . 67
5.3.5 États suspects du simulateur d'environnement . . . 69
5.4 Calcul des ensembles de vecteurs d'entrées pertinents . . . 69
5.5 Approximations . . . 72
5.6 Stratégies de sélection d'un vecteur d'entrées . . . 76
5.6.1 Intersection . . . 77
5.6.2 Union . . . 77
5.6.3 Paresseuse . . . 78
5.7 Guidage systématique . . . 79
5.7.1 Cas général . . . 79
5.7.2 Cas de la stratégie *paresseuse* . . . 80
5.8 Problème du nombre d'instants à considérer . . . 81
5.8.1 Premières intuitions . . . 81
5.8.2 Des cas plus complexes . . . 81
5.9 Algorithmes . . . 82
5.9.1 Stratégie *intersection* et *union* . . . 82
5.9.2 Stratégie *paresseuse* . . . 83
5.10 Annexe: exemple d'un climatiseur . . . 84
5.10.1 Principes informels du climatiseur . . . 84

5.10.2 Description détaillée de l'environnement du climatiseur 85
5.10.3 L'environnement du climatiseur issu de l'analyse de Lutess 89
5.10.4 Calcul des états accessibles et des vecteurs d'entrées pertinents 91

6 Évaluation des stratégies proposées 97
6.1 Mesure de l'efficacité des séquences de test 97
6.2 Modélisation des services téléphoniques 99
6.2.1 Généralités 99
6.2.2 Interface entre l'environnement et le système 101
6.2.3 Prédicats 102
6.3 Propriétés de sûreté utilisées 102
6.3.1 Propriétés générales 103
6.3.2 Propriétés spécifiques 103
6.4 Aspects d'implantation 109
6.5 Situations suspectes atteintes 110
6.5.1 Présentation des résultats 110
6.5.2 CFBL - CFBL 110
6.5.3 CFBL - CELL 113
6.5.4 CELL - CW 115
6.5.5 CELL - TWC 118
6.5.6 CND - TCS 122
6.5.7 CW - TCS 123
6.6 Bilan de l'expérience 126
6.6.1 Quelques remarques particulières aux stratégies et approximations . . 126
6.6.2 Bilan général de l'expérience 129
6.6.3 Guide d'écriture des propriétés de sûreté 130

7 Vers un guidage plus précis 133
7.1 Motivations 133
7.2 Apprentissage par l'expérience 134
7.2.1 Principe 134
7.2.2 Construction des informations sur les vecteurs de sorties 135
7.2.3 Adaptation du calcul des vecteurs d'entrées pertinents 136
7.3 Utilisation d'un nouvel opérateur 139
7.3.1 Principe 139
7.3.2 Modification du calcul des vecteurs d'entrées pertinents 141
7.4 Discussions 142
7.4.1 Vers une notion de critère d'arrêt 142
7.4.2 Connaissances partielles des vecteurs de sortie, avantages et inconvénients 144

8 Conclusions 145

A L'analyseur d'états suspects 155
A.1 Utilisation de l'analyseur . . . 155
A.1.1 Fonctionnalités de l'analyseur . . . 155
A.1.2 Options de la ligne de commandes . . . 155
A.2 Architecture de l'analyseur . . . 156
A.2.1 Généralités . . . 156
A.2.2 Interface des objets . . . 157
Objet *AnalyzeFiles* . . . 157
Objet *HTable* . . . 158
Objet *ListHTElement* . . . 159
Objet *HTElement* . . . 160

B Définition des nœuds Lustre 163
B.1 Opérateurs de logique temporelle . . . 163
B.1.1 Opérateur $Implies(A, B)$ . . . 163
B.1.2 Opérateur $Always_Since(A, B)$ . . . 163
B.1.3 Opérateur $Once_Since(A, B)$ . . . 164
B.1.4 Opérateur $Always_From_To(A, B, C)$ . . . 164
B.1.5 Opérateur $Once_From_To(A, B, C)$ . . . 164
B.2 Manipulations des tableaux de température . . . 165
B.2.1 Test de l'égalité de deux températures . . . 165
B.2.2 Test de la supériorité de deux températures . . . 165
B.2.3 Test de l'infériorité de deux températures . . . 166
B.2.4 Modélisation de l'augmentation de température . . . 166
B.2.5 Modélisation de la diminution de température . . . 167
B.3 Le climatiseur . . . 167

Chapitre 1

Introduction

La complexité des logiciels ne cesse de s'accroître tant du point de vue de leurs tailles que des services qu'ils rendent. Il en résulte une difficulté croissante à assurer la qualité des logiciels, et à effectuer toutes les phases de conception et notamment celle concernant la validation. La validation regroupe la recherche des erreurs dans un logiciel avant de le mettre en service et l'évaluation de la confiance que l'on peut porter dans la qualité de ce logiciel.

Cependant, cette qualité globale n'étant pas formellement définie, maîtriser le risque induit par ce flou devient un problème majeur.

Pour faire face à cet enjeu, deux approches ont émergé, l'une fondée sur la preuve[1], l'autre sur le test. Au début, elles étaient opposées par les visions très tranchées du problème de la qualité qu'avaient les mondes académique et industriel, le premier ne croyant qu'en la preuve et le second préconisant le test. Maintenant, ces approches partagent au moins le souci d'avoir les meilleurs fondements possibles et de reposer sur une grande automatisation.

Il est clair que le "*zéro défaut*" ou "*presque zéro défaut*" ne peut être garanti que par le recours à des techniques formelles de preuve. Il est également évident qu'un logiciel prouvé a un coût souvent inacceptable pour le secteur industriel, quand cette preuve est possible. Le recours au test comme moyen principal de validation est donc un problème largement abordé.

Test des logiciels informatiques

Le but du test des logiciels peut être défini de diverses manières allant jusqu'à la controverse. Nous avons recueilli ci-dessous les trois définitions du test les plus courantes:

- l'IEEE définit le test comme "l'exécution ou l'évaluation d'un logiciel ou d'un composant, par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus".

[1] La preuve peut être réalisée par des techniques aussi bien de déduction que d'évaluation de modèle ("*model checking*").

- l'AFCIQ[2] définit le test comme "une technique de contrôle consistant à s'assurer, au moyen de son exécution, que le comportement d'un programme est conforme à des données préétablies".
- Myers [Mye79] définit le test comme "*l'activité qui exécute un programme dans l'intention de trouver des erreurs*"[3].

On note bien que les définitions proposées par l'IEEE et l'AFCIQ insistent sur le caractère "bon fonctionnement" du logiciel. En d'autres termes, le test a pour but de montrer que le logiciel remplit les fonctions pour lesquelles il a été conçu. Cette approche est notamment utilisée dans le cadre du test de conformité dans le domaine des protocoles de communication en s'appuyant sur des spécifications formelles. À l'opposé, Myers considère que le but du test est de détecter des erreurs car tous les programmeurs sont convaincus que les programmes contiennent des erreurs. L'idée est alors de construire des techniques de test produisant des données de test qui favorisent la mise en évidence des erreurs du logiciel.

La différence d'approche du test entre les définitions de l'IEEE ou l'AFCIQ et celle de Myers se retrouve également au niveau de la notion de réussite ou d'échec de l'exécution d'une donnée de test. En effet, dans les définitions de l'IEEE ou de l'AFCIQ, l'exécution d'une donnée de test sera considérée comme réussie si les sorties calculées par le logiciel sont conformes aux fonctionnalités qu'il doit remplir. Au contraire, dans le cas de la définition de Myers, l'exécution d'une donnée de test sera considérée comme réussie si cette dernière à permis de mettre en évidence une erreur dans le logiciel.

Dans la pratique, une technique de test définit un critère de sélection des données de test permettant de construire un jeu d'essai. La plupart des techniques de test peuvent être classées dans l'une des deux catégories suivantes:

- le test boîte noire: pour construire les jeux de test, ces techniques reposent sur des spécifications, au minimum sur le profil des fonctions du programme;
- le test boîte blanche: les techniques utilisent le code source du programme pour construire les données de test. Par exemple, le testeur peut se fixer comme objectif de "couvrir" toutes les instructions contenues dans le programme et construire ensuite l'ensemble des données de test lui permettant d'atteindre cet objectif.

Une difficulté du test réside dans le problème de l'arrêt. Beaucoup de stratégies proposent par définition une notion d'arrêt en produisant des jeux d'essais de taille finie: l'arrêt de la génération des données de test (et donc du test) pour satisfaire la couverture des instructions d'un programme se produira lorsque toutes les instructions auront été couvertes par l'ensemble des données de test. Cependant, comment déterminer si un jeu d'essai est suffisant pour avoir confiance dans le programme? Ne devrions nous pas utiliser plusieurs jeux d'essai issus de différentes techniques de test? Dans cette optique, l'arrêt du test se pose plutôt en termes d'efficacité du test, c'est à dire en termes de capacité des données de test à détecter des erreurs.

Mesurer l'efficacité d'une technique de test n'est pas facile. Le critère de sélection des données de test définit un objectif qui, s'il est atteint, offre une probabilité minimale de

[2] L'Association Française pour le Contrôle Industriel et la Qualité.
[3] "*Testing is the process of executing a program with the intent of finding errors*".

cerner une certaine classe d'erreur[4]. Cependant, la mise en œuvre d'une technique de test n'offre pas la garantie de révéler les erreurs selon leurs classes. C'est pourquoi le critère de sélection est parfois complété par le choix précis de données de test fondé sur l'expérience du testeur.

Une fois le jeu d'essai construit, il faut exécuter une à une les données de test qui le constituent et déterminer ceux qui ont réussi et ceux qui ont échoué. Dans toute sa généralité, savoir si l'exécution d'une donnée de test sur un programme P a réussi ou échoué est indécidable. Cependant, adapté à un domaine particulier, il est possible d'associer automatiquement un résultat attendu à une donnée de test donné. Ce problème est connu sous le nom de *problème de l'oracle*.

Problématique

Le processus de test repose sur la constitution et la mise en œuvre d'un jeu d'essais. Un jeu d'essai est un ensemble de données de test, c'est à dire des valeurs pour les paramètres d'entrée du logiciel. Ce processus peut être décomposé en trois grandes étapes qui mènent respectivement à la génération (sélection) des données de test du jeu d'essai, l'exécution du programme sous test avec chacune des données de test et enfin l'analyse fondée sur un oracle. Un oracle est un mécanisme qui détermine si la sortie calculée par le logiciel est correcte par rapport à la donnée de test fournie en entrée.

Parmi ces trois étapes, le choix des données de test conditionne l'ensemble des résultats qui pourront être obtenus du processus de test. La capacité des données de test sélectionnées à révéler les erreurs contenues dans le logiciel se présente comme un problème fondamental pour l'ensemble du processus de test et pour la confiance que l'on va pouvoir accorder au logiciel. Pour rendre ce choix le plus pertinent possible, les méthodes (ou techniques) utilisées pour la sélection des données de test tendent à se spécialiser en fonction des erreurs recherchées et des particularités des logiciels testés: le travail que nous présentons dans cette thèse porte de manière plus précise sur le test des logiciels critiques représentés par des logiciels réactifs synchrones.

Construire les jeux d'essais, exécuter les tests et enfin contrôler les résultats de façon totalement manuelle s'avère être une activité fastidieuse, génératrice d'erreur et très coûteuse en temps pour l'ingénieur chargé du test. Elle requiert les trois tâches suivantes:

- analyser les spécifications et/ou le code et choisir les données de test en s'appuyant sur une ou plusieurs techniques de test;
- construire et coder l'ensemble des éléments permettant d'exécuter les données de test sélectionnées et de recueillir les résultats produits par le logiciel;
- analyser les sorties du logiciel pour s'assurer de leur correction vis-à vis des données de test. Cette dernière tâche est certainement la plus délicate: la moindre inattention de la part de testeur peut impliquer la non-détection d'une erreur mise en évidence par le test.

[4]On ne s'intéresse pas ici à définir ces classes.

Afin de réduire ces coûts, l'automatisation des différentes étapes du test s'avère être une solution possible. Elle permet également de réduire le plus possible l'intervention du testeur et les erreurs qu'il peut induire.

Logiciels critiques

Parmi l'ensemble des logiciels, les logiciels critiques nécessitent un effort particulier pour leur validation. Les logiciels critiques sont une catégorie de logiciels où une défaillance peut entraîner une catastrophe humaine ou économique: nous pouvons citer l'exemple d'un accélérateur de particules destine à la radio-thérapie dont le programme de contrôle défectueux a entraîné la mort de plusieurs patients en 1986 [Jac89] ou encore la défaillance du logiciel de contrôle d'Ariane V qui a entraîné la destruction du lanceur.

Afin d'augmenter le plus possible la confiance que l'on peut placer dans un logiciel critique, des efforts sont faits à tous les niveaux de leur développement. En particulier les méthodes de spécifications formelles tentent de se généraliser. Un exemple significatif de ces pratiques est l'utilisation de la méthode B pour la conception des parties critiques du logiciel gérant la ligne de métro automatique *Météor* de Paris.

À l'autre extrémité du processus de développement, nous utilisons différentes techniques de validation des logiciels qui vont de la preuve au test. Du point de vue du test, la formalisation du ou des critères de sélection des données de test, ainsi que l'automatisation de leur génération contribue efficacement à l'amélioration de la qualité des logiciels. Ainsi, les efforts du testeur devront se porter non plus sur la conception des données de test, mais sur les objectifs du test (la description de ces objectifs).

Logiciels synchrones et logiciels réactifs

Les logiciels synchrones interagissent continuellement avec leur environnement. De plus, chaque fois que le logiciel reçoit de nouvelles entrées de la part de l'environnement, le calcul des sorties du logiciel doit être fait en un temps nul. De la même façon, les temps de communication entre le logiciel, ses différents composants et l'environnement doivent aussi être nuls.

La particularité des logiciels synchrones leur permet d'être particulièrement bien adaptés à la description des logiciels réactifs [HP85]. Les systèmes réactifs sont une catégorie de logiciels qui doivent toujours réagir aux sollicitations de leur environnement et plus rapidement que toute évolution de ce dernier. Les modéliser en tant que logiciels synchrones permet de prendre en compte cette obligation de réactivité.

D'autre part, les logiciels réactifs synchrones sont particulièrement bien adaptés à la description des logiciels critiques. En effet, les logiciels critiques sont généralement utilisés dans des systèmes contrôle/commande où le logiciel doit agir sur son environnement pour éviter toute catastrophe: il est donc indispensable qu'un tel logiciel ne manque aucune évolution de son environnement. Le modèle synchrone est notamment utilisé par *Airbus* pour les logiciels de contrôle du vol de ses avions.

Contribution de la thèse

Le but de ce travail est de contribuer à l'amélioration des techniques de validation des logiciels, et plus particulièrement des logiciels critiques pouvant être écrits comme des logiciels réactifs synchrones. Notre travail succède à ceux de Ioannis Parissis [Par96], Lydie Du Bousquet [DB99] et Nicolas Zuanon [Zua00b] sur la génération automatique de données de test. L'ensemble de ces travaux a donné naissance à l'outil LUTESS. Cet outil permet de générer des données de test en simulant le comportement de l'environnement: nous nous plaçons ici dans le cadre du test boîte noire. LUTESS utilise une version exécutable du logiciel sous test: les seuls éléments connus sont les spécifications de l'environnement du logiciel sous test.

Au cours de ce travail, nous avons conçu et adapté à LUTESS de nouvelles stratégies pour la sélection des données de test. Par défaut, LUTESS[5] choisit de façon aléatoire et équiprobable des données de test compatibles avec l'environnement. Les stratégies que nous proposons utilisent l'information contenue dans les propriétés de sûreté pour guider la génération automatique des données de test vers celles qui pourraient mener le logiciel sous test à leurs violations. Cette approche se situe totalement dans le cadre de la définition du test donnée par Myers. En effet, le guidage que nous proposons doit permettre de choisir les données de test qui ont le plus de chance de révéler une violation des propriétés de sûreté ou, plus exactement écarter les données de test qui n'ont aucune chance de révéler une violation des propriétés de sûreté.

De plus, les propriétés que nous considérons ont la particularité de pouvoir être exprimées en fonction des événements du passé. Toute la difficulté de notre approche consiste à être capable de recréer automatiquement l'ensemble des événements nécessaires à l'observation d'une violation des propriétés de sûreté.

Les différentes stratégies que nous avons proposées ont été évaluées sur des études de cas. Au cours de cette expérimentation, nous avons dû définir un critère de comparaison entre les différentes stratégies afin d'évaluer l'efficacité d'une stratégie à mener le logiciel sous test dans une situation où les propriétés de sûreté peuvent être violées.

À la suite de cette expérimentation, nous nous sommes intéressés à différentes possibilités d'évolution de nos stratégies afin d'accroître leur efficacité. Cette réflexion nous a notamment permis d'ébaucher un critère d'arrêt du test basé sur les propriétés de sûreté.

Organisation du document

Le manuscrit s'articule en deux grandes parties:

- La première partie concerne la situation de notre travail dans les domaines du test à partir de spécification (chapitre 2) et des logiciels réactifs synchrones (chapitre 3). Nous y présentons, en particulier, l'outil LUTESS (chapitre 4).
- La seconde partie constitue le cœur de notre travail. Nous y présentons les techniques de génération automatique des données de test guidé par les propriétés de sûreté (chapitre 5), l'évaluation de ces techniques (chapitre 6) et les évolutions que nous pouvons

[5] Le chapitre 4 sera dédié à l'étude complète de l'outil LUTESS.

envisager dans des développements futurs (chapitre 7). Enfin, nous donnons au chapitre 8 les conclusions et perspectives de ce travail.

Première partie

Contexte du travail: Test à partir de spécifications et test des logiciels réactifs synchrones

Chapitre 2

Validation à partir de spécifications

Dans ce chapitre, nous nous intéressons à quelques catégories de langages de spécifications pour lesquels des techniques de génération automatique des données de test ont été définies.

2.1 Test à partir de spécifications

2.1.1 Spécifications et logiciels

Le travail que nous présentons dans ce manuscrit s'inscrit dans le cadre du test fonctionnel à partir de spécifications. Classiquement, ces techniques sont basées sur une spécification déterminant les comportements possibles (ou attendus) du logiciel sous test et d'un critère de test sur ces dernières. Ensuite, des données de test sont générées plus ou moins automatiquement pour satisfaire le critère de sélection définis sur la spécification.

Les spécifications relatives à un logiciel se décomposent en trois groupes:

- *les spécifications fonctionnelles* décrivent les comportements attendus du logiciel, en général chaque fonction du logiciel séparément. Ces spécifications se décomposent en deux parties. Chacune des fonctions du logiciel est décrite à l'aide de son profil incluant le domaine de définition de la fonction et son domaine des valeurs[1] et d'une ou plusieurs relations liant les entrées aux sorties de la fonction;
- Le deuxième groupe contient les spécifications décrivant les propriétés globales s'appliquant au logiciel dans son environnement. On les dénomme "*propriétés*". Typiquement, la description des propriétés de sûreté tombe dans cette catégorie: elles décrivent globalement les comportements sûrs du logiciel en fonction de l'évolution de son environnement;
- Enfin, le dernier type de spécification concerne la description de l'environnement. Ces spécifications ne sont pas directement liées au fonctionnement du logiciel, mais décrivent comment l'environnement d'exécution du logiciel évolue. On les appelle "*contraintes d'environnement*".

[1] On suppose que le domaine de définition correspond aux paramètres ou variables d'entrées du programme, et qu'il n'y a pas d'état interne du programme ou de son contexte matériel influant sur le résultat.

2.1.2 Génération automatique de données de test

Test à partir de spécifications algébriques

Dans le cas idéal, il faudrait tester un programme exhaustivement pour être sûr qu'il ne contient aucune erreur. Cela revient à créer une donnée de test pour chaque *n-uplet* de valeurs du domaine de définition[2] du programme. Cependant, la taille de ce domaine (même fini), est trop grande et rend illusoire le test exhaustif. Prenons l'exemple d'un compilateur: il n'est pas possible de le tester exhaustivement étant donné que le nombre de programmes que l'on peut écrire dans un langage donné est infini. Nous devons sélectionner un petit nombre d'exemples (ou données de test) significatifs pour s'assurer de la correction de notre compilateur. Ce souci de produire des données de test ayant des propriétés équivalentes au test exhaustif se retrouve dans les travaux de Gilles Bernot, Marie-Claude Gaudel et Bruno Marre [BGM91]. Ils proposent d'introduire la notion de contexte de test permettant de regrouper des hypothèses sur le logiciel, l'ensemble des données de test et l'oracle. L'idée est de construire par raffinements successifs un contexte de test ayant les mêmes propriétés que le test exhaustif. À chaque raffinement, les données de test du contexte de test doivent conserver deux propriétés:

- *non biaisé*: un ensemble de données de test est dit *non biaisé* s'il ne rejette pas de programmes corrects.
- *valide*: un ensemble de données de test est dit *valide* s'il accepte uniquement des programmes corrects

Des hypothèses permettent de réduire l'ensemble des données de test sélectionnées. Les deux hypothèses les plus courantes sont l'hypothèse de régularité et l'hypothèse d'uniformité:

- soit L le logiciel sous test et t une mesure de la complexité. L'hypothèse de régularité d'ordre k suppose que si le logiciel L se comporte correctement pour tout $t \leq k$, alors il se comportera correctement quel que soit t. Considérons un logiciel permettant de calculer $n!$ de façon itérative. Dans ce cas, si le logiciel calcule correctement $5!$, l'hypothèse de régularité supposera que le logiciel calculera correctement $n!$ (peu importe le nombre d'itérations, le résultat restera correct);
- l'hypothèse d'uniformité suppose que si le logiciel produit un résultat correct pour une donnée de test d'un domaine D, alors le résultat sera correct pour toutes les données du même domaine D. Par exemple, cette hypothèse est utilisée dans le cadre du test par partition [RC81] où l'ensemble de données de test est constitué d'une valeur dans chacun des éléments de la partition.

Dans [BGM91], les auteurs ont appliqué leur théorie dans un outil permettant de produire des données de test à partir de spécifications algébriques. Les spécifications algébriques permettent de décrire les opérations associées aux structures de données d'un programme (*sorte*). Chacune des opérations est définie par une signature et un ensemble d'invariants:

- la signature permet de décrire les ensembles d'entrées et de sorties de l'opération;
- les invariants permettent de décrire comment la structure de donnée évolue après une opération.

[2] Cet ensemble est également appelé domaine des entrées.

Considérons un exemple de spécifications algébriques représentant une liste d'entiers triée:

$\Delta NATLISTusesNAT$

$\Delta S = \{NatList\}$
$S_{Obs} = \{Nat, Bool\}$

$\Delta \Sigma =$

$$\begin{array}{lcrcrcl} empty & : & & & & \rightarrow & NatList \\ cons & : & Nat & * & NatList & \rightarrow & NatListe \\ sorted & : & & & NatList & \rightarrow & Boolean \\ insert & : & Nat & * & NatList & \rightarrow & NatListe \end{array}$$

$Generators: \ empty, \ cons$

$\Delta Ax =$

$$\begin{array}{lcl} sorted(empty) & = & true \\ sorted(cons(N1, empty)) & = & true \\ sorted(cons(N1, cons(N2, L))) & = & and(\leq (N1, N2), sorted(cons(N2, L))) \\ insert(N1, empty) & = & cons(N1, empty) \\ \leq (N1, N2) = true & \Rightarrow & insert(N1, cons(N2, L)) = cons(N1, cons(N2, L)) \\ \leq (N1, N2) = false & \Rightarrow & insert(N1, cons(N2, L)) = cons(N2, cons(N1, L)) \end{array}$$

où $N1$ et $N2$ sont des variables de sorte Nat et L de sorte $NatListe$. Prenons l'axiome définissant l'opération de tri: l'hypothèse de régularité d'ordre 2 permet de clore l'ensemble de listes triées en sélectionnant les éléments suivants:

$$\begin{array}{rcl} sorted(cons(N1, cons(N2, empty))) & = & and(\leq (N1, N2), cons(N2, empty))) \\ sorted(cons(N1, cons(N2, cons(N3, empty)))) & = & and(\leq (N1, N2), \\ & & sorted(cons(N2, cons(N3, empty)))) \end{array}$$

La réduction du nombre des données de test avec l'hypothèse de régularité n'est pas suffisante: les variables entières permettent de définir un trop grand nombre de données de test. L'hypothèse d'uniformité permet de réduire cet ensemble de données de test en instanciant les variables $N1$, $N2$ et $N3$:

$$\begin{array}{rcl} sorted(cons(5, cons(3, empty))) & = & and(\leq (5, 3), cons(5, empty))) \\ sorted(cons(4, cons(2, cons(6, empty)))) & = & and(\leq (4, 2), sorted(cons(2, cons(6, empty)))) \end{array}$$

Un outil qui permet de générer automatiquement des données de test en s'appuyant sur des spécifications algébriques et les hypothèses de régularité et d'uniformité a été développé. Cet outil a permis de générer un ensemble de données de test significatif pour l'exemple non trivial de la liste d'entiers triée, qui doivent être soumises au logiciel sous test: la correction des réponses du logiciel est vérifiée à partir des axiomes présents dans les spécifications. D'autre part, des expérimentations de cette méthode ont été faites sur les spécifications du métro automatique de la ville de Lyon [DM91].

Dans le prolongement de [BGM91], les travaux présentés dans [AMLG99] propose une solution pour générer automatiquement des données de test aux limites: beaucoup d'erreurs sont dues à un mauvais traitement des cas aux limites, ou encore, à une sous-évaluation de la taille des données que le programme aura à traiter. La solution proposée utilise des spécifications algébriques bornées. Ces spécifications sont constituées de deux parties:

- la "*partie idéale*", selon les termes de l'auteur, permet de spécifier des structures de données et les opérations associées par l'intermédiaire de spécifications algébriques telles qu'elles sont utilisées dans [BGM91];
- la "*partie bornée*" permet de définir les bornes des structures de données spécifiées dans la partie idéale.

La technique de test présentée dans [AMLG99] a notamment permis de générer automatiquement des arbres binaires de recherche sur des entiers (16 bits) constitués de quinze nœuds avec une profondeur de dix.

Test à partir de spécifications B

D'autres langages de spécifications comme B ou Z décrivent les opérations d'un logiciel par l'intermédiaire de *pre* et *post* conditions. Les travaux de Bruno Legeard et Fabien Peureux [LP01] proposent une technique de génération des données de test aux limites en s'appuyant sur une spécification en B du logiciel. L'objectif de ces travaux est de s'assurer l'adéquation entre les spécifications du logiciel et le cahier des charges: les spécifications B sont utilisées ici comme une spécification pour le test permettant la génération automatique de données de test et non dans le cadre du développement du code du logiciel. Les spécifications B du logiciel sont traduites en différents ensembles de contraintes [PLT00] permettant de représenter l'évolution de la machine B par un automate. L'état d'un tel automate correspond aux valeurs des variables de la machine B. Le domaine des variables de la machine B est ensuite partitionné en fonction des prédicats qui les définissent: l'utilisation de $IF ... THEN ... ELSE$ permet, par exemple de décomposer le domaine des variables entrant dans l'expression de la condition selon le fait qu'elles rendent *vrai* ou *faux* l'évaluation de cette dernière. Cette partition des domaines des variables permet d'obtenir différents comportements du logiciel à couvrir. La stratégie de couverture fonctionnelle des spécifications est complétée par une stratégie de sélection des données de test aux bornes des domaines précédemment définis. Afin de mettre en œuvre cette dernière stratégie, un ensemble d'états aux limites est construit. Un état est dit aux limites si au moins une des variables qui le composent, est évaluée avec une valeur limite (ces valeurs sont définies à partir du partitionnement du domaine de définition de la variable). L'approche proposée dans [PLT00] se décompose en trois parties:

- construction pour chaque état aux limites s d'une séquence de données de test qui permet de mener le système spécifié par la machine B de son état initial à l'état s;
- une fois le système placé dans l'état s, le test de l'opération est effectué par l'appel de cette dernière;
- enfin, des opérations d'observations sont appelées et un verdict est rendu: les résultats obtenus sur la simulation du système doivent être les mêmes que ceux obtenus sur

l'implantation du système. Si une erreur est détectée sur un sous-domaine alors les auteurs considèrent qu'elle se produira pour l'ensemble du sous-domaine: ceci correspond à l'hypothèse d'uniformité présentée dans [BGM91].

Dans [LPU02], une évaluation de la technique proposée a été faite sur un sous-ensemble de la norme GSM 11-11. Ce travail met notamment en parallèle un jeu de test construit manuellement par les ingénieurs de SCHLUMBERGER avec celui qui est généré automatiquement par les travaux de Bruno Legeard et Fabien Peureux. La technique de génération automatique engendre plus de 200 données de test. Cependant, sur les 200 données de test produites, un nombre important se sont révélées peu intéressantes à cause des répétitions. Globalement, cette étude a permis de montrer que les données de test générées automatiquement couvrent environ 85% des données de test produites manuellement tout en économisant 30% du temps de conception.

Test à partir de spécifications booléennes

Les travaux que nous avons présentés précédemment sont basés sur des descriptions ensemblistes des données et des propriétés ou *pre/post* conditions définissant comment les opérations doivent faire évoluer les données. Dans [WGS94], Weyuker, Goradia et Singh utilisent une description booléenne liant les entrées aux sorties du logiciel. Ils proposent différentes stratégies pour générer automatiquement des données de test à partir d'une spécification booléenne. Chacune des stratégies proposées est évaluée sur un sous-ensemble des spécifications d'un logiciel permettant d'éviter les collisions d'avion en vol (TCAS II). Le principe de base des stratégies décrites est de sélectionner des données de test de telle sorte que la modification de la valeur d'une seule variable entraîne la modification de la valeur de vérité de la formule booléenne utilisée comme spécification. L'efficacité des stratégies est mesurée par l'intermédiaire du test mutationnel. Le taux de mutants tués est de l'ordre 98%. Ce taux a été comparé à celui obtenu avec un ensemble de données de test de même taille généré aléatoirement. Dans le cas du test aléatoire, le taux de mutants tués varie entre 43% et 87%.

Ces descriptions booléennes peuvent également être représentées par des graphes causes-effets [Mye79, Elm73]. Ces derniers permettent d'écrire les spécifications d'un logiciel en associant les causes aux effets. Les causes représentent des conditions d'entrée alors que les effets représentent les conditions de sortie. À chaque cause et chaque effet est associé un nœud. Les nœuds sont reliés entre eux par un des quatre opérateurs (*identité, non, et, ou*) représentés sur la figure 2.1. Il est possible d'utiliser des conditions intermédiaires entre les causes et les effets en associant également un nœud à chaque condition intermédiaire. Précisons également qu'il est possible de mettre des contraintes sur les causes[3] (par exemple deux causes, a et b ne peuvent pas se produire en même temps).

Considérons un programme simple à trois entrées (i_1, i_2 et i_3) et deux sorties (o_1 et o_2). Nous souhaitons que notre programme émette la sortie o_1 lorsque i_1 et i_2 sont égales à vrai, alors que la sortie o_2 est émise lorsque nous avons seulement i_3. Cette spécification peut être représentée par le graphe causes-effets de la figure 2.2.

[3]Nous donnons ici uniquement les éléments dont nous avons besoin: nous ne détaillerons le formalisme utilisé pour l'expression des contraintes sur les causes.

FIG. 2.1: Opérateurs des graphes causes-effets

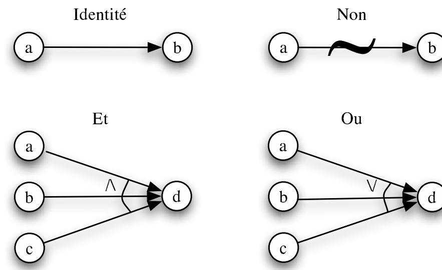
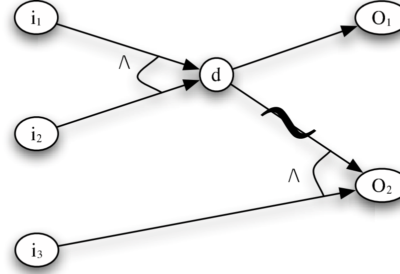


FIG. 2.2: Exemple de graphes causes-effets



Pour construire le graphe causes-effets associé à notre exemple (figure 2.2), nous introduisons le nœud intermédiaire d qui correspond à la conjonction des causes i_1 et i_2. Depuis le nœud d, nous obtenons l'effet o_1 par identité, alors que l'effet o_2 est obtenu par la conjonction de i_3 avec la négation de d.

Beitz et Rout donnent dans [BR93] la description d'un outil pour la génération automatique de données de test basée sur les graphes causes-effets. Ils définissent une donnée de test comme un couple (i, o) où o est la sortie que doit générer le logiciel sous test lorsque i est donnée en entrée. L'outil présenté dans [BR93] permet de construire un graphe causes-effets à partir d'un langage de spécification. Les données de test sont ensuite dérivées en trois étapes:

- premièrement, un parcours du graphe causes-effets est effectué afin de déterminer l'ensemble des combinaisons d'entrées possibles;
- dans un second temps, les combinaisons d'entrées sont regroupées en classes d'équivalence en fonction des parties activées dans le graphe causes-effets;
- enfin, une donnée de test est choisie dans chaque classe d'équivalence.

Les travaux présentés dans [VTP94, TVPL94, VTP95] utilisent également les graphes causes-effets pour représenter les spécifications du logiciel sous test. Contrairement à Beitz et Rout [BR93] qui utilisent une heuristique directement sur les graphes causes-effets, les auteurs commencent par convertir les graphes causes-effets en formules booléennes. Chaque formule booléenne est construite à partir de plusieurs prédicats reliés les uns aux autres par les opérateurs booléens *and* et *or*. Un prédicat peut être soit une variable booléenne, soit

une expression relationnelle du type

$$E_1 \ < rel_{op} > \ E_2$$

où E_1 et E_2 sont des expressions arithmétiques, et $< rel_{op} >$ est un opérateur de comparaison. Ils définissent ensuite différentes classes de faute possible sur les formules booléennes et les critères que doit respecter un ensemble de données de test afin qu'il soit efficace pour la détection de la classe de faute considérée. Ces travaux caractérisent sept classes de faute sur les formules booléennes:

- fautes sur les opérateurs booléens: il s'agit d'une mauvaise utilisation des opérateurs booléens, par exemple l'écriture d'un *and* à la place d'un *or* ou une utilisation abusive de la négation;
- fautes sur les opérateurs de relations: les erreurs appartenant à cette classe résultent d'une mauvaise utilisation des opérateurs $< rel_{op} >$, par exemple $E_1 \geq E_2$ au lieu de $E_1 > E_2$;
- fautes de parenthèsage;
- fautes de variables booléennes: utilisation d'une variable à la place d'une autre;
- expressions arithmétiques incorrectes;
- opérande en trop: les fautes appartenant à cette classe sont dues à la présence de prédicats “en trop” dans l'expression booléenne.
- manque d'une opérande: les fautes appartenant à cette classe sont dues à l'absence d'un ou plusieurs prédicats dans l'expression booléenne.

Les travaux de [VTP94, TVPL94, VTP95] s'intéressent plus particulièrement aux trois premières classes de fautes pour lesquelles deux critères ont été définis :

- BOR[4] : ce critère impose aux jeux de test de garantir la détection des fautes d'opérateurs booléens.
- BRO[5] : ce second critère étend le premier en imposant aux jeux de test de garantir également les fautes d'opérateurs relationnels.

L'ensemble des travaux présenter dans [VTP94, TVPL94, VTP95] ont pour but de valider la pertinence des critères BOR et BRO sur l'exemple de la chaudière et sur n versions d'un programme (écrit par n personnes différentes à partir d'une même spécification). Les résultats de ces expériences ont montré que les algorithmes proposés ont un pouvoir de couverture des critères BOR et BRO équivalent à celui de l'algorithme d'Elmendorf [Elm73] pour un nombre de données de test réduit de l'ordre de la moitié. Une autre comparaison avec la génération aléatoire des données de test montre également que les algorithmes proposés ont au moins le même pouvoir de couverture des critères pour un nombre de données de test moins important.

Une autre approche présentée dans [ZC03] propose de spécifier les comportements du logiciel sous test par un automate non déterministe. Le test est constitué de deux phases:

---

[4] BOR: Boolean OperatoR.

[5] BRO: Boolean and Relational Operator.

- amener le logiciel sous test dans un état cible de la spécification;
- tester la validité d'une propriété contre la spécification dans l'état cible.

Toute la difficulté de cette approche consiste à construire une séquence d'entrées qui va mener le logiciel sous test dans l'état cible. En effet, si dans un automate déterministe une même séquence d'entrées mène au même état, ce n'est plus vrai pour un automate non déterministe. Les travaux présentés dans [ZC03] définissent un algorithme qui permet de construire une séquence d'entrées menant le logiciel sous test dans un état de sa spécification décrite à l'aide d'un automate non déterministe.

D'autres approches comme dans [RC81] proposent de partitionner le domaine d'entrée d'un logiciel en classes d'équivalence. Les classes d'équivalence sont déterminées à partir des spécifications soit en utilisant les propriétés des différents domaines de définition des variables d'entrées, soit en associant les classes d'équivalence aux différentes fonctionnalités du logiciel. L'ensemble des données de test est construit en choisissant une entrée par classe d'équivalence.

L'ensemble des résultats obtenus dans les différents travaux que nous avons présentés montre que l'efficacité du test est significativement augmentée par rapport au test aléatoire lorsqu'une stratégie de sélection des données de test est appliquée. Cependant, les mesures d'efficacité d'une stratégie sont liées aux types d'erreur recherchés. En conséquence, il est nécessaire de bien identifier les différents types d'erreur afin d'appliquer les stratégies les mieux adaptées. Enfin, l'avantage du test aléatoire réside dans le fait qu'il est facile de produire une très grande quantité de données de test et que leur génération ne sera pas influencée par un type d'erreur particulier.

2.1.3 Construction de l'oracle

Les techniques de test que nous avons présentées précédemment utilisent des spécifications fonctionnelles pour la génération des données de test. Avec de telles spécifications, il est possible de déduire (ou construire) un oracle qui déterminera automatiquement si le résultat obtenu à la suite de l'exécution d'une donnée de test est correct ou non. Une première idée simple pour la construction d'un oracle consiste à associer à chaque donnée de test la sortie attendue. Une telle solution peut être mise en œuvre manuellement par une analyse des spécifications par le testeur. Cependant, une construction manuelle de l'oracle est génératrice d'erreur: cette étape cruciale pour la détection d'erreur dans le logiciel doit être en grande partie automatisée.

Dans [RLO92], Debra Richardson, Stéphanie Leif Aha et Owen O'Malley proposent une solution pour permettre de déduire l'oracle de façon systématique. La solution proposée s'adresse plus particulièrement au logiciels réactifs testés à partir de leurs spécifications. Ces travaux proposent de construire à partir de la spécification formelle du logiciel une abstraction interprétable du logiciel (oracle). Dans une seconde étape, il est nécessaire pour le testeur de construire la relation qui lie l'ensemble des sorties et l'état observé du logiciel à l'oracle: de cette façon, il est possible de s'assurer que le comportement attendu du logiciel correspond à celui qui est observé.

2.1.4 Environnement du logiciel

Notre travail se distingue des approches que nous venons de présenter par le fait que nous nous servons non pas d'une spécification pour déterminer quels seraient les comportements du logiciel sous test, mais pour simuler l'évolution de l'environnement d'exécution du logiciel sous test. Bien que notre spécification de l'environnement soit donnée sous la forme d'expressions booléennes, les critères de couverture couramment utilisés pour déterminer l'ensemble des données de test pouvant couvrir la spécification ne sont pas pertinents dans notre cas: en effet, cela signifierait que nous cherchons à couvrir l'environnement et non le logiciel (ou ses spécifications, ses fonctionnalités!). Une présentation complète de l'outil LUTESS (dans lequel s'inscrit notre travail) sera donnée au chapitre 4.

2.2 Problèmes liés à l'évaluation des techniques de test

L'approche adoptée dans [FI98] diffère de celle de [VTP94, TVPL94, VTP95] par le fait que nous nous situons ici dans le domaine du test structurel (ou boîte blanche). Bien que nous nous situions dans le domaine du test à partir de spécification, ces travaux utilisent des critères de couverture équivalent à ceux que nous venons d'étudier. En effet, l'idée est toujours de générer des données de test dans le but de couvrir un critère non plus exprimé sur les spécifications mais directement sur le code du programme. Dans le cas de ce travail, Frankl et Iakounenko souhaitent évaluer deux critères de test structurel (couverture des décisions et couverture de toutes les utilisations des variables[6]) sur un programme de configuration d'une antenne satellite de l'ESA[7] représentant 10000 lignes de code C.

Dans ce travail, un accent particulier est mis sur les difficultés rencontrées lors de l'évaluation d'un critère de manière significative: si nous considérons un programme P, sa spécification S et un critère C, alors nous pouvons déterminer un grand nombre d'ensembles de données de test satisfaisant C pour P et S; certains de ces ensembles pouvant mener à la détection d'une ou plusieurs erreurs alors que d'autres n'en détecteront aucune. Ces difficultés peuvent également se retrouver au niveau des spécifications d'autant plus que les critères de couverture des spécifications (en particulier dans les travaux de [VTP94, TVPL94, VTP95]) servant à la génération des données de test sont semblables à ceux utilisés dans le cadre du test structurel.

2.3 Validation des propriétés de sûreté

Dans [MP95], Manna et Pnueli définissent les propriétés de sûreté comme devant être valides (c'est à dire évaluées à *vrai*) dans tous les états du logiciel. Ces propriétés peuvent être vues d'une façon très générale comme des invariants du logiciel et peuvent être considérées comme un cas particulier de spécification: spécification des comportements sûrs du programme.

[6] Une définition de ces critères de test structurel peut être trouvé dans [XRK00].

[7] European Space Agency.

Nous pouvons aborder la validation de propriétés de sûreté soit par la vérification formelle (ou preuve), soit par le test. Les techniques de validation basées sur la preuve des propriétés de sûreté ont l'avantage de pouvoir garantir totalement le respect des propriétés de sûreté lorsque la preuve réussit; cependant, si la preuve ne peut aboutir (à cause d'un manque de puissance de calcul par exemple), nous ne pouvons rien déduire sur la validité de la propriété. En revanche, le test aboutit toujours: dès qu'une donnée de test est exécutée, nous avons un résultat pour cette donnée de test. Toute la difficulté du test consiste à bien choisir l'ensemble des données de test afin de détecter le maximum de fautes.

Le travail présenté par O. Laurent, P. Michel et V. Wiels dans [LMW01] propose de réduire l'effort consacré au test et à la simulation en introduisant de la vérification formelle. Ce travail montre une application possible des méthodes formelles dans le cadre de la validation d'un logiciel de contrôle de vol pour *Airbus A340*. Il constitue une première évaluation de la vérification formelle de propriétés sur un contrôleur de vol *Airbus* à partir de technique de *Model-Checking*. Les résultats de cette étude montrent qu'il est aisé de vérifier les propriétés à l'aide d'outils[8] une fois qu'elles sont écrites. Cependant, cette étude montre aussi des difficultés comme le fait que certaines fonctions soient directement implantées en *C* et doivent être remplacées par des assertions caractérisant leurs comportements. Ce dernier point constitue un véritable point faible pour la preuve du logiciel: ici la preuve est faite sur le code du logiciel dans lequel une partie du code est remplacée par des assertions. En conclusion de ce travail, les auteurs considèrent la preuve comme un moyen de réduire l'effort de test et de simulation sur une partie du logiciel.

Toujours dans le domaine de la vérification des propriétés de sûreté, les travaux présentés dans [BCC+03] proposent de construire un modèle du programme directement à partir du code *C*. Les auteurs de ces travaux insistent également sur le fait qu'il est assez difficile d'écrire les propriétés de sûreté d'un logiciel critique et s'orientent vers la preuve de l'absence de certaines classes de fautes (par exemple, division par zéro ou encore accès à un tableau en dehors des bornes).

Cependant, les approches totalement automatiques basées sur le *model checking* se trouvent souvent confrontées au problème d'explosion combinatoire du nombre d'états à explorer. Afin de limiter ce problème, Heitmeyer propose dans [HKL+98] de combiner des techniques d'abstraction et de model checking pour détecter des violations de propriétés de sûreté au niveau des spécifications. Trois méthodes d'abstraction sont proposées et définies formellement:

- une première abstraction permet la suppression des variables non utilisées. Elle consiste à construire un modèle des spécifications en éliminant toutes les variables qui n'interviennent pas dans l'évaluation d'une propriété (analogue au *slicing*);
- la seconde abstraction permet de définir un ensemble de variables V tel que pour toutes variables $v \in V$, il existe une variable $\hat{v}$ qui soit la seule variable dont la valeur dépende directement de celle de v: il est alors possible de supprimer dans le modèle abstrait les variables contenues dans l'ensemble V;
- la dernière abstraction proposée permet de simplifier les domaines de définition des variables. Par exemple, il est possible de réduire la complexité d'une variable entière en considérant un petit nombre d'intervalles de valeurs.

[8] Pour cette étude, deux outils de model-checking ont été utilisés: *Lesar* [Rat92] et *Lucifer* [Lju99].

La technique proposée dans [HKL+98] consiste à construire un modèle des spécifications par l'intermédiaire d'une où plusieurs abstractions puis à utiliser des techniques de "model checking" pour vérifier les propriétés de sûreté sur l'abstraction. Si un contre-exemple est mis en évidence sur le modèle abstrait, il est nécessaire de le vérifier sur le modèle concret: les différentes étapes du contre-exemple abstrait obtenu peuvent être séparées par des événements concrets masqués par les abstractions. L'ensemble de tous ces événements doit être retrouvé pour constituer le contre-exemple concret.

Une autre approche basée sur le *theorem proving* a été développée par [RB01] afin de prouver des propriété de sûreté sur des programme Ladder Diagram (langage de programmation pour les automates programmables industriels). Les approches basées sur le *theorem proving* sont fortement dépendantes du type de propriété pour lesquelles elles ont été définies. Leurs utilisations requièrent souvent une grande expertise de la part de l'utilisateur, notamment pour donner aux outils de bons "schémas de preuve".

Chapitre 3

Introduction à Lustre et son utilisation pour le test

Nous avons présenté au chapitre précédent quelques approches pour le test des logiciels à partir de spécifications. Avec ce chapitre, nous abordons le thème central de notre travail: le test des logiciels réactifs synchrones spécifiés en Lustre. Nous décrivons d'abord les principales caractéristiques des logiciels réactifs synchrones. Nous nous intéressons ensuite au langage Lustre dédié à la spécification et à la programmation des applications synchrones; nous en donnons les principes et surtout en esquissons l'usage comme moyen d'écriture de propriétés de logique temporelle. En présentant la modélisation d'un programme Lustre sous la forme d'un automate de Mealy, nous montrons le lien entre tout code Lustre et une machine d'états finis. Un court résumé des travaux menés en France sur le test d'applications spécifiées ou programmées en Lustre clôt ce chapitre.

3.1 Logiciels réactifs synchrones

3.1.1 Logiciels réactifs

Les logiciels réactifs sont définis dans [HP85] comme des logiciels qui interagissent de façon permanente avec leur environnement (voir figure 3.1). Ils sont opposés aux logiciels transformationnels qui, lorsqu'ils sont activés, prennent une donnée en entrée, la transforment, retournent le résultat obtenu et stoppent leur exécution. D'autre part, le principe de réactivité se traduit par le fait que le logiciel doit toujours répondre aux sollicitations de son environ-

Fig. 3.1: Logiciel réactif

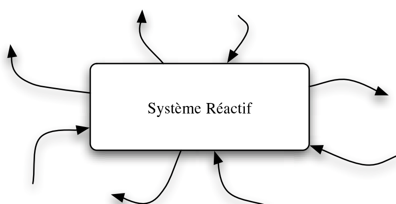
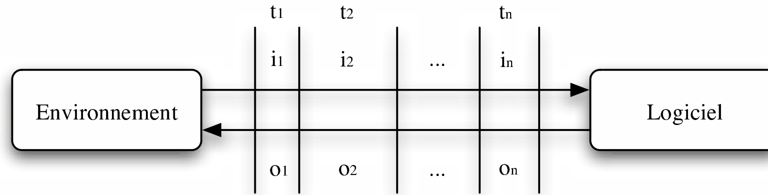


FIG. 3.2: Logiciel synchrone



nement: aucune entrée ne doit mener à un blocage du logiciel[1].

3.1.2 Modèle synchrone

Dans le cadre de ce travail, nous considérons les logiciels réactifs de la même manière que dans [HCRP91]: nous leur imposons de devoir calculer leurs sorties plus rapidement que toute évolution de leur environnement. Cette propriété constitue une restriction par rapport à la définition donnée dans [HP85], mais elle nous permet de distinguer les logiciels réactifs des logiciels interactifs qui imposent leur temps de réponse à l'environnement.

Le fonctionnement des logiciels synchrones peut être caractérisé par une boucle infinie au cours de laquelle, successivement (voir figure 3.2):

1. le logiciel reçoit une entrée de son environnement;

2. il calcule les sorties;

3. il émet les sorties calculées vers son environnement.

Ce comportement permet de dater précisément chaque événement interne du programme par rapport au flot des événements externes. Ainsi, il est possible de définir une suite d'instants logiques où, à chaque instant t_k, le logiciel reçoit une entrée i_k de l'environnement puis émet une sortie o_k, le tout avec un temps de calcul des sorties supposé nul.

Les logiciels synchrones au sens strict ne peuvent être que théoriques: en effet, quelle que soit la puissance de la machine utilisée pour l'exécution du logiciel, il existera toujours un temps δt, non nul, nécessaire pour le calcul des sorties.

L'hypothèse de synchronisme permet de considérer un logiciel comme synchrone si et seulement si le temps de réponse δt du logiciel est inférieur au temps minimum Δt nécessaire à l'observation de toutes évolutions significatives de l'environnement. La notion d'évolution significative de l'environnement est liée à la tâche particulière pour laquelle le logiciel est conçu et ne peut être spécifiée que par un expert du domaine dans lequel le logiciel sera exécuté. Un logiciel sera dit réactif synchrone si ce dernier vérifie l'hypothèse de synchronisme.

[1] On note que cette définition admet que certaines entrées soient perdues, l'idée étant acceptée par exemple qu'un système d'échantillonnage soit un exemplaire de système réactif.

3.2 Un langage dédié aux logiciels synchrones: Lustre

Différents langages, comme Esterel [BDS91], Signal [LGGLBLM91] ou Lustre [CHPP87], sont adaptés à la construction d'application réactives synchrones. Nous présenterons ici uniquement le langage Lustre sur lequel a été développé l'outil Lutess. Cette présentation porte seulement sur les notions dont nous aurons besoin dans la suite du manuscrit.

3.2.1 Introduction au langage

Une manière naturelle d'introduire une dimension temporelle dans les langages à flot de données consiste, par exemple, à mettre en relation le temps et le rythme des données dans le flot. En effet, les variables manipulées s'interprètent naturellement comme des fonctions du temps.

Le langage Lustre [CHPP87] a été spécifiquement développé pour l'écriture des logiciels réactifs synchrones. Lustre est un langage flot de donnée, dans lequel une dimension temporelle à été introduite en mettant en relation le temps et le rythme des données dans le flot. Ainsi, les variables manipulées s'interprètent naturellement comme des fonctions du temps. Chaque variable ou expression désigne un flot. Un flot est une suite infinie de valeurs d'un type donné associée à une horloge[2] représentant une suite d'instants. De ce fait, l'évolution d'un programme Lustre est découpée en instants logiques: à chaque instant t correspond une valeur de la suite infinie définissant la valeur du flot pour l'instant t.

Un flot possède la n-ième valeur de sa suite de valeurs au n-ième instant de son horloge. Tout programme ou fragment de programme a un comportement cyclique, qui définit son *horloge de base*[3], à partir de laquelle toutes les autres horloges sont dérivées: un flot dont l'horloge est l'horloge de base prend sa n-ième valeur au n-ième cycle d'exécution du programme.

Un programme Lustre consiste en des équations, et est organisé en nœuds. Une équation relie une variable et une expression (par exemple $X = E$ où X est la variable). Cette équation spécifie un invariant temporel: "$S = E$" signifie qu'à chaque instant t, $x(t) = e(t)$.

★ **Nœud 3.1** : *Structure syntaxique*

```
node n(i : τ; ...)returns(o : τ; ...)
     var l : τ;
let
     o = f(i, l);
tel
```

Dans un nœud Lustre, nous pouvons distinguer trois catégories de flots (voir le nœud 3.1): les flots d'entrée (i), les flots de sortie (o) et les flots locaux (l). Les flots d'entrée et de

[2]Le langage Lustre donne la possibilité de définir plusieurs horloges évoluant à des rythmes différents: nous ne détaillons pas ces fonctionnalités qui ne sont pas utilisées dans le cadre de ce travail.

[3]D'autres horloges, plus "lentes" peuvent être définies à l'aide de flots à valeurs booléennes: tout flot booléen peut être utilisé pour définir une horloge, qui est la suite des instants où sa valeur est *vrai* .

Fig. 3.3: Comportement du nœud *edge*

x	0	0	1	1	0	1
edge(x)	0	0	1	0	0	1

t

sortie définissent le profil du nœud et sont obligatoires, alors que les flots locaux permettent de calculer des valeurs intermédiaires afin de faciliter le calcul des flots de sortie. À chaque instant, chaque flot a une valeur significative.

Chaque flot de sortie ou local est défini par une équation. Le terme "*équation*" est à prendre au sens mathématique, c'est à dire que l'ordre dans lequel sont écrites les équations dans un nœud n'a pas d'importance: toutes les équations d'un nœud LUSTRE sont évaluées dans le même instant.

LUSTRE permet de manipuler des flots de type booléen, entier ou réel avec les opérateurs arithmétiques et logiques standards qui combinent les flots "point à point": par exemple, $X = A \ and \ B$ signifie $\forall t, \ x(t) = a(t) \ and \ b(t)$. Notons également l'existence d'un opérateur booléen particulier: $\#(f_{b_1}, ..., f_{b_n})$. Cet opérateur retourne *vrai* à l'instant t si au plus un des flots $f_{b_1}, ..., f_{b_n}$ est *vrai* au même instant t.

LUSTRE offre également à l'utilisateur deux opérateurs temporels élémentaires qui apportent un mécanisme pour accéder aux valeurs précédentes d'un flot:

- l'opérateur *pre* sert à définir un flot à partir d'un autre par un décalage d'un instant dans le passé. Si f représente le flot $\{f_0, f_1, \ ..., \ f_n\}$, alors $pref$ sera le flot $\{nil, f_0, f_1, ..., f_n\}$, nil représentant une valeur indéfinie.
- l'opérateur $\rightarrow$ (suivi de) permet d'initialiser un flot. Si f et g représentent les flots $\{f_0, f_1, ..., f_n\}$ et $\{g_0, g_1, ..., g_n\}$, alors $f \rightarrow g$ sera le flot $\{f_0, g_1, g_2, ..., g_n\}$.

Le nœud LUSTRE suivant définit l'opérateur *edge* (ou "*flot montant*"). Cet opérateur retourne la valeur *vrai* lorsque le flot booléen donné en paramètre passe de *faux* à *vrai* (voir figure 3.3):

★ **Nœud 3.2** : edge: flot montant

```
node edge(X : bool) returns(ed : bool)
let
    ed = false -> not pre X and X;
tel
```

À l'instant initial, *edge* est *faux* indépendamment du signal X; pour tous les autres instants, *edge* est *vrai* dans le cas où x a été évalué à *faux* à l'instant précédent et à *vrai* à l'instant courant. La figure 3.3 montre l'évolution de la valeur de *edge* en fonction de celle de X, où 0 et 1 dénotent respectivement les valeurs *faux* et *vrai* .

3.2.2 Tableaux et nœuds récursifs en LUSTRE

Dans ce paragraphe, nous nous limitons à la présentation des structures de données et de contrôle que nous utilisons dans les différents exemples du manuscrit.

Les tableaux et les nœuds récursifs ne sont que des facilités syntaxiques: ils n'augmentent pas l'expressivité du langage. LUSTRE a été conçu pour la programmation d'applications critiques. Sa compilation impose donc que la quantité de mémoire nécessaire à l'exécution du programme soit bornée et prédictible pour éviter les erreurs de saturation mémoire. De ce fait, la taille des tableaux et le nombre d'appels récursifs d'un nœud doivent être connus à la compilation.

Tableaux

Un tableau est déclaré de la manière suivante: monTab : $\tau\hat{}n$, où monTab sera défini comme un tableau de type τ contenant n éléments.

Remarque 3.1 *n représente une valeur entière constante. Cette valeur peut être explicite, par exemple 5. Elle peut aussi être le résultat de l'évaluation d'une expression entière calculée à partir de constantes*[4]*.*

LUSTRE offre aux programmeurs deux constructeurs sur les tableaux:

- [ ]: si $E_0, E_1, ..., E_{n-1}$ sont n expressions du même type τ, alors l'expression $[E_0, E_1, ..., E_{n-1}]$ définit un tableau de type $\tau\hat{}n$ contenant n éléments où le i^e élément correspond à E_i $(i \in [0..n-1]$.
- |: si T_1 et T_2 sont deux tableaux de type $\tau\hat{}m$ et $\tau\hat{}n$; l'expression $T_1|T_2$ définit la concaténation des deux tableaux de type $\tau\hat{}(m+n)$.

L'accès aux éléments d'un tableau peut se faire de deux façons: soit au i^e élément du tableau T par l'expression $T[i]$, soit à un sous-tableau par l'expression $T[i..j]$ représentant le tableau T' de type $\tau\hat{}(j-i+1)$ avec $0 <= i <= j < n$ tel que:

$$T'[k] = T[i+k], \quad avec \quad k \in [0..(j-i)].$$

Lors de la compilation, un tableau est décomposé en autant de variables qu'il y a d'éléments.

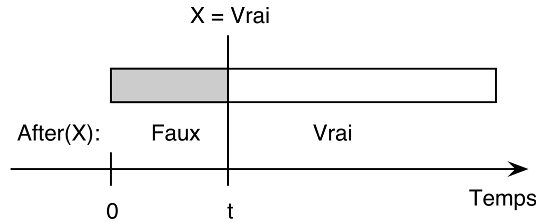
Nœuds récursifs

La définition d'un nœud récursif nécessite l'utilisation de l'opérateur conditionnel statique "**with** ... **then** ... **else** ..." qui est "exécuté" lors de la compilation. Ceci permet d'obtenir après la compilation un "nœud" non récursif.

Cette fonctionnalité du langage LUSTRE autorise la définition de nœuds "*génériques*" opérant sur des tableaux de valeurs. Prenons l'exemple de l'opérateur LIN_OR. Cet opérateur prend en paramètre un tableau Tab de valeurs booléennes et une constante n de type entier définissant la taille du tableau. Le résultat de LIN_OR est *vrai* si au moins un des éléments du tableau est évalué à *vrai* :

[4] LUSTRE autorise la définition de constantes: **const** maConst = *expr*, où *expr* désigne soit une constante explicite soit une expression constante.

FIG. 3.4: Aspects temporels de l'opérateur $after(A)$



★ **Nœud 3.3** : *Opérateur LIN_OR*

```
node LIN_OR(const n: int; Tab: bool^n) returns (Ok: bool);

let
    Ok = with n =1 then Tab[0]
        else Tab[0] or LIN_OR(n − 1, Tab[1..N − 1]);
tel
```

3.2.3 Propriétés de logique temporelle

Grâce à l'opérateur *pre*, LUSTRE est un cadre adéquat pour exprimer des phénomènes temporels, et, il est même possible de considérer LUSTRE comme une logique temporelle du passé [PH88]. Des opérateurs temporels peuvent être définis par le programmeur. Par exemple, $after(X)$ (voir nœud 3.4 et figure 3.4) est un prédicat qui a toujours la valeur *vrai* après la première observation d'une occurrence de X.

★ **Nœud 3.4** : *code* LUSTRE *du nœud* $after$.

```
node after(X: bool) returns (aft: bool);
let
    aft = false → pre(X or aft);
tel
```

Nous pouvons littéralement interpréter le comportement de l'opérateur $after$ du point de vue d'une logique temporelle du passé comme suit (ce point de vue correspond exactement au raisonnement que tiendrait un programmeur pour définir le code du nœud $after$):

- Si nous sommes à l'instant initial, $after$ est nécessairement évalué à *faux* . En effet, aucun événement externe ne peut se produire avant cet instant;
- Dans le cas contraire, l'opérateur $after$ est évalué à *vrai* lorsque X a pris la valeur *vrai* à l'instant précédent, ou bien si l'opérateur $after$ était déjà évalué à *vrai* .

Le tableau 3.1 montre l'évolution du nœud $after(X)$ et des flots qui le constituent au cours des dix premiers instants, t représentant l'horloge.

Cet aspect du langage LUSTRE s'avère bien adapté à l'écriture de propriétés invariantes décrivant, soit des comportements attendus du logiciel sous test, soit des contraintes sur les

TAB. 3.1: Évaluation du nœud *after*.

t	0	1	2	3	4	5	6	7	8	9
X	0	0	0	0	1	0	0	0	1	1
pre(X or aft)	nil	0	0	0	0	1	1	1	1	1
aft	0	0	0	0	0	1	1	1	1	1

entrées du logiciel sous test liées à son environnement d'exécution. Les opérateurs[5] les plus couramment utilisés sont les suivants:

- $Always_From_To(A, B, C)$ est *faux* si la valeur de A a été au moins une fois à *faux* entre les instants où B et C ont pris la valeur *vrai* . Cet opérateur sert à s'assurer que l'événement A s'est toujours produit entre les instants où l'événement B et l'événement C se sont produits;
- $Once_From_To(A, B, C)$ est *faux* si la valeur de A a toujours été *faux* entre les instants où B et C ont pris la valeur *vrai* . Cet opérateur permet de s'assurer que l'événement A s'est produit au moins une fois entre les instants où l'événement B et l'événement C se sont produits;
- $Always_Since(A, B)$ est *vrai* si la valeur de A a toujours été *vrai* depuis la dernière évaluation à *vrai* de la valeur de B. Cet opérateur s'assure que l'événement A s'est toujours produit depuis la dernière occurrence de l'événement B;
- $Once_Since(A, B)$ est *vrai* si la valeur de A a au moins une fois été *vrai* depuis la dernière évaluation à *vrai* de la valeur de B. Cet opérateur s'assure que l'événement A s'est produit au moins une fois depuis la dernière occurrence de l'événement B;
- $Implies(A, B)$ représente la valeur de l'implication: $A \Rightarrow B$.

3.3 Automate associé à un programme LUSTRE

L'idée, ici, est de montrer sur un exemple comment est modélisé [Glo89] l'ensemble des comportements possibles d'un programme LUSTRE par un automate de contrôle. Nous reprenons l'exemple du nœud *edge* que nous avons présenté au paragraphe 3.2.1.

3.3.1 Arbre des exécutions

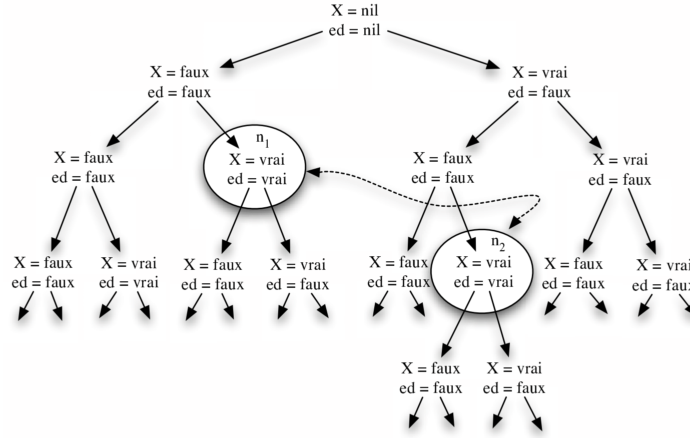
L'arbre des exécutions d'un programme LUSTRE dont les nœuds sont réduits aux valeurs des variables d'entrées et de sorties[6] représente tous les comportements possibles de ce dernier. Par nature, la profondeur d'un tel arbre est infinie étant donné qu'un programme LUSTRE n'est pas sensé s'arrêter.

L'arbre d'exécution du nœud *edge* est représenté par la figure 3.5. La racine de cet arbre porte les valeurs des variables X et *ed* lors du lancement du programme, c'est à dire avant le premier instant. Ces valeurs ne sont pas définies (*nil*). Lorsque le programme reçoit sa première entrée, *ed* est évalué à *faux* quelle que soit la valeur de la variable X.

[5] Nous donnons le code LUSTRE de chacun de ces opérateurs dans l'annexe B.1.

[6] On peut construire un arbre d'exécution dont les nœuds incluent en plus des variables d'état [Glo89].

FIG. 3.5: Arbre des exécutions.



3.3.2 Automate de contrôle

Dans l'exemple précédent, pour tous les instants suivant l'instant initial, la valeur de *ed* dépend des valeurs précédentes et courantes de X. Si nous choisissons deux nœuds de l'arbre, n_1 et n_2 (voir figure 3.5), de sorte que la valeur de X soit la même à la fois dans le nœud sélectionné et le nœud père, alors les sous-arbres descendant de ces deux nœuds sont identiques. De cette façon il est possible de replier l'arbre d'exécution sur lui-même afin d'obtenir l'automate de contrôle du programme LUSTRE[7].

La construction de l'automate de contrôle d'un programme LUSTRE permet d'obtenir un modèle fini des comportements. Intuitivement, chaque état s de l'automate représentera l'histoire de ce qui s'est passé avant d'atteindre s. Cette structure de contrôle est un automate de Mealy [HU79]: les valeurs des entrées et sorties figurent sur les transitions.

En pratique, un état est composé d'un ensemble de variables d'état. La variable d'état *init* qui est associée à l'expression $true \to false$ sert à repérer l'état initial. Pour chaque expression **pre** E du programme, une variable d'état est créée; cette variable permet de conserver la valeur précédente de E. Dans le cas du nœud *edge*, il y a deux variables d'état (voir automate de contrôle figure 3.6):

- $init = true \to false$;
- $sv_1 = X$.

3.4 Validation des logiciels basés sur LUSTRE

Différents outils basés sur le langage LUSTRE ont vu le jour. L'ensemble de ces outils offre des capacités de preuve (LESAR [Rat92, HLR92]), de test fonctionnel (LUTESS [Par96] et LURETTE [RWNH98]), structurel ([Maz94]) ou même les deux à la fois (GATeL [MA00]).

[7]Nous ne présentons pas ici les détails de la fonction de bissimulation [Glo89] qui existe entre l'arbre d'exécution et l'automate de contrôle d'un programme LUSTRE.

FIG. 3.6: Automate de contrôle.

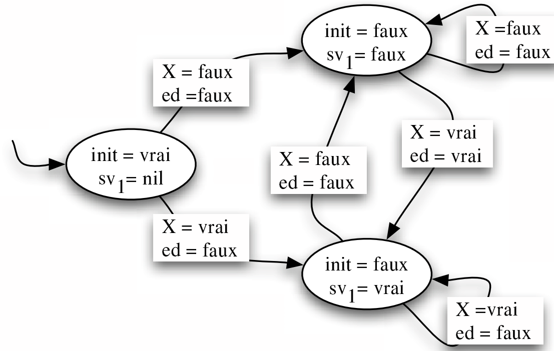
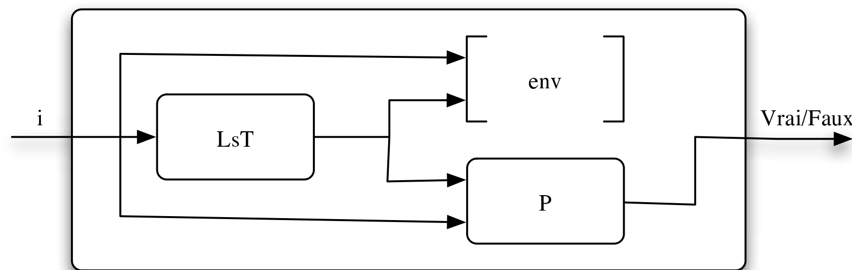


FIG. 3.7: Programme de vérification.



3.4.1 LESAR

LESAR [Rat92, HLR92] est un outil de *model checking* développé pour le langage LUSTRE. Cet outil est basé sur le principe des observateurs synchrones (voir figure 3.7): le logiciel LsT et la propriété P que l'on souhaite prouver sont englobés dans un même programme de vérification ayant une interface d'entrée identique à celle du logiciel à tester et une sortie booléenne unique représentant la valeur de vérité de la propriété à prouver. Des contraintes sur les entrées, env, représentant une description du comportement de l'environnement du logiciel sont ajoutées afin de ne considérer que les comportements réalistes.

LESAR commence par construire l'automate associé au programme de vérification LUSTRE. Il parcourt ensuite tous les états où les contraintes d'environnement env sont vérifiées: dans chacun de ces états, LESAR s'assure que la propriété est toujours évaluée à *vrai* . Cependant, ce type d'approche souffre du problème de l'explosion combinatoire du nombre d'états à explorer: ce nombre croît (au pire) de manière exponentielle avec le nombre n de variables d'état contenues dans le programme: $nb_{state} \leq 2^n$[8].

3.4.2 Test structurel en LUSTRE

Les critères de couvertures traditionnellement utilisés pour le test structurel des logiciels ne semblent pas être pertinents dans le cas des logiciels écrits en LUSTRE: les équations

[8] LESAR travaille uniquement avec des données booléennes.

définissant un tel logiciel sont évaluées en parallèle ce qui pourrait laisser penser à une couverture instantanée des instructions du logiciel avec une donnée de test.

L'approche proposée dans [Maz94] ne définit pas de critère de test structurel sur le langage Lustre à proprement parlé, mais sur l'automate de contrôle construit à partir de logiciel écrit en Lustre. Plus précisément, l'automate est transformé en un modèle stochastique du comportement du logiciel en associant à chaque transition une probabilité: une séquence d'entrée est ensuite calculée de telle sorte que la probabilité de couverture des états, des transitions ou des séquences de deux transitions soit supérieure à une valeur seuil fixée au préalable.

3.4.3 Lutess

L'outil Lutess [Par96], qui sera plus largement développé au chapitre suivant, propose d'aborder la validation des logiciels réactifs synchrones par le test. Le but de cet outil est de pouvoir générer aléatoirement de données de test respectant des contraintes d'environnement: nous parlerons de *simulateur d'environnement*. Nous considérerons une séquence de test de longueur n comme une suite de n couple d'entrées/sortie (i_k, o_k) pour $k \in (1..n)$.

Un avantage d'un outil de test comme Lutess par rapport à un outil de model checking est qu'il souffre moins souvent du problème d'explosion combinatoire et qu'il permet, une fois le simulateur d'environnement construit, de rendre un verdict quelle que soit la séquence de test considérée:

- soit la séquence de test obtenue a permis d'exhiber un comportement erroné du logiciel sous test;
- soit nous considérons que le logiciel ne présente pas de comportement erroné après qu'une séquence de test suffisamment longue et sans erreur, sans pour autant pouvoir en être totalement sûr.

3.4.4 Lurette

Lurette [RWNH98] est un outil de test développé par le laboratoire Verimag de Grenoble. De la même façon que Lutess, Lurette permet une simulation aléatoire du comportement de l'environnement du logiciel sous test. Lurette construit automatiquement une séquence de test en générant et soumettant au logiciel sous test un vecteur d'entrées respectant des contraintes d'environnement; chacune des entrées composant le vecteur pouvant être de type entier ou booléen.

Contrairement à Lutess qui connecte le simulateur d'environnement à une version exécutable logiciel sous test, Lurette construit un programme exécutable à partir du code C[9] du logiciel sous test, des assertions (environnement) et d'un oracle.

Cette approche basée sur le modèle des observateurs [HLR93] (voir figure 3.7) offre la possibilité de connaître complètement l'état du système et de l'environnement: cette information permet en particulier d'évaluer, dans chaque état du système, les différentes évolutions possibles du système en fonction de l'entrée choisie avant de lui soumettre.

[9]Ce code est en général obtenu par la compilation d'un langage de plus au niveau dédié au développement des logiciels réactifs synchrones.

3.4.5 GATeL

L'outil GATEL [MA00] permet de générer des séquences de test en se basant d'une part sur des objectifs de test écrits en LUSTRE étendu et d'autre part une description sous forme de nœuds LUSTRE du programme sous test (dans le cas du test fonctionnel) ou le programme LUSTRE lui-même (dans le cas du test structurel). Intuitivement, GATEL propose une solution permettant de générer automatiquement une séquence de test menant vers un objectif définis par le testeur.

L'approche proposée dans GATEL se décompose en trois parties:

- l'outil commence par construire la séquence d'entrées qui va permettre d'atteindre l'objectif de test fixé par le testeur;
- ensuite, l'outil doit exécuter la séquence de test précédemment construite;
- enfin, l'oracle doit décider si le test est un succès ou non. Ici deux cas sont possibles: soit nous étions dans le cadre du test fonctionnel et la description LUSTRE du programme peut être assimilée à une spécification exécutable du logiciel sous test permettant de déterminer des valeurs de sortie à comparer à celles fournies par le logiciel sous test, soit nous étions dans le cadre du test structurel et l'oracle doit s'appuyer sur des propriétés annexes pour pouvoir décider du succès du test.

Chapitre 4

LUTESS: un environnement de test pour les logiciels synchrones

Notre contribution portant sur une extension des fonctionnalités de l'outil de test LUTESS, ce chapitre est consacré à une introduction à cet outil. Après une présentation du harnais de test qu'apporte LUTESS, nous nous concentrons sur les techniques de génération automatique de données de test intégrées dans cet outil. Le rappel des fondements théoriques de LUTESS et de la technique d'implantation du simulateur d'environnement conclut la présentation du contexte rapproché de nos travaux.

4.1 L'outil LUTESS

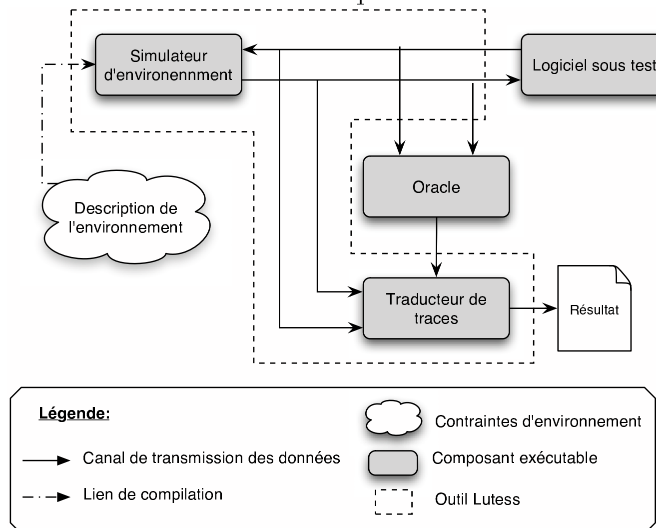
4.1.1 Introduction à LUTESS

LUTESS [Par96] est un outil de test développé pour des logiciels réactifs synchrones dont l'environnement est spécifié en LUSTRE et dont les interfaces sont composées de signaux ou variables booléennes. LUTESS permet de réaliser un test fonctionnel en boîte noire visant à la manifestation des défauts (recherche d'erreurs) présents dans le logiciel à valider. L'outil LUTESS permet une génération automatique des données de test booléennes qui respectent les contraintes d'environnement définies par l'utilisateur. Il offre aussi la possibilité de guider le choix des données de test selon différents critères (introduits au §4.1.2).

LUTESS permet d'interconnecter (voir figure 4.1) le logiciel à tester, un oracle, un collecteur/transformateur de traces et un simulateur d'environnement. Le logiciel à tester est un code binaire synchrone. À partir des données d'entrée et de sortie du logiciel à tester, l'oracle sert à émettre un verdict sur la concordance entre les sorties du logiciel et les attentes du testeur. Il peut être produit par le testeur sous la forme d'un programme ou engendré automatiquement à partir de propriétés spécifiées en LUSTRE. L'oracle doit aussi être un code synchrone. Toutes les valeurs (entrées/sorties/verdicts) sont collectées et peuvent être présentées de façon sélective. Le simulateur d'environnement est construit automatiquement par LUTESS à partir des contraintes d'environnement. À chaque instant du comportement cyclique du logiciel sous test, le simulateur d'environnement engendre un vecteur d'entrées[1],

[1] "Vecteur d'entrées" est un abrégé de "vecteur de valeurs pour les variables d'entrée".

FIG. 4.1: Principe de LUTESS



l'envoie au logiciel sous test et en reçoit la réponse.

Définition 4.1 *Un vecteur d'entrées est dit valide à l'instant courant lorsqu'il respecte l'ensemble des contraintes d'environnement à cet instant. Nous notons $V_{i_{env}}$ l'ensemble de ces vecteurs d'entrées.*

Le processus par défaut de sélection d'un vecteur d'entrées (appelé génération aléatoire) consiste à choisir aléatoirement et de manière équiprobable un vecteur d'entrée respectant les contraintes d'environnement.

Cependant, ce comportement par défaut ne traduit pas toujours une simulation réaliste de l'environnement du logiciel: parmi tous les vecteurs d'entrées valides à l'instant courant, certains sont plus probables que d'autres, traduisant ainsi des événements plus ou moins fréquents de l'environnement réel. Dans ce but, il a été développé et intégré dans LUTESS des profils opérationnels [DB99, dBOR98] qui facilitent le test statistique. Ces derniers donnent la possibilité au testeur de spécifier une probabilité d'occurrence d'un vecteur d'entrées sous une condition donnée.

Malgré tout, certains comportements des logiciels sont particulièrement difficiles à mettre en évidence. En effet, un logiciel peut avoir des modes de fonctionnement qui correspondent à des situations exceptionnelles, et/ou nécessitant une longue séquence d'initialisation difficile à produire aléatoirement. LUTESS donne la possibilité de guider la génération des données de test par des schémas comportementaux [Zua00b] (ou scénarios) fournis par le testeur.

Enfin, LUTESS peut utiliser les propriétés de sûreté imposées au logiciel sous test pour guider la génération des données de test. Cette technique [OP94a, PV01a, PV01b] permet de générer les vecteurs d'entrées les plus pertinents possibles afin de mettre le logiciel sous test dans une situation où la valeur de vérité de la propriété de sûreté ne dépend que de la réaction du logiciel (valeur de ses sorties).

4.1.2 Les techniques de sélection des vecteurs d'entrées

Globalement, la sélection d'un vecteur d'entrées par une technique donnée s'effectue en deux temps:

1. définition d'un sous-ensemble, $V_{val} \subseteq V_{i_{env}}$, de vecteurs d'entrées valides à l'instant t;
2. choix d'un vecteur d'entrées particulier parmi ceux de V_{val}.

Génération aléatoire

La génération aléatoire utilisée par défaut dans LUTESS consiste à:

1. ne pas restreindre l'ensemble de vecteurs d'entrées valides et nous avons:

$$V_{val} = V_{i_{env}}.$$

2. choisir un vecteur d'entrées dans V_{val} de manière équiprobable.

Les profils opérationnels

Les profils opérationnels permettent au testeur de rendre l'occurrence d'un événement plus ou moins probable selon une condition donnée. Considérons par exemple un capteur de température pouvant également émettre un signal "*défectueux*" lorsque ce dernier est en panne: les profils opérationnels permettent de rendre rare une panne du capteur si son fonctionnement était correct, mais aussi de rendre la continuité de la panne très probable si le capteur était déjà en panne à l'instant précédent. Les profils opérationnels sont définis par des triplets (*ev*, *prob*, *cond*), où l'événement *ev* a une probabilité d'occurrence *prob* sous la condition *cond*. Si nous reprenons l'exemple de notre capteur, nous définirons le profil opérationnel suivant:

$$(d\acute{e}fectueux,\ 0.1,\ \textbf{not pre}\ d\acute{e}fectueux)$$
$$(d\acute{e}fectueux,\ 0.9,\ \textbf{pre}\ d\acute{e}fectueux)$$

Les profils opérationnels s'intègrent dans LUTESS comme suit:

1. La construction de l'ensemble V_{val} utilisé lorsque des profils opérationnels sont définis est identique à celle de la génération aléatoire (comportement par défaut):

$$V_{val} = V_{i_{env}};$$

2. Les profils opérationnels modifient le processus de sélection du vecteur d'entrées à soumettre au logiciel sous test: ils permettent de remplacer la sélection équiprobable des vecteurs d'entrées par une sélection respectant une loi (élémentaire) de probabilité.

Les schémas comportementaux

L'écriture de schémas comportementaux permet de guider la génération des données de test dans une direction retenue par le testeur, en décrivant un comportement du logiciel à observer. LUTESS engendre ensuite, automatiquement, les vecteurs d'entrées qui mènent à la réalisation de toutes les étapes du schéma comportemental. Les schémas comportementaux sont définis à partir de deux listes imbriquées de conditions:

- la liste des "*conditions d'instant*" (*cond*) décrit la succession des étapes du schéma comportemental;
- la liste des "*conditions d'intervalle*" (*intercond*) décrit les événements que l'on ne souhaite pas observer entre deux étapes du schéma comportemental.

Prenons l'exemple d'une communication téléphonique entre deux personnes A et B. Nous avons le schéma suivant:

$$cond = (A\ décroche, A\ appelle\ B, B\ décroche, A\ ou\ B\ raccroche)$$

$$intercond = (A\ raccroche, A\ raccroche, vrai)$$

A commence par décrocher son combiné, puis appelle B. Entre ces deux événements, nous ne voulons pas que A raccroche son combiné. Une fois que le numéro de B est composé, le schéma exprime le fait que nous souhaitions que B décroche son combiné pour pouvoir établir la communication; nous ne voulons toujours pas que A raccroche entre l'instant où il compose le numéro de B, et l'instant où B décroche. Enfin la communication se termine lorsque l'une des deux personnes (A ou B) raccroche; entre l'établissement de la communication et sa fin, nous n'avons pas d'événement à interdire.

À chaque instant, le test guidé par des schémas comportementaux partitionne l'ensemble des vecteurs d'entrées valides $V_{i_{env}}$ en trois classes:

- la classe C_p des vecteurs d'entrées qui font progresser le schéma comportemental, c'est à dire les vecteurs d'entrées qui, dans l'état courant, permettent de passer d'une étape à une autre du schéma comportemental;
- la classe C_r des vecteurs d'entrées qui font régresser le schéma comportemental, c'est à dire les vecteurs d'entrées qui bloquent sa progression et nécessite de recommencer ce dernier depuis son début;
- la classe C_n des vecteurs d'entrées neutres qui ne font ni régresser, ni progresser le schéma comportemental.

Ensuite, l'ensemble V_{val} est associé à une des trois classes C_p, C_r ou C_n selon une probabilité définie par le poids de chaque classe:

$$\wp(C_p) = \frac{P_{C_p}}{P_{C_p} + P_{C_r} + P_{C_n}}, \quad \wp(C_r) = \frac{P_{C_r}}{P_{C_p} + P_{C_r} + P_{C_n}}, \quad \wp(C_n) = \frac{P_{C_n}}{P_{C_p} + P_{C_r} + P_{C_n}},$$

où P_{C_p}, P_{C_r}, P_{C_n} représentent respectivement les poids des classes C_p, C_r, C_n définis par le testeur.

Ensuite, LUTESS procède à la sélection du vecteur d'entrées à soumettre au logiciel sous test par un tirage aléatoire équiprobable dans V_{val}.

Le test guidé par les propriétés de sûreté

Le test guidé par les propriétés de sûreté sera plus largement développé au chapitre 5. Son but est de placer le logiciel dans une situation où il peut violer ces propriétés de sûreté. Celles-ci sont exprimées en LUSTRE: une importante difficulté réside dans le fait qu'elles

TAB. 4.1: Modes de fonctionnement de LUTESS.

Mode 1	$V_{val} = V_{i_{env}}$ *sélection équiprobable*	Test aléatoire
Mode 2	$V_{val} = V_{i_{env}}$ *sélection avec profils opérationels*	Test à probabilités définies
Mode 3	$V_{val} = guidage\ par\ scénario(V_{i_{env}})$ *sélection équiprobable*	Test équiprobable guidé par scénario
Mode 4	$V_{val} = guidage\ par\ propriété\ de\ sûreté(V_{i_{env}})$ *sélection équiprobable*	Test équiprobable guidé par les propriétés de sûreté

peuvent faire référence au passé. En conséquence, LUTESS devra être capable de générer automatiquement les vecteurs d'entrées qui vont construire, instant après instant, ce passé.

Le test guidé par les propriétés de sûreté décompose l'ensemble $V_{i_{env}}$ en n sous-ensembles: $V_{i_0}, V_{i_1}, ..., V_{i_{n-1}}$, où V_{i_k} représente l'ensemble des vecteurs d'entrées à l'instant courant t, pouvant mener le logiciel sous test à une violation de la propriété à l'instant $t + k$, $k \in [0..n-1]$. Différentes heuristiques ont été développées pour construire l'ensemble V_{val} à partir de $V_{i_0}, V_{i_1}, ..., V_{i_{n-1}}$.

De la même façon que les schémas comportementaux, le test guidé par les propriétés de sûreté agit sur la construction de V_{val}. Ensuite, la sélection du vecteur d'entrées se fait de façon aléatoire équiprobable dans V_{val}.

Modes de fonctionnement

La courte étude précédente des différentes techniques que propose LUTESS montre qu'il y a trois façons de partitionner l'ensemble $V_{i_{env}}$ de tous les vecteurs d'entrées valides à l'instant courant, chacune étant associée à des heuristiques de construction d'un ensemble $V_{val} \subseteq V_{i_{env}}$, et deux façon de sélectionner un vecteur d'entrées à soumettre au logiciel sous test parmi V_{val}. Le tableau 4.1 résume les quatre modes de fonctionnement qui ont été implantés dans LUTESS.

4.1.3 Structure syntaxique d'un nœud de test

L'ensemble des contraintes d'environnement qui permet à LUTESS de construire le simulateur d'environnement (voir paragraphe 3.4.3) est exprimé dans un nœud de test. La syntaxe d'un nœud de test est fondée sur celle d'un nœud LUSTRE; elle correspond à une extension de la grammaire de LUSTRE [Ray91] repérée par le nouveau mot clef **testnode**. La forme générale d'un nœud de test est présentée ci-dessous.

★ **Nœud 4.1** : *Structure syntaxique d'un nœud de test.*

testnode env(*sorties booléennes du logiciel sous test*)
returns (*entrées booléennes du logiciel sous test*)
var
  variables locales
let
  environment($C_1, \dots , C_n$);
  définition des variables locales
tel

Un nœud de test a pour paramètres d'entrée les sorties du logiciel sous test et pour paramètres-résultats les entrées à fournir au logiciel. Viennent ensuite les déclarations des variables locales nécessaires à l'expression des contraintes d'environnement. Le corps du nœud de test se situe entre les mots clefs **let** et **tel**: l'opérateur **environment** permet d'exprimer les contraintes d'environnement $C_1, \dots , C_n$, suivies des définitions des variables locales sous la forme d'équations LUSTRE. Dans un nœud de test, les variables locales offrent la possibilité au testeur de définir des valeurs intermédiaires permettant de simplifier l'écriture des contraintes d'environnement: leur utilisation est facultative. On note que ce nœud ne contient aucune équation pour définir les valeurs des paramètres-résultats. En effet, les valeurs émises par un nœud de test sont construites de manière non-déterministe parmi toutes celles satisfaisant les contraintes d'environnement.

4.1.4 Exemple d'un climatiseur

Pour illustrer l'écriture d'un nœud de test, nous allons utiliser l'exemple d'un contrôleur de climatiseur. L'environnement du contrôleur est composé d'un interrupteur, d'un capteur émettant trois signaux de température et d'une soufflerie (voir figure 4.2). Le contrôleur utilise les signaux de température et la position de l'interrupteur pour commander la soufflerie et répondre aux besoins de l'utilisateur; ces quatre signaux sont décrits ci-dessous.

Marche est *vrai* lorsque le climatiseur est sous tension (interrupteur "*On/Off*" commandé par l'utilisateur).

TempInf est *vrai* lorsque la température lue par le capteur est inférieure à la température souhaitée par l'utilisateur.

TempOk est *vrai* lorsque la température lue par le capteur est égale à la température souhaitée par l'utilisateur.

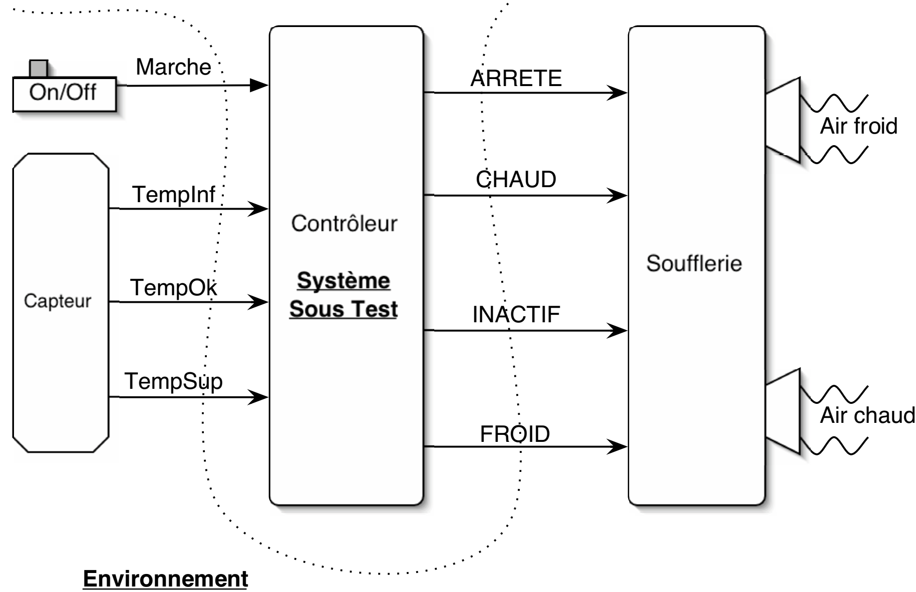
TempSup est *vrai* lorsque la température lue par le capteur est supérieure à la température souhaitée par l'utilisateur.

Le capteur permet à la fois de régler la température souhaitée par l'utilisateur et de mesurer la température ambiante. Le résultat de la comparaison des deux températures aboutit à l'envoi d'un des trois signaux: $TempInf$, $TempOk$ ou $TempSup$.

Les commandes qui permettent au contrôleur d'agir sur la soufflerie sont au nombre de quatre:

ARRETE permet de mettre la soufflerie hors tension. Ce signal est égal à *vrai* lorsque l'interrupteur est sur la position "*off*".

FIG. 4.2: Le climatiseur et son environnement.



INACTIF met le climatiseur en veille: le climatiseur continue de vérifier que la température ambiante correspond à la température souhaitée par l'utilisateur ($TempOk = vrai$), alors que la soufflerie ne diffuse ni air chaud, ni air frais.

CHAUD active la soufflerie pour diffuser de l'air chaud. Cette commande intervient lorsque la température ambiante est inférieure à celle voulue par l'utilisateur ($TempInf = vrai$).

FROID active la soufflerie pour diffuser de l'air frais. Cette commande intervient lorsque la température ambiante est supérieure à celle voulue par l'utilisateur ($TempSup = vrai$).

Nous modélisons le comportement de l'environnement par quatre contraintes (voir nœud 4.2) que nous classons en deux groupes. Le premier groupe composé des deux premières contraintes impose qu'un seul signal soit actif parmi les trois signaux de température: $TempInf$, $TempOk$ et $TempSup$.

Le second groupe, formé des deux dernières contraintes, s'assure de l'évolution continue de la température qui veut que cette dernière soit égale à celle choisie par l'utilisateur entre les instants où elle lui est inférieure puis supérieure.

Pour exprimer les deux dernières contraintes, nous avons besoin de l'opérateur

$$Once_From_To(B, A, C)$$

présenté au paragraphe 3.2.3.

★ **Nœud 4.2** : *environnement du climatiseur*

```
testnode env(ARRETE, INACTIF, CHAUD, FROID: bool)
returns (Marche: bool; TempInf, TempOk, TempSup: bool)
let
   environment(
      -- Au plus un signal de température actif à la fois:
      -- Contrainte 1:
      TempInf or TempOk or TempSup,
      -- Contrainte 2:
      #(TempInf, TempOk, TempSup)
      ----------------------------
      -- Évolution de la température:
      -- Contrainte 3:
      Once_From_To(TempOk, TempInf, TempSup),
      -- Contrainte 4:
      Once_From_To(TempOk, TempSup, TempInf)
   );
tel
```

4.2 Fondements théoriques de LUTESS

Nous présentons dans ce paragraphe le modèle formel du simulateur d'environnement. Nous commençons par donner une sémantique opérationnelle au nœud de test basée sur les traces d'exécutions [Rat92, Par96]: intuitivement, ces traces correspondent aux valeurs prises par les variables du nœud de test et peuvent être représentées sur un arbre des exécutions tel qu'il a été présenté au paragraphe 3.3: une trace d'exécution correspond au parcours d'un chemin de l'arbre.

Dans une seconde partie, nous verrons comment nous pouvons représenter un simulateur d'environnement à partir d'un automate de Mealy. De la même façon que pour un nœud LUSTRE, la construction d'un tel automate est rendue possible grâce à l'existence d'une fonction de bissimulation entre l'arbre des exécutions et l'automate de contrôle [Glo89].

4.2.1 Sémantique des traces d'un nœud de test

La sémantique opérationnelle de LUTESS [Par96] est définie de la même manière que la sémantique opérationnelle de LUSTRE [Rat92] à l'aide des traces d'exécution. Cette sémantique considère uniquement la mémoire à l'instant courant et celle à l'instant précédent.

Soit les ensembles I de tous les identificateurs contenus dans le nœud de test, et V de toutes les valuations possibles pour les identificateurs de I: une mémoire σ est une fonction associant à chaque variable du nœud de test une valeur de V:

$$\sigma : I \rightarrow V$$

Une séquence non vide de mémoires peut être représentée par le parcours d'une branche de l'arbre des exécutions du nœud de test.

Soient σ et σ' deux mémoires à deux instants consécutifs. Nous notons par $\sigma, \sigma' \vdash E \mid v$ le fait que l'expression E prenne la valeur v après l'évaluation de σ, σ' et par $\sigma, \sigma' \vdash E$ le fait que

l'expression E puisse être appliquée après l'évaluation de σ, σ'. Par ailleurs, nous noterons par $\perp$ la mémoire indéfinie. Nous avons également besoin de deux valeurs particulières, *out* et *nil*:

- *out* sert à assigner les variables de sortie à l'instant courant afin d'interdire l'utilisation de leurs valeurs: au moment où LUTESS choisit le vecteur d'entrées à soumettre au logiciel, il ne peut pas connaître les valeurs des variables de sortie du logiciel calculées en fonction du vecteur d'entrées choisi. Ainsi, *out* représente la valeur "pas encore connue";
- *nil* représente la valeur indéfinie.

Remarque 4.1 *Dans le cadre de cette sémantique, nous faisons l'hypothèse que pour les expressions du type pre x, x ne peut être qu'une variable. Cette hypothèse n'altère en rien la généralité de la sémantique. En effet, dans le cas où nous aurions* $var = pre\ X_{expr}$, *nous pouvons utiliser le principe de substitution de façon à obtenir :*

$$var = pre\ X_{expr} \equiv \begin{cases} var_{temp} = X_{expr}; \\ var = pre\ var_{temp}; \end{cases}$$

Règles de la sémantique des traces

Nous donnons ci-dessous les règles de la sémantique des traces d'un nœud de test. La règle de l'opérateur **environment** spécifie que si toutes les contraintes d'environnement $C_k,\ k \in [1..r]$ sont satisfaites dans le contexte défini par les deux mémoires consécutives σ, σ', alors l'opérateur **environment** est applicable dans ce même contexte.

- Compatibilité avec l'opérateur **environment**.

$$\frac{\sigma, \sigma' \vdash C_1 \mid true, ..., \sigma, \sigma' \vdash C_r \mid true}{\sigma, \sigma' \vdash \mathbf{environment}(C_1, ..., C_r)}$$

L'ensemble des autres règles est construit sur le même modèle que la règle associée à l'opérateur **environment**.

- K est une constante booléenne et k sa valeur.
  $\sigma, \sigma' \vdash K \mid k$
- x est une variable d'entrée ou locale booléenne.
  $\sigma, \sigma' \vdash x \mid \sigma'(x)$
- x est une variable de sortie booléenne.
  $\sigma, \sigma' \vdash x \mid out$
- E est une expression booléenne.

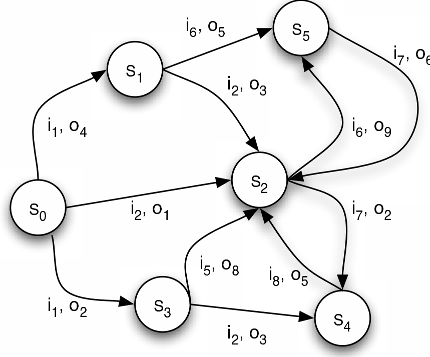
$$\frac{\sigma, \sigma' \vdash E \mid true}{\sigma, \sigma' \vdash not\ E \mid false} \qquad \frac{\sigma, \sigma' \vdash E \mid false}{\sigma, \sigma' \vdash not\ E \mid true}$$

- E_1, E_2 et E_3 sont des expressions booléennes, α représente la valeur *nil* ou *out*.

$$\frac{\sigma, \sigma' \vdash E_1 \mid true}{\sigma, \sigma' \vdash if\ E_1\ then\ E_2\ else\ E_3 \mid E_2} \qquad \frac{\sigma, \sigma' \vdash E_1 \mid false}{\sigma, \sigma' \vdash if\ E_1\ then\ E_2\ else\ E_3 \mid E_3}$$

$$\frac{\sigma, \sigma' \vdash E_1 \mid \alpha}{\sigma, \sigma' \vdash if\ E_1\ then\ E_2\ else\ E_3 \mid \alpha}$$

FIG. 4.3: Simulateur d'environnement



- E_1, E_2 sont des expressions booléennes.
  $\sigma, \sigma' \vdash E_1 \ or \ E_2 \mid if \ E_1 \ then \ true \ else \ E_2$
  $\sigma, \sigma' \vdash E_1 \ and \ E_2 \mid if \ E_1 \ then \ E_2 \ else \ false$
  $\dfrac{\bot, \sigma' \vdash E_1 \mid v_1}{\bot, \sigma' \vdash E_1 \rightarrow E_2 \mid v_1} \quad \dfrac{\sigma, \sigma' \vdash E_2 \mid v_2}{\sigma, \sigma' \vdash E_1 \rightarrow E_2 \mid v_2}$
- x est soit une variable d'entrée, soit une variable de sortie, soit une variable locale.
  $\bot, \sigma' \vdash pre(x) \mid nil \quad \sigma, \sigma' \vdash pre(x) \mid \sigma(x)$
- Compatibilité avec les équations.
  $\dfrac{\sigma, \sigma' \vdash E \mid v, \ \sigma'(x) = v}{\sigma, \sigma' \vdash x = E} \quad \dfrac{\sigma, \sigma' \vdash P_1 \mid v_1, \ \sigma, \sigma' \vdash P_2 \mid v_2}{\sigma, \sigma' \vdash P_1; P_2}$

4.2.2 Simulateur d'environnement

Formellement, le simulateur d'environnement dérivé d'un nœud de test est construit à partir d'un automate de Mealy construit de manière analogue à l'automate de contrôle d'un nœud LUSTRE présenté au paragraphe 3.3.

Définition 4.2 *Soit M un simulateur d'environnement (voir figure 4.3):*

$$M = (S_{env}, s_{init_{env}}, I, O, f_{env}, env, part_{env}, sel_{env})$$

- *S_{env} est un ensemble fini d'états et $s_{init_{env}}$ est l'état initial.*
- *I est l'ensemble des variables d'entrée du système sous test.*
- *O est l'ensemble des variables de sortie du système sous test.*
- *$f_{env} : S_{env} \times V_I \times V_O \rightarrow S_{env}$ est la fonction de transition, où V_I (resp. V_O) représente l'ensemble des valuations des variables d'entrée (resp. variables de sortie): V_I (resp. V_O) représente tous les vecteurs d'entrées (resp. de sorties).*
- *$env \subseteq S_{env} \times V_I$ représente la description de l'environnement.*
- *$V_{val} = part_{env}(V_{i_{env}})$ est la méthode qui partitionne l'ensemble des vecteurs d'entrées valides $V_{i_{env}}$ et construit l'ensemble V_{val} , avec:*

$$V_{i_{env}} = \{i | (s, i) \in env\}$$

– *$sel_{val}(V_{val})$ est la méthode qui permet de choisir dans V_{val} un vecteur d'entrées à soumettre au logiciel sous test.*

Dans le cas du test aléatoire (mode 1 du tableau 4.1), la méthode $part_{env}(V_{i_{env}})$ construit l'ensemble V_{val} tel que:

$$V_{val} = V_{i_{env}}.$$

Ensuite, la méthode sel_{val} choisit aléatoirement un vecteur d'entrées de V_{val} de manière équiprobable.

Le fonctionnement du simulateur d'environnement peut être comparé à une sorte de machine sorties/entrées. Classiquement, une machine entrées/sorties dérivée d'une machine de Mealy lit des entrées, puis calcule ses sorties et l'état suivant en fonction de l'état courant et des entrées lues. Or le fonctionnement du simulateur d'environnement est inversé; chaque instant du simulateur d'environnement peut être décomposé en deux "demi-instants":

1. LUTESS génère le vecteur d'entrées à soumettre au logiciel (émission d'une sortie du simulateur);
2. LUTESS reçoit le vecteur de sortie émis par le logiciel (acquisition d'une entrée du simulateur) et peut calculer le nouvel état de du simulateur.

Le modèle théorique que nous donnons ici ne garantit pas que le simulateur construit soit réactif, c'est à dire qu'il existe une entrée valide à soumettre au logiciel sous test dans tous les états atteignables de l'environnement. Bien qu'il soit théoriquement possible de calculer l'ensemble des états accessibles de l'environnement par l'intermédiaire d'un plus petit point fixe [HLR92], ce calcul s'avère en pratique souvent impossible. Le choix qui a été fait dans LUTESS est de considérer a priori l'environnement comme réactif sans vérification. Si une situation de blocage est détectée au cours de la génération des données de test, LUTESS émet un diagnostic pour en informer l'utilisateur.

L'état du simulateur d'environnement est représenté par un ensemble de variables booléennes d'état. L'état mémorise l'ensemble des valeurs passées qui sont nécessaires au simulateur pour déterminer d'une part l'ensemble des vecteurs d'entrées valides de l'environnement, et d'autre part l'état suivant du simulateur.

La fonction de transition est alors décomposée en un ensemble de fonctions de transitions partielles: une par variable d'état. Dans ce document, nous parlerons indifféremment de fonction de transition ou d'ensemble de fonctions de transitions partielles.

Reprenons l'exemple du contrôleur de climatiseur introduit au paragraphe 4.1.4. La génération automatique du simulateur d'environnement conduit à une machine d'états finis comprenant cinq variables d'états:

– sv_0 correspond à la variable *init* de l'automate de contrôle présenté au paragraphe 3.3;
– sv_1 mémorise le fait que $TempSup$ est, ou a été, *vrai*;
– sv_2 a la valeur *vrai* si $TempOk$ a été observé depuis la dernière occurrence de $TempSup$ ou alors si $TempSup$ n'a jamais pris la valeur *vrai*;
– sv_3 mémorise le fait que $TempInf$ soit, ou a été, *vrai*;
– sv_4 est affecté de la valeur *vrai* si $TempOk$ a été observé depuis la dernière occurrence de $TempInf$ ou alors si $TempInf$ n'a jamais pris la valeur *vrai* .

La fonction de transition 4.1 représente celle de l'environnement du climatiseur. sv_x et sv'_x dénotent respectivement la valeur courante et la valeur suivante de la variable d'état.

★ **Fonction de transition 4.1** : *climatiseur*

$$
\begin{array}{rl}
sv'_0 = & sv_0 \wedge false \\
sv'_1 = & TempSup \vee \textbf{if}\ sv_0\ \textbf{then}\ false\ \textbf{else}\ sv_1\ \textbf{fi} \\
sv'_2 = & \textbf{if}\ TempSup\ \textbf{then}\ TempOk \\
 & \textbf{else if if}\ sv_0\ \textbf{then}\ false\ \textbf{else}\ sv_1\ \textbf{fi then}\ (\mathbf{TempOk} \vee \mathbf{sv_2}) \\
 & \quad \textbf{else}\ true \\
 & \quad \textbf{fi} \\
 & \textbf{fi} \\
sv'_3 = & TempInf \vee \textbf{if}\ sv_0\ \textbf{then}\ false\ \textbf{else}\ sv_3\ \textbf{fi} \\
sv'_4 = & \textbf{if}\ TempInf\ \textbf{then}\ TempOk \\
 & \textbf{else if if}\ sv_0\ \textbf{then}\ false\ \textbf{else}\ sv_3\ \textbf{fi then}\ (\mathbf{TempOk} \vee \mathbf{sv_4}) \\
 & \quad \textbf{else}\ true \\
 & \quad \textbf{fi} \\
 & \textbf{fi}
\end{array}
$$

La relation $env \subseteq S \times V_I$ représentant les contraintes d'environnement est construite automatiquement par LUTESS et est représentée par l'expression booléenne ci-dessous: cette dernière permet de décider si le couple (s, i) appartient à env, où i est un vecteur d'entrées et s un état.

★ **Relation d'environnement 4.1** : *climatiseur*

$$
\begin{array}{r|l}
Contrainte\,1 : & (TempInf \vee TempOk \vee TempSup) \\
 & \wedge \\
Contrainte\,2 : & ((TempInf \wedge \neg(TempOk \vee TempSup)) \vee \\
 & \quad (\neg TempInf \wedge TempOk \wedge \neg TempSup) \vee \\
 & \quad (\neg TempOk)) \\
 & \wedge \\
Contrainte\,3 : & \mathbf{if\ if}\ sv_0\ \mathbf{then}\ false\ \mathbf{else}\ sv_3\ \mathbf{fi} \wedge TempSup\ \mathbf{then} \\
 & \qquad \mathbf{if}\ TempInf\ \mathbf{then}\ TempOk \\
 & \qquad \mathbf{else\ if\ if}\ sv_0\ \mathbf{then}\ false\ \mathbf{else}\ sv_3\ \mathbf{fi\ then}\ (TempOk \vee sv_4) \\
 & \qquad\qquad \mathbf{else}\ true \\
 & \qquad\qquad \mathbf{fi} \\
 & \mathbf{else}\ true \\
 & \mathbf{fi} \\
 & \wedge \\
Contrainte\,4 : & \mathbf{if\ if}\ sv_0\ \mathbf{then}\ false\ \mathbf{else}\ sv_1\ \mathbf{fi} \wedge TempInf\ \mathbf{then} \\
 & \qquad \mathbf{if}\ TempSup\ \mathbf{then}\ TempOk \\
 & \qquad \mathbf{else\ if\ if}\ sv_0\ \mathbf{then}\ false\ \mathbf{else}\ sv_1\ \mathbf{fi\ then}\ (TempOk \vee sv_2) \\
 & \qquad\qquad \mathbf{else}\ true \\
 & \qquad\qquad \mathbf{fi} \\
 & \mathbf{else}\ true \\
 & \mathbf{fi}
\end{array}
$$

La relation d'environnement 4.1 montre le lien entre l'expression booléenne et les contraintes d'environnement dont elle est déduite (voir nœud de test 4.2). La présentation de la relation d'environnement a été décomposée en quatre parties correspondant respectivement aux quatre contraintes d'environnement.

L'expression des deux premières contraintes de l'environnement du simulateur se retrouve aisément dans l'expression des deux premières parties de la relation d'environnement 4.1.

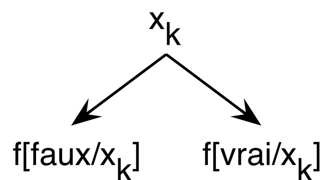
Les contraintes 3 et 4 correspondent aux deux dernières contraintes du nœud de test (celles utilisant l'opérateur $Once_From_To(B, A, C)$). Analysons la partie de l'expression booléenne associée à la contrainte 3: $Once_From_To(TempOk, TempInf, TempSup)$.

- si le simulateur ne se trouve pas dans l'état initial ($sv_0 = false$) et qu'une occurrence de $TempInf$ a déjà été observée ($sv_3 = true$) et que $TempSup = true$ alors nous devons déjà avoir observé une occurrence de $TempOk$ depuis la dernière occurrence de $TempInf$ ($sv_4 = false$)[2];

- sinon, dans tous les autres cas il n'y a pas de contrainte.

[2]Nous avons réduit notre analyse aux seuls cas possibles au regard des autres contraintes d'environnement alors que la traduction l'opérateur $Once_From_To$ obtenue dans la relation d'environnement est plus large: elle correspond à la sémantique de ce dernier sans tenir compte des autres contraintes d'environnement.

FIG. 4.4: Décomposition de Shannon: principe.



4.3 Technique d'implantation du simulateur d'environnement

Au paragraphe 4.2.2, nous avons vu que le simulateur d'environnement peut être représenté par une machine de Mealy étendue à nombre fini d'états: la fonction de transition et la relation *env* décrivant les contraintes d'environnement sont exprimées à l'aide d'expressions booléennes. Une manière efficace de représenter et d'évaluer une expression booléenne est d'utiliser un bdd[3]. Dans cette partie, nous présentons d'une part les *bdd* qui constituent le cœur de LUTESS, et d'autre part le principe d'implantation de la génération équiprobable des vecteurs d'entrées [Par96].

4.3.1 Les bdd

Les bdd ont été introduits par [Ake78]. Un ensemble d'opérateurs permettant de manipuler les fonctions booléennes a été défini sur le bdd par [Bry86].

Les bdd sont basés sur les arbres de Shannon qui sont fondés sur le principe de la décomposition d'une fonction selon une variable.

Définition 4.3 *La décomposition d'une fonction $f : \mathbb{B}^n \to \mathbb{B}$ selon une variable $x_k, k \in [1..n]$, est:*

$$f_{\overline{x_k}} = f[\text{faux}/x_k], f_{x_k} = f[vrai/x_k]$$
$$f = (\neg x_k \wedge f_{\overline{x_k}}) \vee (x_k \wedge f_{x_k})$$

où $f[b, x_k]$ désigne le résultat de la substitution de la variable x_k par le valeur $b \in \mathbb{B}$ dans f avec $\mathbb{B} = \{\text{vrai}, \text{faux}\}$ (voir figure 4.4).

Par itérations successives sur toutes les variables d'une fonction booléenne f, nous obtenons l'arbre de Shannon représentant f. Cet arbre est unique pour un ordre fixé des variables (l'ordre choisi étant arbitraire). Chaque niveau de l'arbre est associé à une variable de f; le sous-arbre gauche (resp. droit) étant associé au cas où la variable est évaluée à *faux* (resp. *vrai*). Prenons l'exemple de la fonction f suivante:

$$f(a, b, c, d) = (a \vee b) \Rightarrow (c \wedge d)$$

La décomposition de f selon la variable a est:

$$(\neg a \wedge \underbrace{(b \Rightarrow (c \wedge d))}_{f[faux/a]}) \vee (a \wedge \underbrace{((b \wedge c \wedge d) \vee (\neg b \wedge c \wedge d))}_{f[vrai/a]})$$

[3] "*binary decision diagram*": arbre de décision binaire.

FIG. 4.5: Arbre de Shannon pour f.

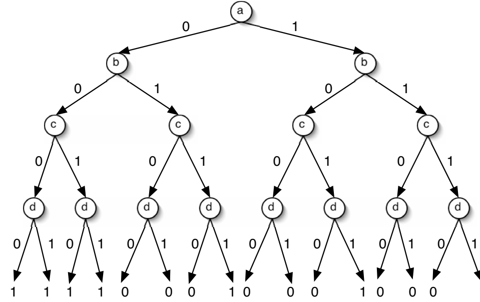
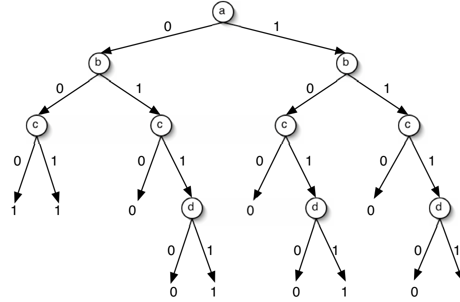


FIG. 4.6: Élimination des nœuds redondants: étape 1.



En appliquant la décomposition de Shannon sur les trois autres variables, nous obtenons l'arbre de Shannon de f représenté en figure 4.5.

Les bdd permettent d'optimiser la représentation de l'arbre de Shannon d'une fonction booléenne par deux transformations:

- **Élimination des nœuds redondants:** cette transformation est opérée des feuilles vers la racine. Elle consiste à éliminer tous les nœuds qui ont deux fils identiques. Ainsi, le nœud supprimé prend la valeur des fils.
- **Partage des sous-graphes isomorphes:** cette transformation permet la mise en commun de deux sous-arbres identiques.

En reprenant l'exemple de la fonction $f(a, b, c, d)$ représentée par l'arbre de Shannon de la figure 4.5, nous obtenons un bbd en trois étapes: les figures 4.6 et 4.7 montrent comment, en deux étapes, les nœuds redondants sont supprimés. Dans une troisième étape, le partage des sous-arbres isomorphes conduit au bbd présenté sur la figure 4.8.

4.3.2 Principes d'implantation du simulateur d'environnement

LUTESS utilise les bdd à chaque fois qu'il est nécessaire de représenter une fonction booléenne ou un ensemble de contraintes booléennes. Dans le cas de la génération aléatoire équiprobable, chacune des fonctions de transitions partielles associées aux variables d'état est représentée par un bbd, ainsi que l'ensemble des contraintes d'environnement. Dans ce paragraphe nous exposons le principe de la sélection équiprobable d'un vecteur d'entrées et celui du calcul de l'état suivant qui seront illustrés sur l'exemple du climatiseur.

FIG. 4.7: Élimination des nœuds redondants: étape 2.

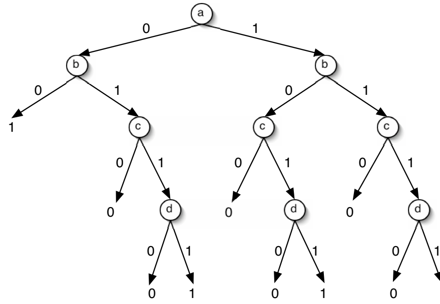
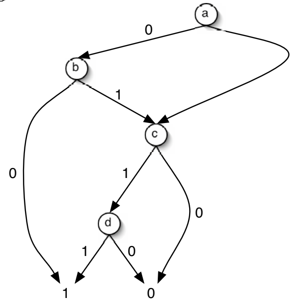


FIG. 4.8: Partage des sous-arbres isomorphes: étape 3.



Sélection d'un vecteur d'entrées

Le bdd représentant les contraintes d'environnement (nommé bdd d'environnement) est organisé en deux parties (voir figure 4.9): la partie haute du bdd contient les variables d'état nécessaire à l'évaluation des contraintes d'environnement, alors que la partie basse contient les entrées du logiciel sous test. Cette configuration du bdd d'environnement permet de calculer très facilement l'ensemble $V_{i_{env}}$ des vecteurs d'entrées valides dans l'état courant: il suffit de parcourir la partie du bdd associé aux variables d'état en fonction de la valeur courante de chacune d'entre elles. Lorsque toutes les variables d'état contenues dans le bdd d'environnement ont été parcourues, nous obtenons un sous-bdd représentant $V_{i_{env}}$.

Après avoir déterminé l'ensemble $V_{i_{env}}$, LUTESS doit s'assurer de sélectionner un vecteur d'entrées de façon équiprobable. Un vecteur d'entrées valide est repéré dans le sous-bdd représentant l'ensemble $V_{i_{env}}$ à l'instant courant par un chemin issu de la racine de ce dernier

FIG. 4.9: Schéma du bdd d'environnement.

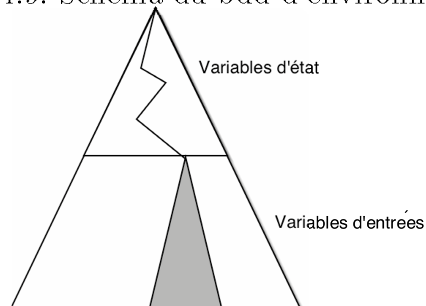
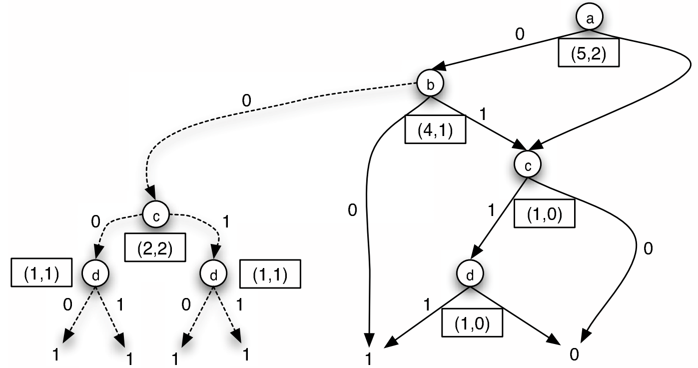


FIG. 4.10: Etiquetage du bdd.



et menant à une feuille ayant pour valeur *vrai* . Un tel chemin sera dit valide. La sélection équiprobable d'un vecteur d'entrées parmi l'ensemble de tous les vecteurs d'entrées valides, est garanti par un étiquetage du bdd[4]. À chaque nœud du bdd est attachée une étiquette (v_1, v_2): v_1 indique le nombre de vecteurs valides dans le sous-bdd gauche, alors que v_2 donne le nombre de vecteurs valides dans le sous-bdd droit. Cette étiquette permet de calculer la probabilité qu'une variable d'entrée i donnée associée à un nœud du bdd ait la valeur *vrai* ou *faux* . La variable i prendra la valeur *faux* avec la probabilité:

$$\wp(i, faux) = \frac{v_1}{v_1 + v_2}.$$

De la même façon, nous exprimons la probabilité que la variable i ait la valeur *vrai* par:

$$\wp(i, vrai) = \frac{v_2}{v_1 + v_2}.$$

La figure 4.10 représente le bdd étiqueté de la fonction $f(a, b, c, d)$ du paragraphe 4.3.1. Notons que l'étiquetage tient compte des variables qui n'apparaissent pas dans le bdd (c'est à dire qui n'influent pas sur la valeur de la fonction): par exemple, si les variables a et b sont évaluées à *faux* , alors quelles que soient les valeurs des variables c et d, nous arriverons toujours sur une feuille évaluée à *vrai* (ce qui correspond à quatre sous-vecteurs valides à partir de c inclus représentés en pointillés sur la figure 4.10).

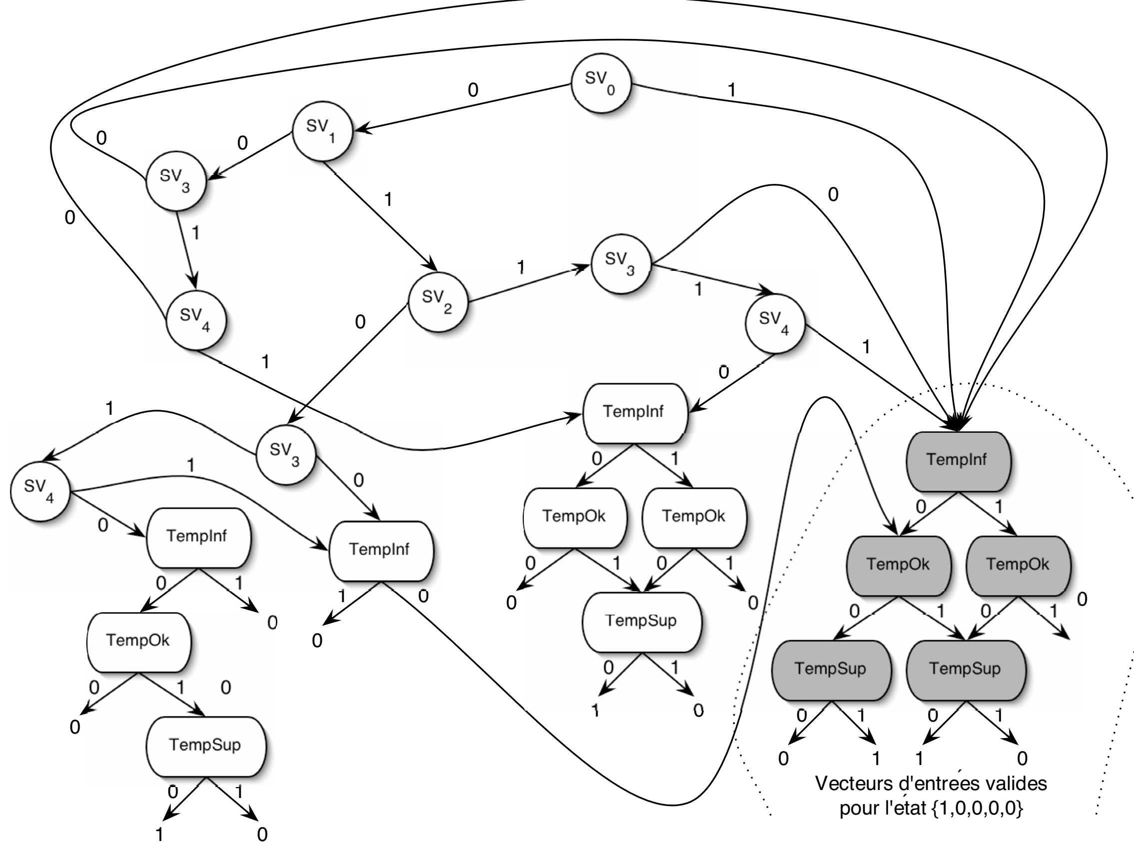
Supposons que la fonction $f(a, b, c, d)$ représente l'ensemble $V_{i_{env}}$ d'un simulateur. Il y a alors sept vecteurs d'entrées valides dont le vecteur $\{a = 0, b = 0, c = 1, d = 0\}$. Pour ce dernier, la probabilité d'être sélectionné est:

$$\begin{array}{rcccccccc} \wp(\{a=0, b=0, c=1, d=0\}) & = & \wp(a=0) & \times & \wp(b=0) & \times & \wp(c=1) & \times & \wp(d=0) \\ & = & \frac{5}{7} & \times & \frac{4}{5} & \times & \frac{2}{4} & \times & \frac{1}{2} \\ & = & \frac{1}{7} & & & & & & \end{array}$$

Nous pourrions faire ce même calcul pour les sept vecteurs d'entrées valides pour confirmer qu'ils ont tous la même probabilité d'être choisis.

[4]Nous ne donnons pas ici l'algorithme d'étiquetage du bdd (voir [Par96]).

Fig. 4.11: Le bdd d'environnement du climatiseur.



Calcul de l'état suivant

Le calcul de l'état suivant est fait par le parcours de chaque bdd associé à chacune des variables d'état. Soient q le nombre de variables d'état, $f_i \, (i \in [0..q-1])$ la i[e] fonction de transition associée à la variable sv_i, β_i le bbd représentant f_i et x une variable de f_i. Pour chacune des variables x de f_i, le parcours du nœud de β_i associé à la variable x s'effectue en sélectionnant le sous-arbre gauche (resp. droit) si x est évaluée à *faux* (resp. *vrai*); lorsqu'une feuille β_i est atteinte, la nouvelle valeur de sv_i est déterminée par la valeur de la feuille.

Illustration sur le climatiseur

Reprenons l'exemple du climatiseur présenté au paragraphe 4.1.4. L'ensemble des contraintes d'environnement est représenté sur le bdd de la figure 4.11. Notons que le partage des sous-arbres isomorphes suppose également que les feuilles de l'arbre ayant la même valeur soient mises en commun. Sur la figure 4.11 les feuilles de même valeur n'ont pas été mises en commun pour des raisons de lisibilité de cette dernière.

Remarque 4.2 *Nous notons la valeur d'un état à un instant donné par une suite de cinq valeurs:*

$$\{sv_0, sv_1, sv_2, sv_3, sv_4\}, \, avec \, sv_i \in [0..1]$$

TAB. 4.2: Vecteurs d'entrées valides pour le climatiseur dans l'état initial.

Vecteur	Marche	TempInf	TempOk	TempSup
v_1	1	1	0	0
v_2	0	1	0	0
v_3	1	0	1	0
v_4	0	0	1	0
v_5	1	0	0	1
v_6	0	0	0	1

À l'instant initial, nous sommes dans l'état $\{1,0,0,0,0\}$. LUTESS commence par sélectionner l'ensemble des vecteurs d'entrées valides en parcourant le bdd d'environnement (voir figure 4.11) en fonction de la valeur de l'état courant. Nous obtenons six vecteurs d'entrées possibles (voir table 4.2). Remarquons que l'entrée $Marche$ n'est pas présente dans le bdd d'environnement: ceci signifie que l'entrée $Marche$ n'est pas contrainte et que sa valeur peut être soit vraie, soit fausse, indépendamment des valeurs prises par $TempInf$, $TempOk$ et $TempSup$.

Parmi l'ensemble des vecteurs d'entrées valides, LUTESS en choisit un de façon équiprobable[5]: supposons que c'est v_5. Une réponse correcte du logiciel sous test au vecteur v_5 pourrait être le vecteur de sortie:

$$v_s = \{ARRETE = 0, INACTIF = 0, CHAUD = 0, FROID = 1\}.$$

À partir de v_s, du vecteur d'entrée v_5 et de l'état courant $\{1,0,0,0,0\}$, nous pouvons calculer l'état suivant et recommencer le processus de sélection d'un nouveau vecteur d'entrées en utilisant ces valeurs dans la fonction de transition 4.1. Le nouvel état sera alors $\{0,1,0,0,1\}$:

★ **Fonction de transition 4.2** : *Calcul du nouvel état*

$$\begin{aligned}
sv_0' : 0 = \;& 1 \wedge 0 \\
sv_1' : 1 = \;& 1 \vee \textbf{if}\ 1\ \textbf{then}\ 0\ \textbf{else}\ 0\ \textbf{fi} \\
sv_2' : 0 = \;& \textbf{if}\ 1\ \textbf{then}\ 0 \\
& \textbf{else}\ \textbf{if}\ \textbf{if}\ 1\ \textbf{then}\ 0\ \textbf{else}\ 0\ \textbf{fi}\ \textbf{then}\ (\mathbf{0} \vee \mathbf{0}) \\
& \quad \textbf{else}\ 1 \\
& \quad \textbf{fi} \\
& \textbf{fi} \\
sv_3' : 0 = \;& 0 \vee \textbf{if}\ 1\ \textbf{then}\ 0\ \textbf{else}\ 0\ \textbf{fi} \\
sv_4' : 1 = \;& \textbf{if}\ 0\ \textbf{then}\ 0 \\
& \textbf{else}\ \textbf{if}\ \textbf{if}\ 1\ \textbf{then}\ 0\ \textbf{else}\ 0\ \textbf{fi}\ \textbf{then}\ (\mathbf{0} \vee \mathbf{0} \\
& \quad \textbf{else}\ 1 \\
& \quad \textbf{fi} \\
& \textbf{fi}
\end{aligned}$$

[5]Nous n'avons pas représenté l'étiquetage du bdd sur la figure 4.11 pour ne pas alourdir la présentation.

Nous ne donnons pas ici le détail des algorithmes de simulation aléatoire de l'environnement. Ils font partie du travail présenté dans [Par96].

Deuxième partie

Test guidé par les propriétés de sûreté

Chapitre 5

Test guidé par les propriétés de sûreté

5.1 Motivations

Au chapitre 4, nous avons exposé les méthodes dont dispose LUTESS pour sélectionner les vecteurs d'entrées à soumettre au logiciel sous test. Cette thèse porte sur les moyens de mieux guider la sélection des vecteurs d'entrées afin de placer le logiciel en situation de violer les propriétés de sûreté. Ceci répond au souci de s'assurer du bon comportement du logiciel dans les situations critiques.

LUTESS offre par l'intermédiaire des schémas comportementaux [Zua00b] (voir paragraphe 4.1.2) un moyen efficace d'amener l'environnement dans une situation critique pour le logiciel. Une telle situation crée pour le logiciel sous test les conditions propices à la violation d'une propriété de sûreté qui sont au nombre de deux:

- l'existence d'un vecteur d'entrées valide dans l'environnement qui rend l'évaluation de la propriété de sûreté dépendant uniquement du vecteur de sorties;
- une valuation du vecteur de sorties qui rend l'évaluation la propriété de sûreté à *faux* .

Un schéma comportemental définit deux listes d'événements de l'environnement:

- la liste $(cond_1, cond_2, ..., cond_n)$ des événements que l'on souhaite observer dans la séquence de données de test à générer par LUTESS;
- la liste $(inter_cond_1, inter_cond_2, ..., inter_cond_{n-1})$ des événements qui ne doivent pas apparaître dans les intervalles entre deux événements de la liste précédente: $inter_cond_1$ ne doit pas être observé entre les événements $cond_1$ et $cond_2$.

Si nous considérons l'exemple simple de la propriété $Ps = Once_From_To(o_1, i_1, i_2)$: la situation critique s'étend entre les événements i_1 et i_2 de l'environnement où le logiciel doit produire la sortie o_1. Un schéma qui pourrait mener à l'observation d'une violation de cette dernière serait:

$$\begin{array}{rcl} cond & : & (i_1, i_2) \\ intercond & : & (vrai) \end{array}$$

L'idée est de guider LUTESS pour qu'il génère i_1, puis i_2. Cependant, pour qu'une violation effective de Ps soit observée, il faut en plus que le logiciel n'ait pas généré o_1 entre les événements i_1 et i_2.

La difficulté pour le testeur lorsqu'il travaille à partir de spécifications, est d'extraire des propriétés de sûreté la situation critique et les différentes étapes du schéma comportemental qui vont guider LUTESS vers cette dernière.

Dans [OP94a], pour guider le test avec les propriétés de sûreté, il avait été proposé de favoriser les vecteurs d'entrées tels que la valeur de vérité de la propriété de sûreté ne dépende que du vecteur de sorties calculé par le logiciel à l'état courant. Cette approche a le défaut d'être dépendante du "*hasard*": pour que la solution proposée dans [OP94a] soit applicable, il faut que l'environnement se trouve déjà dans une situation critique, ce qui dépend de la séquence de test aléatoirement générée.

Nous proposons [PV01b, PV01a] ici un moyen pour LUTESS de construire automatiquement la séquence des vecteurs d'entrées qui permet de mener le logiciel dans une situation critique à partir de la seule description des propriétés de sûreté. Nous proposons dans ces travaux une solution qui permet de prévoir les vecteurs d'entrées à engendrer à l'instant courant pour qu'une violation de la propriété de sûreté puisse être observée à un instant futur.

5.2 Expression des propriétés de sûreté dans le nœud de test

5.2.1 Structure syntaxique

Les propriétés de sûreté qui vont guider la sélection des vecteurs d'entrées sont décrites dans le nœud de test de manière similaire aux contraintes d'environnement. La forme générale d'un nœud de test prenant en compte les propriétés de sûreté est donnée avec le nœud 5.1. De la même façon que les contraintes d'environnement (voir paragraphe 4.1.3), les propriétés de sûreté sont exprimées au sein d'une liste d'expressions booléennes LUSTRE et identifiées par le nouvel opérateur **safety**.

★ **Nœud 5.1** : *Structure syntaxique d'un nœud de test guidé par les propriétés de sûreté:*

testnode env(*sorties du logiciel sous test*)
returns (*entrées du logiciel sous test*)
var
  variables locales
let
  environment($C_1, \ldots, C_n$);
  safety($Ps_1, \ldots, Ps_n$);
  définition des variables locales
tel

L'ensemble des propriétés de sûreté que nous introduisons dans le nœud de test seront uniquement utilisées comme des guides pour le choix du vecteur d'entrées à générer: une violation des propriétés de sûreté ne pourra être observée que si ces propriétés sont aussi intégrées à l'oracle de notre système sous test[1].

[1] Voir le principe de l'architecture de LUTESS présentée au paragraphe 4.1.1.

5.2.2 Sémantique

La prise en compte des propriétés de sûreté pour guider la sélection des vecteurs d'entrées par l'intermédiaire de l'opérateur **safety** nécessite d'ajouter une nouvelle règle à l'ensemble des règles de sémantique défini au paragraphe 4.2.1:

– compatibilité avec l'opérateur **safety**.

$$\frac{\sigma, \sigma' \vdash P_{s_1} \mid out \wedge ... \wedge \sigma, \sigma' \vdash P_{s_r} \mid out}{\sigma, \sigma' \vdash \mathbf{safety}(P_{s_1}, ..., P_{s_r})}$$

Dans l'état courant de l'environnement, une propriété de sûreté P_s est évaluée avec la valeur *out*[2]. En effet, nous avons vu au paragraphe 4.2.2 qu'au demi-instant où LUTESS construit le vecteur d'entrées, le vecteur de sorties du logiciel n'est pas encore connu. Or, la valeur d'une propriété de sûreté dépend à la fois de l'état, du vecteur d'entrées et du vecteur de sorties courant. La valeur des propriétés de sûreté ne peut donc pas être connue tant que le logiciel sous test n'a pas calculé son vecteur de sorties.

5.3 Simulateur guidé par les propriétés de sûreté

Nous présentons dans ce paragraphe le modèle formel d'un simulateur d'environnement guidé par les propriétés de sûreté. Nous commençons par donner quelques exemples simples de sélections de vecteurs d'entrées guidées par ces propriétés. Ensuite, nous définissons la notion de chemin dans l'environnement et de propriétés de sûreté. Ces deux derniers éléments sont nécessaires pour caractériser formellement le simulateur d'environnement guidé par les propriétés de sûreté. Nous terminons ce paragraphe par la définition d'un état suspect de ce simulateur: cette notion d'état suspect est un moyen de comparer les différentes heuristiques de sélection d'un vecteur d'entrées.

5.3.1 Problématique

Le principe du test guidé par les propriétés de sûreté est de choisir un vecteur d'entrées tel que la valeur de vérité de la propriété de sûreté dépende uniquement de la valeur du vecteur de sorties calculée par le logiciel sous test. Considérons un programme, très simple, n'ayant qu'une seule entrée i et qu'une seule sortie o et supposons que nous souhaitons valider la propriété suivante:

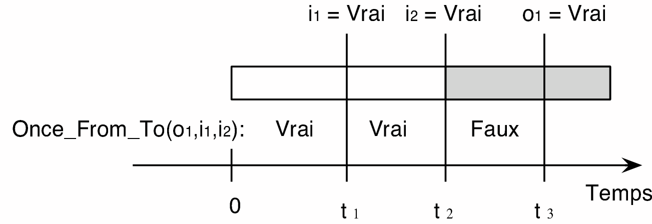
$$P_{s_1} : \; i \Rightarrow o.$$

Le test guidé par les propriétés de sûreté favorise la génération de l'entrée $i = vrai$ à condition que les contraintes d'environnement le permettent. En effet, la valeur de vérité de P_{s_1} ne dépend alors que de la sortie o. En revanche, si $i = faux$ était choisi, P_{s_1} aurait été évaluée à *vrai* quelle que soit la valeur de la sortie o calculée par le logiciel.

Les propriétés de sûreté que nous considérons sont écrites en LUSTRE qui permet de les exprimer en fonction d'événements passés. Rechercher une violation d'une propriété requiert

[2]Nous rappelons que la valeur *out* représente la valeur "*pas encore connue*" (voir paragraphe 4.2.1).

FIG. 5.1: Opérateur *Once_From_To*.



donc de générer à l'instant courant les événements qui permettront d'observer une violation de la propriété à un instant futur. Soit la propriété:

$$P_{s_2} : \; true \rightarrow \mathbf{pre} \; i \Rightarrow o$$

où i est une entrée du logiciel et o une sortie. Si l'on veut mettre le logiciel en situation de violer la propriété de sûreté P_{s_2}, il faut être capable de générer l'entrée $i = vrai$ à l'instant courant pour que la propriété soit violable à l'instant suivant.

Considérons un dernier exemple repris du paragraphe 5.1 basé sur un opérateur de logique temporelle:

$$P_{s_3} = Once_From_To(o_1, i_1, i_2)$$

où o_1 est une sortie du logiciel sous test et i_1, i_2 deux entrées de ce même logiciel. Comme le montre la figure 5.1, cette propriété est violée si LUTESS génère les entrées $i_1 = vrai$ puis "un peu plus tard" $i_2 = vrai$ sans que le logiciel ait eu le temps de produire la sortie $o_1 = vrai$.

Le problème considéré ici est la reconstitution automatique du passé nécessaire pour mener l'environnement dans une situation telle que le logiciel puisse violer ses propriétés de sûreté.

5.3.2 Chemin dans l'environnement

La définition d'un chemin de l'environnement nécessite d'introduire la notation

$$E^n \Leftrightarrow \underbrace{E \times ... \times E}_{n \; fois},$$

où E désigne un ensemble et n un entier positif.

Définition 5.1 *Soit* $M_{env} = (S_{env}, s_{init_{env}}, I, O, f_{env}, env, part_{env}(V_{I_{env}}), sel_{env})$ *un simulateur d'environnement. Nous définissons un chemin* p *issu de* $s_0 \in S_{env}$ *et de longueur* n *comme une suite d'états* $(s_0, ..., s_n) \in S_{env}^{n+1}$ *tel que:*

$$\exists (i_0, ..., i_{n-1}) \in V_I^n, \exists (o_0, ..., o_{n-1}) \in V_O^n \; tel \; que$$
$$\forall k \in [0..n-1], s_{k+1} = f_{env}(s_k, i_k, o_k) \; et \; (i_k, s_k) \in env.$$

Intuitivement, la longueur n d'un chemin $(s_0, ..., s_n)$ correspond à un nombre de transitions parcourues.

Nous parlerons de chemin pour désigner une suite d'états liés par la fonction de transition qui mène d'un état vers un autre. Intuitivement, nous pouvons relier un chemin de l'environnement à un flot LUSTRE (voir §3.2.1) en considérant un chemin comme une partie finie d'un flot. Ceci nous permet de décomposer un chemin de longueur n en n instants logiques successifs. Nous notons l'instant courant par t ou $t+0$ et par $t+k$ le $k^{ième}$ instant après l'instant courant.

5.3.3 Propriétés de sûreté

Les propriétés de sûreté étant exprimées en LUSTRE, elles peuvent être représentées par un automate comme illustré dans le paragraphe 3.3.

Définition 5.2 *Soit Ps une propriété de sûreté telle que:*

$$Ps = (S_{Ps}, s_{init_{Ps}}, I, O, f_{Ps}, Saf_{Ps})$$

- S_{Ps} *représente l'ensemble des états de la propriété de sûreté;*
- $s_{init_{Ps}}$ *est l'état initial;*
- *I (resp. O) représente l'ensemble des variables d'entrées (resp. de sorties) du logiciel sous test;*
- $f_{Ps} : S_{Ps} \times V_I \times V_O \rightarrow S_{Ps}$ *représente la fonction de transition, V_I et V_O étant respectivement l'ensemble des valuations des variables d'entrées et de sorties;*
- $Saf_{Ps} : S \times V_I \times V_O \rightarrow \{\text{vrai}, \text{faux}\}$ *définit la fonction d'évaluation de la propriété.*

Dans la suite du manuscrit nous parlerons indifféremment de la ou des propriétés de sûreté. En effet, une propriété de sûreté est un invariant temporel et une conjonction d'invariants temporels reste un invariant temporel. Ainsi, il est possible de considérer la conjonction de l'ensemble des propriétés de sûreté d'un logiciel comme une seule propriété de sûreté.

Afin de pouvoir guider la génération à partir des propriétés de sûreté, il est nécessaire d'étendre le simulateur d'environnement défini au paragraphe 4.2.2 de telle sorte que les propriétés de sûreté y soient intégrées. Nous parlerons alors de simulateur guidé par les propriétés de sûreté.

5.3.4 Simulateur d'environnement guidé par les propriétés de sûreté

Soient $M_{env} = (S_{env}, s_{init_{env}}, I, O, f_{env}, env, part_{env}(V_{I_{env}}), sel_{env})$ un simulateur d'environnement tel que défini au paragraphe 4.2.2 et $Ps = (S_{Ps}, s_{init_{Ps}}, I, O, f_{Ps}, Saf_{Ps})$ une propriété de sûreté. Afin de définir un simulateur d'environnement guidé par les propriétés de sûreté, il faut en premier lieu que M_{env} et Ps disposent des même variables d'entrées et de sorties (I et O). Nous devons ensuite définir le nouvel ensemble d'états S, l'état initial s_{init} et la nouvelle fonction de transition f:

- $S \subseteq S_{env} \times S_{Ps}$;
- $s_{init} = (s_{init_{env}}, s_{init_{Ps}})$;

- $f : (S_{env} \times S_{Ps}) \times V_I \times V_O \rightarrow (S_{env} \times S_{Ps})$. Soient les états s_{env} et s'_{env} (resp. s_{Ps} et s'_{Ps}) appartenant à S_{env} (resp. S_{Ps}). Soit $i \in V_I$ et $o \in V_O$ tel que $s'_{env} = f_{env}(s_{env}, i, o)$ et $s'_{Ps} = f_{Ps}(s_{Ps}, i, o)$. Nous avons alors:

$$\begin{aligned}(s'_{env}, s'_{Ps}) &= f((s_{env}, s_{Ps}), i, o) \\ &= (f_{env}(s_{env}, i, o), f_{Ps}(s_{Ps}, i, o))\end{aligned}$$

Enfin, nous devons adapter la description de l'environnement env et la fonction Saf_{Ps} au nouvel ensemble d'états:

- $env \subseteq (S_{env} \times S_{Ps}) \times V_I$;
- $Saf_{Ps} : (S_{env} \times S_{Ps}) \times V_I \times V_O \rightarrow \{vrai, faux\}$.

Un simulateur guidé par les propriétés de sûreté peut alors être défini de manière analogue au simulateur d'environnement présenté au paragraphe 4.2.2.

Définition 5.3 *Soit M_{saf} un simulateur guidé par les propriétés de sûreté Ps.*

$$M_{saf} = (S, s_{init}, I, O, f, env, part_{env}(V_{I_{env}}, Saf_{Ps}), sel_{env})$$

où:

- *S est l'ensemble des états du simulateur M. Un état représente l'ensemble des états de l'environnement et des propriétés de sûreté: il mémorise toutes les valeurs du passé nécessaires aux calculs de l'état suivant et des propriétés de sûreté;*
- *s_{init} est l'état initial;*
- *I est l'ensemble des variables d'entrées;*
- *O est l'ensemble des variables de sorties;*
- *$f : S \times V_I \times V_O \rightarrow S$ est la fonction de transition, où V_I (resp. V_O) représente l'ensemble des valuations des variables d'entrées (resp. de sorties);*
- *$env \subseteq S \times V_I$ représente les contraintes d'environnement;*
- *$V_{val} = part_{env}(V_{I_{env}}, Saf_{Ps})$ est une méthode qui détermine en fonction de l'ensemble des vecteurs d'entrées valides ($V_{I_{env}}$) et des propriétés de sûreté (Saf_{Ps}), l'ensemble V_{val} des vecteurs d'entrées dans lequel un vecteur d'entrées sera choisi;*
- *$sel_{env}(V_{val})$ est la méthode qui permet de choisir un vecteur d'entrées à soumettre au logiciel sous test.*

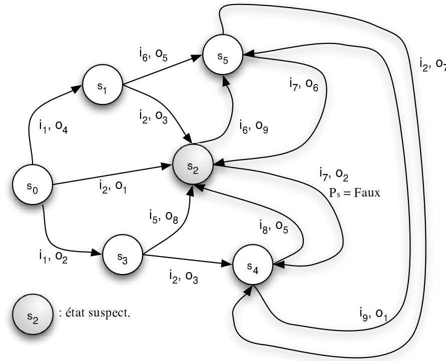
La nouvelle méthode $part_{env}(V_{I_{env}}, Saf_{Ps})$ que nous présentons ici se décompose en deux étapes:

1. On considère l'ensemble des chemins P_k de longueur k (avec $k \in [0..n]$) issus de l'état courant s_{t+0} et n la longueur maximale des chemins[3] à explorer. Nous définissons alors l'ensemble $V_{I_k}(s_{t+0})$ des vecteurs d'entrées pertinents d'ordre k pouvant mener à l'observation d'une violation de la propriété P_s à l'issue d'un chemin de P_k. Nous regroupons ensuite les ensembles de vecteurs d'entrées $V_{I_k}(s_{t+0})$ dans une liste $\Xi_n(s_{t+0})$, tel que:

$$\Xi_n(s_{t+0}) = [V_{I_0}(s_{t+0}), .., V_{I_n}(s_{t+0})]$$

[3] Nous discuterons au paragraphe 5.8 du choix de n.

FIG. 5.2: Représentations d'un état suspect.



2. On construit à l'aide de l'une des trois stratégies *intersection*, *union* et *paresseuse*, l'ensemble des vecteurs d'entrées V_{val} à partir de $\Xi_n(s_{t+0})$.

Nous présenterons aux paragraphes 5.4 et 5.5 la formalisation du calcul des ensembles $V_{I_k}(s_{t+0})$ qui permettent de construire $\Xi_n(s_{t+0})$; au paragraphe 5.10 nous donnerons un exemple détaillé du calcul des $V_{I_k}(s_{t+0})$ alors que le paragraphe 5.6 s'intéressera à la description des stratégies proposées pour la génération de l'ensemble V_{val}.

5.3.5 États suspects du simulateur d'environnement

Intuitivement, un état est dit *suspect* si le simulateur d'environnement peut générer, dans cet état, un vecteur d'entrées qui met le logiciel en situation de fournir une réponse invalidant la propriété de sûreté Ps.

Définition 5.4 *Soit* $M_{saf} = (S, s_{init}, I, O, f, env, part_{env}(V_{I_{env}}, Saf_{Ps}), sel_{env})$ *un simulateur guidé par les propriétés de sûreté. Un état* $s \in S$ *du simulateur est dit suspect pour les propriétés de sûreté ssi*

$$\exists i \in V_I, \exists o \in V_O \text{ tel que } \neg Saf_{Ps}(s, i, o) \text{ et } (s, i) \in env$$

La définition 5.4 est illustrée par la figure 5.2 où la valeur de la propriété de sûreté n'est précisée que dans le cas où elle est fausse. Seul l'état s_2 est suspect étant donné qu'il est le seul état où il existe une transition pour laquelle la propriété est évaluée à *faux* .

Remarque 5.1 *L'état suspect permet de caractériser formellement une situation critique. En effet, un état suspect remplit par définition les deux conditions définies au paragraphe 5 qui peuvent être résumé par l'existence d'un couple de vecteurs d'entrées/sorties tel que la valeur de la propriété de sûreté ne dépende que du vecteur de sorties.*

5.4 Calcul des ensembles de vecteurs d'entrées pertinents

Intuitivement, un vecteur d'entrées pertinents d'ordre k est un vecteur d'entrées qui peut mener à un état suspect à l'issue d'un chemin de longueur k. Nous définissons ci-dessous

l'ensemble $V_{I_k}(s_{t+0})$ (définition 5.6) des vecteurs d'entrées pertinents d'ordre k que nous pouvons engendrer à partir de l'état courant s_{t+0}. Dans ce but, il est nécessaire de pouvoir déterminer l'ensemble des états accessibles par un chemin de longueur k issu de l'état courant s_{t+0} et du vecteur d'entrées i_{t+0} (définition 5.5): s'il existe au moins un état suspect parmi l'ensemble des états accessibles, alors nous aurons $i_{t+0} \in V_{I_k}(s_{t+0})$.

La définition de l'ensemble $V_{I_k}(s_{t+0})$ et le calcul des états accessibles utilisent l'opérateur *constrain* (noté $\uparrow$) tel qu'il est défini dans la thèse de Pascal Raymond [Ray91]. Cet opérateur permet de construire la fonction "f *sachant* g":

$$\text{"}f \uparrow g \equiv \textbf{si } g \textbf{ alors } f \textbf{ sinon } \textit{indéterminé}\text{"}$$

Cet opérateur permet de simplifier l'expression de f en faisant l'hypothèse que g est *vrai* .

Définition 5.5 *Soit un simulateur M, guidé par les propriétés de sûreté:*

$$M_{saf} = (S, s_{init}, I, O, f, env, part_{env}(V_{I_{env}}, Saf_{Ps}), sel_{env}),$$

$S_k(s_{t+0}, i_{t+0})$, l'ensemble des états de M accessibles par un chemin de longueur n, est calculé par induction de la manière suivante avec $k \in [0..n]$:

$$\begin{cases} S_0(s_{t+0}, i_{t+0}) & = & \{s_{t+0}\} \\ S_1(s_{t+0}, i_{t+0}) & = & \{s \in S | \exists o_{t+0} \in V_O, \\ & & \quad s = f(s_{t+0}, i_{t+0}, o_{t+0}) \uparrow env(i_{t+0}, s_{t+0})\} \\ S_k(s_{t+0}, i_{t+0}) & = & \{s \in S | \exists s_{t+k-1} \in S_{k-1}(s_{t+0}, i_{t+0})), \exists (i_{t+k-1} o_{t+k-1}) \in V_I \times V_O, \\ & & \quad s = f(s_{t+k-1}, i_{t+k-1}, o_{t+k-1}) \uparrow env(i_{t+k-1}, s'_{t+k-1})\} \end{cases}$$

Le calcul par induction de $S_n(s_{t+0}, i_{t+0})$ est initialisé par $S_0(s_{t+0}, i_{t+0})$ et $S_1(s_{t+0}, i_{t+0})$. $S_1(s_{t+0}, i_{t+0})$ représente l'ensemble des états accessibles à partir de s_{t+0} et du vecteur d'entrées i_{t+0} en une transition. Ensuite, à chaque pas de l'induction, nous calculons l'ensemble des états $S_k(s_{t+0}, i_{t+0})$ à partir de $S_{k-1}(s_{t+0}, i_{t+0})$ jusqu'à ce que $k = n$. Nous représentons $S_k(s_{t+0}, i_{t+0})$ par intention en utilisant une expression booléenne. Cette représentation nous permet de calculer efficacement $S_k(s_{t+0}, i_{t+0})$ en utilisant le principe de substitution.

Cette façon de calculer $S_n(s_{t+0})$ nous donne en fait une approximation du nombre d'états réellement accessibles au bout d'un chemin de longueur n. En effet, à chaque étape du calcul nous considérons les couples de vecteurs d'entrées/sorties tel que:

- le vecteur d'entrées respecte les contraintes d'environnement;
- tous les vecteurs de sorties du logiciel sont envisagés (nous ne faisons pas d'hypothèse sur les réactions du logiciel).

Une fois un vecteur d'entrées sélectionné, le logiciel répond par un seul vecteur de sorties ce qui réduit l'ensemble d'états accessibles à l'instant courant par rapport à celui calculé.

Nous pouvons maintenant, à partir de $S_k(s_{t+0}, i_{t+0})$, définir l'ensemble $V_{I_k}(s_{t+0})$ des vecteurs d'entrées pertinents d'ordre k.

Définition 5.6 *Soit $M_{saf} = (S, s_{init}, I, O, f, env, part_{env}(V_{I_{env}}, Saf_{Ps}), sel_{env})$ un simulateur guidé par les propriétés de sûreté:*

$$V_{I_k}(s_{t+0}) = \{i_{t+0} \in V_I | \exists s_{t+k} \in S_k(s_{t+0}, i_{t+0}), \exists (i_{t+k}, o_{t+k}) \in V_I \times V_O, \\ \neg Saf_{Ps}(s_{t+k}, i_{t+k}, o_{t+k}) \uparrow env(i_{t+k}, s_{t+k})\}$$

où s_{t+k}*,* i_{t+k} *et* o_{t+k} *représentent respectivement l'état du simulateur d'environnement, le vecteur d'entrées et le vecteur de sorties du logiciel sous test à l'issue d'un chemin de longueur* k*.*

Considérons un programme très simple avec quatre entrées i_1, i_2, i_3 et i_4 ainsi qu'une sortie o_1. L'environnement de notre programme est constitué de deux contraintes:

$$\begin{array}{ll} (1) & After(i_2) \Rightarrow \neg i_1 \\ (2) & \neg(\mathit{false} \rightarrow \mathbf{pre}\ i_2) \Rightarrow \neg i_3 \end{array}$$

La première contrainte interdit de produire i_1 après une occurrence de i_2, alors que la seconde interdit de produire i_3 si elle n'est pas précédée d'une occurrence de i_2. Enfin, nous souhaitons valider la propriété de sûreté suivante:

$$(After(i_1) \wedge (\mathit{false} \rightarrow \mathbf{pre}\ i_4 \wedge i_3)) \Rightarrow o_1$$

À partir de ces deux contraintes d'environnement et de cette propriété de sûreté, LUTESS construit le simulateur d'environnement dont la description est donnée ci-après. Chaque fonction de transition partielle (associée à chacune des variables d'état) est accompagnée de sa signification dans la description de l'environnement.

$$\underbrace{\begin{array}{lcl} sv_0 & = & sv_0 \wedge false \\ & = & EstInitial \\ sv_1 & = & i_1\ \mathbf{or\ if}\ sv_0\ \mathbf{then}\ false\ \mathbf{else}\ sv_1 \mathbf{fi} \\ & = & After(i_1) \\ sv_2 & = & i_4 \\ & = & \mathbf{pre}\ i_4 \\ sv_3 & = & i_2\ \mathbf{or\ if}\ sv_0\ \mathbf{then}\ false\ \mathbf{else}\ sv_3 \mathbf{fi} \\ & = & After(i_2) \\ sv_4 & = & i_2 \\ & = & \mathbf{pre}\ i_2 \end{array}}_{\textit{Fonction de transition.}}$$

$$\underbrace{\begin{array}{l} \mathbf{if\ if}\ sv_0\ \mathbf{then}\ false\ \mathbf{else}\ sv_3\ \mathbf{fi\ then} \\ \qquad\qquad \mathbf{not}\ i_1 \\ \mathbf{else}\ true\ \mathbf{fi} \\ \mathbf{and\ if\ not\ if}\ sv_0\ \mathbf{then}\ false\ \mathbf{else}\ sv_4 \mathbf{fi\ then} \\ \qquad\qquad \mathbf{not}\ i_3 \\ \mathbf{else}\ true\ \mathbf{fi} \end{array}}_{\textit{Représentation des contraintes d'environnement.}}$$

Remarque 5.2 *Nous ne donnerons pas ici le détail des calculs des vecteurs d'entrées pertinents*[4] *mais seulement les vecteurs d'entrées obtenus.*

Le tableau 5.1 représente les ensembles de vecteurs d'entrées pertinents $V_{I_k}(s_{t+0})$ obtenus avec la définition 5.6 pour des chemins de longueur $k \in [0..3]$. Dans chaque expression booléenne représentant un ensemble $V_{I_k}(s_{t+0})$, nous avons remplacé les variables d'état par leur signification (voir la fonction de transition ci-dessus).

[4] Voir le paragraphe 5.10 pour un exemple complet du calcul des vecteurs d'entrées pertinents.

Remarque 5.3 *Une entrée du vecteur d'entrées qui n'apparaît pas dans l'expression booléenne de $V_{I_k}(s_{t+0})$ signifie que la valeur de cette entrée n'a pas d'influence sur la capacité du vecteur à mener l'environnement dans un état suspect.*

Les vecteurs d'entrées qui peuvent mener à une violation des propriétés de sûreté par un chemin de longueur 0 sont:

$$V_{I_0}(s_{t+0}) = \{i_1 \in \{vrai, faux\}, i_2 \in \{vrai, faux\}, i_3 = vrai\}$$

à condition que le simulateur ne se trouve pas dans l'état initial et que l'entrée $i_1 = vrai$ ait déjà été engendrée et enfin que les entrées $i_2 = vrai$ et $i_4 = vrai$ aient été engendrées à l'instant précédent.

$$S_{suspect} = \{sv_0 = faux, sv_1 = vrai, sv_2 = vrai, sv_3 \in \{vrai, faux\}, sv_4 = vrai\}$$

Cette condition caractérise pour $V_{I_0}(s_{t+0})$ l'ensemble des états suspects, étant donnée que $V_{I_0}(s_{t+0})$ représente les vecteurs d'entrées pouvant mener à l'observation instantanée d'une violation des propriétés de sûreté.

Considérons maintenant les vecteurs d'entrées appartenant à $V_{I_2}(s_{t+0})$. Dans le cas où le simulateur se trouve dans un état tel que l'entrée i_1 a déjà été observée (c'est à dire $After(i_1) = vrai$), nous pourrons toujours atteindre un état suspect par un chemin de longueur 2 quel que soit le vecteur d'entrées engendré. En revanche, si le simulateur se trouve dans un état où l'entrée i_2 a déjà été observée alors que l'entrée i_1 ne l'a pas été [5], alors aucun vecteur d'entrées ne pourra atteindre d'état suspect par un chemin de longueur 2. Pour tous les autres états du simulateur, nous avons:

$$\begin{aligned} V_{I_2}(s_{t+0}) &= \{i_1 = vrai, i_2 \in \{vrai, faux\}, i_3 \in \{vrai, faux\}\} \bigcup \\ &\quad \{i_1 = faux, i_2 = faux, i_3 \in \{vrai, faux\}\} \end{aligned}$$

L'ensemble $V_{I_1}(s_{t+0})$ est construit sur le même principe que ceux que nous venons de présenter et l'ensemble $V_{I_3}(s_{t+0})$ est identique à $V_{I_2}(s_{t+0})$.

Sur cet exemple jouet, la prise en compte des contraintes d'environnement n'a aucune incidence sur le calcul des vecteurs d'entrées pertinents. Cependant, l'environnement peut être suffisamment important pour que sa prise en compte aboutisse à des temps de calcul trop importants. Nous proposons dans le paragraphe 5.5, différentes approximations permettant de calculer un sur ensemble de $V_{I_k}(s_{t+0})$, en limitant voire supprimant la prise en compte des contraintes d'environnement.

5.5 Approximations du calcul des vecteurs d'entrées pertinents

Les définitions que nous avons données au paragraphe 5.4 permettent de déterminer les vecteurs d'entrées pertinents pour atteindre un état suspect au bout d'un chemin de longueur k en utilisant toutes les informations disponibles dans l'état courant du simulateur:

[5] Déduit à partir de l'ordonnancement des conditions par les **else if** successifs.

TAB. 5.1: Vecteurs d'entrées pertinents calculés en fonction de la longueur du chemin.

Chemin de longueur 0 $V_{I_0}(s_{t+0})$	**if** $\neg EstInitial \wedge After(i_1) \wedge$ **pre** $i_4 \wedge$ **pre** i_2 **then** i_3
Chemin de longueur 1 $V_{I_1}(s_{t+0})$	**if** $EstInitial$ **then** $i_1 \wedge i_2 \wedge i_4$ **else if** $After(i_1)$ **then** $i_2 \wedge i_4$ **else if** $After(i_2)$ **then** $false$ **else** $i_1 \wedge i_2 \wedge i_4$
Chemin de longueur 2 $V_{I_2}(s_{t+0})$	**if** $EstInitial$ **then** $i_1 \vee \neg i_2$ **else if** $After(i_1)$ **then** $true$ **else if** $After(i_2)$ **then** $false$ **else** $i_1 \vee \neg i_2$
Chemin de longueur 3 $V_{I_3}(s_{t+0})$	**if** $EstInitial$ **then** $i_1 \vee \neg i_2$ **else if** $After(i_1)$ **then** $true$ **else if** $After(i_2)$ **then** $false$ **else** $i_1 \vee \neg i_2$

en particulier, nous utilisons toujours la relation *env* pour ne garder que les vecteurs d'entrées valides dans l'environnement à chaque étape du calcul.

Cependant, la relation *env* peut être de très grande taille et dans un souci d'efficacité, nous sommes amenés à considérer une des trois méthodes de calcul suivantes selon les cas:

- *"Tout Environnement"* correspond aux calculs définis au paragraphe 5.4, c'est à dire en tenant toujours compte de l'environnement. Dans ce cas, nous faisons le calcul le plus précis que nous pouvons faire en fonction des connaissances que nous avons du système: propriétés de sûreté et contraintes d'environnement;
- *"Semi Environnement"* est une approximation qui néglige la relation *env* dans le calcul des états accessibles, le calcul des ensembles de vecteurs d'entrées pertinents $V_{I_k}(s_{t+0})$ restant identiques à celui de la définition 5.6. Nous remplaçons le calcul des états accessibles de la définition 5.5 par:

$$\begin{cases} S_0(s_{t+0}, i_{t+0}) & = & \{s_{t+0}\} \\ S_1(s_{t+0}, i_{t+0}) & = & \{s \in S | \forall o_{t+0} \in V_O, \\ & & \quad s = f(s_{t+0}, i_{t+0}, o_{t+0})\} \\ S_k(s_{t+0}, i_{t+0}) & = & \{s \in S | \forall s'_{t+k} \in S_{k-1}(s_{t+0}), \forall (i_{t+k}, o_{t+k}) \in V_I \times V_O, \\ & & \quad s = f(s'_{t+k}, i_{t+k}, o_{t+k})\}; \end{cases}$$

- l'approximation *"Sans Environnement"* reprend le calcul des états accessibles tel qu'il est fait dans le cadre de l'approximation *"Semi Environnement"* et néglige en plus les contraintes d'environnement dans le calcul des ensembles de vecteurs d'entrées pertinents $V_{I_k}(s_{t+0})$:

$$V_{I_k}(s_{t+0}) = \{i_{t+0} \in V_I | \exists s_{t+k} \in S_k(s_{t+0}, i_{t+0}), \exists (i_{t+k}, o_{t+k}) \in V_I \times V_O, \neg Saf_{Ps}(s_{t+k}, i_{t+k}, o_{t+k})\}$$

Dans ce cas nous ne tenons jamais compte de l'environnement. Ici, on ne tient compte

TAB. 5.2: Vecteurs d'entrées pertinents calculés par *"Semi Environnement"* .

Chemin de longueur 0	**if** $\neg EstInitial \wedge After(i_1) \wedge$ **pre** $i_4 \wedge$ **pre** i_2 **then** i_3
Chemin de longueur 1	**if** $EstInitial$ **then** $i_1 \wedge i_2 \wedge i_4$ **else if** $After(i_1)$ **then** $i_2 \wedge i_4$ **else** $i_1 \wedge i_2 \wedge i_4$
Chemin de longueur 2	$true$
Chemin de longueur 3	$true$

TAB. 5.3: Vecteurs d'entrées pertinents calculés par *"Sans Environnement"* .

Chemin de longueur 0	**if** $\neg EstInitial \wedge After(i_1) \wedge$ **pre** i_4 **then** i_3
Chemin de longueur 1	**if** $EstInitial$ **then** $i_1 \wedge i_4$ **else if** $After(i_1)$ **then** i_4 **else** $i_1 \wedge i_4$
Chemin de longueur 2	$true$
Chemin de longueur 3	$true$

que des vecteurs d'entrées qui doivent être générés pour atteindre un état suspect, sans se soucier de savoir si ces derniers respectent les contraintes de l'environnement[6].

Afin de comprendre les effets des différentes approximations en fonction de la longueur du chemin à explorer, nous reprenons l'exemple introduit au paragraphe 5.4 et utilisons les deux éléments suivants:

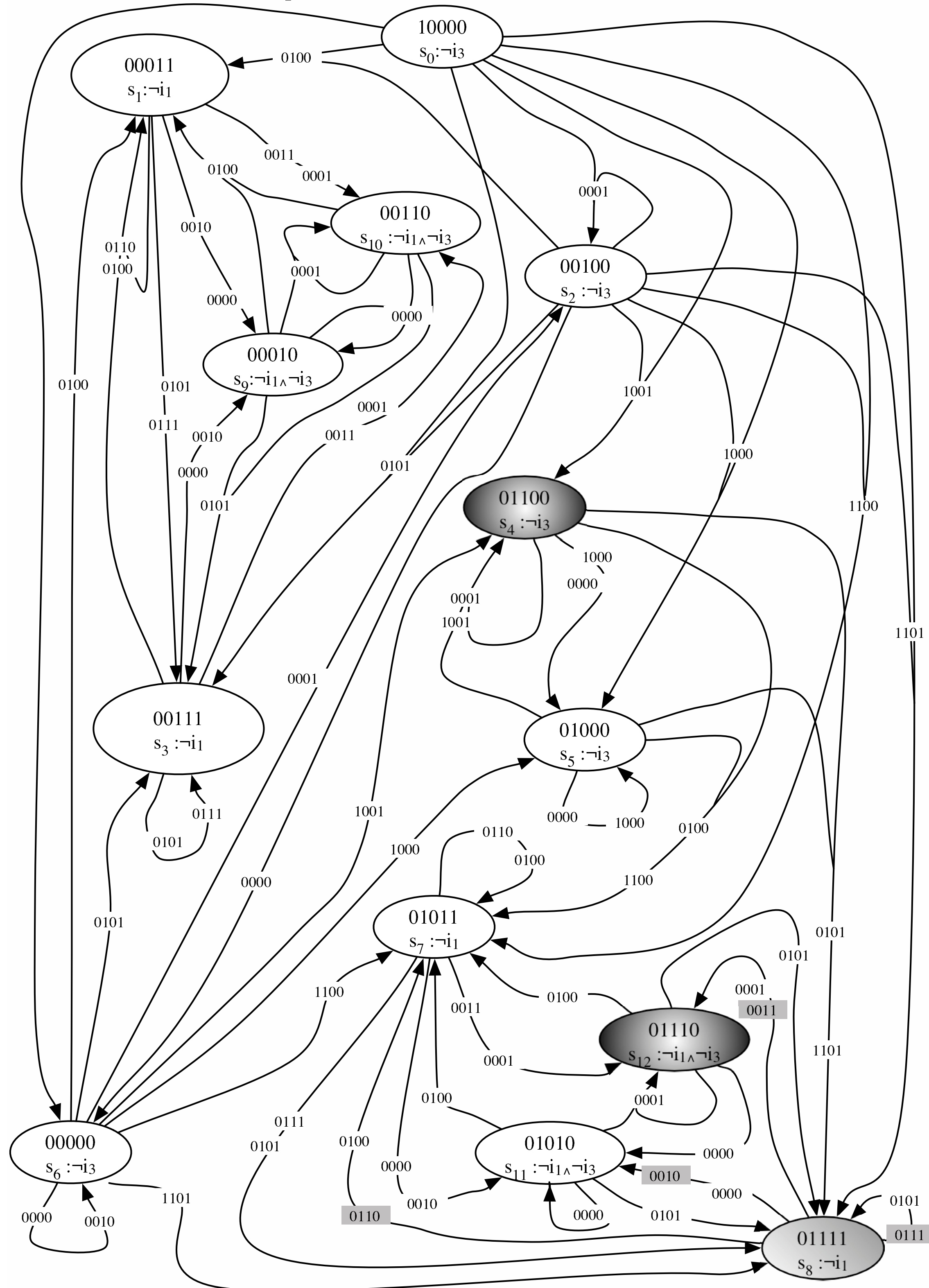
- la figure 5.3 représente le simulateur d'environnement. La valeur des neuf états ($S0$ à $S8$) est représentée par le vecteur de cinq bits: $sv_0..sv_4$. Dans le cas de cet exemple, la fonction de transition ne dépend pas de la valeur de la sortie o_1. Les valeurs notées sur les transitions correspondent aux différents vecteurs $i_1..i_4$ d'entrées possibles menant d'un état à un autre (les vecteurs d'entrées qui ne sont pas représentés sont ceux qui sont interdits par les contraintes d'environnement);
- les ensembles $V_{I_k}(s_{t+0})$ calculés pour les méthodes *"Semi Environnement"* et *"Sans Environnement"* sont respectivement donnés dans les tableaux 5.2 et 5.3 de manière similaire à ceux de la méthode *"Tout Environnement"* donnés dans le tableau 5.1 du paragraphe 5.4.

Pour un chemin de longueur 0, les méthodes *"Tout Environnement"* et *"Semi Environnement"* donnent les même résultats: nous pouvons générer i_3 pour mettre le logiciel sous test en situation de violer P_s uniquement si nous sommes après i_1, que i_4 et i_2 ont été évaluées à *vrai* à l'instant précédent. En revanche, la méthode *"Sans Environnement"* ne s'aperçoit pas qu'il est nécessaire de générer i_2 à l'instant précédent pour pouvoir générer i_3.

Pour un chemin de longueur 1, la méthode *"Sans Environnement"* ne prévoit pas de générer i_2 afin de pouvoir produire i_3 à l'instant suivant alors que les deux autres méthodes

[6] Ici, nous cherchons à obtenir des vecteurs d'entrées pertinents qui peuvent mener à la violation d'une propriété à l'issue d'un chemin de longueur k; parmi ces derniers, la seconde partie de la méthode $part_{env}(V_{i_{env}}, Saf_{Ps})$ engendrera l'ensemble V_{val} compatible avec les contraintes d'environnement (voir §5.6).

Fig. 5.3: Représentations du simulateur d'environnement.



le font. D'autre part, la méthode *"Tout Environnement"* est la seule à prévoir qu'il est impossible de détecter un état suspect si nous sommes après i_2 et que i_1 n'a pas encore été générée.

Pour un chemin de longueur supérieure à 1, seule la méthode *"Tout Environnement"* permet de savoir que l'on ne doit pas générer i_2 avant i_1 sous peine de ne plus pouvoir atteindre d'état suspect (contrainte d'environnement (1)). Les méthodes *"Semi Environnement"* et *"Sans Environnement"* ne parviennent pas à traiter correctement le degré de liberté qui existe au niveau des opérateurs de logique temporelle tels qu'ils sont définis dans l'annexe B.1. Dans le cas de l'opérateur $After(i_1)$, utilisé dans l'exemple ci-dessus, son degré de liberté induit que pour les chemins de longueur supérieure à 1 nous avons le choix de générer ou de ne pas générer i_1 à l'instant courant. Or, si nous choisissons de ne pas générer i_1 à l'instant courant, nous ne devons pas générer i_2 pour ne pas être bloqué par la contrainte d'environnement (1).

L'interprétation de ces expressions booléennes sur la figure 5.3 montre que les méthodes *"Tout Environnement"* et *"Semi Environnement"* considèrent uniquement l'état s_8 comme suspect, alors que la méthode *"Sans Environnement"* considère les trois états s_4, s_8 et S_{12} comme suspects (les états suspects sont ceux caractérisés par le chemin de longueur 0). Il est en effet impossible pour la méthode *"Sans Environnement"* de prévoir qu'il faut générer l'entrée $i_2 = vrai$ à l'instant courant pour être en mesure de générer i_3 à l'instant suivant: c'est pour cette raison qu'elle considère s_4 et s_{12} comme suspects.

Pour les chemins de longueur supérieure à 1, seule la méthode *"Tout Environnement"* les gère correctement en interdisant la génération de l'entrée i_2 avant i_1 (selon les contraintes d'environnement, il n'est plus possible de générer i_1 après une occurrence de i_2). Pour les autres méthodes, elles considèrent qu'il sera toujours possible de trouver un chemin vers un état suspect quel que soit le vecteur d'entrées généré.

5.6 Stratégies de sélection d'un vecteur d'entrées

Nous avons présenté au paragraphe 5.4 le calcul des vecteurs d'entrées pertinents permettant de construire

$$\Xi_n(s_{t+0}) = [V_{I_0}(s_{t+0}), .., V_{I_n}(s_{t+0})].$$

La construction de $\Xi_n(s_{t+0})$ constitue la première des deux étapes de la méthode $part_{env}$ que nous avons définies au paragraphe 5.3.4.

Ici, nous présentons la seconde étape de la méthode $part_{env}$. Cette étape vise à engendrer l'ensemble V_{val} dans lequel LUTESS choisit un vecteur d'entrées. Nous avons défini trois stratégies, *intersection, union* et *paresseuse* qui permettent de construire V_{val} à partir de $\Xi_n(s_{t+0})$.

Remarque 5.4 *Si la stratégie choisie ne permet pas, pour l'état courant, de déterminer un vecteur d'entrées pouvant mener à un état suspect de l'environnement au bout d'un chemin de longueur inférieure ou égale à n (c'est à dire $V_{val} = \emptyset$), alors nous choisissons un vecteur d'entrées parmi tous les vecteurs d'entrées valides dans l'environnement. Nous avons alors: $V_{val} = V_{I_{env}}$.*

5.6.1 Intersection

La stratégie *d'intersection* construit la conjonction de tous les éléments de $\Xi_n(s_{t+0})$:

$$V_{val} = \bigcap_{i=0}^{n} \Xi_n[i](s_{t+0}).$$

Intuitivement, cette stratégie essaye de déterminer un vecteur d'entrées qui puisse conduire la séquence de test dans un état suspect quelle que soit la longueur du chemin comprise entre 0 et n.

La stratégie *intersection* est bien adaptée dans le cas de propriétés comme $P_s = ((\textbf{pre}\ i) \vee i) \Rightarrow o$ où il existe une entrée i qui permet de mener l'environnement vers un état suspect indifféremment de la longueur du chemin comprise entre 0 et 1.

Considérons maintenant un programme très simple possédant deux entrées i_1 et i_2 et une sortie o_1 et la propriété $P_s = ((\textbf{pre}\ i_1) \wedge i_2) \Rightarrow o_1$. Dans ce cas, il n'existe pas de valuation unique pour le vecteur d'entrées qui permette à la fois de mener l'environnement dans un état suspect au bout d'un chemin de longueur 0 ou 1. La stratégie par *intersection* retournera dans ce cas $V_{val} = \emptyset$.

L'exemple que nous venons de prendre montre les limites de la stratégie *intersection*. Cette stratégie se trouve bloquée dès lors qu'il faut générer une séquence de vecteurs d'entrées avec des valeurs différentes à chaque instant d'un chemin menant vers un état suspect.

5.6.2 Union

Cette seconde stratégie construit l'union de tous les éléments de $\Xi_n(s_{t+0})$:

$$V_{val} = \bigcup_{i=0}^{n} \Xi_n[i](s_{t+0}).$$

Intuitivement, cette stratégie regroupe tous les vecteurs d'entrées qui peuvent mener à un état suspect de l'environnement à l'issue d'un chemin de longueur comprise entre 0 et n.

Reprenons la propriété $P_s = ((\textbf{pre}\ i_1) \wedge i_2) \Rightarrow o_1$. Les ensembles de vecteurs d'entrées pertinents pour cette propriété sont:

- $V_{I_0}(s_{t+0}) = \textbf{if pre}\ i_1\ \textbf{then}\ i_2\ \textbf{else}\ false$;
- $V_{I_1}(s_{t+0}) = i_1$.

Lorsque nous calculons l'union de $V_{I_0}(s_{t+0})$ et de $V_{I_1}(s_{t+0})$ nous obtenons:

$$V_{val} = (\textbf{if pre}\ i_1\ \textbf{then}\ i_2\ \textbf{else}\ false) \vee (i_1).$$

Cet ensemble permet à la fois d'engendrer i_1 lorsque **pre** i_1 est *faux* et i_2 ou i_1 dans le cas contraire. Cette stratégie permet d'engendrer la séquence de vecteurs d'entrées telle que i_1 puis i_2 soient évalués à *vrai* en deux instants successifs.

Considérons maintenant la propriété $P_s = (\textbf{pre}\ i_1 \wedge i_2) \vee (\textbf{pre}\ i_3 \wedge i_4) \Rightarrow o$. Nous avons deux façons de mener l'environnement dans un état suspect pour la propriété P_s: soit nous commenons par générer i_1 puis i_2, soit nous générons i_3, puis i_4; dans les deux cas, les deux

entrées doivent être générées consécutivement. La stratégie *d'union* permet de générer i_1 ou i_3 pour atteindre un état suspect au bout d'un chemin de longueur 1, mais également de générer i_2 ou i_4 si respectivement i_1 ou i_3 a été générée à l'instant précédent. Étant donnée que toutes ces entrées ont toutes la même chance d'être choisies par la stratégie *d'union*, nous pouvons avoir une séquence: i_3, i_1, i_3, ... qui ne mène jamais vers un état suspect.

Cette stratégie permet de choisir des vecteurs d'entrées qui peuvent mener à un état suspect en un nombre d'instants k compris entre 0 et n; mais elle ne permet pas de garantir un bon ordonnancement dans le choix des vecteurs d'entrées pour atteindre cet état.

5.6.3 Paresseuse

La stratégie *paresseuse* consiste à déterminer un vecteur d'entrées qui mène le plus rapidement possible vers un état suspect de l'environnement.

$V_{val} = \varnothing; i = 0;$
tant que $V_{val} = \varnothing \wedge i < 0$ **faire**
  $V_{val} = \Xi[i] \cap V_{I_{env}};$
  $i = i + 1;$
fin tant que;

Cette stratégie permet de générer directement la bonne séquence des vecteurs d'entrées (si les contraintes de l'environnement le permettent) qui peut mener vers un état suspect de l'environnement. Considérons un système où l'on souhaite tester la propriété $P_s = ((\mathbf{pre}\ i_1) \wedge i_2) \Rightarrow o_1$. Pour pouvoir être dans une situation où l'on peut observer une violation de la propriété P_s, nous devons générer i_1 suivi de i_2. Supposons que ni i_1, ni i_2 aient été générées à l'instant précédent. Dans ce cas, la stratégie *paresseuse* commence par regarder s'il existe un vecteur d'entrées v_1 qui puisse mettre le système sous test en situation de violer P_s par un chemin de longueur zéro (c'est à dire que l'état courant serait suspect). Un tel vecteur v n'existe pas puisque nous avons supposé que i_1 n'a pas encore été générée. La stratégie *paresseuse* regarde alors s'il existe un vecteur d'entrées v_2 qui mène à un état suspect par un chemin de longueur 1: un tel vecteur existe, par exemple elle pourra choisir le vecteur $v_2 = \{i_1 = vrai,\ i_2 = faux\}$. Lorsque le simulateur d'environnement aura atteint l'état suivant, la stratégie paresseuse pourra alors trouver un vecteur permettant de mettre le système sous test en situation de violer P_s par un chemin de longueur zéro.

La sélection des vecteurs d'entrées par la stratégie *paresseuse* telle qu'elle est présentée ici présente un défaut illustré par l'exemple suivant:

$$P_s = (\mathbf{pre}\ \mathbf{pre}\ i_1 \vee \mathbf{pre}\ i_2) \wedge i_3 \Rightarrow o.$$

La figure 5.4 montre les vecteurs d'entrées à générer en fonction de la longueur du chemin choisie pour atteindre un état suspect. Supposons que ni i_1, ni i_2 n'ont été produites au cours des deux derniers instants. La stratégie *paresseuse* commence par chercher s'il existe un vecteur d'entrées dans l'état courant de l'environnement tel qu'une réponse du logiciel puisse rendre la propriété P_s fausse à l'état courant: aucun vecteur d'entrées ne sera trouvé. La stratégie regardera alors les vecteurs d'entrées qui peuvent mener à un état suspect par un chemin de longueur 1: elle trouvera par exemple le vecteur $\{i_1 = faux,\ i_2 = vrai,\ i_3 = vrai\}$;

Fig. 5.4: Vecteurs d'entrées pertinents.

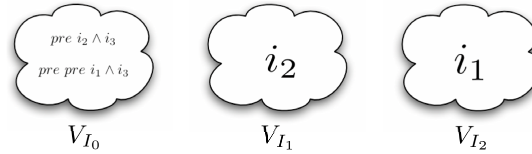
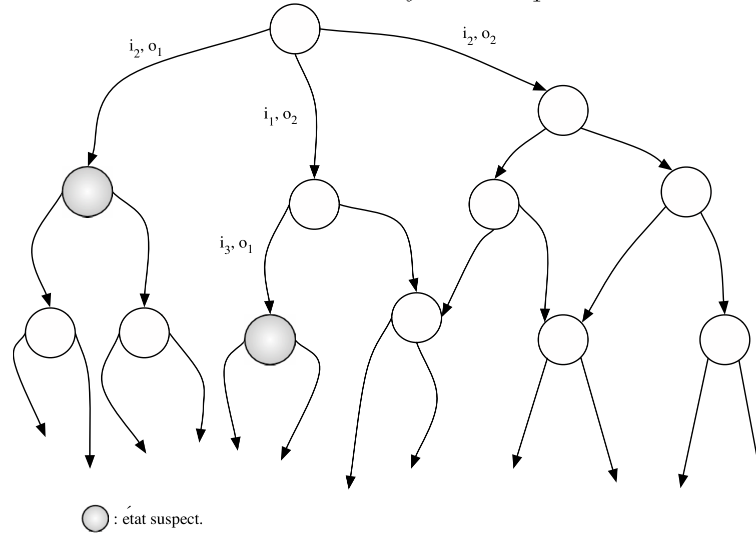


Fig. 5.5: Environnement: choix non systématique des vecteurs d'entrées.



elle ne cherchera donc pas de vecteurs d'entrées pouvant mener à un état suspect par un chemin de longueur 2. La stratégie *paresseuse* va avoir le défaut de tester P_s toujours de la même manière: i_2 à l'instant courant puis i_3 à l'instant suivant.

5.7 Guidage systématique de la sélection des vecteurs d'entrées

Dans ce paragraphe, nous discutons des effets que pourrait produire une utilisation systématique du guidage de la sélection des vecteurs d'entrées. Nous abordons cette discussion en deux parties: tout d'abord dans le cas général du guidage par les propriétés de sûreté puis ensuite pour le cas particulier de la stratégie paresseuse.

5.7.1 Cas général

Les stratégies que nous venons de proposer ont pour but de déterminer des vecteurs d'entrées pouvant mener à des états suspects de l'environnement. Cependant, elles ne peuvent pas garantir qu'un état suspect sera systématiquement atteint étant donné que nous ne connaissons pas quelles vont être les réactions du système sous test au cours de l'intervalle d'instants que nous explorons et que, selon l'approximation choisie, nous considérons les contraintes d'environnement de façon plus ou moins partielle.

Considérons la figure 5.5. Cette figure reproduit partiellement la fonction de transition d'un simulateur d'environnement: l'état courant se trouvant au sommet de la figure et les états suspects étant grisés. Supposons que nous utilisons nos stratégies avec $n = 1$, c'est à dire que nous essayons de déterminer des vecteurs d'entrées qui peuvent mener à un état suspect de l'environnement pour des chemins de longueur inférieure ou égale à 1. Dans ce cas, nos stratégies vont générer i_2 pour tenter de mener l'environnement dans un état suspect par un chemin de longueur 1 à condition que le système sous test génère la sortie o_1. Or si le système génère la sortie o_2, nous ne pourrons pas atteindre d'état suspect avant au moins trois instants et nous nous sommes coupés de la possibilité d'atteindre un autre état suspect par la séquence (i_1, o_2), (i_3, o_1).

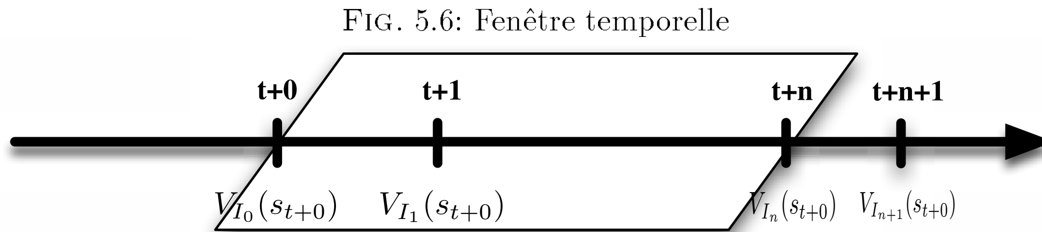
Pour ne pas se priver de vecteurs d'entrées qui peuvent être intéressants, mais qui ne sont pas détectables par nos stratégies, nous introduisons une utilisation probabiliste des stratégies décrites ci-dessus. L'idée est de choisir un vecteur d'entrées dans V_{val} avec une probabilité $\mathcal{P}_{V_{val}}$ ou un vecteur d'entrées parmi tous les vecteurs d'entrées compatibles avec l'environnement avec la probabilité $(1 - \mathcal{P}_{V_{val}})$: nous avons fixé arbitrairement la valeur de $\mathcal{P}_{V_{val}}$ égale à $0,9$.

5.7.2 Cas de la stratégie *paresseuse*

Dans le cas particulier de la stratégie *paresseuse*, ne pas choisir systématiquement le premier ensemble V_{val} calculé par cette dernière permet également de résoudre le problème lorsque nous avions le choix entre plusieurs vecteurs d'entrées (voir §5.6). En effet, nous avions vu qu'une fois que la stratégie *paresseuse* a détecté un vecteur d'entrées pouvant mener vers un état suspect par un chemin de longueur k, elle ne cherche plus à savoir s'il existe un chemin de longueur supérieure à k pouvant mener à un autre état suspect. Or, si nous appliquons le principe du choix non systématique, nous donnons à la stratégie *paresseuse* la possibilité de trouver un vecteur d'entrées menant vers un état suspect par chemin de longueur supérieure à k , alors que nous avons déjà trouvé un vecteur d'entrées menant vers un état suspect par chemin de longueur k. Nous redéfinissons la stratégie *paresseuse* comme ci-dessous où la fonction $prob(x)$ retourne la valeur vraie avec la probabilité x:

$V_{val} = \varnothing; i = 0;$
tant que $V_{val} = \varnothing \wedge i < 0$ **faire**
    if $prob(0,9)$ **then**
        $V_{val} = \Xi[i] \cap V_{I_{env}};$
    else
        $V_{val} = \varnothing;$
    $i = i + 1;$
fin tant que;

FIG. 5.6: Fenêtre temporelle



5.8 Problème du nombre d'instants à considérer

5.8.1 Premières intuitions

Les différentes stratégies que nous avons présentées dans ce chapitre utilisent toutes un paramètre n déterminant la longueur maximale des chemins qui sont explorés afin de déterminer les vecteurs d'entrées qui peuvent mener vers un état suspect. Considérons par exemple la propriété

$$Ps_1: \; true \; \rightarrow \textbf{pre} \; i \Rightarrow o$$

où i et o sont respectivement une entrée et une sortie du logiciel sous test. Avec cette propriété, il est évident que nous ne pouvons rien faire à l'instant courant $(t+0)$ pour mettre le logiciel en situation de violer Ps si nous avons pas eu $i = vrai$ à l'instant $t-1$. En revanche, lorsque nous calculons les vecteurs d'entrées pertinents pour atteindre un état suspect à l'instant $t+1$, nous nous apercevons qu'il faut générer $i = vrai$ à l'instant courant. Si maintenant nous cherchions à déterminer quels seraient les vecteurs d'entrées à générer pour atteindre un état suspect à l'instant $t+2$, alors nous nous apercevrions qu'il faut générer $i = vrai$ à l'instant $t+1$. Or cette information ne nous intéresse pas: nous voulons connaître les vecteurs d'entrées à générer à l'instant courant. Ici, nous devons donc prendre $n = 1$. Au-delà de 1, nous n'obtenons aucune information supplémentaire.

Intuitivement, nous pouvons considérer une fenêtre temporelle commençant à l'instant courant $t+0$ et de largeur n (voir figure 5.8.1) où nous pouvons prévoir quelles entrées doivent être générées afin de mener le système sous test vers un état suspect pour des chemins de longueur inférieure ou égale à n. Au-delà de cette fenêtre, les vecteurs d'entrées obtenus ne sont plus relatifs à l'instant courant, mais à des instants futurs. Les expériences que nous avons menées montrent que la largeur de cette fenêtre correspond au nombre de "*pre imbriqués*" que contient la propriété de sûreté. En d'autres termes, si une propriété de sûreté Ps s'exprime en faisant référence à des événements du passé vieux d'au plus p instants, alors la largeur de la fenêtre temporelle sera égale à p et nous aurons $n = p$.

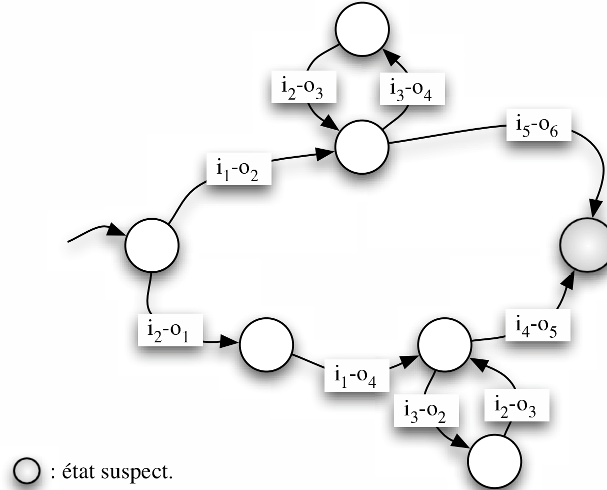
5.8.2 Des cas plus complexes

L'expérience montre que nous obtenons souvent:

$$V_{I_n}(s_{t+0}) = V_{I_{n+1}}(s_{t+0}).$$

En d'autres termes, pour des chemins de longueurs supérieures ou égales à n, l'ensemble des vecteurs d'entrées pertinents pour atteindre un état suspect devient stable. Nous étions dans

FIG. 5.7: Propriété alternante.



ce cas avec l'exemple donné au paragraphe 5.4.

Remarque 5.5 *dans le cas où* $V_{I_n}(s_{t+0}) = V_{I_{n+1}}(s_{t+0}) =$ vrai[7]*, nous devons considérer uniquement les ensembles* $[V_{I_0}(s_{t+0})..V_{I_{n-1}}(s_{t+0})]$ *pour la stratégie d'union: en effet faire l'union avec* $V_{I_n}(s_{t+0}) =$ vrai *aurait pour effet d'absorber tous les vecteurs d'entrées pertinents pouvant mener vers un état suspect pour des chemins de longueurs inférieures à* n*.*

Afin d'éviter au testeur de définir lui-même la valeur de n, nous pouvons envisager d'utiliser la stabilité de $V_{I_n}(s_{t+0})$ lorsque n est assez grand dans le but d'arrêter le calcul des $V_{I_k}(s_{t+0})$. Cependant, il est relativement facile de trouver un contre exemple montrant que cette stabilité peut n'être jamais atteinte: considérons une propriété de sûreté représentée par l'automate de la figure 5.7. Le test guidé par les propriétés de sûreté va construire des ensembles $V_{I_{2k}} = \{i_1\}$ et $V_{I_{2k+1}} = \{i_2\}$ pour $k > 1$. Nous n'avons pas de convergence des ensembles de vecteurs d'entrées pertinents.

Afin de déterminer automatiquement la valeur de n, il faudrait parcourir l'automate de la propriété qui a été défini au paragraphe 5.3.3 pour déterminer le plus long chemin sans boucle. Dans le cas de la propriété représentée sur la figue 5.7, nous obtiendrions un chemin maximum sans boucle de longueur 3, et donc nous trouvons $n = 3$.

5.9 Algorithmes liés au test guidé par les propriétés de sûreté

5.9.1 Stratégie *intersection* et *union*

L'introduction des stratégies *intersection* et *union* dans LUTESS nécessite la modification du calcul de V_{val} par rapport à celle introduite dans [Par96]. À chaque pas dans la boucle,

[7]Ceci signifie que nous pouvons toujours trouver un chemin de longueur supérieure à n (à partir de l'instant courant) menant vers un état suspect quel que soit le vecteur d'entrées choisi à l'instant courant.

les algorithmes 5.1 et 5.2 calculent:

1. l'ensemble des états accessibles à l'instant $t+k$ (l'instant courant étant désigné par $t+0$);
2. l'ensemble des entrées V_{I_k} que l'on peut générer à l'instant courant et pouvant mener à un état suspect à l'instant $t+k$;
3. L'intersection ou l'union avec le calcul partiel de V_{val} effectué au pas précédent.

★ **Algorithme 5.1** : *Stratégie* intersection*: calcul de* V_{val}

$V_{val} := V_{I_{env}}$;
$s := current_state$;
$k := 0$;
tant que $k < n$ **faire**

$$\begin{aligned} V_{I_k} \;:=\; \{ & i \in V_I \mid \exists (i_0, ..., i_k) \in V_I^{k+1}, \exists (o_0, ..., o_k) \in V_O^{k+1}, \\ & \exists (s_0, ..., s_k) \in S^{k+1},\ tel\ que \\ & \forall j \in [0..k],\ i = i_0 \wedge s = s_0 \wedge \\ & s_{j+1} = f(s_j, i_j, o_j) \uparrow env(s_j, i_j) \wedge \\ & \neg P_s(s_{t+k}, i_{t+k}, o_{t+k}) \uparrow env(s_j, i_j)\} \end{aligned}$$

$V_{val} = V_{val} \cap V_{I_k} \cap V_{I_{env}}$;
$k := k + 1$;
fin tant que;

★ **Algorithme 5.2** : *Stratégie* union*: calcul de* V_{val}

$V_{val} := \varnothing$;
$s := current_state$;
$k := 0$;
tant que $k < n$ **faire**

$$\begin{aligned} V_{I_k} \;:=\; \{ & i \in V_I \mid \exists (i_0, ..., i_k) \in V_I^{k+1}, \exists (o_0, ..., o_k) \in V_O^{k+1}, \\ & \exists (s_0, ..., s_k) \in S^{k+1},\ tel\ que \\ & \forall j \in [0..k],\ i = i_0 \wedge s = s_0 \wedge \\ & s_{j+1} = f(s_j, i_j, o_j) \uparrow env(s_j, i_j) \wedge \\ & \neg P_s(s_{t+k}, i_{t+k}, o_{t+k}) \uparrow env(s_j, i_j)\} \end{aligned}$$

$V_{val} := V_{val} \cup (V_{I_k} \cap V_{I_{env}})$;
$k := k + 1$;
fin tant que;

5.9.2 Stratégie *paresseuse*

La stratégie paresseuse construit l'ensemble des vecteurs d'entrées pouvant mener à un état suspect pour chaque instant de l'intervalle $[t+0, ..., t+n-1]$.

★ **Algorithme 5.3** : *Stratégie* paresseuse*: calcul de* V_{val}

$V_{val} := \emptyset$;
$s := current_state$;
$k := 0$;
tant que $k < n \wedge V_{val} = \emptyset$ **faire**

$$V_{I_k} := \{i \in V_I \mid \exists(i_0,...,i_k) \in V_I^{k+1}, \exists(o_0,...,o_k) \in V_O^{k+1}, \exists(s_0,...,s_k) \in S^{k+1},\ tel\ que\ \forall j \in [0..k],\ i = i_0 \wedge s = s_0 \wedge s_{j+1} = f(s_j,i_j,o_j) \uparrow env(s_j,i_j) \wedge \neg P_s(s_{t+k},i_{t+k},o_{t+k}) \uparrow env(s_j,i_j)\}$$

$V_{val} := V_{I_k} \cap V_{I_{env}}$;
$k := k+1$;
fin tant que;
si $V_{val} = \emptyset$ **alors**
$V_{val} := V_{I_{env}}$;

5.10 Annexe: calcul détaillé des vecteurs d'entrées pertinents pour un climatiseur

5.10.1 Principes informels du climatiseur

Afin d'illustrer les techniques de génération de données de test guidées par les propriétés de sûreté, nous allons modifier l'exemple du climatiseur présenté au paragraphe 4.1.4. Dans un souci de simplicité, nous allons utiliser l'approximation *"Sans Environnement"* qui néglige la relation d'environnement dans le calcul des états accessibles et dans la détermination des vecteurs d'entrées pertinents.

Nous présentons sur la figure 5.8 notre nouveau système de climatisation. Le signal d'entrée **Marche** et les signaux de sortie **ARRETE**, **CHAUD**, **INACTIF** et **FROID** restent identiques à ceux présentés dans l'exemple 4.1.4.

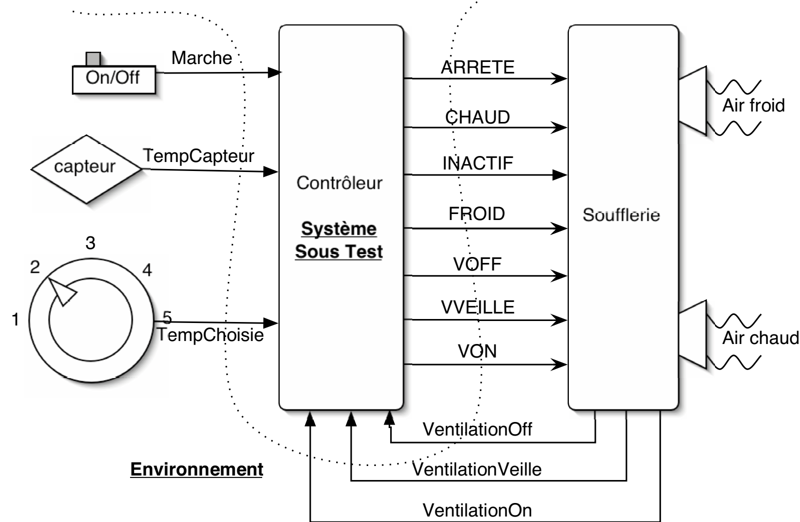
En revanche, ce système de climatisation se distingue du précédent en deux points: d'une part nous modifions la façon d'acquérir la température ambiante et celle souhaitée par l'utilisateur, et d'autre part nous dotons notre climatiseur d'une nouvelle soufflerie.

L'acquisition de la température ambiante et le réglage de la température désirée s'effectuent de la façon suivante:

- la température ambiante est mesurée par l'intermédiaire d'un thermomètre (capteur) et transmise au contrôleur sur l'entrée **TempCapteur**;
- la température souhaitée par l'utilisateur est réglée à l'aide d'un thermostat puis transmise au contrôleur sur l'entrée **TempChoisie**.

La nouvelle soufflerie, plus complexe, peut se trouver dans l'un des trois états Off, $Veille$ et On:

FIG. 5.8: Le climatiseur et son environnement.



- l'état Off: la ventilation de la soufflerie est arrêtée. Le contrôleur doit s'assurer que la soufflerie est dans l'état Off avant d'envoyer la commande **ARRETE**;
- l'état $Veille$: la ventilation de la soufflerie tourne au ralenti. Le contrôleur ordonne à la soufflerie de passer dans cet état lorsque la température ambiante est égale à la température souhaitée par l'utilisateur;
- l'état On: la ventilation de la soufflerie tourne à vitesse normale. Le climatiseur peut ordonner la production de chaleur ou de froid à la soufflerie uniquement si cette dernière se trouve dans l'état On; si ce n'est pas le cas, le contrôleur doit préalablement demander à la soufflerie de passer dans l'état On.

Le contrôleur est informé de l'état dans lequel se trouve la soufflerie par trois signaux d'entrée: **VentilationOff, VentilationVeille, VentilationOn**. Notons également que le changement d'état de la soufflerie ne peut être demandé que par le contrôleur en utilisant une des trois commandes: **SOFF, SVEILLE, SOFF**.

La propriété de sûreté que nous souhaitons valider, s'assure que le climatiseur se met à produire de la chaleur ou que la température désirée par l'utilisateur est égale à la température ambiante ou encore que l'utilisateur a arrêté le climatiseur au plus tard trois unités de temps après avoir constaté que la température ambiante est plus faible que la température souhaitée par l'utilisateur.

5.10.2 Description détaillée de l'environnement du climatiseur

Nous représentons les valeurs de température par un système "d'échantillonnage" à l'aide un tableau de cinq booléens. Soit une température représentée par le tableau T: T possède exactement un élément ayant pour valeur *vrai* (codage "*un parmi n*" de la température), $T[0] = vrai$ étant la plus faible valeur de température et $T[4] = vrai$ représentant la plus forte valeur de température. Nous ne donnerons ici que les définitions informelles des nœuds dont nous avons besoin pour la description de l'environnement et la manipulation des ta-

bleaux de températures: la définition LUSTRE de chacun des nœuds présentés ci-dessous peut être trouvée dans l'annexe B.2. Afin d'écrire une modélisation du comportement de l'environnement, nous avons besoin de primitives permettant de comparer et manipuler les tableaux de température. Cinq primitives ont été définies:

egal(const taille: int; T1, T2: bool^taille) returns (eq: bool) retourne *vrai* ssi les deux températures représentées par les tableaux $T1$ et $T2$ sont égales;

sup(const taille: int; T1, T2: bool^taille) returns (gt: bool) retourne *vrai* ssi la température représentée par le tableau $T1$ est supérieure à la température représentée par le tableau $T2$;

inf(const taille: int; T1, T2: bool^taille) returns (ls: bool) retourne *vrai* ssi la température représentée par le tableau $T1$ est inférieure à la température représentée par le tableau $T2$;

augmente(T1, T1prime: bool^TAILLE_TEMP) returns (aug: bool) retourne *vrai* ssi la température représentée par le tableau $T1prime$ est supérieure ou égale à la température représentée par le tableau $T1$;

diminue(T1, T1prime: bool^TAILLE_TEMP) returns (dim: bool) retourne *vrai* ssi la température représentée par le tableau $T1prime$ est inférieure ou égales à la température représentée par le tableau $T1$.

L'état de la soufflerie est représenté par le tableau *etatSoufflerie*. À chaque instant, exactement un élément du tableau possède la valeur vraie: cette valeur peut être comparée à la place qu'occupe un jeton dans un tableau. Cette représentation nous permet de faire évoluer l'état de la soufflerie en fonction des demandes du contrôleur en déplaçant le jeton d'une case à l'autre du tableau: à chaque instant, le jeton peut se déplacer au plus d'une case. Ainsi, nous pouvons prendre en compte le temps d'accélération ou de ralentissement de la soufflerie entre deux états de cette dernière. Nous définissons deux primitives pour gérer l'état de la soufflerie:

next(st: bool^TAILLE_ETAT_SOUFFLERIE) returns (new_st: bool^TAILLE_ETAT_SOUFFLERIE) permet de faire avancer le jeton d'une case dans le tableau représentant l'état de la soufflerie;

previous(st: bool^TAILLE_ETAT_SOUFFLERIE) returns (new_st: bool^TAILLE_ETAT_SOUFFLERIE) permet de faire reculer le jeton d'une case dans le tableau représentant l'état de la soufflerie.

La description de l'environnement de notre contrôleur de climatiseur nécessite également l'introduction de quatre autres primitives. Les deux premières sont deux opérateurs de logique temporelle, alors que les deux dernières concernent des aspects plus généraux sur les tableaux et expressions LUSTRE:

always_from_to(B, A, C) returns (X: bool) retourne *vrai* ssi B est toujours *vrai* entre les occurrences de A et de C;

once_from_to(B, A, C) returns (X: bool) retourne *vrai* ssi au moins une occurrence de B est observée entre les occurrences de A et de C.

ExactementUn(const n: int; A: bool^n) returns (EXU: bool) retourne *vrai* si le tableau A de taille n contient uniquement un élément ayant pour valeur *vrai*;

delay(const d: int; val: bool) returns (valAftDelay: bool) retourne la valeur de *val* après un délai de d instants.

★ **Nœud 5.2** : *environnement du climatiseur*

```
testnode env(ARRETE, INACTIF, CHAUD, FROID, SOFF, SVEILLE, SON: bool)
returns (Marche, SoufflerieOff, SouffllerieOn, SouffllerieVeille: bool;
         TempCapteur, TempChoisie: bool^5)
var
    etatSoufflerie : bool^TAILLE_ETAT_SOUFFLERIE;
let
  environment(
    -- Contraintes sur le choix de la température par l'utilisateur:
    -- P_env1
    always_from_to(egal(TAILLE_TEMP, INIT_TEMP -> pre TempChoisie, TempChoisie),
        false -> pre (not egal(TAILLE_TEMP, INIT_TEMP -> pre TempChoisie,
                          TempChoisie)),
        delay(5, edge(egal((TAILLE_TEMP, INIT_TEMP -> pre TempChoisie),
                          TempChoisie)))),
    -- Évolution de la température en fonction du fonctionnement du climatiseur:
    -- P_env2
    if CHAUD then
            augmente(pre TempCapteur, TempCapteur)
    else if FROID then
            diminue(pre TempCapteur, TempCapteur)
    else if INACTIF or ARRETE then
            (augmente(pre TempCapteur, TempCapteur) or
            diminue(pre TempCapteur, TempCapteur) or
            egal(TAILLE_TEMP, pre TempCapteur, TempCapteur)),
    -- P_env3
    once_from_to(egal(TAILLE_TEMP, TempChoisie, TempCapteur),
            sup(TAILLE_TEMP, TempChoisie, TempCapteur),
            inf(TAILLE_TEMP, TempChoisie, TempCapteur)),
    -- P_env4
    once_from_to(egal(TAILLE_TEMP, TempChoisie, TempCapteur),
            inf(TAILLE_TEMP, TempChoisie, TempCapteur),
            sup(TAILLE_TEMP, TempChoisie, TempCapteur)),
    -- Modélisation de la température: les tableaux TempChoisie et TempCapteur contiennent
    -- exactement un élément égal à Vrai:
    -- P_env5
    ExactementUn(TAILLE_TEMP, TempCapteur),
    -- P_env6
    ExactementUn(TAILLE_TEMP, TempChoisie))
    -- Modélisation du comportement de la soufflerie.
    -- P_env7
    if etatSoufflerie[ST_OFF] then SouffllerieOff and not(SouffllerieVeille or SouffllerieOn)
    else if etatSoufflerie[ST_VEILLE] then SouffllerieVeille and not(SouffllerieOff or SouffllerieOn)
    else if etatSoufflerie[ST_ON] then SouffllerieOn and not(SouffllerieVeille or SouffllerieOff)
    else not SouffllerieOn and not SouffllerieVeille and not SouffllerieOff,
    -- À l'état initial, le climatiseur est arrêté:
    -- P_env8
    not Marche -> true
  );
  safety(
    once_from_to(CHAUD or not Marche or egal(TAILLE_TEMP, TempChoisie, TempCapteur),
            sup(TAILLE_TEMP, TempChoisie, TempCapteur),
            delay(3, sup(TAILLE_TEMP, TempChoisie, TempCapteur)))
  );
    -- Définition des variables locales:
  etatSoufflerie = INIT_ST -> if pre(SON or (SVEILLE and SouffllerieOff)) then
                                  next(pre etatSoufflerie)
                              else if pre (SOFF or (SVEILLE and SouffllerieOn)) then
                                  previous(pre etatSoufflerie)
                              else
                                  pre etatSoufflerie;
tel
```

Nous modélisons le comportement de l'environnement par six propriétés:

- La première (P_{env1}) définit le comportement de l'utilisateur vis-à-vis du réglage de la température souhaitée. Cette propriété impose de ne pas changer de façon permanente la température choisie par l'utilisateur. Dans ce but nous imposons que la température choisie par l'utilisateur reste constante au moins pendant les cinq instants suivant la dernière variation de température.
- Les trois propriétés suivantes modélisent l'évolution de la température en fonction du fonctionnement du climatiseur. La première d'entre-elles (P_{env2}) permet de faire augmenter (resp. baisser) la température ambiante lorsque le climatiseur produit de la chaleur (resp. fraîcheur); de plus lorsque le climatiseur est inactif ou arrêté, la température peut évoluer librement. Les deux autres propriétés (P_{env3} et P_{env4}) assurent la continuité de l'évolution de la température ambiante. Pour passer d'une situation où la température ambiante est inférieure à la température choisie par l'utilisateur à une situation où la température ambiante est supérieure à la température choisie par l'utilisateur, la température ambiante est alors obligée de passer par une étape où elle est égale à la température choisie par l'utilisateur (et *vice versa*).
- Les propriétés P_{env5} et P_{env6} posent des contraintes sur la représentation de la température. Nous avons vu que la température est représentée par un tableau de cinq booléens avec un codage "*un parmi n*". Ces deux propriétés imposent à l'environnement de générer uniquement des tableaux de température contenant exactement un élément ayant pour valeur *vrai* .
- La propriété P_{env7} fixe l'état de la soufflerie en fonction de la place occupée par le jeton dans le tableau *etatSoufflerie*.
- La dernière propriété de notre environnement (P_{env8}) s'assure que le climatiseur est arrêté à l'état initial.

5.10.3 L'environnement du climatiseur issu de l'analyse de LUTESS

Une description détaillée de l'environnement de cette nouvelle version du climatiseur est donnée à la fin de ce chapitre, au paragraphe 5.10.2. La génération automatique du simulateur d'environnement conduit à une machine d'états finis contenant 34 variables d'états. Nous ne présenterons ici que les fonctions de transitions partielles des variables d'état intervenant dans l'expression de la propriété de sûreté:

Ps en LUSTRE	$Once_From_To(CHAUD$ **or not** $Marche$ **or** $egal(TAILLE_TEMP,\ TempChoisie,\ TempCapteur)$, $sup(TAILLE_TEMP, TempChoisie, TempCapteur)$, $delay(3, sup(TAILLE_TEMP, TempChoisie, TempCapteur)))$
	$\equiv$

Expression interne à LUTESS de Ps	**if** $lv_1 \wedge ($**if** sv_0 **then** $false$ **else** sv_2 **fi**$)$ **then**
	 lv_2;
	 else
	 $true$
	 fi

Lorsque le climatiseur détecte que la température souhaitée par l'utilisateur est supérieure à la température ambiante mesurée par le capteur, la propriété de sûreté que nous avons

décrite ci-dessus s'assure que le climatiseur se trouve dans l'une des trois situations suivantes au plus tard trois instants après avoir détecté la différence de température:

- le climatiseur produit de la chaleur;
- le climatiseur n'est pas en marche;
- la température ambiante est redevenue égale à la température souhaitée par l'utilisateur.

L'expression de la propriété de sûreté obtenue après l'analyse de la description de l'environnement par LUTESS fait intervenir les variables d'état sv_0 et sv_2 et deux variables locales (lv_1 et lv_2). La définition de ces deux variables locales et les deux fonctions de transitions partielles définissant sv_0 et sv_2 font intervenir trois nouvelles variables locales (lv_3, lv_4 et lv_5) et quatre nouvelles variables d'état: sv_1, sv_3, sv_4 et sv_5. Nous donnons ci-dessous les définitions des variables locales et d'état dont nous avons besoin.

$$\underbrace{\begin{array}{rl}
lv_1 = & \textbf{if } sv_0 \textbf{ then } false \textbf{ else } sv_1 \textbf{ fi} \\
lv_2 = & \textbf{if } lv_4 \textbf{ then } lv_3 \\
& \textbf{else if } lv_1 \textbf{ then } lv_3 \vee sv_5 \\
& \textbf{else } true \\
& \textbf{fi fi} \\
lv_3 = & CHAUD \vee \neg Marche \vee lv_5 \\
lv_4 = & \textbf{if } ((TempChoisie[0] \wedge TempCapteur[0]) \vee (\neg(TempChoisie[0] \vee TempCapteur[0]))) \textbf{ then} \\
& \quad \textbf{if } ((TempChoisie[1] \wedge TempCapteur[1]) \vee (\neg(TempChoisie[1] \vee TempCapteur[1]))) \textbf{ then} \\
& \quad\quad \textbf{if } ((TempChoisie[2] \wedge TempCapteur[2]) \vee (\neg(TempChoisie[2] \vee TempCapteur[2]))) \textbf{ then} \\
& \quad\quad\quad \textbf{if } ((TempChoisie[3] \wedge TempCapteur[3]) \vee (\neg(TempChoisie[3] \vee TempCapteur[3]))) \textbf{ then} \\
& \quad\quad\quad\quad false \\
& \quad\quad\quad \textbf{else } TempCapteur[3] \textbf{ fi} \\
& \quad\quad \textbf{else } TempCapteur[2] \textbf{ fi} \\
& \quad \textbf{else } TempCapteur[1] \textbf{ fi} \\
& \textbf{else } TempCapteur[0] \textbf{ fi} \\
lv_5 = & ((TempChoisie[0] \wedge TempCapteur[0]) \vee \neg(TempChoisie[0] \vee TempCapteur[0])) \wedge \\
& ((TempChoisie[1] \wedge TempCapteur[1]) \vee \neg(TempChoisie[1] \vee TempCapteur[1])) \wedge \\
& ((TempChoisie[2] \wedge TempCapteur[2]) \vee \neg(TempChoisie[2] \vee TempCapteur[2])) \wedge \\
& ((TempChoisie[3] \wedge TempCapteur[3]) \vee \neg(TempChoisie[3] \vee TempCapteur[3])) \wedge \\
& ((TempChoisie[4] \wedge TempCapteur[4]) \vee \neg(TempChoisie[4] \vee TempCapteur[4]))
\end{array}}_{Variables\ locales}$$

★ **Fonction de transition 5.1** : *fonctions de transition de l'environnement du climatiseur.*

$$\begin{array}{rl}
sv'_0 = & sv_0 \wedge false \\
sv'_1 = & lv_4 \vee lv_1 \\
sv'_2 = & \textbf{if } sv_0 \textbf{ then } false \textbf{ else } sv_3 \textbf{ fi} \\
sv'_3 = & \textbf{if } sv_0 \textbf{ then } false \textbf{ else } sv_4 \textbf{ fi} \\
sv'_4 = & lv_4 \\
sv'_5 = & lv_2 \\
sv'_6 = & ...
\end{array}$$

- sv_0 prend pour valeur *vrai* dans l'état initial, puis prend la valeur *faux* pour tous les autres états. Cette variable d'état permet à LUTESS de marquer l'état initial (elle est équivalente à la variable *init* de l'automate de contrôle LUSTRE: voir §3.3).

- sv_1 est égale à la disjonction des variables locales lv_1 et lv_4: lv_1 est égale à la valeur courante de sv_1 (sauf dans l'état initial où $lv_1 = faux$; lv_4 représente le test **sup**(TempChoisie, TempCapteur)[8]. Étant donné les définitions des variables locales lv_1 et lv_4, nous pouvons dire que sv_1 prend pour valeur *vrai* quand la température choisie par l'utilisateur est (ou a été) supérieure à la température ambiante.

- sv_2 représente la valeur précédente de sv_3, sv_3 représente la valeur précédente de sv_4 et sv_4 représente la valeur précédente de lv_4. sv_2 représente l'expression **delay**(3, **sup**(TempChoisie, TempCapteur)) par l'intermédiaire des variables d'état sv_3 et sv_4.

- sv_5 représente la valeur précédente de lv_2. lv_2 prend la valeur *faux* lorsque le contrôleur ne demande pas (ou n'a pas demandé) de produire de chaleur, ou que la température ambiante n'est (ou n'a pas été) égale à celle choisie par l'utilisateur[9], ou encore que le climatiseur n'est (ou n'a pas été) arrêté alors que la température choisie par l'utilisateur est (ou a été) supérieure à la température ambiante

Afin de simplifier les expressions et la compréhension des fonctions de transitions partielles et des définitions des variables locales, nous remplacerons dans la suite de ce document l'expression brute des variables $lv4$ et lv_5 issues de l'analyse du nœud de test par leurs significations:

- $lv_4 = sup(TempChoisie, TempCapteur)$;

- $lv_5 = egale(TempChoisie, TempCapteur)$.

5.10.4 Calcul des états accessibles et des vecteurs d'entrées pertinents

Plaçons nous dans l'état initial (instant 0), c'est à dire $sv_0 = true$, toutes les autres variables d'état ayant la valeur fausse. L'ensemble des états accessibles à l'issue d'un chemin de longueur 1 et l'ensemble des vecteurs d'entrées pertinents sont obtenus comme ci-dessous: nous commençons par calculer la valeur des variables locales à l'instant courant, ce qui nous permet de calculer la valeur des variables d'état à l'instant 1.

Remarque 5.6 *La notation* $nomVariable^n$ *signifie que nous considérons la valeur de la variable* $nomVariable$ *à l'issue d'un chemin de longueur* n*.*

[8] La température est représentée par un vecteur de bit en utilisant un codage 1 parmi n.

[9] L'égalité entre la température choisie par l'utilisateur et la température ambiante est déterminée par la valeur de la variable locale lv_5.

$$
\begin{aligned}
lv_1^0 &= \textbf{if } sv_0^0 \textbf{ then } false \textbf{ else } sv_1^0 \textbf{ fi} \\
&= \textbf{if } true \textbf{ then } false \textbf{ else } \ldots \textbf{ fi} \\
&= false \\
lv_2^0 &= \textbf{if } lv_4^0 \textbf{ then } lv_3^0 \\
&\quad\ \textbf{else if } lv_1^0 \textbf{ then } lv_3^0 \vee sv_5^0 \\
&\quad\ \textbf{else } true \\
&\quad\ \textbf{fi fi} \\
&= \textbf{if } sup(TempChoisie^0,\ TempCapteur^0) \textbf{ then} \\
&\qquad\quad egale(TempChoisie^0,\ TempCapteur^0) \\
&\quad\ \textbf{else} \\
&\qquad\quad true \\
&\quad\ \textbf{fi} \\
lv_3^0 &= CHAUD^0 \vee \neg Marche^0 \vee lv_5^0 \\
&= CHAUD^0 \vee \neg Marche^0 \vee egale(TempChoisie^0,\ TempCapteur^0) \\
lv_4^0 &= sup(TempChoisie^0,\ TempCapteur^0) \\
lv_5^0 &= egale(TempChoisie^0,\ TempCapteur^0)
\end{aligned}
$$

Variables locales à l'état initial.

$$
\begin{aligned}
sv_0^1 &= sv_0^0 \wedge false \\
&= false \\
sv_1^1 &= lv_4^0 \vee lv_1^0 \\
&= sup(TempChoisie^0,\ TempCapteur^0) \\
sv_2^1 &= \textbf{if } sv_0^0 \textbf{ then } false \textbf{ else } sv_3^0 \textbf{ fi} \\
&= false \\
sv_3^1 &= \textbf{if } sv_0^0 \textbf{ then } false \textbf{ else } sv_4^0 \textbf{ fi} \\
&= false \\
sv_4^1 &= lv_4^0 \\
&= sup(TempChoisie^0,\ TempCapteur^0) \\
sv_5^1 &= lv_2^0 \\
&= \textbf{if } sup(TempChoisie^0,\ TempCapteur^0) \textbf{ then} \\
&\qquad\quad CHAUD^0 \vee \neg Marche^0 \vee egale(TempChoisie^0,\ TempCapteur^0) \\
&\quad\ \textbf{else} \\
&\qquad\quad true \\
&\quad\ \textbf{fi} \\
sv_6^1 &= \ldots
\end{aligned}
$$

États accessibles par un chemin de longueur 1.

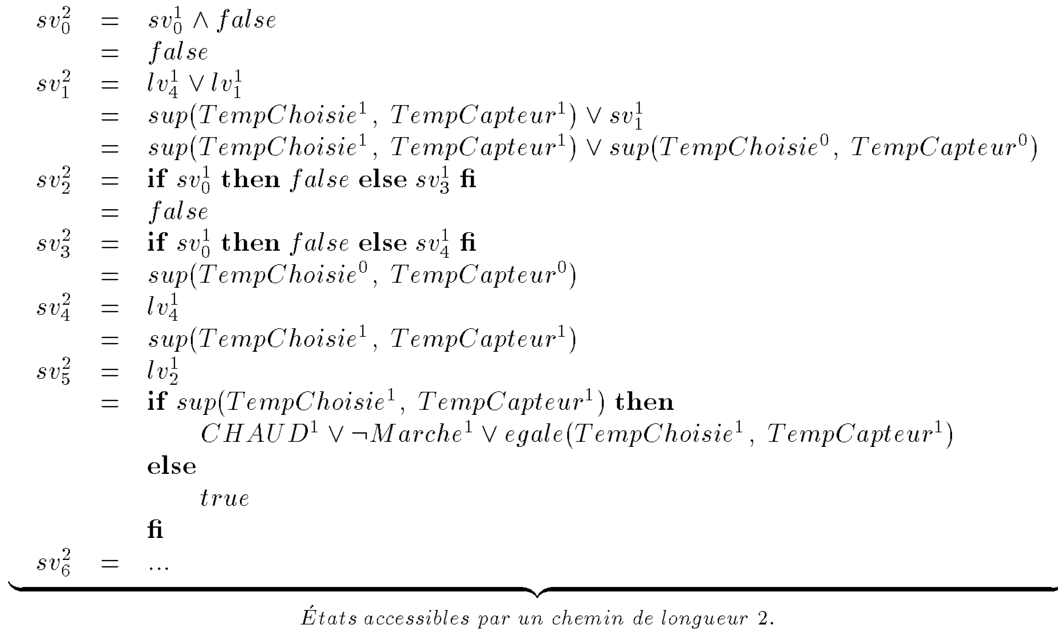
Nous substituons aux variables locales entrant dans l'expression de la propriété de sûreté leurs valeurs à l'instant 1. Nous obtenons alors une expression booléenne toujours égale à la valeur fausse. Ceci signifie qu'il n'existe pas de vecteurs d'entrées pouvant mener vers un état suspect au bout d'un chemin de longueur 1 à partir de l'état initial.

$$
\begin{aligned}
&\neg(\textbf{if} sup(TempChoisie^0,\ TempCampteur^0) \wedge false \textbf{ then} \\
&\qquad\quad \ldots \\
&\quad\ \textbf{else} \\
&\qquad\quad true \\
&\quad\ \textbf{fi})
\end{aligned}
$$

Vecteurs d'entrées pertinents pouvant mener à un état suspect par un chemin de longueur 1.

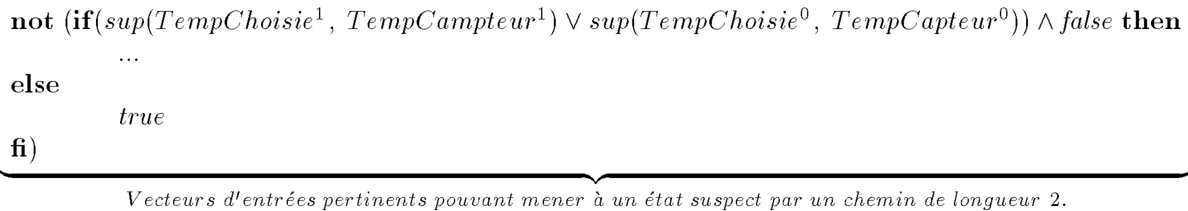
De manière similaire nous pouvons calculer l'ensemble des états accessibles pour des chemins de longueur 2 ainsi que les ensembles de vecteurs d'entrées pertinents V_{I_2} (nous ne

détaillerons pas le calcul intermédiaire des variables locales):



$$
\begin{array}{rcl}
sv_0^2 & = & sv_0^1 \wedge false \\
 & = & false \\
sv_1^2 & = & lv_4^1 \vee lv_1^1 \\
 & = & sup(TempChoisie^1, TempCapteur^1) \vee sv_1^1 \\
 & = & sup(TempChoisie^1, TempCapteur^1) \vee sup(TempChoisie^0, TempCapteur^0) \\
sv_2^2 & = & \textbf{if } sv_0^1 \textbf{ then } false \textbf{ else } sv_3^1 \textbf{ fi} \\
 & = & false \\
sv_3^2 & = & \textbf{if } sv_0^1 \textbf{ then } false \textbf{ else } sv_4^1 \textbf{ fi} \\
 & = & sup(TempChoisie^0, TempCapteur^0) \\
sv_4^2 & = & lv_4^1 \\
 & = & sup(TempChoisie^1, TempCapteur^1) \\
sv_5^2 & = & lv_2^1 \\
 & = & \textbf{if } sup(TempChoisie^1, TempCapteur^1) \textbf{ then} \\
 & & \quad CHAUD^1 \vee \neg Marche^1 \vee egale(TempChoisie^1, TempCapteur^1) \\
 & & \textbf{else} \\
 & & \quad true \\
 & & \textbf{fi} \\
sv_6^2 & = & \ldots
\end{array}
$$

États accessibles par un chemin de longueur 2.



$$
\begin{array}{l}
\textbf{not } (\textbf{if}(sup(TempChoisie^1, TempCampteur^1) \vee sup(TempChoisie^0, TempCapteur^0)) \wedge false \textbf{ then} \\
\quad \ldots \\
\textbf{else} \\
\quad true \\
\textbf{fi})
\end{array}
$$

Vecteurs d'entrées pertinents pouvant mener à un état suspect par un chemin de longueur 2.

De la même façon que pour les chemins de longueur 1, il n'existe pas de vecteur d'entrées pertinent qui permette de mener vers un état suspect par un chemin de longueur 2. Cette situation correspond bien à notre propriété de sûreté étant donné que cette dernière à pour but de s'assurer que le climatiseur a réagi en au plus trois instants: nous ne pouvons donc pas violer la propriété de sûreté en moins de trois instants.

Enfin, nous allons déterminer quels sont les vecteurs d'entrées qui peuvent mener à un état suspect par des chemins de longueur 3.

$$
\underbrace{
\begin{array}{lll}
sv_0^3 & = & sv_0^2 \wedge false \\
 & = & false \\
sv_1^3 & = & lv_4^2 \vee lv_1^2 \\
 & = & sup(TempChoisie^2,\ TempCapteur^2) \vee sup(TempChoisie^1,\ TempCapteur^1) \vee \\
 & & sup(TempChoisie^0,\ TempCapteur^0) \\
sv_2^3 & = & \textbf{if}\ sv_0^0\ \textbf{then}\ false\ \textbf{else}\ sv_3^0\ \textbf{fi} \\
 & = & sup(TempChoisie^0,\ TempCapteur^0) \\
sv_3^3 & = & \textbf{if}\ sv_0^0\ \textbf{then}\ false\ \textbf{else}\ sv_4^0\ \textbf{fi} \\
 & = & sup(TempChoisie^1,\ TempCapteur^1) \\
sv_4^3 & = & lv_4^0 \\
 & = & sup(TempChoisie^2,\ TempCapteur^2) \\
sv_5^3 & = & lv_2^0 \\
 & = & \textbf{if}\ sup(TempChoisie^2,\ TempCapteur^2)\ \textbf{then} \\
 & & \quad CHAUD^2 \vee \neg Marche^2 \vee egale(TempChoisie^2,\ TempCapteur^2) \\
 & & \textbf{else} \\
 & & \quad \textbf{if}\ sv_1^2\ \textbf{then} \\
 & & \quad\quad CHAUD^2 \vee \neg Marche^2 \vee egale(TempChoisie^2,\ TempCapteur^2) \vee sv_5^2 \\
 & & \quad \textbf{else} \\
 & & \quad\quad true \\
 & & \quad \textbf{fi} \\
 & & \textbf{fi} \\
 & = & \textbf{if}\ sup(TempChoisie^2,\ TempCapteur^2)\ \textbf{then} \\
 & & \quad CHAUD^2 \vee \neg Marche^2 \vee egale(TempChoisie^2,\ TempCapteur^2) \\
 & & \textbf{else} \\
 & & \quad \textbf{if}\ sup(TempChoisie^1,\ TempCapteur^1)\ \textbf{then} \\
 & & \quad\quad CHAUD^2 \vee \neg Marche^2 \vee egale(TempChoisie^2,\ TempCapteur^2) \vee \\
 & & \quad\quad \textbf{if}\ sup(TempChoisie^1,\ TempCapteur^1)\ \textbf{then} \\
 & & \quad\quad\quad CHAUD^1 \vee \neg Marche^1 \vee egale(TempChoisie^1,\ TempCapteur^1) \\
 & & \quad\quad \textbf{else} \\
 & & \quad\quad\quad true \\
 & & \quad\quad \textbf{fi} \\
 & & \quad \textbf{else} \\
 & & \quad\quad true \\
 & & \quad \textbf{fi} \\
 & & \textbf{fi} \\
sv_6^3 & = & \ldots
\end{array}
}_{\textit{États accessibles par un chemin de longueur 3.}}
$$

$$
\underbrace{
\begin{array}{l}
\textbf{not}\ (\textbf{if}(sup(TempChoisie^2,\ TempCampteur^2) \vee sup(TempChoisie^1,\ TempCampteur^1) \vee \\
\quad sup(TempChoisie^0,\ TempCampteur^0)) \wedge sup(TempChoisie^0,\ TempCampteur^0)\ \textbf{then} \\
\quad\quad \textbf{if} sup(TempChoisie^3, TempCapteur^3) \textbf{then} \\
\quad\quad\quad\quad egale(TempChoisie^3, TempCapteur^3) \\
\quad\quad \textbf{else} \\
\quad\quad\quad\quad true \\
\quad\quad \textbf{fi} \\
\textbf{else} \\
\quad\quad true \\
\textbf{fi})
\end{array}
}_{\textit{Vecteurs d'entrées pertinents pouvant mener à un état suspect par un chemin de longueur 3.}}
$$

Pour pouvoir atteindre un état suspect par un chemin de longueur 3, nous devons générer à l'instant courant une température choisie par l'utilisateur supérieure à la température ambiante. Cette contrainte se trouve dans la condition du premier "**if**" de l'expression représentant les vecteurs d'entrées pertinents:

$$\ldots\textbf{if}\ \ldots \wedge\ sup(TempChoisie^0,\ TempCapteur^0)\ \textbf{then}\ldots$$

Si nous ne générons pas $sup(TempChoisie^0,\ TempCapteur^0)$ alors la propriété sera toujours évaluée à *vrai* (au moins pour les trois instants qui suivent). Tous les autres éléments entrants dans l'expression des vecteurs d'entrées pertinents pouvant mener à un état suspect par un chemin de longueur 3 font référence à des valeurs qui devront être générées aux instant $t+1$, $t+2$ et $t+3$ ($t+0$ étant l'instant courant): nous ne pouvons donc pas agir dessus à l'instant courant et nous supposons que nous pourrons les choisir de telle sorte que nous puissions atteindre un état suspect.

Chapitre 6

Évaluation des stratégies proposées

Dans ce chapitre, nous évaluons l'efficacité relative des stratégies que nous avons proposées au chapitre 5. Nous rappelons que le but de ces stratégies est de construire l'ensemble V_{val} des vecteurs d'entrées pertinents pouvant mener à l'observation d'une violation des propriétés de sûreté. Ce chapitre est décomposé en six paragraphes. Nous définissons dans le premier paragraphe (§6.1) une mesure relative de l'efficacité des stratégies. Le paragraphe 6.2 présente la modélisation des services téléphoniques qui servent de support à notre expérimentation. Dans la partie 6.3, nous donnons une description détaillée des propriétés de sûreté que nous utilisons. Le paragraphe 6.4 correspond à une évaluation des capacités de calcul disponibles pour la construction des simulateurs d'environnement; son suivant détaille les résultats obtenus en termes de nombre d'états suspects atteints. Enfin, la partie 6.6 dresse un bilan de l'expérimentation.

6.1 Mesure de l'efficacité des séquences de test

L'efficacité du test peut être mesurée de manière directe en considérant le nombre d'erreurs révélées par ce dernier ou de manière indirecte par la satisfaction d'un critère (dit de génération) dont on sait qu'il est relié à la révélation d'erreurs. La mesure de l'efficacité du test doit répondre au moins à l'un des deux objectifs ci-dessous:

1. évaluer des séquences de test qui résultent d'une campagne de test. Dans ce cas, il s'agit d'une mesure de l'efficacité "*absolue*", c'est à dire s'assurer que les séquences de test ont atteint le critère préalablement fixé.
2. comparer différentes techniques de sélection des données de test entre elles. Ici, la mesure de l'efficacité est *relative*; en d'autres termes, la mesure doit permettre d'évaluer chacune des techniques les unes par rapport aux autres en fonction d'un objectif fixé.

Souvent, les critères de sélection des données de test servent également de critères d'adéquation [Wey86], c'est à dire qu'ils permettent de mesurer si le processus de test a été correctement conduit; en particulier, un critère d'adéquation peut être assimilé à un critère d'arrêt du test. De tels critères donnent une mesure "*absolue*" du test et sont bâtis soit à partir de modèles de fautes sur le code ou les spécifications du logiciel, soit à partir d'une notion de couverture

(code ou spécification) avec l'hypothèse sous-jacente que si les tests engendrés assurent la couverture choisie alors la qualité requise du test est atteinte.

Une approche classique pour évaluer l'efficacité *relative* entre différents critères de sélection est de procéder par analyse mutationnelle. Cette technique consiste à rejouer les séquences de test sur un ensemble de programmes très légèrement modifiés les uns par rapport aux autres appelés *mutants*: plus le nombre de mutants détectés par le test (programmes dont le comportement "dérivant" est révélé) est grand, plus il sera considéré comme efficace. Les travaux présentés dans [FI98] utilisent ce principe pour comparer l'efficacité relative de deux critères de sélection.

LUTESS considère le logiciel comme une boîte noire, c'est à dire que nous ne disposons d'aucun modèle de fautes du logiciel, nous ne faisons aucune hypothèse sur les comportements de ce dernier et nous ne disposons pas de son code. Dans ce contexte, nous ne pouvons pas utiliser l'analyse mutationnelle pour évaluer l'efficacité de nos stratégies. D'autre part, l'objectif de nos travaux est de valider les comportements du logiciel dans des situations critiques. Une manière directe de mesurer l'efficacité de nos techniques serait de compter le nombre de violations des propriétés de sûreté. Une telle mesure a le défaut d'être dépendante de l'implantation du logiciel; en particulier, dans le cas où aucune violation n'est détectée, il n'est pas possible de savoir si ce résultat est dû à de mauvais cas de test ou à une implantation correcte vis-à-vis de la propriété testée. Malgré tout, le testeur a besoin d'informations lui permettant:

- de savoir quand s'arrêter;
- d'évaluer la qualité des séquences de test engendrées par LUTESS.

Nous proposons d'utiliser le nombre d'états suspects (cf. §5.3.5) atteints par les différentes stratégies pour les comparer entre elles. Ces états caractérisent les situations critiques et lorsqu'une telle situation est atteinte, les propriétés de sûreté ne peuvent être violées que si le logiciel ne réagit pas correctement: être capable de tester ces situations dangereuses nous semble être plus caractéristique des performances de nos stratégies car cette mesure est indépendante des choix d'implantation du système sous test. À partir de la notion d'état suspect, nous avons défini trois mesures:

1. le *nombre total d'états suspects atteints* exprime le nombre de fois où le logiciel a été confronté à une situation critique. Cette mesure indique le nombre de fois où la valeur des propriétés de sûreté ne dépendait que de la valeur du vecteur de sorties du logiciel;
2. le *nombre total d'états suspects distincts atteints* permet d'affiner l'information apporter par le *nombre total d'états suspects atteints*. Cette mesure exprime le nombre de situations critiques différentes auxquelles le logiciel a été confronté. Il est possible de compléter cette mesure en précisant la proportion de chacune d'entre elles par rapport au nombre total de situations critiques atteintes;
3. le *nombre d'états suspects atteints par une stratégie A et non atteint par une stratégie B* permet de comparer les situations critiques atteintes par A et pas par B. Dans ce cas, nous cherchons à comparer les stratégies entre elles.

Par analogie avec le test structurel où la confiance dans le logiciel augmente avec le (ou les) taux de couverture retenu(s), notre confiance dans le logiciel augmente avec le nombre de

situations critiques atteintes. Cependant, nous ne pouvons pas être sûrs d'avoir testé toutes les situations critiques; au mieux nous pouvons estimer le nombre maximum d'états suspects (et donc de situations critiques) sans avoir la garantie qu'ils soient tous accessibles.

6.2 Modélisation des services téléphoniques

6.2.1 Généralités

L'étude de cas qui sert de support à notre évaluation est extraite du concours pour la détection d'interaction de services FIDC1[1] [GBGT98] qui a eu lieu en marge de la conférence FIW'98[2]. Au cours de ce concours, LUTESS a remporté le premier prix [dBZ98]. Le choix de cet exemple a été dicté par deux raisons:

- d'une part la base des services téléphoniques développés dans le cadre du concours nous a permis d'obtenir un grand nombre d'exemples avec un ensemble de propriétés que les services doivent respecter;
- d'autre part, j'avais une connaissance assez importante de ces exemples pour avoir participé en tant qu'étudiant de magistère aux campagnes de test des services téléphoniques lors du concours FIDC1.

L'interaction de services consiste à détecter les erreurs qui peuvent survenir lorsque plusieurs services (téléphoniques dans notre cas) fonctionnent ensemble alors que leur fonctionnement indépendant ne provoque pas d'erreur. Prenons par exemple le cas d'un service permettant d'afficher le numéro de l'appelant sur le combiné de l'appelé; supposons également que l'appelant soit abonné au service de liste rouge, c'est à dire qu'il ne souhaite pas communiquer son numéro de téléphone aux autres utilisateurs. Lorsque ces deux services fonctionnent ensemble, une interaction se produit si le service d'affichage du numéro révèle le numéro de l'abonné à la liste rouge.

Lors du concours FIDC1, les interactions avaient été découvertes à l'aide de schémas comportementaux. Ces schémas décrivaient les comportements attendus des abonnés (environnement) pour observer un fonctionnement parallèle des services considérés et éventuellement détecter une interaction. Ici, nous souhaitons utiliser les propriétés de sûreté construites à partir des éléments suivants:

1. une description de *situations suspectes*, c'est à dire des situations où les deux services sont activés en même temps;
2. une description des *comportements* attendus du système.

En général, nous pouvons écrire les propriétés de sûreté sous la forme:

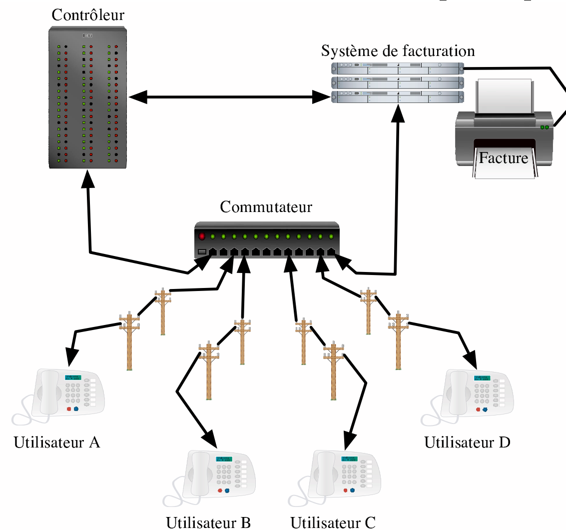
$$\textit{situations suspectes} \Rightarrow \textit{comportements}.$$

La figure 6.1 représente le modèle de notre système de services téléphoniques. Ce système est composé:

- des téléphones des abonnés; dans notre évaluation, nous considérons un réseau de quatre abonnés nommés A, B, C, D;

[1] 1*st* Feature Interaction Detection Contest.
[2] FIW: Feature Interaction Workshop.

FIG. 6.1: Modèle du réseau téléphonique.



- d'un système de facturation chargé de gérer la facturation des communications et des services téléphoniques utilisés par chacun des abonnés;
- d'un contrôleur permettant la mise en œuvre de l'ensemble des services téléphoniques;
- d'un commutateur aiguillant les communications entre les trois autres éléments du système.

Les modèles synchrones (ou partiellement synchrones) de ce système et les modes de composition - intégration des services sont largement décrits par ailleurs ([DB99, Zua00b]). Ils ne sont pas repris ici.

Nous disposions de douze spécifications de services décrites par les organisateurs du concours. Les services ont été testés deux à deux ce qui nous a conduit à implanter 78 paires de services[3]. Parmi ces dernières, j'ai sélectionné un sous-ensemble de 6 paires de services; chaque paire constitue un système indépendant.

Chaque système que nous allons utiliser pour l'évaluation des stratégies proposées au chapitre 5 est constitué du service de base (POTS[4]) et de deux autres services complémentaires[5]. Le service de base permet d'assurer les communications "classiques", à savoir établir une communication entre deux abonnés et gérer la facturation des communications passées par chacun des abonnés, alors que les deux services complémentaires permettent d'offrir des fonctionnalités optionnelles comme le transfert d'appel ou la conversation à trois. Nous donnons ci-dessous une brève description des services qui fournissent la matière de cette étude:

- CFBL: *Call Forward on Busy Line* permet à un abonné ayant souscrit au service de transférer ses appels vers un autre abonné du réseau lorsque sa ligne est occupée.
- CELL: *Cellular* est un service de téléphone cellulaire.

[3]Dans le cadre du concours FIDC1.

[4]POTS: Plain Old Telephon System.

[5]Notons que les deux services complémentaires peuvent être identiques si l'on souhaite chercher les interactions d'un service avec lui-même.

- CW: *Call Waiting* permet à un abonné ayant souscrit à ce service d'être averti lorsqu'un nouvel appel arrive alors qu'il est déjà en communication. Ce service permet également de mettre en attente le premier appel pour prendre le second. Notons que la ligne sera occupée si un troisième appel survient.
- CND: *Call Number Delivery* permet à un abonné ayant souscrit à ce service de faire afficher le numéro de l'appelant sur le téléphone.
- TCS: *Terminal Call Screening* permet à un abonné ayant souscrit à ce service de définir une liste noire d'abonnés dont il ne veut pas recevoir de communication.
- TWC: *Three Way Calling* est un service qui permet de tenir une conversation téléphonique à trois.

6.2.2 Interface entre l'environnement et le système

L'environnement de notre système téléphonique doit modéliser le comportement des quatre abonnés à partir des actions décrites ci-dessous:

- $OffHook(A)$: l'abonné A a décroché son téléphone.
- $OnHook(A)$: l'abonné A a raccroché son téléphone.
- $Dial(A, B)$: l'abonné A compose le numéro de l'abonné B.

Symétriquement le système téléphonique peut émettre des commandes vers les téléphones des abonnés afin, par exemple, d'établir une communication:

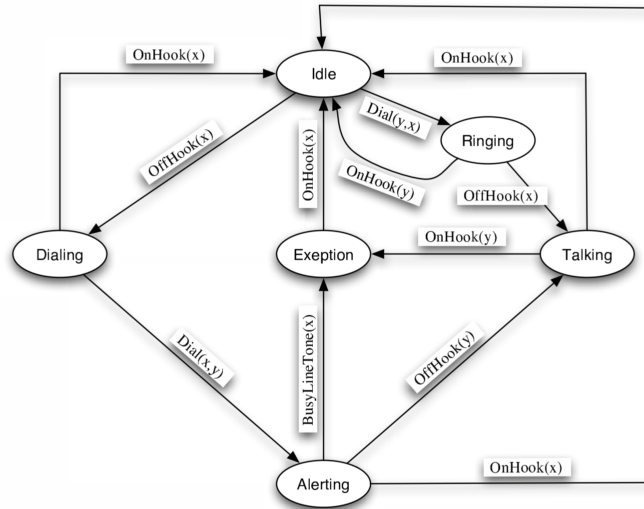
- $DialTone(A)$: l'abonné A reçoit un signal sonore indiquant que le téléphone de son correspondant est en train de sonner.
- $Ringing(A, B)$: le téléphone de l'abonné A sonne pour un appel en provenance de l'abonné B.
- $BusyLineTone(A)$: l'abonné A reçoit un signal sonore indiquant que le correspondant qu'il cherche à joindre est occupé.
- $Display(A, M)$: un message M est affiché sur le téléphone de l'abonné A.

Notre système téléphonique est considéré comme suffisamment rapide pour qu'on admette qu'à chaque instant, il ne puisse se produire qu'au plus un événement. Par exemple, si un abonné décroche son téléphone à l'instant t, il composera le numéro de son correspondant au mieux à l'instant $t + 1$.

Chaque téléphone peut se trouver dans un des six états *idle*, *dialing*, *alerting*, *ringing*, *talking*, *exception*.

- idle: le téléphone est raccroché; l'abonné peut recevoir une communication ou initier une communication en décrochant.
- dialing: l'abonné a décroché son téléphone; le système attend qu'il compose le numéro de son correspondant.
- alerting: l'abonné a composé un numéro et le téléphone de son correspondant sonne.
- ringing: cet état indique que le téléphone de l'abonné est en train de sonner.
- talking: une communication a été établie entre deux abonnés.

FIG. 6.2: Digramme de transition pour un téléphone



– exception: la communication avec l'appelé ne peut pas être établie ou elle a été interrompue par l'abonné qui était en communication.

La figure 6.2 représente la fonction de transition entre les états du téléphone de l'abonné x dans le cas d'une communication classique entre les abonnés x et y.

6.2.3 Prédicats

Nous rappelons que LUTESS ne peut générer que des données booléennes: l'ensemble des communications entre le système sous test et l'environnement est réalisé par des signaux. Pour faciliter le codage de l'environnement et l'interprétation des vecteurs booléens, nous utilisons l'ensemble des prédicats suivants:

- $OffHook(x)$: *vrai* si le téléphone de l'abonnée x est décroché.
- $OnHook(x)$: *vrai* si le téléphone de l'abonnée x est raccroché.
- $Dial(x,y)$: *vrai* si l'abonné x a composé le numéro de l'abonné y.
- $BusyLineTone(x)$: *vrai* si la ligne du correspondant de l'abonné x est occupée.
- $Alerting(x)$: *vrai* si le téléphone de l'abonné x se trouve dans l'état **alerting**.
- $Talking(x)$: *vrai* si le téléphone de l'abonné x se trouve dans l'état **talking**.

Nous n'avons décrit dans ce paragraphe que les principaux prédicats nécessaires pour notre étude. D'autres prédicats pourront être introduits dans un contexte particulier au fur et à mesure de la description des propriétés testées.

6.3 Propriétés de sûreté utilisées

Dans le cadre de cette évaluation, nous avons utilisé à la fois des propriétés spécifiques à chacun des services considérés et des propriétés plus générales qui peuvent s'appliquer à tous les systèmes téléphoniques construits pour le concours FIDC1.

6.3.1 Propriétés générales

Pour chacun des systèmes testés, les deux propriétés générales ci-dessous ont servi de base à la validation de l'évolution de l'état interne d'un poste téléphonique.

$$\begin{aligned}(1) \;:\; & always_from_to(Talking(A),\\ & \quad \mathbf{pre}\ \mathbf{pre}\ OffHook(A) \wedge \mathbf{pre}\ Dial(A,B) \wedge OffHook(B),\\ & \quad OnHook(B) \vee OnHook(A))\\ (2) \;:\; & always_from_to(Alerting(A),\\ & \quad \mathbf{pre}\ OffHook(A) \wedge Dial(A,B),\\ & \quad OffHook(B) \vee OnHook(A))\end{aligned}$$

Ces deux propriétés ne permettent pas de valider l'ensemble des comportements décrits par la figure 6.2. La propriété (1) s'assure que le téléphone de A se trouve dans l'état **talking** lorsqu'une communication est établie entres les abonnés A et B sur l'initiative de A. La propriété (2) s'assure que le téléphone de A se trouve dans l'état **alerting** après que A ait composé le numéro de B.

6.3.2 Propriétés spécifiques

Chaque système que nous avons considéré est doté de propriétés spécifiques.

CFBL - CFBL

Dans le système construit à partir du service de base (POTS) et du transfert d'appel lorsque la ligne est occupée (CFBL), nous cherchons à mettre en évidence les interactions de CFBL avec lui-même. Dans ce cas, nous considérons que tous les utilisateurs sont abonnés au service CFBL et que A transfère vers B, B vers C, C vers D et enfin D vers A. Plus particulièrement, nous allons nous intéresser à des propriétés de sûreté qui visent à prévenir les boucles infinies qui peuvent être créées à la suite de plusieurs transferts d'appel en cascade.

Dans un souci de simplicité, nous demandons à notre système de ne pas autoriser plus d'un transfert pour éviter le problème des boucles infinies. La propriété $(CFBL - CFBL\,1)$ décrit une situation précise où le système doit stopper les transferts d'appel:

- l'abonné A commence par décrocher son téléphone;
- l'abonné B décroche à son tour son téléphone;
- dans une troisième étape, l'abonné C décroche lui aussi son téléphone;
- enfin, l'abonné C appelle l'abonné A.

Lorsque C appelle A, A a déjà décroché son téléphone et l'appel est transféré vers B qui lui aussi a déjà décroché son téléphone. Dans ce cas, les transferts d'appel doivent cesser et l'abonné C doit recevoir un signal lui indiquant que la ligne est occupée. Notons que l'abonné A ne peut pas raccrocher pendant les trois événements qui suivent $OffHook(A)$ puisqu'il peut se produire au plus un événement par instant.

(CFBL - CFBL 1)

$$\begin{array}{r}
((\mathbf{pre\ pre\ pre}\ OffHook(A))\wedge \\
(\mathbf{pre\ pre}\ (OffHook(B)))\wedge \\
(\mathbf{pre}\ (OffHook(C)))\wedge \\
Dial(C,A)) \quad \Rightarrow \quad BusyLineTone(St_C)
\end{array}$$

La propriété $(CFBL - CFBL\,1)$ présente deux défauts dans le cas des services téléphoniques: elle impose:

1. une succession d'événements précis avant que l'on puisse atteindre la situation suspecte;
2. un ordre des appels des abonnés.

L'idée des deux propriétés suivantes est de permettre plus de liberté dans les enchaînements d'appels et d'événements, au prix d'une complexité croissante du simulateur d'environnement.

La propriété $(CFBL - CFBL\ 2)$ donne plus de liberté sur la séquence d'événements à produire tout en continuant d'imposer les événements à un abonné précis. Elle impose que nous soyons entre l'événement $OffHook(A)$ et $OnHook(A)$ c'est à dire que l'abonné A a son téléphone décroché; nous imposons également la même contrainte à l'abonné B. Ensuite, nous voulons que l'abonné C appelle l'abonné A (entre les événements $Dial(C,A)$ et $OnHook(C)$). Enfin, l'opérateur $Once_Since$ nous permet d'ordonner une partie des événements: A et B doivent avoir décroché avant que C appelle A.

(CFBL - CFBL 2)

$$\begin{array}{r}
(Between(OffHook(A), OnHook(A))\wedge \\
Between(OffHook(B), OnHook(B))\wedge \\
Between(Dial(C,A), OnHook(C))\wedge \\
Once_Since(Between(OffHook(A), OnHook(A))\wedge \\
Between(OffHook(B), OnHook(B)), \\
Dial(C,A))) \quad \Rightarrow \quad BusyLineTone(st_C)
\end{array}$$

La propriété $(CFBL - CFBL\ 3)$ généralise la propriété $(CFBL - CFBL\ 2)$ en ne précisant pas *a priori* qui fait quoi. En d'autres termes l'instanciation des abonnés que nous avions dans $(CFBL - CFBL\,2)$ est laissé à LUTESS. Pour écrire cette propriété nous devons introduire un nouveau prédicat qui permettent de déterminer les conditions d'abonnement au service de transfert d'appel:

- $Transfert(x,y)$ est vrai si l'abonné x transfert ses appels vers l'abonné y.

(CFBL - CFBL 3)

$$\begin{array}{r}
(Transfert(x,y)\wedge \\
Transfert(z,x)\wedge \\
Between(OffHook(x), OnHook(x))\wedge \\
Between(OffHook(y), OnHook(y))\wedge \\
Between(Dial(z,x), OnHook(z))\wedge \\
Once_Since(Between(OffHook(x), OnHook(x))\wedge \\
Between(OffHook(y), OnHook(y)), \\
Dial(z,x))) \quad \Rightarrow \quad BusyLineTone(st_z)
\end{array}$$

CFBL - CELL

On ajoute au service de base le transfert d'appel (CFBL) et un service de téléphone cellulaire (CELL). Lors d'un transfert d'appel, le coût supplémentaire de l'appel entre le poste de l'appelé et le poste vers lequel l'appel est transféré est à la charge de l'abonné qui fait transférer ses communications. Dans le cas d'un transfert vers un téléphone cellulaire, le service de transfert d'appel doit facturer une communication "air" au lieu d'une communication normale. Les propriétés énoncées ci-dessous visent à vérifier le bon fonctionnement de cette facturation.

Tous les abonnés du réseau téléphonique ont souscrit au transfert d'appel (CFBL): A transfère ses appels vers C, B transfère vers A, C vers B et enfin D vers A. En revanche, seuls les abonnés A et C ont souscrit au téléphone cellulaire (CELL).

Comme dans le cas du système construit à partir des services CFBL - CFBL, nous allons écrire trois propriétés permettant de vérifier la facturation "air". La première ($CFBL - CELL\ 1$) décrit précisément la séquence d'événements attendue pour observer une facturation "air" dans le cadre d'un transfert d'appel: l'abonné A décroche, puis l'abonné B décroche à son tour. B appelle A; or A est occupé, l'appel est donc transféré vers C qui répond. Nous devons observer une facturation "air" au nom de l'abonné A pour la communication de A vers C.

(CFBL - CELL 1)

$$
\begin{aligned}
(\textbf{pre pre pre } & OffHook(A) \wedge \\
\textbf{pre pre } & OffHook(B) \wedge \\
\textbf{pre } & Dial(B, A) \wedge \\
& OffHook(C)) \quad \Rightarrow \quad AirBegin(A))
\end{aligned}
$$

Remarque 6.1 *Nous avons introduit un nouveau prédicat $AirBegin(x)$ qui prend la valeur vraie lorsque la facturation "air" est déclenchée pour l'abonné x.*

Nous ajoutons également deux propriétés qui généralisent graduellement ($CFBL - CELL\ 1$). Nous commençons, avec ($CFBL - CELL\ 2$) par donner plus de liberté dans l'enchaînement des événements décrits dans ($CFBL - CELL\ 1$), alors que ($CFBL - CELL\ 3$) ne définit pas précisément qui appelle qui:

(CFBL - CELL 2)

$$
\begin{aligned}
& (Between(OffHook(A), OnHook(A)) \wedge \\
& Between(OffHook(B), OnHook(B)) \wedge \\
& Between(Dial(B, A), OnHook(B)) \wedge \\
& OffHook(C) \wedge \\
& Once_Since(Between(OffHook(A), OnHook(A)) \wedge \\
& \textbf{not } Between(OffHook(C), OnHook(C)) \wedge \\
& Between(OffHook(B), OnHook(B)), \\
& Dial(B, A)) \wedge \\
& Once_Since(Between(OffHook(A), OnHook(A)) \wedge \\
& Between(OffHook(B), OnHook(B)) \wedge \\
& Dial(B, A), \\
& OffHook(C))) \quad \Rightarrow \quad AirBegin(A)
\end{aligned}
$$

(CFBL - CELL 3)

$$
\begin{array}{r}
(Cell(x) \wedge Cell(z) \wedge \\
Transfert(x,z) \wedge \\
Between(OffHook(x), OnHook(x)) \wedge \\
Between(OffHook(y), OnHook(y)) \wedge \\
Between(Dial(y,x), OnHook(y)) \wedge \\
OffHook(z) \wedge \\
Once_Since(Between(OffHook(x), OnHook(x)) \wedge \\
\textbf{not } Between(OffHook(z), OnHook(z)) \wedge \\
Between(OffHook(y), OnHook(y)), \\
Dial(y,x)) \wedge \\
Once_Since(Between(OffHook(x), OnHook(x)) \wedge \\
Between(OffHook(y), OnHook(y)) \wedge \\
Dial(y,x), \\
OffHook(z))) \quad \Rightarrow \quad AirBegin(x)
\end{array}
$$

Cette dernière propriété nécessite l'introduction du prédicat $Cell(x)$ qui retourne *vrai* lorsque l'abonné x a souscrit au service de téléphone cellulaire.

Nous écrivons également trois propriétés plus simples qui permettent de tester la facturation "air" dans le cadre d'un simple appel issu d'un abonné ayant souscrit au service de téléphone cellulaire. De même que pour les trois premières propriétés, $(CFBL - CELL\ 4)$ décrit précisément la séquence d'événements attendue, alors que $(CFBL - CELL\ 5)$ et $(CFBL - CELL\ 6)$ généralisent progressivement $(CFBL - CELL\ 4)$:

(CFBL - CELL 4)

$$
\begin{array}{l}
(\textbf{pre pre } OffHook(A) \wedge \\
\quad \textbf{pre } Dial(A,B)) \wedge \\
\quad\quad OffHook(B) \quad \Rightarrow \quad AirBegin(A)
\end{array}
$$

Remarque 6.2 *Nous ne donnons pas ici le détail des propriétés* $(CFBL - CELL\ 5)$ *et* $(CFBL - CELL\ 6)$ *qui sont construites à partir de la propriété* $(CFBL - CELL\ 4)$ *et sur le même modèle que les propriétés* $(CFBL - CELL\ 2)$ *et* $(CFBL - CELL\ 3)$.

CELL - CW

Nous étudions ici un système téléphonique constitué du service de base, d'un service de téléphone cellulaire (CELL) et d'un service de signal d'appel (CW). Nous considérons deux catégories de propriétés:

- la première s'assure du bon fonctionnement de la facturation "air" dans le cadre d'une simple communication entre un abonné au téléphone filaire et un abonné au téléphone cellulaire: dans ce but nous reprendrons les propriétés $(CFBL - CELL\ 4)$, $(CFBL - CELL\ 5)$ et $(CFBL - CELL\ 6)$ que nous renommons dans ce système par $(CELL - CW\ 1)$, $(CELL - CW\ 2)$ et $(CELL - CW\ 3)$;
- la seconde catégorie s'assure du bon fonctionnement de la facturation lorsqu'un abonné au téléphone cellulaire appelle un abonné déjà en communication et ayant souscrit au

service d'appel.

Dans ce système, les abonnés A et C ont souscrit au téléphone cellulaire alors que tous les abonnés ont souscrit au service de signal d'appel. Afin de tester le bon fonctionnement de la facturation "air" avec le service de signal d'appel, nous allons établir une communication entre les abonnés B et D. Pendant cette communication, l'abonné A va appeler l'abonné D. Dès que l'abonné D mettra B en attente pour prendre la communication de A par l'intermédiaire du service de signal d'appel, nous devrons observer une facturation "air" sur le compte de l'abonné A.

CELL - CW 4

$$
\begin{array}{r}
(\textbf{pre pre pre pre pre } OffHook(B)\wedge \\
\textbf{pre pre pre pre } Dial(B,D)\wedge \\
\textbf{pre pre pre } OffHook(D)\wedge \\
\textbf{pre pre } OffHook(A)\wedge \\
\textbf{pre } Dial(A,B)\wedge \\
Flash(B)) \;\Rightarrow\; AirBegin(A)
\end{array}
$$

La propriété $(CELL - CW\ 4)$ introduit une nouvelle action: $Flash(B)$. Cette action consiste à raccrocher puis décrocher le téléphone de façon très rapide. Dans le cadre du signal d'appel, cette action permet de permuter d'un correspondant à l'autre lorsque qu'un abonné déjà en communication reçoit un nouvel appel.

De la même manière que pour les deux autres systèmes précédents, nous généralisons la propriété $(CELL-CW\ 4)$ par deux propriétés: $(CELL-CW\ 5)$ et $(CELL-CW\ 6)$. Ces deux propriétés sont construites sur le même modèle que les propriétés $(CFBL-CELL\ 2)$ et $(CFBL-CELL\ 3)$.

CELL - TWC

Pour ce quatrième système téléphonique, nous associons toujours au service de base le service de téléphone cellulaire (CELL) avec cette fois un service de conversation téléphonique entre trois abonnés (TWC). Les propriétés que nous avons retenues pour ce système concernent toujours la facturation "air" du service de téléphone cellulaire. Cette fois, nous souhaitons nous assurer qu'une facturation "air" est effectuée lorsqu'un abonné ayant souscrit au service de conversation à trois et ayant déjà établi une communication, établit une communication avec un troisième correspondant ayant souscrit au téléphone cellulaire. Ici, l'action $Flash(A)$ prévient le système que l'abonné A va composer le numéro du troisième correspondant.

CELL - TWC 1

$$
\begin{array}{r}
\textbf{pre pre pre pre pre } OffHook(C)\wedge \\
\textbf{pre pre pre pre } Dial(C,A)\wedge \\
\textbf{pre pre pre } OffHook(A)\wedge \\
\textbf{pre pre } Flash(A)\wedge \\
\textbf{pre } Dial(A,B)\wedge \\
OffHook(B) \;\Rightarrow\; AirBegin(A)
\end{array}
$$

Nous construisons également les propriétés $(CELL-TWC\ 2)$ et $(CELL-TWC\ 2)$ qui correspondent aux deux généralisations successives de la propriété $(CELL-TWC\ 1)$.

CND - TCS

Ce système téléphonique est composé en plus du service de base, d'un service d'affichage du numéro de l'appelant (CND) et d'un service permettant à un abonné de définir une liste noire correspondant à des numéros indésirables dont il ne veut pas recevoir de communication (TCS). Nous allons écrire ici des propriétés dont le but est de s'assurer que le numéro d'un abonné dont on ne veut pas recevoir d'appel ne s'affiche pas sur le combiné de l'appelé.

Tous les abonnés du réseau ont souscrit au service CND, alors que seuls les abonnés B et C ont souscrit au service TCS:

- l'abonné B ne souhaite pas recevoir d'appel des abonnés A et C;
- l'abonné C ne souhaite pas recevoir d'appel de l'abonné D.

La propriété $(CND - TCS1)$ s'assure que l'abonné B ne verra pas s'afficher le numéro de l'abonné C dont il ne souhaite pas recevoir d'appel. Cette propriété utilise le prédicat $NoDisplay(B)$: il est *vrai* lorsque le dispositif d'affichage du téléphone de B est vide.

CND - TCS 1

$$\begin{array}{r} \textbf{pre}\ OffHook(C)\wedge \\ Dial(C,B) \ \Rightarrow \ NoDisplay(B) \end{array}$$

Cette propriété ne constitue qu'un cas particulier de ce que nous souhaitons vérifier, et comme pour tous les autres systèmes précédents, nous avons écrit les propriétés $(CND - TCS\ 2)$ et $(CND - TCS\ 3)$ qui généralisent progressivement $(CND - TCS\ 1)$.

CW - TCS

Pour ce dernier système téléphonique, nous avons considéré le service de base auquel nous avons ajouté le service de signal d'appel (CW) et le service de liste noire (TCS). Nous utilisons des propriétés qui permettent de s'assurer qu'un abonné mis sur la liste noire d'un autre abonné ne peut pas appeler ce dernier par l'intermédiaire du signal d'appel.

Dans ce système tous les abonnés ont souscrit aux deux services. Dans le cadre du service TCS, A ne souhaite pas recevoir d'appel de D, B ne souhaite pas recevoir d'appel de C, C ne souhaite pas recevoir d'appel de A et enfin D ne souhaite pas recevoir d'appel de C.

La propriété $(CW - TCS1)$ consiste à établir une communication entre les abonnés B et C, puis à faire appeler C par A. L'abonné C ayant souscrit à CW peut être averti qu'un abonné cherche à le joindre (A dans notre cas). Or, comme C a également souscrit à TCS en précisant qu'il ne souhaite pas recevoir d'appel de A, il ne doit pas être averti de cet appel et A doit recevoir une annonce lui indiquant qu'il ne peut pas joindre C (le prédicat $Announce(A)$ est *vrai* lorsque l'annonce est émise vers l'abonné A).

CW - TCS 1

$$\begin{array}{r} \textbf{pre pre pre pre}\ OffHook(B)\wedge \\ \textbf{pre pre pre}\ Dial(B,C)\wedge \\ \textbf{pre pre}\ OffHook(C)\wedge \\ \textbf{pre}\ OffHook(A)\wedge \\ Dial(A,C) \ \Rightarrow \ Announce(A) \end{array}$$

Nous généralisons ensuite la propriété $(CW - TCS\ 1)$ en appliquant une méthode

légèrement différente de celle que nous avons utilisée jusqu'à maintenant. Dans ($CW-TCS$ 2), nous caractérisons les états suspects comme étant ceux compris entre l'instant où A appelle C et l'instant où A raccroche, à condition que A et C aient tous deux décroché avant l'appel de A, et que C ait été appelé par B avant que lui-même ne décroche.

CW - TCS 2

$$
\begin{array}{l}
Once_Since(OffHook(B), Dial(B,C))\wedge \\
Once_Since(Dial(B,C), OffHook(C))\wedge \\
Once_Since(OffHook(C), Dial(A,C))\wedge \\
Once_Since(OffHook(A), Dial(A,C))\wedge \\
\quad Between(Dial(A,C), OnHook(A)) \;\Rightarrow\; Announce(A)
\end{array}
$$

La propriété ($CW-TCS$ 3) reprend la même structure que ($CW-TCS$ 2), mais sans définir a priori le rôle des différents abonnés dans la propriété. Les prédicats $isTcs(x)$ et $AOnBlackListOfB(y,x)$ permettent respectivement de savoir si l'abonné x a souscrit au service de liste noire (TCS) et si l'abonné y et sur la liste noire de x.

CW - TCS 3

$$
\begin{array}{r}
isTcs(x) \wedge isCw(x)\wedge \\
\neg AOnBlackListOfB(y,x) \wedge AOnBlackListOfB(z,x)\wedge \\
Once_Since(OffHook(y), Dial(y,x))\wedge \\
Once_Since(Dial(y,x), OffHook(x))\wedge \\
Once_Since(OffHook(x), Dial(z,x))\wedge \\
Once_Since(OffHook(z), Dial(z,x))\wedge \\
Between(Dial(z,x), OnHook(z)) \;\Rightarrow\; Announce(z)
\end{array}
$$

6.4 Aspects d'implantation

La machine qui a été utilisée au cours de cette évaluation est un $Pentium\ III$ $700MHz$ doté de $512Mo$ de RAM et utilisant Linux. En moyenne la construction du simulateur d'environnement guidé par les propriétés de sûreté prend de quelques minutes à quelques dizaines de minutes. Cependant, les limites de notre simulateur semblent être atteintes avec les propriétés qui laissent au système le soin de choisir les abonnés (celles qui s'expriment en fonction des utilisateurs x, y ou z): la construction de l'environnement a été arrêtée au bout de 24h de calcul.

Les propriétés les plus générales nécessitent de laisser le simulateur instancier aléatoirement les abonnés x, y et z. Nous avons donc dû introduire trois entrées fictives à notre système téléphonique afin que Lutess puisse générer des valeurs pour x, y et z. Dans cette situation, il pourrait sembler intéressant de pouvoir générer aléatoirement des variables locales au simulateur d'environnement afin de ne pas être obligé de modifier l'interface entre le simulateur d'environnement et le système sous test. Cependant, pour écrire l'oracle, nous devons connaître les conditions de test et notamment comment sont affectés les abonnés x, y et z: dans l'architecture actuelle de Lutess, ces informations doivent être présentes à l'interface du simulateur d'environnement.

TAB. 6.1: Tableau de présentation des résultats.

	...	Nom Stratégie α	...
...			
Nom Stratégie β		x	
...			
États suspects atteints		y/z	

6.5 Situations suspectes atteintes

6.5.1 Présentation des résultats

Nous donnons ici un guide de lecture pour interpréter les tableaux[6] résumant les états suspects atteints pour chacune des propriétés présentées au paragraphe 6.3. Ces tableaux permettent de connaître le nombre d'états suspects qu'une stratégie a atteint et qui n'ont pas été atteints par une autre stratégie. Ils donnent également, pour chacune des stratégies, le nombre d'états suspects distincts atteints par rapport au nombre total d'états suspects atteints.

La table 6.1 représente un tableau générique utilisé pour mise en forme des résultats obtenus:

- x représente le nombre d'états suspects atteints par la stratégie α (sur la colonne) et qui n'ont pas été atteints par la stratégie β (sur la ligne);
- y est le nombre d'états suspects distincts atteints;
- z est le nombre total d'états suspects atteints.

Pour chacune des propriétés présentées au paragraphe 6.3, LUTESS a construit une séquence de 30000 instants (c'est à dire que 30000 vecteurs d'entrées ont été sélectionnés puis exécutés par le système) par stratégie. Notons que le test s'arrête à l'issue des 30000 instants. En d'autres termes, même si la violation d'une propriété est détectée, le test continue jusqu'à la fin des 30000 instants.

Cette expérimentation nous permet de comparer les stratégies que nous avons définies au chapitre 5 entre elles mais aussi avec la sélection aléatoire équiprobable des vecteurs d'entrées (*témoin*) et avec la solution de guidage *instantané* du choix d'un vecteur d'entrées [OP94b].

6.5.2 CFBL - CFBL

Remarques préliminaires

La propriété ($CFBL - CFBL$ 1) est la seule propriété pour ce système avec laquelle nous avons pu construire le générateur de test quelle que soit la stratégie et l'approximation choisie.

En revanche, dès que nous utilisons les propriétés généralisées, LUTESS n'arrive à construire le générateur de test en général que pour l'approximation *"Sans Environnement"* pour la

[6] Ces tableaux sont construits automatiquement à partir des fichiers contenant la liste états suspects (voir l'annexe A).

TAB. 6.2: Résultats pour $CFBL - CFBL\,1$

Approximation *"Tout Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	1	0	1	1
Instantanée	0	0	0	0	0
Intersection	1	2	0	2	2
Union	0	0	0	0	0
Paresseuse	0	0	0	0	0
DS/S	1 / 1	2 / 21	0 / 0	2 / 666	2 / 585

Approximation *"Semi Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	1	0	1	1
Instantanée	0	0	0	0	0
Intersection	1	2	0	2	2
Union	0	0	0	0	0
Paresseuse	0	0	0	0	0
DS/S	1 / 1	2 / 21	0 / 0	2 / 585	2 / 585

Approximation *"Sans Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	2	2
Instantanée	0	0	0	2	2
Intersection	0	0	0	2	2
Union	0	0	0	0	0
Paresseuse	0	0	0	0	0
DS/S	0 / 0	0 / 0	0 / 0	2 / 666	2 / 664

propriété $(CFBL - CFBL\,2)$[7]. Dans le cas de la propriété $(CFBL - CFBL\,3)$, LUTESS n'a pas réussi à construire de générateur de test quelle que soit la stratégie et l'approximation choisie. La construction d'un générateur d'environnement pour cette propriété nécessite plus de 24 heures.

Concernant les propriétés générales (1) et (2) qui s'intéressent à l'état du poste téléphonique, LUTESS a pu construire un générateur de test pour toutes les stratégies avec les approximations *"Semi Environnement"* et *"Sans Environnement"* définies au paragraphe 5.5.

Étude des résultats

Dans le cas de l'approximation *"Tout Environnement"*, la propriété $(CFBL - CFBL\,1)$ n'a jamais été violée quelle que soit la stratégie choisie. En revanche, l'ensemble des états suspects atteints est résumé dans les tableaux 6.2.

Pour la propriété $CFBL - CFBL\ 1$, les résultats montrent que les stratégies d'union et paresseuse sont capables de tester un plus grand nombre de fois les situations suspectes (nombre total d'états suspects atteints). En revanche, le nombre de situations suspectes distinctes atteintes semble stable indépendamment de la stratégie utilisée; notons toutefois que les stratégies d'union et paresseuse continuent d'atteindre le même nombre d'états suspects lorsque l'approximation augmente contrairement aux autres stratégies qui voient leurs nombres d'états suspects atteints décroître. Enfin, le tableau 6.3 montre que les états suspects atteints par la stratégie paresseuse sont les même quelle que soit l'approximation

[7]Pour la propriété $(CFBL - CFBL\ 2)$, LUTESS a également réussi à construire un générateur pour la séquence témoin et la stratégie instantanée pour l'approximation *"Semi Environnement"* .

TAB. 6.3: Comparaison des résultats pour la stratégie paresseuse en fonction de l'approximation

	Paresseuse (*"Tout Environnement"*)	Paresseuse (*"Semi Environnement"*)	Paresseuse (*"Sans Environnement"*)
Paresseuse (*"Tout Environnement"*)	0	0	0
Paresseuse (*"Semi Environnement"*)	0	0	0
Paresseuse (*"Sans Environnement"*)	0	0	0
DS/S	2 / 585	2 / 585	2 / 664

choisie. De plus, en utilisant conjointement les résultats des tableaux 6.2 et 6.3, on déduit que les stratégies union et paresseuse atteignent les mêmes états suspects. En conséquence, la comparaison des états suspects atteints par la stratégie union en fonction de l'approximation donnerait les même résultats que ceux obtenus avec la stratégie paresseuse.

L'absence de situation suspecte atteinte par la stratégie d'intersection s'explique simplement par le fait qu'il n'est pas possible de définir un même vecteur d'entrées unique qui puisse mener à un état suspect par des chemins de longueur 0, 1, 2 ou 3. En conséquence, la stratégie d'intersection ne peut pas guider la sélection des vecteurs d'entrées.

Le tableau 6.4 représente les résultats obtenus pour la propriété ($CFBL - CFBL\,2$) avec l'approximation *"Sans Environnement"* . Avec cette seconde propriété, le nombre d'états suspects distincts atteint avec les stratégies instantanée et intersection est légèrement supérieur à celui atteint par les stratégies union et paresseuse. Cependant, la différence la plus importante avec les résultats de la propriété ($CFBL - CFBL\,1$) est le très grand nombre d'états suspects atteint par les stratégies instantanée, intersection et paresseuse où approximativement 50% des états de la séquence sont suspects. Dans le cas de ($CFBL - CFBL\,2$), dès que le simulateur se trouve dans un état suspect à l'instant t, il est très facile de le conserver pour les n prochains instants en interdisant aux abonnés A, B et C de raccrocher. En effet:

- lorsque la stratégie instantanée se trouve dans un état suspect, elle est capable de déterminer qu'il ne faut pas faire raccrocher les abonnés A, B et C pour pouvoir observer une violation de la propriété; ce qui a aussi pour conséquence de conserver l'état suspect;
- lorsque la stratégie intersection se trouve dans un état suspect et que le vecteur d'entrées qui consiste à ne pas faire raccrocher les abonnés A, B et C permet de conserver l'état suspect, il s'ensuit que ce même vecteurs pourra mener à une observation d'une violation de la propriété à l'instant $t+1$, $t+2$ ... $t+n$. Ainsi la stratégie intersection est capable de conserver l'état suspect.
- la stratégie union peut conserver l'état suspect en ne faisant pas raccrocher les abonnés A, B et C. En revanche, elle s'aperçoit également que si A décroche, puis B et enfin que C appelle A alors un état suspect peut être obtenu. Comme la stratégie union considère tous les vecteurs d'entrées pertinents avec la même priorité quel que soit le nombre d'instants nécessaires pour atteindre un état suspect, elle pourra facilement choisir de faire raccrocher les abonnés A, B et C dans le but de faire à nouveau décrocher A par la suite;
- la stratégie paresseuse favorise les vecteurs d'entrées qui peuvent mener à l'observation

TAB. 6.4: Résultats pour $CFBL - CFBL\,2$

Approximation *"Sans Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	35	35	30	31
Instantanée	0	0	0	4	4
Intersection	0	0	0	4	4
Union	0	9	9	0	6
Paresseuse	0	8	8	5	0
DS/S	0 / 0	35 / 16689	35 / 16689	30 / 353	31 / 15686

d'une violation de la propriété le plus rapidement possible. En conséquence, dès qu'elle va atteindre un état suspect, elle va facilement le conserver en ne faisant pas raccrocher les abonnés A, B et C.

Ces premiers résultats mettent en évidence une certaine complémentarité entre les différentes stratégies. En effet, toutes les stratégies semblent atteindre un nombre d'états suspects distincts à peu près équivalent, alors que chacune des stratégies atteint un petit nombre d'états suspects non atteint par les autres stratégies.

6.5.3 CFBL - CELL

Remarques préliminaires

Les résultats que nous avons obtenus pour le système $(CFBL - CFBL)$ nous ont montré qu'il pouvait être difficile voire impossible de construire un simulateur de l'environnement lorsque les propriétés devenaient complexes. Pour cette raison, nous avons choisi de simplifier les contraintes d'environnement du système $(CFBL - CELL)$. Cette simplification ne porte que sur des comportements de l'abonné que nous ne souhaitions pas voir et que nous avions préalablement interdits. Par exemple, nous avions interdit à un abonné de décrocher et raccrocher successivement son téléphone pour éviter de voir apparaître dans les séquences de longues périodes de décrocher/raccrocher présentant peu d'intérêt pour le test.

Avec ces simplifications de l'environnement, nous avons pu tester un plus grand nombre de propriétés (comparer au système $(CFBL - CFBL)$. En effet, ici seules les propriétés $(CFBL - CELL\,3)$ et $(CFBL - CELL\,6)$ n'ont pas pu être testées faute d'avoir pu construire le simulateur d'environnement dans un temps raisonnable (dans notre cas, inférieur à 24h). Toutes les autres propriétés ont pu être testées dans les trois approximations.

Afin d'essayer de tester les dernières propriétés pour lesquelles LUTESS n'a pas pu construire le simulateur d'environnement, nous pourrions envisager de continuer à simplifier la description de l'environnement. Cependant, nous n'appliquerions plus toutes les contraintes physiques qui existent sur un téléphone comme par exemple celle qui impose qu'il faut décrocher entre deux actions "raccrocher". Dans la suite de cette expérimentation, nous avons fait le choix de conserver un environnement réaliste au risque de voir les propriétés les plus générales échouer lors de la construction du simulateur.

TAB. 6.5: Résultats pour $CFBL - CELL$ 1

Approximation *"Tout Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	2	2
Instantanée	0	0	0	2	2
Intersection	0	0	0	2	2
Union	0	0	0	0	0
Paresseuse	0	0	0	0	0
DS/S	0 / 0	0 / 0	0 / 0	2 / 90	2 / 85

Approximation *"Semi Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	2	2
Instantanée	0	0	0	2	2
Intersection	0	0	0	2	2
Union	0	0	0	0	0
Paresseuse	0	0	0	0	0
DS/S	0 / 0	0 / 0	0 / 0	2 / 83	2 / 90

Approximation *"Sans Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	2	2
Instantanée	0	0	0	2	2
Intersection	0	0	0	2	2
Union	0	0	0	0	0
Paresseuse	0	0	0	0	0
DS/S	0 / 0	0 / 0	0 / 0	2 / 84	2 / 79

Étude des résultats

Le nombre d'états suspects atteints par chaque stratégie lors du test de la propriété ($CFBL - CELL$ 1) en fonction de l'approximation choisie est donné dans les trois tableaux de la table 6.5.

Les résultats que nous obtenons avec la propriété ($CFBL - CELL$ 1) donnent toujours l'avantage aux stratégies *union* et *paresseuse* quelle que soit l'approximation choisie: elles sont les seules à atteindre des états suspects pour ($CFBL - CELL$ 1). De plus, si nous analysons les résultats de l'oracle pour cette propriété, nous obtenons une violation de la propriété chaque fois qu'un état suspect est atteint. Après vérification de la propriété utilisée, nous pouvons affirmer que nos stratégies ont mis en évidence une interaction entre les deux services téléphonique qui a pour conséquence une erreur de facturation. Cette interaction avait également été découverte lors du concours de 1998 [dBZ98].

Le nombre d'états suspects distincts atteints par les stratégies *union* et *paresseuse* ne varie pas avec l'approximation choisie: nous obtenons toujours 2 états suspects distincts. Ces résultats vont dans le sens des premiers résultats que nous avions eus avec la première paire de services. Nous avions alors constaté que pour la propriété $CFBL - CFBL$ 1, le nombre d'états suspects atteint par la stratégie paresseuse ne variait pas avec l'approximation choisie.

En revanche, aucune stratégie, quelle que soit l'approximation choisie, n'a pu atteindre d'état suspect[8] avec la propriété ($CELL - CFBL$ 2)[9]. Un tel résultat peut s'expliquer de deux manières:

[8]Nous ne présentons pas ici de tableau de résultats où toutes les valeurs sont identiquement égales à 0.

[9]Nous rappelons que ($CELL - CFBL$ 2) est la première généralisation de ($CELL - CFBL$ 1) (voir page 105).

TAB. 6.6: Résultats pour $CFBL - CELL\ 4$

Approximation *"Tout Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	10	10	0	9
Instantanée	0	0	0	0	1
Intersection	0	0	0	0	1
Union	0	10	10	0	9
Paresseuse	0	2	2	0	0
DS/S	0 / 0	10 / 450	10 / 450	0 / 0	9 / 476

Approximation *"Semi Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	2	2	2	1
Instantanée	0	0	0	0	1
Intersection	0	0	0	0	1
Union	0	0	0	0	1
Paresseuse	0	2	2	2	0
DS/S	8 / 20	10 / 450	10 / 450	10 / 450	9 / 476

Approximation *"Sans Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	2	2	2	1
Instantanée	0	0	0	0	1
Intersection	0	0	0	0	1
Union	0	0	0	0	1
Paresseuse	0	2	2	2	0
DS/S	8 / 20	10 / 450	10 / 450	10 / 450	9 / 476

- soit la propriété que l'on considère peut être mal écrite et ne jamais être violable;
- soit le choix du nombre d'instants n à explorer est mal choisi;

Après avoir vérifié la valeur de n puis essayé différentes valeurs de n menant toujours à aucun état suspect, l'analyse des ensembles de vecteurs d'entrées pertinents[10] permettant d'atteindre un état suspect a révélé que nous étions en présence d'ensembles vides. Ceci indique que la propriété ne peut pas être violée.

La propriété ($CFBL - CELL\ 5$) correspond à la première généralisation de la propriété ($CFBL - CELL\ 4$). Les résultats que nous avons obtenus avec cette propriété sont présentés dans les trois tableaux de la table 6.7. Nous observons que le nombre d'états suspects atteints reste globalement identique indépendamment de l'approximation choisie, sauf pour la stratégie paresseuse qui obtient de meilleurs résultats lorsque les contraintes de l'environnement sont prises en compte dans le calcul des vecteurs d'entrées pertinents menant à des états suspects.

6.5.4 CELL - CW

Remarques préliminaires

Tout comme pour le système constitué des services $CELL$ et $CFBL$, LUTESS n'a pas été capable de construire le simulateur d'environnement pour les propriétés ($CELL - CW\ 3$) et ($CELL - CW\ 6$) qui correspondent aux propriétés les plus générales. Nous rappelons que

[10] Nous rappelons que ces ensembles sont représentés par des bdd qui dans le cas de la propriété ($CELL - CFBL\ 2$) étaient tous égaux à *faux* .

TAB. 6.7: Résultats pour $CFBL - CELL$ 5

Approximation *"Tout Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	5	5	2	14
Instantanée	1	0	0	1	13
Intersection	1	0	0	1	13
Union	4	7	7	0	15
Paresseuse	4	7	7	3	0
DS/S	29 / 523	33 / 20560	33 / 20560	27 / 493	39 / 22602

Approximation *"Semi Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	5	5	2	14
Instantanée	1	0	0	1	13
Intersection	1	0	0	1	13
Union	4	7	7	0	15
Paresseuse	4	7	7	3	0
DS/S	29 / 523	33 / 20560	33 / 20560	27 / 493	39 / 22602

Approximation *"Sans Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	5	5	2	4
Instantanée	1	0	0	1	3
Intersection	1	0	0	1	3
Union	4	7	7	0	5
Paresseuse	2	5	5	1	0
DS/S	29 / 523	33 / 20560	33 / 20560	27 / 493	31 / 20823

ces propriétés n'imposent pas *a priori* qui doit faire quoi mais seulement des contraintes sur les abonnés comme par exemple l'abonné x doit avoir souscrit au service CW. L'échec systématique de la construction du simulateur avec les propriétés laissant le soin à LUTESS de décider quels rôles joueront les différents abonnés du système dans la séquence des événements qui doit mener à la violation de la propriété nous laisse penser que nous atteignons les limites de ce que nous pouvons calculer avec notre matériel en un temps raisonnable (inférieur à 24h).

Étude des résultats

Les résultats obtenus pour cette nouvelle paire de services sont conformes avec ceux précédemment obtenus. Dans le cas de ($CELL - CW$ 1), chaque fois qu'un état suspect a été atteint, nous avons constaté une violation de l'oracle.

Lorsque l'on regarde les résultats obtenus pour la propriété ($CELL - CW$ 2) (voir les tableaux de résultats 6.9), qui correspond à la première généralisation de la propriété ($CELL - CW$ 1), nous constatons que le nombre d'états suspects atteints par une stratégie donnée est nettement plus important que pour la propriété initiale (ici ($CELL - CW$ 1)).

Cette différence entre le nombre d'états suspects atteint avec la propriété généralisée et celui atteint par la propriété fortement guidée s'observe pour toutes les paires de services que nous avons testées jusqu'ici. Une propriété fortement guidée définie une suite très précise d'événements qui doivent se produire afin d'être en mesure d'observer sa violation. Le nombre d'états suspects défini par une propriété fortement guidée est, par construction, inférieur à celui défini par une propriété générale.

Avec la propriété $CELL - CW$ 5, LUTESS n'a pas pu atteindre d'état suspect. Ces

TAB. 6.8: Résultats pour $CELL - CW\ 1$

Approximation *"Tout Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	1	0	1	3
Instantanée	0	0	0	1	3
Intersection	0	1	0	1	3
Union	0	1	0	0	2
Paresseuse	0	1	0	0	0
DS/S	8 / 18	9 / 510	8 / 18	9 / 953	11 / 899

Approximation *"Semi Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	1	0	1	3
Instantanée	0	0	0	1	3
Intersection	0	1	0	1	3
Union	0	1	0	0	2
Paresseuse	0	1	0	0	0
DS/S	8 / 18	9 / 510	8 / 18	9 / 953	11 / 899

Approximation *"Sans Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	1	0	1	3
Instantanée	0	0	0	1	3
Intersection	0	1	0	1	3
Union	0	1	0	0	2
Paresseuse	0	1	0	0	0
DS/S	8 / 18	9 / 510	8 / 18	9 / 953	11 / 899

TAB. 6.9: Résultats pour $CELL - CW\ 2$

Approximation *"Tout Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	2	2	1	7
Instantanée	0	0	0	0	6
Intersection	0	0	0	0	6
Union	0	1	1	0	6
Paresseuse	2	3	3	2	0
DS/S	27 / 537	29 / 20642	29 / 20642	28 / 520	32 / 22592

Approximation *"Semi Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	2	2	1	7
Instantanée	0	0	0	0	6
Intersection	0	0	0	0	6
Union	0	1	1	0	6
Paresseuse	2	3	3	2	0
DS/S	27 / 537	29 / 20642	29 / 20642	28 / 520	32 / 22592

Approximation *"Sans Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	2	2	1	4
Instantanée	0	0	0	0	3
Intersection	0	0	0	0	3
Union	0	1	1	0	3
Paresseuse	0	1	1	0	0
DS/S	27 / 537	29 / 20642	29 / 20642	28 / 520	31 / 20200

TAB. 6.10: Résultats pour $CELL - CW\ 4$

Approximation *"Tout Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	2	3
Instantanée	0	0	0	2	3
Intersection	0	0	0	2	3
Union	0	0	0	0	1
Paresseuse	0	0	0	0	0
DS/S	0 / 0	0 / 0	0 / 0	2 / 74	3 / 70

Approximation *"Semi Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	2	3
Instantanée	0	0	0	2	3
Intersection	0	0	0	2	3
Union	0	0	0	0	1
Paresseuse	0	0	0	0	0
DS/S	0 / 0	0 / 0	0 / 0	2 / 82	3 / 63

Approximation *"Sans Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	2	3
Instantanée	0	0	0	2	3
Intersection	0	0	0	2	3
Union	0	0	0	0	1
Paresseuse	0	0	0	0	0
DS/S	0 / 0	0 / 0	0 / 0	2 / 82	3 / 63

résultats sont dus au fait qu'il n'existe pas de vecteur d'entrées pouvant mener la propriété vers un état suspect(de la même façon que pour la propriété ($CELL - CFBL\ 2$)).

Les deux dernières propriétés correspondent aux deux propriétés générales (voir paragraphe 6.3.1). Nous rappelons que ces deux propriétés ne sont pas construites à partir d'une implication (comme c'est le cas des propriétés spécifiques aux services testés) mais à partir de l'opérateur de logique temporelle $always_from_to$. Or malgré cette différence de construction de la propriété, les résultats que nous obtenons restent cohérents avec l'ensemble des résultats que nous avons déjà obtenus.

6.5.5 CELL - TWC

Remarques préliminaires

Les résultats que nous obtenons sur ce système confirment d'une manière générale ceux que nous avons obtenus pour les systèmes précédents:

- la stratégie paresseuse permet d'atteindre un plus grand nombre d'états suspects distincts;
- peu de différence entre les différentes approximations;
- complémentarité des stratégies.

De la même façon que pour les trois premiers systèmes étudiés, LUTESS ne parvient pas à construire le simulateur d'environnement (quelle que soit l'approximation choisie) pour la propriété $CELL{-}TWC\ 3$ qui correspond à la plus grande généralisation de $CELL{-}TWC\ 1$.

TAB. 6.11: Résultats pour $CELL - CW\ 7$

Approximation *"Tout Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	35	30
Instantanée	0	0	0	35	30
Intersection	0	0	0	35	30
Union	0	0	0	0	4
Paresseuse	0	0	0	9	0
DS/S	0 / 0	0 / 0	0 / 0	35 / 17981	30 / 24918

Approximation *"Semi Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	35	30
Instantanée	0	0	0	35	30
Intersection	0	0	0	35	30
Union	0	0	0	0	4
Paresseuse	0	0	0	9	0
DS/S	0 / 0	0 / 0	0 / 0	35 / 15320	30 / 24978

Approximation *"Sans Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	35	30
Instantanée	0	0	0	35	30
Intersection	0	0	0	35	30
Union	0	0	0	0	4
Paresseuse	0	0	0	9	0
DS/S	0 / 0	0 / 0	0 / 0	35 / 15320	30 / 24988

TAB. 6.12: Résultats pour $CELL - CW\ 8$

Approximation *"Tout Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	4	4	4	17
Instantanée	0	0	0	0	16
Intersection	0	0	0	0	16
Union	0	0	0	0	16
Paresseuse	14	17	17	17	0
DS/S	28 / 247	32 / 2031	32 / 2031	32 / 22161	31 / 24975

Approximation *"Semi Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	4	4	4	17
Instantanée	0	0	0	0	16
Intersection	0	0	0	0	16
Union	0	0	0	0	16
Paresseuse	14	17	17	17	0
DS/S	28 / 247	32 / 2031	32 / 2031	32 / 22161	31 / 24975

Approximation *"Sans Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	32	32	32	31
Instantanée	0	0	0	0	16
Intersection	0	0	0	0	16
Union	0	0	0	0	16
Paresseuse	0	17	17	17	0
DS/S	0 / 0	32 / 1781	32 / 2031	32 / 22161	31 / 24975

TAB. 6.13: Résultats pour $CELL - TWC$ 1

Approximation *"Tout Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	2	2
Instantanée	0	0	0	2	2
Intersection	0	0	0	2	2
Union	0	0	0	0	0
Paresseuse	0	0	0	0	0
DS/S	0 / 0	0 / 0	0 / 0	2 / 71	2 / 84

Approximation *"Semi Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	2	2
Instantanée	0	0	0	2	2
Intersection	0	0	0	2	2
Union	0	0	0	0	0
Paresseuse	0	0	0	0	0
DS/S	0 / 0	0 / 0	0 / 0	2 / 75	2 / 72

Approximation *"Sans Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	2	2
Instantanée	0	0	0	2	2
Intersection	0	0	0	2	2
Union	0	0	0	0	0
Paresseuse	0	0	0	0	0
DS/S	0 / 0	0 / 0	0 / 0	2 / 73	2 / 72

Étude des résultats

Les résultats obtenus avec la propriété $CELL-TWC$ 1 sont très proches de ceux obtenus avec la propriété $CELL - CW$ 4. Dans les deux cas, seules les stratégies union et paresseuse permettent d'atteindre des états suspects. La séquence d'événement qui mène dans un état suspect est un peu plus longue pour ces deux propriétés que pour les précédentes (suite de 5 événements au lieu de 3 en moyenne). Ces résultats tendent à montrer que l'efficacité du guidage est d'autant plus significative que la séquence menant à un état suspect est longue.

Dans le cas de la propriété ($CELL-TWC$ 2), aucune stratégie n'arrive à atteindre d'états suspects quelle que soit l'approximation choisie. Les causes de ces résultats sont identiques à celles qui ont conduit aux même résultats pour les propriétés ($CELL - CFBL$ 2) et ($CELL - CW$ 5).

Les deux dernières propriétés correspondent aux propriétés générales qui ont été utilisées pour tous les systèmes que nous avons testés dans cette étude. Les résultats obtenus pour ($CELL - TWC$ 4) et ($CELL - TWC$ 5) sont pratiquement identiques d'une paire de service à l'autre. Rappelons que ces propriétés ne dépendent pas de la spécificité d'un service téléphonique en particulier mais portent sur le déroulement d'une communication classique entre deux abonnés commune à toutes les paires de services. Il n'est donc pas surprenant de se trouver face aux même résultats.

Tab. 6.14: Résultats pour $CELL - TWC$ 4

Approximation *"Tout Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	36	25
Instantanée	0	0	0	36	25
Intersection	0	0	0	36	25
Union	0	0	0	0	4
Paresseuse	0	0	0	15	0
DS/S	0 / 0	0 / 0	0 / 0	36 / 17981	25 / 24918

Approximation *"Semi Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	34	23
Instantanée	0	0	0	34	23
Intersection	0	0	0	34	23
Union	0	0	0	0	4
Paresseuse	0	0	0	15	0
DS/S	0 / 0	0 / 0	0 / 0	34 / 15320	23 / 24978

Approximation *"Sans Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	34	27
Instantanée	0	0	0	34	27
Intersection	0	0	0	34	27
Union	0	0	0	0	4
Paresseuse	0	0	0	11	0
DS/S	0 / 0	0 / 0	0 / 0	34 / 15320	27 / 24935

Tab. 6.15: Résultats pour $CELL - TWC$ 5

Approximation *"Tout Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	4	4	4	17
Instantanée	0	0	0	0	16
Intersection	0	0	0	0	16
Union	0	0	0	0	16
Paresseuse	14	17	17	17	0
DS/S	28 / 247	32 / 2031	32 / 2031	32 / 22161	31 / 24975

Approximation *"Semi Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	4	4	4	17
Instantanée	0	0	0	0	16
Intersection	0	0	0	0	16
Union	0	0	0	0	16
Paresseuse	14	17	17	17	0
DS/S	28 / 247	32 / 1781	32 / 1781	32 / 22161	31 / 24975

Approximation *"Sans Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	4	4	4	17
Instantanée	0	0	0	0	16
Intersection	0	0	0	0	16
Union	0	0	0	0	16
Paresseuse	14	17	17	17	0
DS/S	28 / 247	32 / 2031	32 / 2031	32 / 22161	31 / 24975

TAB. 6.16: Résultats pour $CND - TCS$ 1

Approximation *"Tout Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	1	0	0	7
Instantanée	0	0	0	0	7
Intersection	0	1	0	0	7
Union	0	1	0	0	7
Paresseuse	0	1	0	0	0
DS/S	8 / 31	9 / 421	8 / 31	8 / 26	15 / 7207

Approximation *"Semi Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	1	0	3	2
Instantanée	0	0	0	3	2
Intersection	0	1	0	3	2
Union	0	1	0	0	1
Paresseuse	0	1	0	2	0
DS/S	8 / 31	9 / 421	8 / 31	11 / 757	10 / 756

Approximation *"Sans Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	3	2
Instantanée	8	0	8	11	10
Intersection	0	0	0	3	2
Union	0	0	0	0	1
Paresseuse	0	0	0	2	0
DS/S	8 / 31	0 / 0	8 / 31	11 / 757	10 / 455

6.5.6 CND - TCS

Remarques préliminaires

La propriété ($CND - TCS$ 1) que nous utilisons avec ce système est relativement simple à comparer de la plupart de celles que nous avons utilisées. Dans sa version la plus stricte, elle comporte deux événements:

- l'abonné C décroche;
- C appelle B.

Dans ces conditions, nous ne devrions pas voir le numéro de l'abonné C s'afficher sur le téléphone de B. Malgré tout, LUTESS ne parvient toujours pas à construire le simulateur d'environnement pour la généralisation de cette propriété dans le cas où nous laissons l'instanciation des abonnés libre ($CND - TCS$ 3).

Les résultats obtenus pour les propriétés $CND - TCS$ 1 et $CND - TCS$ 2 font de la stratégie paresseuse celle qui utilise le mieux l'information contenue dans les contraintes de l'environnement pour le calcul des vecteurs d'entrées pertinents. Ces résultats confirment ceux déjà obtenus avec la propriété $CFBL - CELL$ 5.

Étude des résultats

La propriété $CND - TCS$ 1 met particulièrement en évidence les capacités de la stratégie paresseuse à guider la sélection des vecteurs d'entrées pour atteindre des états suspects. Cette efficacité est d'autant plus significative lorsque les contraintes d'environnement sont prises en compte sur l'ensemble du calcul (approximation *"Tout Environnement"*). Dans ce cas,

TAB. 6.17: Résultats pour $CND - TCS$ 2

Approximation *"Tout Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	1	1	0	13
Instantanée	0	0	0	0	13
Intersection	0	0	0	0	13
Union	0	1	1	0	13
Paresseuse	2	3	3	2	0
DS/S	28 / 490	29 / 7330	29 / 20070	28 / 458	39 / 22632

Approximation *"Semi Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	1	1	3	15
Instantanée	0	0	0	3	15
Intersection	0	0	0	3	15
Union	6	7	7	0	21
Paresseuse	0	1	1	3	0
DS/S	28 / 490	29 / 20070	29 / 19835	25 / 862	43 / 22663

Approximation *"Sans Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	1	1	0	0
Instantanée	0	0	0	0	0
Intersection	0	0	0	0	0
Union	0	1	1	0	0
Paresseuse	0	1	1	0	0
DS/S	28 / 490	29 / 20070	29 / 20070	28 / 458	28 / 20698

la stratégie paresseuse atteint un nombre d'états suspects très supérieur à celui des autres stratégies avec un nombre d'états suspects distincts pratiquement doublé. Cependant, toutes les propriétés ne donnent pas de résultats aussi favorables pour la stratégie paresseuse, en revanche elle se situe pratiquement toujours à un niveau équivalent ou supérieur aux autres stratégies (en terme d'états suspects atteints).

La propriété $CND - TCS$ 2 donne des résultats moins significatifs en terme de nombre total d'états suspects atteints. Par contre, l'analyse des états suspects distincts montre que la stratégie paresseuse en trouve un nombre supérieur de l'ordre de 33% par rapport aux autres stratégies (lorsque l'environnement est pris en compte).

Les deux derniers tableaux (6.18 et 6.19) correspondent aux résultats obtenus avec les propriétés (1) et (2) définies au paragraphe 6.3.1 et communes aux six systèmes. Comme pour les systèmes précédents, seules les stratégies *union* et *paresseuse* permettent d'atteindre des états suspects pour la propriété (1), alors que toutes atteignent des états suspects pour la propriété (2). De plus, les nombres d'états suspects atteints, distincts ou non, restent dans le même ordre de grandeur d'un système à l'autre.

6.5.7 CW - TCS

Remarques préliminaires

Lors de l'écriture des propriétés spécifiques à cette dernière paire de services, nous avons écrit les propriétés généralisées sous une forme légèrement plus simple. Cependant, cette simplification n'a pas été suffisante pour permettre le test de la propriété ($CW - TCS$ 3) qui constitue la plus grande généralisation de ($CW - TCS$ 1) que nous ayons écrite. Mais

Tab. 6.18: Résultats pour $CND - TCS$ 4

Approximation *"Tout Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	25	19
Instantanée	0	0	0	25	19
Intersection	0	0	0	25	19
Union	0	0	0	0	5
Paresseuse	0	0	0	11	0
DS/S	0 / 0	0 / 0	0 / 0	25 / 17853	19 / 24972

Approximation *"Semi Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	20	19
Instantanée	0	0	0	20	19
Intersection	0	0	0	20	19
Union	0	0	0	0	5
Paresseuse	0	0	0	6	0
DS/S	0 / 0	0 / 0	0 / 0	20 / 15048	19 / 24767

Approximation *"Sans Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	20	19
Instantanée	0	0	0	20	19
Intersection	0	0	0	20	19
Union	0	0	0	0	5
Paresseuse	0	0	0	6	0
DS/S	0 / 0	0 / 0	0 / 0	20 / 15048	19 / 24767

Tab. 6.19: Résultats pour $CND - TCS$ 5

Approximation *"Tout Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	3	3	6	7
Instantanée	1	0	0	4	5
Intersection	1	0	0	4	5
Union	0	0	0	0	5
Paresseuse	18	18	18	22	0
DS/S	28 / 266	30 / 1521	30 / 1521	34 / 21829	17 / 24977

Approximation *"Semi Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	3	3	6	7
Instantanée	1	0	0	4	5
Intersection	1	0	0	4	5
Union	0	0	0	0	5
Paresseuse	18	18	18	22	0
DS/S	28 / 266	30 / 1521	30 / 1521	34 / 21829	17 / 24977

Approximation *"Sans Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	3	3	6	7
Instantanée	1	0	0	4	5
Intersection	1	0	0	4	5
Union	0	0	0	0	5
Paresseuse	18	18	18	22	0
DS/S	28 / 266	30 / 1521	30 / 1521	34 / 21829	17 / 24977

TAB. 6.20: Résultats pour $CW - TCS$ 1

Approximation *"Tout Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	2	2
Instantanée	0	0	0	2	2
Intersection	0	0	0	2	2
Union	0	0	0	0	0
Paresseuse	0	0	0	0	0
DS/S	0 / 0	0 / 0	0 / 0	2 / 89	2 / 73

Approximation *"Semi Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	2	2
Instantanée	0	0	0	2	2
Intersection	0	0	0	2	2
Union	0	0	0	0	0
Paresseuse	0	0	0	0	0
DS/S	0 / 0	0 / 0	0 / 0	2 / 75	2 / 91

Approximation *"Sans Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	2	2
Instantanée	0	0	0	2	2
Intersection	0	0	0	2	2
Union	0	0	0	0	0
Paresseuse	0	0	0	0	0
DS/S	0 / 0	0 / 0	0 / 0	2 / 75	2 / 91

le lancement de la séquence de test sans activer la propriété ($CW - TCS$ 3) nous a permis de générer un simulateur d'environnement qui contient les contraintes environnementales nécessaires pour l'instanciation des abonnés génériques X, Y et Z par LUTESS:

$$\neg Identification(X,Y) \wedge LIN_XOR(X)$$
$$\neg Identification(X,Z) \wedge LIN_XOR(Y)$$
$$\neg Identification(Y,Z) \wedge LIN_XOR(Z)$$

L'opérateur $Identification$ permet de s'assurer que les adresses de deux abonnés sont identiques; la négation de cet opérateur permet d'imposer que les abonnés X, Y et Z soient différents. Enfin, l'opérateur LIN_XOR est un opérateur de *ou exclusif* sur un vecteur de booléen: l'adresse d'un abonné étant représentée par un vecteur de booléens en utilisant un codage "un parmi n", nous souhaitons imposer que le vecteur de booléens choisi par LUTESS représente toujours une adresse valide.

Étude des résultats

Pour le système constitué des services CW et TCS, l'analyse des résultats concorde avec celle des autres services. Dans le cas de la propriété $CW - TCS$ 1, l'oracle a mis en évidence une violation pour l'approximation *"Tout Environnement"* , alors qu'aucune violation de $CW - TCS$ 1 n'est détectée avec les autres approximations.

Les résultats obtenus avec la propriété $CW - TCS$ 2 sont particulièrement intéressants. Nous rappelons que cette propriété correspond à la première généralisation de $CW - TCS$ 1, et que nous avons exprimé cette généralisation d'une manière différente des cinq autres paires de services. Mis à part une curiosité qui a conduit à l'échec de la construction du

Tab. 6.21: Résultats pour $CW - TCS$ 2

Approximation *"Tout Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	10	10	8	9
Instantanée	1	0	0	3	4
Intersection	1	0	0	3	4
Union	1	5	5	0	3
Paresseuse	3	7	7	4	0
DS/S	15 / 38	24 / 1561	24 / 1561	22 / 1971	21 / 24835

Approximation *"Sans Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	10	10	7	8
Instantanée	1	0	0	2	2
Intersection	1	0	0	2	2
Union	2	6	6	0	5
Paresseuse	6	9	9	8	0
DS/S	15 / 38	24 / 1561	24 / 1561	20 / 103	17 / 940

simulateur d'environnement pour les stratégies d'intersection, d'union et paresseuse dans le cadre de l'approximation *"Semi Environnement"* , les résultats obtenus pour les deux autres approximations montrent que seuls 15 à 20% des états atteints sont suspects. Ce résultat tend à montrer que notre première façon de généraliser les propriétés rendait relativement facile la conservation de l'environnement dans un état suspect. Notons cependant le cas particulier de la stratégie paresseuse avec l'approximation *"Tout Environnement"* : plus des deux-tiers des états atteints sont suspects. Comparés aux résultats des autres stratégies, nous montrons ici que la stratégie paresseuse est capable d'atteindre très efficacement les états suspects.

Les résultats obtenus avec deux dernières propriétés concernant une communication normale entre deux abonnés sont conformes à ceux obtenus pour les systèmes précédents à l'exception des résultats de la propriété $CW - TCS$ 5 obtenus avec la stratégie intersection et l'approximation *"Sans Environnement"* . Alors que la stratégie intersection a toujours donné de faibles résultats, ici elle obtient le meilleur résultat: 40 états suspects distincts atteints.

6.6 Bilan de l'expérience

6.6.1 Quelques remarques particulières aux stratégies et approximations

D'une manière générale, l'ensemble des résultats (résumés pour l'approximation *"Tout Environnement"* dans le tableau ci-dessous) que nous avons recueilli au cours de cette évaluation montre que:

- Les stratégies d'union et paresseuse permettent d'atteindre un plus grand nombre d'états suspects, ce qui représente un gain par rapport aux stratégies précédemment développées.
- La stratégie paresseuse semble être celle qui tire le plus d'avantages à prendre en compte les contraintes d'environnement dans le calcul des vecteurs d'entrées menant vers des états suspects.

TAB. 6.22: Résultats pour $CW - TCS$ 4

Approximation *"Tout Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	36	26
Instantanée	0	0	0	36	26
Intersection	0	0	0	36	26
Union	0	0	0	0	0
Paresseuse	0	0	0	10	0
DS/S	0 / 0	0 / 0	0 / 0	36 / 18043	26 / 24782

Approximation *"Semi Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	35	27
Instantanée	0	0	0	35	27
Intersection	0	0	0	35	27
Union	0	0	0	0	0
Paresseuse	0	0	0	8	0
DS/S	0 / 0	0 / 0	0 / 0	35 / 15651	27 / 24639

Approximation *"Sans Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	0	0	35	27
Instantanée	0	0	0	35	27
Intersection	0	0	0	35	27
Union	0	0	0	0	0
Paresseuse	0	0	0	8	0
DS/S	0 / 0	0 / 0	0 / 0	35 / 15651	27 / 24654

TAB. 6.23: Résultats pour $CW - TCS$ 5

Approximation *"Tout Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	2	2	11	6
Instantanée	4	0	0	13	6
Intersection	4	0	0	13	6
Union	0	0	0	0	6
Paresseuse	13	11	11	24	0
DS/S	27 / 186	25 / 1736	25 / 1736	38 / 22132	20 / 24954

Approximation *"Semi Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	2	2	11	6
Instantanée	4	0	0	13	6
Intersection	4	0	0	13	6
Union	0	0	0	0	6
Paresseuse	13	11	11	24	0
DS/S	27 / 186	25 / 1736	25 / 1736	38 / 22132	20 / 24954

Approximation *"Sans Environnement"* :

	Témoin	Instantanée	Intersection	Union	Paresseuse
Témoin	0	2	18	11	6
Instantanée	4	0	16	13	6
Intersection	5	1	0	14	6
Union	0	0	16	0	6
Paresseuse	13	11	26	24	0
DS/S	27 / 186	25 / 1736	40 / 1780	38 / 22132	20 / 24954

- La stratégie d'intersection semble être au mieux aussi efficace que la stratégie instantanée. Ceci s'explique par la difficulté d'avoir une propriété pour laquelle on peut trouver un vecteur d'entrées qui puisse mener à un état suspect pour tous les instants explorés par LUTESS.
- Le choix de l'approximation semble, *a priori*, ne pas modifier de façon très significative le nombre d'états suspects atteints; en revanche l'analyse des verdicts de l'oracle montre que le nombre de violations de la propriété augmente avec la précision de l'approximation.
- Nous n'avons pas donné systématiquement le nombre de violations observées pour une propriété et une stratégie donnée, mais uniquement là où nous avons observé une variation du nombre de violations observées en fonction de la stratégie et/ou de l'approximation utilisée. Cependant, pour mieux interpréter cette information, il faudrait la relier à l'état suspect qui a produit la violation. Avec une telle information, nous pourrions connaître quel est le sous-ensemble des états suspects atteints où une violation a été observée; en particulier, il serait intéressant de classer les états suspects en trois catégories:
  1. ceux qui n'ont jamais provoqué de violation de la propriété testée;
  2. ceux qui ont parfois provoqué de violation de la propriété testée;
  3. ceux qui ont toujours provoqué de violation de la propriété testée.

  En revanche, cette mesure a le défaut de dépendre des fautes présentes dans le système testé.

États suspects atteints.					
Propriétés	Témoin	Instantanée	Intersection	Union	Paresseuse
CFBL - CFBL					
$(CFBL-CFBL\ 1)$	1/1	2/21	0/0	2/666	2/585
$(CFBL-CFBL\ 2)$	Pas de résultat				
$(CFBL-CFBL\ 3)$	Pas de résultat				
$(CFBL-CFBL\ 4)$	Pas de résultat				
$(CFBL-CFBL\ 5)$	Pas de résultat				
CELL - CFBL					
$(CELL-CFBL\ 1)$	0/0	0/0	0/0	2/90	2/85
$(CELL-CFBL\ 2)$	0/0	0/0	0/0	0/0	0/0
$(CELL-CFBL\ 3)$	Pas de résultat				
$(CELL-CFBL\ 4)$					
$(CELL-CFBL\ 5)$	29/523	33/20560	33/20560	27/493	39/22602
$(CELL-CFBL\ 6)$	Pas de résultat				
$(CELL-CFBL\ 7)$	0 / 0	0 / 0	0 / 0	36 / 18043	14 / 24782
$(CELL-CFBL\ 8)$	27 / 186	25 / 1736	25 / 1736	38 / 22132	20 / 24954
CELL - CW					
$(CELL-CW\ 1)$	8/18	9/510	8/18	9/953	11/899
$(CELL-CW\ 2)$	25/537	29/20642	29/20642	28/520	32/22592
(Suite des résultats à la page suivante...)					

Propriétés	Témoin	Instantanée	Intersection	Union	Paresseuse
$(CELL - CW\ 3)$	Pas de résultat				
$(CELL - CW\ 4)$	0/0	0/0	0/0	2/74	3/70
$(CELL - CW\ 5)$	0/0	0/0	0/0	0/0	0/0
$(CELL - CW\ 6)$	Pas de résultat				
$(CELL - CW\ 7)$	0/0	0/0	0/0	35/17981	30/24918
$(CELL - CW\ 8)$	28/247	31/2031	32/2031	32/22161	31/24975
CELL - TWC					
$(CELL - TWC\ 1)$	0/0	0/0	0/0	2/71	2/84
$(CELL - TWC\ 2)$	0/0	0/0	0/0	0/0	0/0
$(CELL - TWC\ 3)$	Pas de résultat				
$(CELL - TWC\ 4)$	0/0	0/0	0/0	36/17981	25/24918
$(CELL - TWC\ 5)$	28/247	32/2031	32/2031	32/22161	31/24975
CND - TCS					
$(CND - TCS\ 1)$	8/31	9/421	8/31	8/26	15/7207
$(CND - TCS\ 2)$	28/40	29/7330	29/20070	28/458	39/22632
$(CND - TCS\ 3)$	Pas de résultat				
$(CND - TCS\ 4)$	0/0	0/0	0/0	25/17853	19/24972
$(CND - TCS\ 5)$	28/266	30/1521	30/1521	34/21829	17/24977
CW - TCS					
$(CW - TCS\ 1)$	0/0	0/0	0/0	2/89	2/73
$(CW - TCS\ 2)$	15/38	24/1561	24/1561	22/1971	21/24835
$(CW - TCS\ 3)$	Pas de résultat				
$(CW - TCS\ 4)$	0/0	0/0	0/0	36/18043	26/24782
$(CW - TCS\ 5)$	27/186	25/1736	25/1736	38/22132	20/24954

6.6.2 Bilan général de l'expérience

Les résultats que nous avons obtenus au cours de notre expérimentation montrent que le nombre d'états suspects atteints par les nouvelles stratégies est plus important que le nombre d'états suspects atteints par la sélection purement aléatoire des vecteurs d'entrées ou par la première version du test guidé par les propriétés de sûreté [OP94a].

Nous pouvons également déduire de ces résultats une certaine complémentarité des différentes stratégies: en effet, toutes les stratégies ne couvrent pas exactement le même ensemble d'états suspects. Il semble donc tout à fait pertinent d'utiliser l'ensemble des stratégies disponibles au cours d'une campagne de test.

Cette expérience montre également qu'il est d'autant plus difficile de tenir compte de l'environnement dans le test guidé par les propriétés de sûreté que la propriété testée se généralise. Avec la puissance de calcul dont nous disposions[11], l'approximation *"Sans Environnement"* est celle qui permet de couvrir le plus grand nombre de propriétés.

Les propriétés que nous avons utilisées au cours de cette expérimentation peuvent être classées selon deux catégories [PV03]:

- Les propriétés fortement guidées qui décrivent un à un les d'événements d'une séquence

[11] Pentium III 733 MHz sous Linux, avec 512 MO RAM.

précise qui doit mener à un état suspect sans qu'aucun autre événement ne puisse interférer avec cette séquence.

- Les propriétés générales qui autorisent plus ou moins largement le simulateur d'intercaler des événements entre ceux qui vont mener vers un état suspect.

Du point de vue du test aléatoire, l'utilisation des propriétés générales permet de ne pas favoriser une configuration[12] plutôt qu'une autre. En effet, il n'existe en général aucune raison objective de privilégier une configuration de test plutôt qu'une autre. Cependant, plus la propriété va être générale, plus il sera difficile d'évaluer si toutes ou la plupart des configurations critiques décrites par la propriété ont été testées. De plus, nous pouvons nous trouver face à des propriétés d'une complexité trop grande pour nos moyens de calcul.

L'utilisation de propriétés fortement guidées peut être critiquée sur le fait que l'on teste une seule situation précise (et prédéfinie à l'avance). Cependant, rendre une propriété générale plus précise peut s'avérer indispensable. En particulier, si une violation de la propriété générale est détectée, il est intéressant d'exhiber une propriété fortement guidée qui mettent en évidence cette violation dans un cas particulier. Une telle présentation est souvent plus facile à comprendre que le cas général.

Enfin, cette expérimentation nous a également montré que les résultats obtenus dépendent fortement de la façon dont sont écrites les propriétés. En particulier, il arrive fréquemment que le spécifieur n'intègre l'effet de la totalité des contraintes d'environnement quand il écrit ses propriétés. En conséquence, sous les contraintes d'environnement, une propriété peut être inviolable. Nous nous sommes trouvés dans ce cas pour toutes les propriétés pour lesquelles aucun état suspect n'a été atteint. Nous avons notamment été interpellés par les résultats obtenus avec les propriétés générales des cinq premières paires de services. En effet, sur le nombre total d'états atteints plus des deux tiers étaient suspects. Devant un tel succès, nous avons décidé de modifier la façon d'exprimer la propriété générale de la sixième paire de service afin de s'assurer que le grand nombre d'états suspects atteints n'était pas dû à une propriété qui décrirait la plupart des états du simulateur comme suspect. Avec la nouvelle forme de propriété générale, nous avons constaté que le nombre total d'états suspects représente de l'ordre de 15 à 20% du nombre total d'états atteints sauf pour le cas de la stratégie paresseuse où pratiquement tous les états atteints sont suspects (approximation *"Tout Environnement"*). Ces derniers résultats montrent que la stratégie paresseuse est capable d'atteindre des états suspects très efficacement. Ils confirment également notre intuition à propos de la première façon de généraliser les propriétés (celle utilisée pour les cinq premiers systèmes): le grand nombre d'états suspects obtenus pour ces propriétés est dû au fait que ces dernières définissent une grande partie des états de l'environnement comme suspects.

6.6.3 Guide d'écriture des propriétés de sûreté

Nous n'avons pas pour but de définir exhaustivement toutes les façons d'écrire une propriété de sûreté, mais de décrire comment nous pouvons envisager la construction des propriétés sous une forme générale ou fortement guidée.

[12] Nous désignons par configuration la façon d'ordonner les événements qui peuvent mener dans un état suspect de l'environnement.

Lors de l'expérimentation, la plupart des propriétés utilisées étaient écrites sous la forme d'une implication (propriétés spécifiques à chaque paire de services). Toutes celles exprimées autrement étaient écrites avec l'opérateur *Always_From_To* (propriétés testées par toutes les paires de services). Toutes ces propriétés sont constituées d'au moins deux parties:

- description de la ou des situations suspectes;
- réactions attendues du logiciel lorsqu'une situation suspecte est atteinte.

Nos stratégies analysent les propriétés de sûreté puis déduisent à partir de la description des situations suspectes quels sont les états suspects du simulateur et les événements à générer pour atteindre ces états.

Nous proposons d'écrire les propriétés (générales ou fortement guidées) à partir des opérateurs ci-dessous[13]. Nous désignons par *description* la partie de la propriété qui définit les situations suspectes, et par *réaction* les comportements attendus du logiciel lorsqu'une des situations suspectes est atteinte. Nous avons également besoin, pour certaines formes de propriétés, d'une partie *fin* définissant la terminaison des situations suspectes.

- *description* $\Rightarrow$ *réaction*: l'implication est particulièrement bien adaptée à nos stratégies. La partie *description* contrôlée par l'environnement doit être mise à *vrai* pour pouvoir éventuellement observer une mauvaise *réaction* du système. L'objectif de nos stratégies, c'est de favoriser la séquence d'événements qui permet de rendre régulièrement la partie *description* à *vrai* .
- *Always_Since*(*réaction*, *description*): ce type de propriété impose que la partie *réaction* de la propriété contrôlée par le système sous test reste invariablement à *vrai* dès que la partie *description* a été mise à *vrai* . Notons qu'avec une telle propriété, dès qu'un état suspect s est atteint, tous les autres états, atteint à partir de s, sont également suspects.
- *Always_From_To*(*réaction*, *description*, *fin*): cette dernière propriété est une amélioration de la précédente. Au lieu de rendre tous les états suspects dès que la partie *description* est mise à *vrai* , seul les états entre l'instant où la partie *description* et la partie *fin* ont été mises à *vrai* sont suspects. Cependant, nos stratégies vont avoir tendance à éviter de rendre *vrai* la partie *fin* pour rester dans un état suspect.
- *Once_Since*(*réaction*, *description*): des propriétés écrites avec cet opérateur permettraient de décrire une sorte de comportement préventif du logiciel à une situation suspecte: nous devons observer au moins une fois la partie *réaction* avant la partie *description*. Cet opérateur nous servira plus particulièrement pour la description de la situation suspecte comme nous le montrerons dans la suite de ce paragraphe.
- *Once_From_To*(*réaction*, *description*, *fin*): une propriété écrite avec cet opérateur impose au logiciel de produire la *réaction* entre les instants où la *description* et la *fin* ont été évaluées à *vrai* . Dans ce cas, nos stratégies vont tenter de rendre *vrai* la partie *description* suivi de la partie *fin* avant que le logiciel n'ait pu produire la partie *réaction* entre les deux.

[13] La liste d'opérateurs présentée ici n'a pas vocation à être exhaustive, mais elle correspond aux opérateurs de logique temporelle couramment utilisés avec LUSTRE et qui semblent bien adaptés à nos stratégies aux vues des résultats obtenus.

Remarque 6.3 *L'expression* LUSTRE *de chacun des opérateurs que nous venons de présenter est donnée en annexe B.1.*

La différence entre les propriétés générales et fortement guidées se situe au niveau de l'écriture de la description de la situation suspecte. Dans le cas des propriétés fortement guidées, nous décrivons une suite d'événements consécutifs qui mène vers une situation suspecte de la manière suivante:

$$\textbf{pre pre pre } ev_1 \wedge \textbf{pre pre } ev_2 \wedge \textbf{pre } ev_3.$$

À l'opposé, une propriété générale ne va pas imposer aux événements d'être immédiatement consécutifs. Le but d'une propriété générale est de donner le plus de degré de liberté possible à la génération des cas de test.

$$once_since(ev_1, ev_2) \wedge once_since(ev_2, ev_3) \wedge ev_3.$$

En décrivant la situation suspecte de cette façon, nous indiquons que nous souhaitons que l'événement ev_3 soit précédé de ev_2 lui-même précédé de ev_1 sans les contraindre à être rigoureusement consécutifs. En d'autres termes, le simulateur d'environnement a la possibilité d'intercaler d'autres événements entre ev_1 et ev_2 ou ev_2 et ev_3.

De plus, si nous supposons que les événements ev_1 et ev_2 peuvent être engendrés dans n'importe quel ordre, mais avant l'événement ev_3 alors nous pouvons décrire cet ensemble de situations suspectes par:

$$once_since(ev_1, ev_3) \wedge once_since(ev_2, ev_3) \wedge ev_3.$$

Une propriété de sûreté qui utiliserait cette description des situations suspectes permettrait à LUTESS de générer des séquences du type $(..., ev_1, ..., ev_2, ..., ev_3, ...)$ ou $(..., ev_2, ..., ev_1, ..., ev_3, ...)$.

Chapitre 7

Vers un guidage plus précis

7.1 Motivations

Nous avons proposé au chapitre 5 une solution accompagnée de plusieurs stratégies qui permet de prendre en compte l'information contenue dans les propriétés de sûreté pour générer des vecteurs d'entrées facilitant la violation de ces propriétés.

Nous avons également vu que les ensembles de vecteurs d'entrées pertinents que nous définissons dans la liste $\Xi_n(s_{t+0})$ correspondent toujours à des surapproximations des ensembles de vecteurs d'entrées qui peuvent effectivement mener à une violation des propriétés de sûreté, à cause du simple fait que nous considérons le logiciel sous test comme une boîte noire. Étant donné que nous ne disposons pas d'information sur les valuations du vecteur de sorties généré par le logiciel, nous devons toutes les considérer (y compris celles qui ne sont jamais produites par ce dernier) pour pouvoir calculer les ensembles de vecteurs d'entrées pertinents.

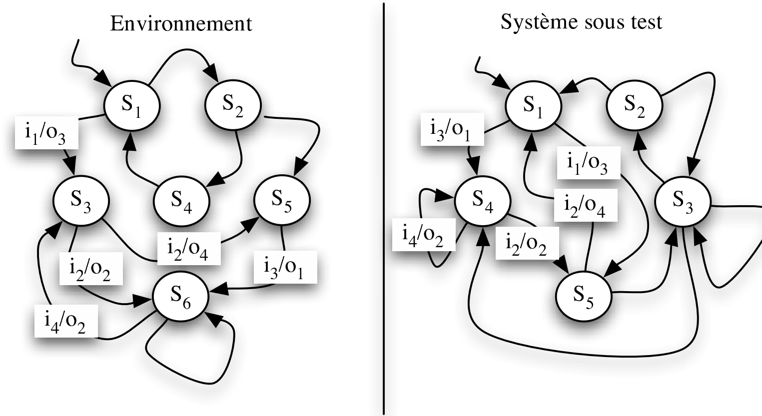
Parmi les trois approximations définies pour le calcul des vecteurs d'entrées pertinents, l'approximation *"Tout Environnement"* est la plus précise dans le sens où elle intègre l'ensemble des contraintes d'environnement à toutes les étapes du calcul. Pour améliorer encore la précision du calcul il est nécessaire d'introduire des informations sur les valeurs des vecteurs de sorties produits par le système sous test, c'est à dire sur son comportement.

Cependant nous souhaitons conserver les principes sur lesquels repose Lutess: en particulier nous souhaitons conserver l'indépendance de Lutess vis-à-vis de la réalisation du logiciel. En conséquence, nous écarterons les approches structurelles qui auraient pu apporter une connaissance sur les réactions de ce dernier.

Nous nous proposons ici d'étudier comment prendre en compte des informations sur les valeurs des vecteurs de sorties et quels pourraient en être les avantages et les inconvénients. Les paragraphes 7.2 et 7.3 présentent deux approches pour prendre en compte ces informations. Nous verrons au paragraphe 7.4.1 comment nous pouvons combiner ces deux propositions dans le but de définir un critère d'arrêt pour une séquence de test. Enfin nous dresserons un bilan de cette étude au paragraphe 7.4.

Notons que les idées que nous développons ici n'ont fait l'objet d'aucune intégration à Lutess. Il s'agit essentiellement de donner quelques ouvertures pour la poursuite du développement de l'outil.

FIG. 7.1: États environnement / système sous test



7.2 Apprentissage par l'expérience

7.2.1 Principe

L'idée est d'utiliser l'information accumulée au cours d'une séquence de test pour que LUTESS "*apprenne*" à connaître les réactions du logiciel sous test. Concrètement, il faut être capable de retrouver pour chaque couple (s, i) de l'environnement quelles ont été les sorties déjà calculées par le logiciel sous test.

Nous suggérons ici que nous pouvons avoir plusieurs vecteurs de sorties différents pour un couple (s, i) de l'environnement. Or, le principe déterministe des logiciels réactifs synchrones semblerait plaider pour un unique vecteur de sorties associé au couple (s, i). Cependant, il faut bien distinguer les états de l'environnement du couple (s, i), des états du système sous test que nous ne connaissons pas (test en boîte noire). En conséquence, lorsque l'environnement revient dans un état précédemment atteint, rien ne nous garantit que le logiciel sous test est lui aussi revenu dans le même état que précédemment: de ce fait, il peut y avoir plusieurs vecteurs de sorties possibles pour un couple (s, i) de l'environnement. Cette situation est illustrée par la figure 7.1: nous y avons représenté uniquement la séquence des couples d'entrées/sorties qui nous intéresse. À la suite du couple d'entrées/sorties (i_1, o_3), l'environnement atteint l'état S_3 tandis que le système se trouve dans l'état S_5. En poursuivant cette séquence d'entrées/sortie par

$$(i_2, o_4), (i_3, o_1), (i_4, o_2)$$

l'environnement se retrouve dans l'état S_3 pour la seconde fois, alors que le système se trouve dans l'état S_4. Ainsi, lorsque l'environnement génère à nouveau l'entrée i_2, le système répond cette fois par la sortie o_2 (lors du premier passage sur l'état S_3 de l'environnement, le système avait répondu par la sortie o_4).

La réalisation d'une solution apprenant les réactions du logiciel au fur et à mesure de l'avancement des séquences de test nous amène à considérer les deux points suivants:

- nous devons modifier le simulateur d'environnement pour lui permettre de construire la relation qui associe à chaque valeur du couple état/vecteur d'entrées rencontrée les

valeurs des vecteurs de sorties observées. En d'autres termes, nous voulons pouvoir accéder facilement à l'ensemble de toutes les sorties générées pour chaque couple (s, i) de l'environnement que nous avons atteint;

- nous devons également revoir l'ensemble des techniques introduites au chapitre 5 qui permettent de calculer les vecteurs d'entrées pertinents afin de prendre en compte l'information sur les vecteurs de sorties calculés par le logiciel.

7.2.2 Construction des informations sur les vecteurs de sorties

Les informations sur les valuations des vecteurs de sorties produites par le logiciel sous test peuvent être représentées par la relation $P_{out} \subseteq S \times V_I \times V_O$ définie de la façon suivante:

Définition 7.1 *Soit M_{saf} un simulateur guidé par les propriétés de sûreté Ps.*

$$M_{saf} = (S, s_{init}, I, O, f, env, part_{env}(V_{I_{env}}, Saf_{Ps}), sel_{env})$$

P_{out} représente l'ensemble de tous les triplets $(s, i, o) \in S \times V_I \times V_O$ déjà observés au cours d'une séquence de test.

- *À l'instant initial $t = 0$, nous avons:*

$$P_{out_0} = \text{vrai}$$

- *À chaque instant t (avec $t > 0$) de la séquence de test en cours de génération, nous ajoutons le triplet (s_t, i_t, o_t) observé à ceux déjà observés aux instants précédents:*

$$P_{out_t} = [(s_t \wedge i_t) \Rightarrow o_t] \wedge P_{out_{t-1}}$$

Un simulateur d'environnement étendu par apprentissage et guidé par les propriétés de sûreté est construit sur la base du simulateur guidé par les propriétés de sûreté[1]: en particulier, les deux simulateurs possèdent le même ensemble d'états, le même état initial et la même fonction de transition. La différence entre les deux simulateurs se situe au niveau de la construction des vecteurs d'entrées pertinents qui doit tenir compte des informations apportées par P_{out}.

Définition 7.2 *Soit M_{app} un simulateur d'environnement étendu par apprentissage et guidé par les propriétés de sûreté:*

$$M_{app} = (S, s_{init}, I, O, f, env, part_{env}(V_{I_{env}}, Saf_{Ps}, P_{out}), sel_{env})$$

- *S est l'ensemble des états du simulateur et s_{init} l'état initial;*
- *I (resp. O) est l'ensemble des variables d'entrées (resp. de sorties);*
- *f est la fonction de transition;*
- *env représente les contraintes d'environnement;*

[1] Voir définition 5.3.

- $V_{val} = part_{env}(V_{I_{env}}, Saf_{Ps}, P_{out})$ *est une méthode qui détermine l'ensemble* V_{val} *où sera choisi le vecteur d'entrées en fonction des vecteurs d'entrées valides* ($V_{I_{env}}$)*, des propriétés de sûreté* (Saf_{Ps}) *et des informations sur les comportements du logiciel* (P_{out})*;*
- $sel_{env}(V_{val})$ *est la méthode qui permet de choisir le vecteur d'entrées à soumettre au logiciel sous test.*

Dans un simulateur étendu par apprentissage, un instant d'une séquence de test se déroule de la façon suivante: au cours d'un premier "demi-instant", un vecteur d'entrées est généré puis soumis au logiciel sous test; au cours du second "demi-instant" (lorsque le vecteur de sorties a été reçu), le simulateur calcule la nouvelle valeur de P_{out}, puis détermine l'état suivant. Pour une séquence de test d'une longueur L (la valeur de L étant fourni par l'utilisateur), le fonctionnement de ce simulateur est représenté par l'algorithme 7.1:

★ **Algorithme 7.1** : *Principe du simulateur d'environnement étendu par apprentissage*

Pour $i := 0$ **jusqu'à** L **faire**
  $V_i := genereVecteurEntrees(V_s)$;
  $emettreVecteurEntree(V_i)$;
  $V_o := attendreVecteurSortie$;
  $P_{out} := comportementSysteme(P_{out}, V_s, V_i, V_o)$;
  $V_s := etatSuivant(V_s, V_i, V_o)$;
Fin pour;

où V_s, V_i et V_o représentent respectivement le vecteur des variables d'états, le vecteur d'entrées et le vecteur de sorties (du logiciel sous test).

7.2.3 Adaptation du calcul des vecteurs d'entrées pertinents

Afin de prendre en compte l'information sur les vecteurs de sorties construite au cours de la séquence de test, nous modifions la génération des vecteurs d'entrées guidée par les propriétés de sûreté de la façon suivante:

Définition 7.3 *Soit* M *un simulateur d'environnement étendu par apprentissage, guidé par les propriétés de sûreté:*

$$M_{app} = (S, s_{init}, I, O, f, env, part_{env}(V_{I_{env}}, Saf_{Ps}, P_{out}), sel_{env})$$

L'ensemble $V_{I_k}(s_{t+0})$ *des vecteurs d'entrées pertinents pouvant mener à une violation des propriétés de sûreté à l'issue d'un chemin de longueur* k *est:*

$$V_{I_k}(s_{t+0}) = \{i_{t+0} \in V_I | \exists s_{t+k} \in S_k(s_{t+0}, i_{t+0}), \exists (i_{t+k}, o_{t+k}) \in V_I \times V_O, \\ (\neg Saf_{Ps}(s_{t+k}, i_{t+k}, o_{t+k}) \uparrow env(i_{t+k}, s_{t+k})) \uparrow P_{out}(s_{t+k}, i_{t+k}, o_{t+k})\}$$

Nous rappelons que la notation $A \uparrow B$ *signifie "*A sachant que B*" (voir §5.4).*

Notons que dans la définition 7.2, la construction de l'ensemble P_{out} fait apparaître des indices temporels, alors que l'utilisation de l'ensemble P_{out} n'en tient pas compte. En fait, la construction de l'ensemble *out* à chaque instant utilise l'ensemble P_{out} construit à l'instant précédent alors que nous utilisons uniquement la valeur courante de l'ensemble P_{out} lorsque nous calculons l'ensemble des vecteurs d'entrées pertinents pour un instant t donné.

Comme pour le calcul des vecteurs d'entrées pertinents, nous devons également modifier le calcul de l'ensemble des états accessibles afin de le rendre plus précis en utilisant la connaissance partielle des comportements du système sous test:

Définition 7.4 *Soit un simulateur étendu par apprentissage M, guidé par les propriétés de sûreté*

$$M_{app} = (S, s_{init}, I, O, f, env, part_{env}(V_{I_{env}}, Saf_{Ps}, P_{out}), sel_{env}),$$

$S_k(s_{t+0})$, l'ensemble des états de M atteignables par un chemin de longueur k, est calculé par induction de la manière suivante:

$$\begin{cases} S_0(s_{t+0}, i_{t+0}) & = & \{s_{t+0}\} \\ S_1(s_{t+0}, i_{t+0}) & = & \{s \in S | \exists o_{t+0} \in V_O, \\ & & \quad s = (f(s_{t+0}, i_{t+0}, o_{t+0}) \uparrow env(i_{t+0}, s_{t+0})) \uparrow \\ & & \quad P_{out}(s_{t+0}, i_{t+0}, o_{t+0})\} \\ S_k(s_{t+0}, i_{t+0}) & = & \{s \in S | \exists s_{t+k-1} \in S_{k-1}(s_{t+0}, i_{t+0}), \exists (i_{t+k_1}, o_{t+k-1}) \in V_I \times V_O, \\ & & \quad s = (f(s_{t+k-1}, i_{t+k-1}, o_{t+k-1}) \uparrow env(i_{t+k-1}, s_{t+k-1})) \uparrow \\ & & \quad P_{out}(s_{t+k-1}, i_{t+k-1}, o_{t+k-1})\} \end{cases}$$

Afin d'illustrer le simulateur étendu par apprentissage et guidé par les propriétés de sûreté, nous allons reprendre l'exemple introduit au paragraphe 5.4. Nous rappelons que nous considérons un logiciel ayant quatre entrées (i_1, i_2, i_3 et i_4) et une sortie (o_1). L'environnement de notre programme était constitué de deux contraintes:

$$\begin{array}{ll} (1) & after(i_2) \Rightarrow \neg i_1 \\ (2) & \neg(false \rightarrow \textbf{pre}\ i_2) \Rightarrow \neg i_3 \end{array}$$

et nous souhaitions valider la propriété de sûreté suivante:

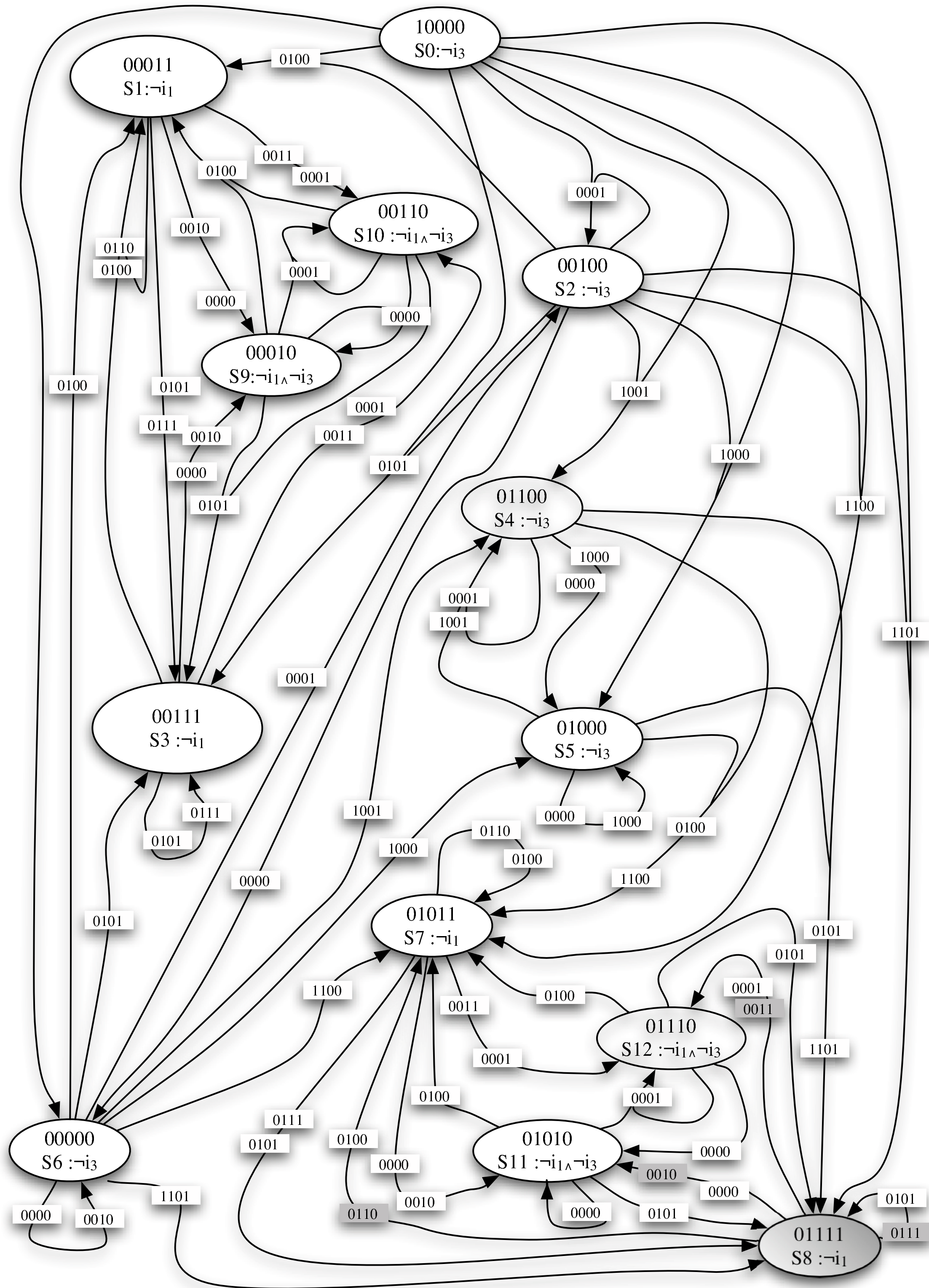
$$(after(i_1) \wedge (false \rightarrow \textbf{pre}\ i_4 \wedge i_3)) \Rightarrow o_1$$

Enfin, nous redonnons sur la figure 7.2 la représentation du simulateur d'environnement où l'état grisé correspond à l'état suspect déterminé par l'approximation *"Tout Environnement"*.. Supposons que nous utilisons la stratégie paresseuse pour générer les vecteurs d'entrées. Lorsque nous nous trouvons dans l'état initial, nous avons $P_{out} = vrai$ et nous devons choisir un vecteur d'entrées parmi ceux représentés dans le tableau 7.1.

Dans l'état initial, il n'existe pas de vecteur d'entrées menant à un état suspect par un chemin de longueur 0: c'est à dire que l'état initial n'est pas suspect! La stratégie paresseuse va alors regarder s'il est possible d'atteindre un état suspect par un chemin de longueur 1 et trouve le vecteur d'entrée (1101). Supposons que LUTESS génère ce vecteur et que le logiciel sous test retourne $o_1 = faux$. Dans ce cas, nous mettons à jour comme ci-dessous:

$$\begin{array}{lll} P_{out} & = & [((sv_0 \wedge \neg sv_1 \wedge \neg sv_2 \wedge \neg sv_3 \wedge \neg sv_4) \wedge (i_1 \wedge i_2 \wedge \neg i_3 \wedge i_4)) \Rightarrow \neg o_1] \wedge \\ & & [vrai] \end{array}$$

Fig. 7.2: Simulateur d'environnement



TAB. 7.1: Vecteurs d'entrées pertinents calculés par *"Tout Environnement"* ($P_{out} = vrai$).

Chemin de longueur 0	**if** $\neg EstInitial(s) \wedge after(i_1) \wedge$ **pre** $i_4 \wedge$ **pre** i_2 **then** i_3 **else** $false$
Chemin de longueur 1	**if** $EstInitial(s)$ **then** $i_1 \wedge i_2 \wedge i_4$ **else if** $after(i_1)$ **then** $i_2 \wedge i_4$ **else if** $after(i_2)$ **then** $false$ **else** $i_1 \wedge i_2 \wedge i_4$
Chemin de longueur 2	**if** $EstInitial(s)$ **then** $i_1 \vee \neg i_2$ **else if** $after(i_1)$ **then** $true$ **else if** $after(i_2)$ **then** $false$ **else** $i_1 \vee \neg i_2$
Chemin de longueur 3	**if** $EstInitial(s)$ **then** $i_1 \vee \neg i_2$ **else if** $after(i_1)$ **then** $true$ **else if** $after(i_2)$ **then** $false$ **else** $i_1 \vee \neg i_2$

Maintenant, l'environnement se trouve dans l'état S_8. Comme il n'existe pas d'information dans P_{out} sur les comportements du système à partir de l'état S_8 de l'environnement, les vecteurs d'entrées pertinents restent inchangés par rapport à ceux représentés dans le tableau 7.1. LUTESS s'aperçoit que l'état S_8 est suspect et qu'il doit engendrer un de ces quatre vecteurs d'entrées: 0010, 0011, 0110, 0111 où l'entrée $i_3 = vrai$. Supposons que LUTESS choisisse le vecteur 0111 et que le logiciel réponde par $o_1 = vrai$, P_{out} est mise à jour, Ps est évaluée à *vrai* et nous restons dans l'état S_8 de l'environnement:

$$\begin{aligned} P_{out} \;=\; & [((\neg sv_0 \wedge sv_1 \wedge sv_2 \wedge sv_3 \wedge sv_4) \wedge (\neg i_1 \wedge i_2 \wedge i_3 \wedge i_4)) \Rightarrow o_1] \wedge \\ & [((sv_0 \wedge \neg sv_1 \wedge \neg sv_2 \wedge \neg sv_3 \wedge \neg sv_4) \wedge (i_1 \wedge i_2 \wedge \neg i_3 \wedge i_4)) \Rightarrow \neg o_1] \wedge \\ & [vrai] \end{aligned}$$

Cette fois, LUTESS va construire l'ensemble des vecteurs d'entrées pertinents sachant que dans l'état S_8 et en produisant le vecteur d'entrées 0111 le logiciel produit o_1 et donc $Ps = vrai$. Ceci a pour conséquence de supprimer le vecteur 0111 des vecteurs d'entrées pertinents.

7.3 Utilisation d'un nouvel opérateur

7.3.1 Principe

Une seconde solution propose d'introduire un nouvel opérateur, **specification**, où le testeur pourrait spécifier partiellement le comportement du logiciel sous test. Nous garderons une structure syntaxique identique à celle des opérateurs **environment** et **safety**:

★ **Nœud 7.1** : *structure syntaxique d'un nœud de test avec l'opérateur* **specification**.

```
testnode env(sorties du logiciel sous test)
returns (entrées du logiciel sous test)
var
   variables locales
let
   environment(C_1, ... , C_n);
   safety(Ps_1, ... , Ps_n);
   specification(Sp_1, ..., Sp_n);
   définition des variables locales
tel
```

Nous retrouvons la structure syntaxique classique d'un nœud de test, où les Sp_k (avec $k \in [1..n]$) sont des expressions booléennes mettant en relation les vecteurs d'entrées et de sorties du logiciel.

D'un point de vue sémantique, nous nous trouvons dans une situation similaire à celle de l'opérateur **safety**: lorsque LUTESS choisit le vecteur d'entrées, il est impossible d'évaluer les valeurs courantes des Sp_k étant donné que la valeur du vecteur de sorties réellement calculée par le logiciel n'est pas encore connue. Nous associons à l'opérateur **specification** la règle suivante[2]:

- compatibilité avec l'opérateur *specification*.

$$\frac{\sigma, \sigma' \vdash Sp_1 \mid out \wedge ... \wedge Sp_n \mid out}{\sigma, \sigma' \vdash specification(Sp_1, ..., Sp_r)}$$

Bien que les opérateurs **specification** et **safety** soient définis par une règle de sémantique équivalente, il est nécessaire de bien distinguer les deux ensembles de contraintes qu'ils contiennent:

- les contraintes définies par l'opérateur **safety** représentent des propriétés que l'on souhaite tester afin de nous assurer que le logiciel fonctionne correctement;
- alors que les contraintes définies par l'opérateur **specification** représentent une spécification partielle du logiciel pour laquelle nous sommes *a priori* "sûrs" des réponses obtenues.

L'introduction de l'opérateur **specification** nous amène à modifier le simulateur d'environnement guidé par les propriétés de sûreté. De la même façon que les propriétés de sûreté[3], l'expression des spécifications du logiciel est faite à partir d'expressions booléennes LUSTRE. L'expression P_{sp} peut donc être représentée comme un automate (voir définition 7.5).

Définition 7.5 *Soit P_{sp} les spécifications partielles du logiciel telles que:*

$$P_{sp} = (S_{sp}, s_{init_{sp}}, I, O, f_{sp}, Spec)$$

- *S_{sp} représente l'ensemble des états des spécifications;*

[2] Il convient d'ajouter cette règle à l'ensemble des règles de sémantique d'un nœud de test que nous avons défini au paragraphe 4.2.1 et à celle du paragraphe 5.2.2 définissant l'opérateur **safety**.

[3] Voir §5.3.3.

- $_{init_{sp}}$ *est l'état initial;*
- *I (resp. O) représente l'ensemble des variables d'entrées (resp. de sorties) du logiciel sous test;*
- $f_{sp} : S_{sp} \times V_I \times V_O \to S_{Ps}$ *représente la fonction de transition,* V_I *et* V_O *étant respectivement l'ensemble des valuations des variables d'entrées et de sorties;*
- $Spec : S \times V_I \times V_O \to \{\text{vrai}, \text{faux}\}$ *définit la fonction d'évaluation des spécifications et retourne* vrai *lorsque les vecteurs d'entrées/sorties sont conformes à ces dernières.*

Nous intégrons les spécifications du logiciel au simulateur d'environnement guidé par les propriétés de sûreté de la même façon que nous avions intégré les propriétés de sûreté au simulateur d'environnement (voir §5.3.4).

Définition 7.6 *Soit un simulateur d'environnement guidé par les propriétés de sûreté:*

$$M_{saf} = (S_{saf}, s_{init_{saf}}, I, O, f_{saf}, env, part_{env}(V_{I_{env}}, Saf_{Ps}), sel_{env}).$$

Soit P_{sp} *les spécifications partielles du logiciel telles que:*

$$P_{sp} = (S_{sp}, s_{init_{sp}}, I, O, f_{sp}, Spec).$$

Un simulateur d'environnement étendu M_{spc} *est défini par:*

$$M_{spc} = (S_{spc}, s_{init_{spc}}, I, O, f_{spc}, env, part_{env}(V_{I_{env}}, Saf_{Ps}, Spec), sel_{env})$$

- $S_{spc} \subseteq S_{saf} \times S_{sp}$ *est l'ensemble des états du simulateur d'environnement étendu;*
- $s_{init_{spc}} = (s_{init_{saf}}, s_{init_{sp}})$ *est l'état initial;*
- *I (resp. O) est l'ensemble des variables d'entrées (resp. de sorties);*
- $f_{spc} : S_{spc} \times V_I \times V_O \to S_{spc}$ *est la fonction de transitions telle que si* s_{saf} *et* s'_{saf} *appartiennent à* S_{saf}*,* s_{sp} *et* s'_{sp} *appartiennent à* S_{sp} *et* (i, o) *appartient à* $V_I \times V_O$ *alors :*
$$\begin{aligned} (s'_{saf}, s'_{sp}) &= f_{spc}((s_{saf}, s_{sp}), i, o) \\ (s'_{saf}, s'_{sp}) &= (f_{saf}(s_{saf}, i, o), f_{spc}(s_{sp}), i, o)) \end{aligned}$$
- $env \subseteq S_{spc} \times V_I$ *représente les contraintes d'environnement;*
- $V_{val} = part_{env}(V_{I_{env}}, Saf_{Ps}, Spec)$ *est une méthode qui détermine en fonction de l'ensemble des vecteurs d'entrées valides* ($V_{I_{env}}$)*, des propriétés de sûreté* (Saf_{Ps}) *et des spécifications partielles du logiciel (Spec), l'ensemble* V_{val} *des vecteurs d'entrées dans lequel un vecteur d'entrées sera choisi;*
- $sel_{env}(V_{val})$ *est la méthode qui permet de choisir un vecteur d'entrées à soumettre au logiciel sous test.*

7.3.2 Modification du calcul des vecteurs d'entrées pertinents

L'ensemble de toutes les contraintes définies par l'opérateur **specification** peut être évalué par la une fonction booléenne $Spec(s, i, o)$. Nous étendons alors la définition 5.6 de la façon suivante:

Définition 7.7 *Soit un simulateur d'environnement étendu M_{spc}:*

$$M_{spc} = (S_{spc}, s_{init_{spc}}, I, O, f_{spc}, env, part_{env}(V_{I_{env}}, Saf_{Ps}, Spec), sel_{env}),$$

$V_{I_k}(s_{t+0})$, l'ensemble des vecteurs d'entrées pertinents est défini par:

$$\begin{aligned} V_{I_k}(s_{t+0}) = \{ i_{t+0} \in V_I | \exists s_{t+k} \in S_k(s_{t+0}, i_{t+0}), \exists (i_{t+k}, o_{t+k}) \in V_I \times V_O, \\ (\neg Saf_{Ps}(s_{t+k}, i_{t+k}, o_{t+k}) \uparrow env(i_{t+k}, s_{t+k})) \uparrow Spec(s_{t+k}, i_{t+k}, o_{t+k}) \} \end{aligned}$$

où s_{t+k}, i_{t+k} et o_{t+k} représentent respectivement l'état du simulateur d'environnement, le vecteur d'entrées et le vecteur de sorties du logiciel sous test à l'issue d'un chemin de longueur k et $S_k(s_{t+0})$ représente l'ensemble des états accessibles par un chemin de longueur k à partir de l'état courant s_{t+0}.

Nous modifions de manière analogue le calcul des états accessibles par un chemin de longueur k afin de prendre également en compte la fonction $Spec(s, i, o)$:

Définition 7.8 *Soit un simulateur d'environnement étendu M_{spc}:*

$$M_{spc} = (S_{spc}, s_{init_{spc}}, I, O, f_{spc}, env, part_{env}(V_{I_{env}}, Saf_{Ps}, Spec), sel_{env}),$$

$S_k(s_{t+0})$, l'ensemble des états de M accessibles par un chemin de longueur k, est calculé par induction de la manière suivante:

$$\begin{cases} S_0(s_{t+0}, i_{t+0}) & = & \{ s_{t+0} \} \\ S_1(s_{t+0}, i_{t+0}) & = & \{ s \in S | \exists o_{t+0} \in V_O, \\ & & \quad s = (f(s_{t+0}, i_{t+0}, o_{t+0}) \uparrow env(i_{t+0}, s_{t+0})) \uparrow \\ & & \quad\quad Spec(s_{t+0}, i_{t+0}, o_{t+0}) \} \\ S_k(s_{t+0}, i_{t+0}) & = & \{ s \in S | \exists s'_{t+k-1} \in S_{k-1}(s_{t+0}, i_{t+0}), \exists (i_{t+k_1}, o_{t+k-1}) \in V_I \times V_O, \\ & & \quad s = (f(s'_{t+k-1}, i_{t+k-1}, o_{t+k-1}) \uparrow env(i_{t+k-1}, s'_{t+k-1})) \uparrow \\ & & \quad\quad Spec(s_{t+k-1}, i_{t+k-1}, o_{t+k-1}) \} \end{cases}$$

Les modifications que nous avons apportées aux calculs des ensembles $V_{I_k}(s_{t+0})$ des vecteurs d'entrées pertinents ne modifient pas la structure globale,[4] $\Xi_n(s_{t+0})$ regroupant tous les ensembles de vecteurs d'entrées pertinents $V_{I_k}(s_{t+0})$ pour $k \in [0..n]$. Il n'est donc pas nécessaire de modifier les algorithmes des stratégies définies au paragraphe 5.6 qui ont pour but de générer un seul ensemble de vecteurs d'entrées pertinents à partir de tous les éléments $V_{I_k}(s_{t+0})$ de $\Xi_n(s_{t+0})$.

7.4 Discussions

7.4.1 Vers une notion de critère d'arrêt

Les travaux présentés au chapitre 2 assimilent en général le critère de sélection des données de test à un critère d'arrêt. En particulier, les travaux présentés dans [VTP94, TVPL94] et

[4]Voir le paragraphe 5.3.4.

[VTP95] sont basés sur la couverture des spécifications du logiciel, ces dernières étant décrites à l'aide d'expressions booléennes.

Dans le cas de LUTESS, nous pouvons définir un critère d'arrêt en combinant les deux approches que nous avons présentées aux paragraphes 7.2 et 7.3. L'idée de base consiste à considérer que nous avons suffisamment testé le logiciel lorsque les spécifications ont été couvertes: pour s'en assurer, nous avons besoin à la fois d'une description des spécifications du logiciel (voir le paragraphe 7.3) et de l'ensemble des situations déjà testées par LUTESS (voir le paragraphe 7.2).

Le critère d'arrêt que nous proposons est basé sur l'analyse de l'expression suivante:

$$C_{arr} = P_{sp}(s, i, o) \uparrow P_{out}(s, i, o).$$

Nous avons suffisamment testé le logiciel lorsque la valeur de C_{arr} ne dépend plus de la valeur des vecteurs de sorties. En effet, ceci signifie que la valeur de $P_{sp}(s, i, o)$ est totalement connue indépendamment des sorties calculées par le logiciel. L'évaluation de l'expression associée à $Carr$ mène à l'une des trois situations:

- $Carr = faux$ signifie que les spécifications n'ont jamais été respectées dans les situations testées;
- $Carr = vrai$ signifie que les spécifications ont toujours été respectées dans les situations testées;
- $Carr = h(s, i)$ représente les couples état/vecteur d'entrées pour lesquelles les spécifications ont été respectées ($h(s, i) = vrai$) ou ne l'on pas été ($h(s, i) = faux$).

Supposons que l'on souhaite tester la propriété $P_{spec} = (i_1 \vee i_2) \Rightarrow o_1$. À l'état initial, nous avons $P_{out} = vrai$[5] (aucun vecteur de sortie n'a été produit) et par suite:

$$P_{spec} \uparrow P_{out}(s, i, o) = P_{sp}(s, i, o).$$

LUTESS produit alors son premier vecteur d'entrées: par exemple, $i_1 = vrai$ et $i_2 = faux$. Si le logiciel répond par $o_1 = vrai$, nous obtenons:

$$P_{sp}(s, i, o) \uparrow P_{out}(s, i, o) = i_2 \Rightarrow o_1$$

Ensuite, si LUTESS engendre le vecteur $i_1 = faux$ et $i_2 = vrai$ alors que le logiciel répond toujours $o_1 = vrai$:

$$P_{sp}(s, i, o) \uparrow P_{out}(s, i, o) = (i_1 \wedge i_2) \Rightarrow o_1$$

Enfin lorsque LUTESS engendre le vecteur $i_1 = vrai$ et $i_2 = vrai$ et que le logiciel répond une troisième fois par $o_1 = vrai$ alors nous obtenons $P_{sp}(s, i, o) \uparrow P_{out}(s, i, o) = vrai$. Dans le cas où le logiciel aurait répondu $o_1 = faux$, nous aurions eu:

$$P_{sp}(s, i, o) \uparrow P_{out}(s, i, o) = \neg(i_1 \wedge i_2).$$

L'exemple simple que nous venons de prendre montre les informations que nous pouvons obtenir de notre critère d'arrêt. Cependant, celui-ci reste qu'une première ébauche d'un véritable critère d'arrêt. Sur la base du critère que nous venons de présenter, nous devons l'étendre en considérant à la fois les contraintes de l'environnement pour ne pas considérer les vecteurs d'entrées interdits par ces dernières et la fonction de transitions pour ne pas s'intéresser à des états inaccessibles de l'environnement.

[5]Nous ne détaillons pas ici la construction séquentielle de P_{out} (voir §7.2).

7.4.2 Connaissances partielles des vecteurs de sortie, avantages et inconvénients

Les différentes possibilités d'évolutions que nous proposons dans ce chapitre pour LUTESS ont pour but d'ajouter une information partielle sur les sorties calculées par le logiciel sous test afin d'améliorer la qualité des vecteurs d'entrées produits. Si une des stratégies permet de détecter un état suspect s_{sus} avec une suite d'entrées/sorties spécifiée dans P_{sp}, alors nous atteindrons très certainement cet état suspect. De plus, si dans cet état s_{sus} il existe une entrée i et une sortie o tel que:

$$\neg Saf_{Ps}(s_{sus}, i, o) \wedge Spec(s_{sus}, i, o)$$

alors nous aurons en plus toutes les chances de détecter une violation des propriétés de sûreté.

Cependant, il faut également garder à l'esprit que les réactions du logiciel peuvent être différentes de leurs spécifications dans le cas où des erreurs seraient présentes. Or si nous utilisons toujours les spécifications partielles du logiciel, nous risquons de ne pas "voir" certains états suspects qui devrait être inaccessibles selon les spécifications, mais qui le sont en réalité à cause d'une erreur dans l'écriture du programme. Cet argument pourrait plaider en faveur d'une solution comme le simulateur d'environnement étendu par apprentissage. En effet, dans ce cas, nous ne supposons plus que le logiciel respecte ses spécifications, mais nous nous basons sur des comportements observés. Malgré tout, il existera toujours une incertitude sur l'état réel dans lequel se trouve le logiciel et auquel nous ne pouvons pas avoir accès (dans le cadre de notre approche boîte noire): si nous nous retrouvons dans une situation de l'environnement déjà observée, rien ne nous garantit que le logiciel se trouve dans l'état où il se trouvait lors de la dernière observation de la situation de l'environnement (voir le §7.2.1); de ce fait, des réactions différentes de celles déjà observées peuvent se produire.

Enfin, il semble qu'il soit actuellement difficile de faire prendre en compte à LUTESS des informations complémentaires pour la génération des vecteurs d'entrées. En effet, aux vues des résultats que nous obtenons au chapitre 6, les capacités de calcul actuelles semblent atteindre leurs limites[6] avec les propriétés généralisées[7].

Malgré tout, l'utilisation d'informations sur les vecteurs de sorties qui peuvent être générés par le logiciel sous test n'est pas réduite à la seule amélioration des performances de la sélection guidée par les propriétés de sûreté des vecteurs d'entrées. En effet, nous avons vu que ces informations pouvaient tout à la fois servir d'oracle pour émettre un diagnostic sur les vecteurs de sorties générés par le logiciel sous test, et aussi, servir à la définition d'un critère d'arrêt tout en essayant de favoriser des vecteurs d'entrées menant à des situations non encore observées.

[6] Notons que nous pourrons toujours trouver des exemples pour lesquels ces extensions sont ou ne sont pas applicables.

[7] Nous désignons par "propriétés généralisées" les propriétés de l'étude de cas du chapitre 6 qui ne spécifiaient pas *a priori* quel était l'abonné associé à un événement du système téléphonique.

Chapitre 8

Conclusions

Contributions de la thèse

Cette thèse s'inscrit dans le domaine de la validation des logiciels réactifs synchrones. Ce type de logiciel doit interagir en permanence avec son environnement à la vitesse imposée par celui-ci. Le travail que nous avons présenté a pour but de s'assurer de la correction de la partie fonctionnelle du logiciel par rapport à des propriétés de type sûreté.

Ce travail fait plus particulièrement suite à celui de Ioannis Parissis [Par96]. L'approche suivie par l'équipe *Vasco* depuis 1994 pour la validation des logiciels réactifs synchrones consiste à simuler aléatoirement le comportement de l'environnement. LUTESS est l'outil qui met en pratique cette approche. Dans son utilisation de base, LUTESS permet de générer aléatoirement des données de test qui respectent un ensemble de contraintes d'environnement de façon équiprobable, c'est à dire que toutes les données de test qui respectent les contraintes d'environnement ont la même probabilité d'être choisies. L'objectif de cette thèse est d'améliorer la qualité des données de test sélectionnées pour favoriser celles qui ont le plus de chance de révéler une erreur.

L'approche que nous suivons avec le développement de LUTESS est radicalement différente des approches que nous avons pu rencontrer dans le domaine du test dit "boîte noire". En effet, beaucoup de travaux se basent sur une analyse des spécifications fonctionnelles du logiciel pour sélectionner les données de test. Au contraire, avec LUTESS, on ne fait aucune hypothèse sur le comportement du logiciel sous test: la sélection des données de test est basée sur une description de l'environnement du logiciel. Pour parvenir à améliorer la sélection des données de test, nous devons fournir à LUTESS des informations complémentaires aux contraintes d'environnement sous forme de profils opérationnels, de schémas comportementaux ou encore de propriétés de sûreté.

Dans le cadre de ce travail, ces informations sont données sous forme de propriétés de sûreté qui décrivent les comportements sûrs du logiciel. Bien qu'il soit assez difficile d'écrire des propriétés de sûreté, elles accompagnent souvent la description des logiciels réactifs synchrones. D'autre part, des travaux comme ceux de [HRS98] proposent de donner un cadre commun à différents corps d'ingénieur pour arriver à exprimer les propriétés de sûreté en ITL. Cette logique temporelle est très proche de celle utilisée par [PH88] pour montrer que le langage LUSTRE sur lequel s'appuie la description des propriétés d'environnement et de

sûreté, est une logique temporelle du passé. Ces travaux peuvent donc contribuer à faciliter l'écriture des propriétés de sûreté pour LUTESS.

Les différentes stratégies de sélection des données de test que nous avons définies et implantées utilisent les propriétés de sûreté comme un guide de test: nous souhaitons choisir les données de test qui ont la plus grande probabilité de mettre en évidence une violation des propriétés de sûreté. Le caractère novateur de nos stratégies par rapport à la première proposition faite dans [OP94a] se situe dans la prise en compte de l'aspect “temporel” de LUSTRE.

Les idées que nous avons présentées ici ont été validées de manière empirique sur une étude de cas de services téléphoniques. Cette expérimentation a été reprise du concours [dBORZ98] gagné par LUTESS. L'ensemble des résultats obtenus a montré une complémentarité des différentes stratégies en termes de nombre d'états suspects atteints, avec un avantage pour les stratégies *union* et *paresseuse*. Nous nous sommes également aperçu des difficultés à écrire correctement la propriété de sûreté confirmant d'un certain point de vue les travaux de [LMW01] portant sur l'évaluation de techniques de validation formelle dans le cadre d'Airbus Industrie, ou encore ceux de [BCC+03] qui proposent de prouver qu'un programme ne contient pas d'erreur d'un type particulier (par exemple division par zéro) au lieu de prouver des comportements attendus du logiciel qui sont difficiles à exprimer.

Propriétés de sûreté et schémas comportementaux

Les schémas comportementaux mis en œuvre par LUTESS permettent de guider la sélection des données de test afin de tester des situations choisies par le testeur. La sélection des données de test guidée par les propriétés de sûreté construit automatiquement la suite des données de test nécessaire pour atteindre une situation où les propriétés peuvent être violées. Ces deux approches permettent de fournir à LUTESS des objectifs de test:

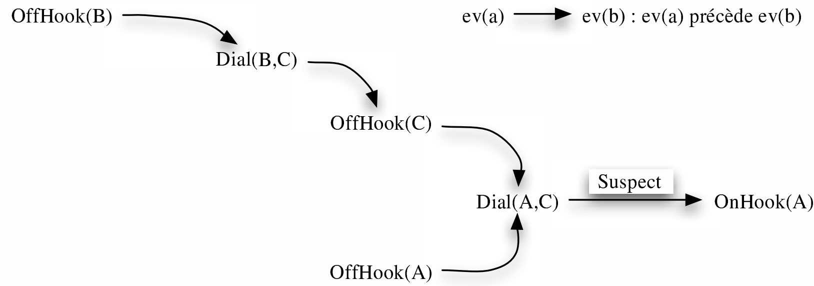
- les schémas comportementaux décrivent des comportements de l'environnement où le testeur souhaite observer quelles sont les réactions du logiciel;
- les propriétés de sûreté décrivent des comportements attendus du logiciel en fonction de l'évolution de son environnement.

Dans [Zua00b], Nicolas Zuanon a montré comment déduire une propriété de sûreté à partir d'un scénario. L'opération inverse est beaucoup plus difficile à mettre en œuvre. En effet, les choix qui ont été faits dans les développements de ces deux approches font que les propriétés de sûreté offrent un pouvoir d'expression plus important que les schémas comportementaux. En d'autres termes, il est possible d'exprimer plusieurs schémas à l'aide d'une propriété. Par exemple, considérons la propriété ($CW-TCS\ 2$) utilisée pour l'évaluation de nos techniques au chapitre 6.

CW - TCS 2

$$
\begin{array}{l}
Once_Since(OffHook(B), Dial(B,C)) \wedge \\
Once_Since(Dial(B,C), OffHook(C)) \wedge \\
Once_Since(OffHook(C), Dial(A,C)) \wedge \\
Once_Since(OffHook(A), Dial(A,C)) \wedge \\
\quad Between(Dial(A,C), OnHook(A)) \Rightarrow Announce(A)
\end{array}
$$

Fig. 8.1: Relation de dépendance des événements de $CW - TCS\ 2$



La figure 8.1 montre la relation de dépendance qui existe entre les différentes étapes qui mènent le logiciel dans un état suspect de l'environnement. Ainsi, nous pouvons déduire au moins quatre schémas comportementaux[1] à partir de cette propriété:

- (OffHook(B), Dial(B, C), OffHook(C), OffHook(A), Dial(A, C));
- (OffHook(B), Dial(B, C), OffHook(A), OffHook(C), Dial(A, C));
- (OffHook(B), OffHook(A), Dial(B, C), OffHook(C), Dial(A, C));
- (OffHook(A), OffHook(B), Dial(B, C), OffHook(C), Dial(A, C));

Cet aspect assez rigide du guidage à partir des schémas comportementaux dans Lutess avait déjà été identifié. Dans [Zua00a], Nicolas Zuanon propose deux opérateurs pour étendre leur l'expressivité:

- un opérateur de priorité: $sc_1 <> sc_2$. Cet opérateur rend l'un des deux schémas prioritaire. En d'autres termes dès qu'un des deux schémas commence, il se retrouve temporairement prioritaire jusqu'à son achèvement;
- un opérateur de concurrence: $sc_1 || sc_2$. La progression des deux schémas peut être entrelaçée.

Malgré cette évolution, les schémas comportementaux n'offrent pas un pouvoir d'expression aussi souple que les propriétés de sûreté. Sur l'exemple de la propriété de sûreté $CW - TCS\ 2$, il faut d'abord identifier tous les scénarios possibles pour ensuite les connectés entres-eux par l'opérateur de priorité.

Vers une méthodologie de test

Nous avons conçu et développé des stratégies de test pour les logiciels réactifs synchrones. Cette classe de logiciels se retrouve essentiellement dans le cadre d'applications critiques. De ce fait, ces logiciels sont accompagnés de leurs spécifications et souvent des propriétés (ou comportements) de sûreté qu'ils doivent respecter. Dès que l'environnement du logiciel sous test est modélisé, nous pouvons lancer une campagne de test en deux phases qui suivent le schéma ci-dessous.

Considérons p propriétés de sûreté; pour chacune des propriétés Ps_i (avec $i \in [1..p]$):

[1] Dans un souci de simplicité, nous faisons abstraction ici des conditions d'intervalles (voir §4.1.2).

1. nous testons la propriété Ps_i jusqu'à ce que le critère d'arrêt[2] soit atteint;
2. nous ajoutons la propriété Ps_i à l'ensemble des propriétés de spécification.

Cette première phase de la campagne de test nous permet de compléter les propriétés de spécifications avec les propriétés de sûreté qui ont été validées par le test. Lorsque toutes les propriétés de sûreté ont été validées, nous passons à la deuxième phase de la campagne de test. Il s'agit de faire générer par LUTESS une séquence de test où nous utilisons la conjonction de toutes les propriétés de sûreté ($\bigcap_{i=1}^{p} Ps_i$) pour le guidage de la sélection des vecteur d'entrées et l'ensemble des spécifications augmenté des propriétés de sûreté. La séquence se termine lorsque le critère d'arrêt a été atteint pour les nouvelles spécifications augmentées des propriétés de sûreté.

Le but de cette seconde phase est de détecter des violations d'une ou plusieurs propriétés de sûreté lorsque plusieurs situations critiques se présentent en même temps. En effet, la réponse à une situation critique peut être incompatible avec la réponse à une autre situation critique qui surviendrait en même temps.

Évolutions et perspectives

Nous avons présenté au chapitre 7 différentes pistes d'évolution de nos techniques ayant pour but de rendre le guidage de la sélection des données de test encore plus précis. Nous avons proposé deux approches:

1. l'apprentissage des réactions du logiciel sous test;
2. prise en compte de spécifications partielles du logiciel.

Ces deux approches ont montré au travers d'expériences simples leurs capacités en rendre le choix des données de test encore plus précis. Cependant, une utilisation systématique d'un guidage trop précis peut conduire à manquer des fautes. En effet, l'incertitude sur la nature et le nombre de fautes présentes dans le logiciel conduit à des divergences entre l'évolution réelle du logiciel et celle attendue par le guidage. En revanche, le rapprochement de ces deux propositions nous a permis d'exhiber une piste pour définir une notion de critère d'arrêt pour LUTESS.

[2] Ce critère se base sur une couverture des spécifications (Voir paragraphe §7.4.1).

Bibliographie

[Ake78] S.B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27:509–516, june 1978.

[AMLG99] A. Arnould, B. Marre, and P. Le Gall. Génération automatique de test à partir de spécifications de structures de données bornées. *Techniques et Sciences Informatique*, 18(3):297–321, mars 1999.

[BCC+03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.

[BDS91] F. Boussinot and R. De Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, september 1991.

[BGM91] G. Bernot, M-C. Gaudel, and B. Marre. Software testing based on formal specifications : a theory and a tool. *Software Engineering Journal*, 6:387–405, 1991.

[BR93] A. M. Beitw and T. P. Rout. A tool for generating test cases. In *Australian Software Engineering Conference*, Sydney, Australia, 1993.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean functions manipulation. *IEEE Transactions on Computers*, pages 667–692, august 1986.

[CHPP87] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. LUSTRE, a declarative language for programming synchronous systems. In *14th Symposium on Principles of Programming Languages (POPL 87), Munich*, pages 178–188. ACM, 1987.

[DB99] Lydie Du Bousquet. Test fonctionnel statistique de systèmes spécifiés en Lustre. Thèse, Université Joseph Fourier, Grenoble, France, Septembre 1999.

[dBOR98] L. du Bousquet, F. Ouabdesselam, and J.-L. Richier. Expressing and implementing operational profiles for reactive software validation. In *9th International Symposium on Software Reliability Engineering*, Paderborn, Germany, november 1998.

[dBORZ98] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Incremental feature validation : a synchronous point of view. In *Feature Interactions in Telecommunications Systems V*. IOS Press, 1998.

[dBZ98] L. du Bousquet and N. Zuanon. Feature interaction detection contest: Lutess testing tool. technical report PFL, IMAG - LSR, Grenoble, France, 1998.

[DM91] P. Dauchy and B. Marre. Test data selection from algebraic specifications : application to an automatic subway module. In *3rd European Software Engineering Conference*, pages 80–100, Milan, Italy, 1991. Springer-Verlag LNCS 550.

[Elm73] W. R. Elmendorf. Cause-effect Graphs in Functional Testing. Tr-00.2487, IBM Software Development Division, New York, USA, 1973.

[FI98] P. G. Frankl and O. Iakoumenko. Further empirical studies of test effectiveness. In *Sixth International Symposium on the Foundation of Software Engineering*, Orlando, Floride, 1998.

[GBGT98] N. Griffeth, R. Blumenthal, J.-C. Gregoire, and O. Tadashi. Feature interaction detection contest. In *Feature Interactions in Telecomunications and Software Systems*, pages 327 – 359. IOS Press, September 1998.

[Glo89] A-C. Glory. Vérification de propriétés de programmes flots de données synchrones. Thèse, Université Joseph Fourier, Grenoble, France, december 1989.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Data Flow Programming Language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[HKL$^+$98] C Heitmeyer, J. Jr. Kirby, B. Labaw, M Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *Transactions on Software Engineering*, 24(11):927–948, 1998.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Programming Language LUSTRE. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, pages 785–793, september 1992.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous Observers and the Verification of Reactive Systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93, Twente*. Workshops in Computing, Springer Verlag, june 1993.

[HP85] D. Harel and A. Pnueli. On development of reactive systems. *Logic and Models of Concurrent Systems, NATO*, F13:477–498, 1985.

[HRS98] K. M. Hansen, A. P. Ravn, and V. Stavridou. From Safety Analysis to Software Requirements. *IEEE Transactions on Software Engineering*, 24(7):573–584, 1998.

[HU79] J. E. Hopcroft and Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[Jac89] Jonathan Jacky. Programmed for disaster: Software errors that imperil lives. *The Sciences*, 29(5):22–27, 1989. September/October.

[LGGLBLM91] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming Real-Time Applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[Lju99] M. Ljung. Formal modelling and automatic verification of LUSTRE programs using np-tools. Master thesis, Royal Institute of Technology, Stockholm, Suède, march 1999.

[LMW01] O. Laurent, P. Michel, and V. Wiels. Using formal verification techniques to reduce simulation and test effort. In *Formal Methods Europe*, pages 465–477, Berlin, Allemagne, Mars 2001. Springer-Verlag.

[LP01] B. Legeard and F. Peureux. Generation of Functional Test Sequences from B Formal Specifications - Presentation and Industrial Case-Study. In *16th International Conference on Automated Software Engineering*, San Diego, USA, 2001.

[LPU02] Bruno Legeard, Fabien Peureux, and Mark Utting. A comparison of the BTT and TTF test-generation methods. In *ZB'2002 – Formal Specification and Development in Z and B*, pages 309–329, 2002.

[MA00] B. Marre and A. Arnould. Génération automatique de séquences de test à partir de description Lustre: GATEL. In *Journées Francophones des Langages Applicatifs*, France, 2000.

[Maz94] C. Mazuet. Stratégies de Test pour des Programmes Synchrones - Application au Langage Lustre. Technical report, Institut National Polytechnique de Toulouse, Toulouse, France, 1994.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems*. Springer-Verlag, 1995.

[Mye79] G. J. Myers. *The Art Of Software Testing*. Wiley-Interscience, 1979.

[OP94a] F. Ouabdesselam and I. Parissis. Testing Safety properties of Synchronous Critical Reactive Software. In *7th International Software Quality Week*, San Francisco, USA, may 1994.

[OP94b] F. Ouabdesselam and I. Parissis. Testing Synchronous Critical Software. In *5th International Symposium on Software Reliability Engineering*, Monterey, USA, november 1994.

[Par96] Ioannis Parissis. Test de logiciels synchrones spécifiés en Lustre. Thèse, Université Joseph Fourier, Grenoble, France, Septembre 1996.

[PH88] D. Pilaud and N. Halbwachs. From a synchronous declarative language to a temporal logic dealing with multiform time. In *Symposium on Formal Techniques in Real Time and Fault Tolerant Systems*, Warwick, september 1988. Springer Verlag.

[PLT00] L. Py, B. Legeard, and B. Tatibouët. évaluation de spécfications formelles en programmation logique avec contraintes ensemblistes - application à l'animation de spécifications B. In *AFADL*, pages 21–35, Grenoble, France, Janvier 2000.

[PV01a] I. Parissis and J. Vassy. Strategies for automated specification-based testing of synchronous software. In *16th International Conference on Automated Software Engineering*, San Diego, USA, November 2001. IEEE.

[PV01b] I. Parissis and J. Vassy. Test des propriétés de sûreté. In *Modélisation des systèmes réactifs*, Toulouse, France, Octobre 2001. hermès.

[PV03] I. Parissis and J. Vassy. Thoroughness of specification-based testing of synchronous programs. In *14th International Symposium on Software Reliability Engineering*, Denver, USA, November 2003. IEEE.

[Rat92] C. Ratel. Définition et réalisation d'un outil de vérification formelle de programmes Lustre: Le système Lesar. Technical report, Université Joseph Fourier, Grenoble, France, 1992.

[Ray91] P. Raymond. Compilation efficace d'un langage déclaratif synchrone : le générateur de code LUSTRE-V3. Technical report, Institut National Polytechnique de Grenoble, Grenoble, France, 1991.

[RB01] J-M Roussel and D. Bruno. Vérification de propriétés de sûreté dans les programmes ladder diagram. In *Modélisation des systèmes réactifs*, Toulouse, France, Octobre 2001. hermès.

[RC81] Debra J. Richardson and Lori A. Clarke. A partition analysis method to increase program reliability. In *Proceedings of the 5th international conference on Software engineering*, pages 244–253. IEEE, 1981.

[RLO92] D. Richardson, S. Leif Aha, and T. O'Malley. Specification-based Test Oracles for Reactive Systems. In *14th Int'l Conf.on Software Engineering*, pages 105–118, Melbourne, Australia, May 1992.

[RWNH98] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, december 1998.

[TVPL94] K. C. Tai, M. A. Vouk, A. Paradkar, and P. Lu. A predicate-based software testing strategy. *IBM Software Journal*, 1994.

[VTP94] M. A. Vouk, K. C. Tai, and A. Paradkar. Empirical studies of predicate-based software testing. In *International Symposium on Software Reliability Engineering*, pages 55–64, 1994.

[VTP95] M. A. Vouk, K. C. Tai, and A. Paradkar. Test generation for boolean expressions. In *International Symposium on Software Reliability Engineering*, pages 106–115, Toulouse, France, 1995.

[Wey86] E. Weyuker. Axiomatizing Software Test Data Adequacy. *IEEE Transactions on Software Engineering*, pages 1128–1138, 1986.

[WGS94] E. Weyuker, T. Goradia, and A. Singh. Automatically Generating Test Data from a Boolean Specification. *IEEE Transactions on Software Engineering*, pages 353–363, May 1994.

[XRK00] S. Xanthakis, P. Régnier, and C. Karapoulios. *Le test des logiciels*. Hermès Sciences, 2000.

[ZC03] F. Zhang and T. Cheug. Optimal transfer trees and distinguishing trees for testing observable nondeterministic finite-state machines. *Transactions on Software Engineering*, 29(1):1–14, 2003.

[Zua00a] N. Zuanon. Modular feature integration and validation in a synchronous context. In *Language Constructs for Describing Features*, pages 213 – 231, Glasgow, UK, May 2000. Springer.

[Zua00b] Nicolas Zuanon. Une méthode de test pour la validation de services de services de télécommunication. Thèse, Université Joseph Fourier, Grenoble, France, Juin 2000.

Annexe A

L'analyseur d'états suspects

Nous présentons ici l'analyseur d'états suspects que nous avons développé. Cette annexe s'articule autour de deux grandes parties. Tout d'abord, nous détaillons les fonctionnalités et l'utilisation de notre analyseur dans la partie A.1. Nous abordons ensuite dans la partie A.2 une description de l'architecture interne de l'analyseur.

A.1 Utilisation de l'analyseur

A.1.1 Fonctionnalités de l'analyseur

Notre analyseur permet de construire automatiquement les tableaux de résultats présentés au chapitre 6 à partir des fichiers contenant la liste des états suspects atteints au cours d'une séquence de test: le format des tableaux de résultats est LaTeX. Dans la suite de cette annexe, nous désignons l'analyseur d'états suspects par "*StateAnalyse*".

La version actuelle de *StateAnalyse* permet à l'utilisateur d'extraire deux types d'informations des séquences d'états suspectes atteintes:

- l'analyse croisée permet de comparer les séquences d'états suspects deux à deux et de déterminer quels sont les états atteints par la première et pas par la seconde, le nombre total d'états suspects atteints et le nombre d'états suspects distincts atteints. L'analyse croisée permet d'obtenir les tableaux du chapitre 6;
- L'affichage des états permet de connaître la valeur de chaque état suspect distinct atteint et sa représentativité en pourcentage par rapport à l'ensemble de tous les états suspects atteints.

A.1.2 Options de la ligne de commandes

Nous détaillons ici l'ensemble des paramètres obligatoires ou facultatifs nécessaires à *StateAnalyse*. Notons qu'une exécution sans paramètre de *StateAnalyse* permet d'afficher une aide en ligne succincte. Nous donnons ci-dessous la liste des éléments nécessaires à *StateAnalyse*:

- **-v** *nbStateVar*: l'option "-v" est suivie d'un entier définissant le nombre de variables d'états;

- **-f** $file_1$ $file_2$ ... $file_n$: l'option "-f" est suivie de la liste des fichiers à analyser (chaque fichier est séparé par un espace).
- **-n** $name_1$ $name_2$ ... $name_n$: l'option "-n" est suivie d'une liste de noms (chaque nom est séparé par un espace). Cette liste est utilisée pour nommer les ligne et les colonnes des tableaux issus de l'analyse croisée. Notons que $name_1$ est associé au fichier $file_1$ et ainsi de suite;
- **-cross** ou **-disp** active respectivement l'analyse croisée ou l'affichage des états suspects.

La listes des éléments obligatoires que nous venons de décrire peut être complétée par un ou plusieurs éléments facultatifs:

- **-o** *output.tex*: cette option définie le nom du fichier dans lequel seront écrits les résultats de l'analyse. Par défaut, ce fichier s'appelle "*result.tex*". Notons que si le fichier dans lequel sont écrits les résultats existe déjà, alors les nouveaux résultats sont ajoutés à la fin du fichier;
- **-m** *macro*: cette option définie le nom de base la macro LaTeX associée aux résultats de l'analyse. Par défaut, la base de la macro s'appelle "*result*". On ajoute au nom de base un suffixe qui dépend de la méthode d'analyse choisie: "*Cross*" ou "*Disp*";
- **-htn** $name_1$ $name_2$ ... $name_n$: cette option permet de renommer les noms des fichiers analysés qui apparaissent dans les commentaires du code LaTeX engendré.

A.2 Architecture de l'analyseur

A.2.1 Généralités

StateAnalyse est programmé en C^{++} (1200 lignes de code). Ce programme est organisé en deux gros modules:

- module "*Analyse*": il est chargé de regrouper les commandes d'analyses des fichiers d'états suspects. Ce module contient également les méthodes nécessaires à l'interprétation de la ligne de commandes et à la lecture des fichiers d'états suspects et est constitué d'un unique objet: $AnalyzeFiles$;
- le module "*HTable*": il regroupe l'ensemble des éléments nécessaires à la structure de données conservant les états suspects.

Structure de données

La structure de données qui mémorise les états suspects atteints est une table de hachage. Nous utilisons une table par fichier d'états suspects. La construction de cette table est décomposée en trois objets:

- La table de hachage proprement dite. Elle constitue l'objet principal qui est manipulé par $AnalyzeFiles$;
- une liste chaînée d'éléments. Cette liste est nécessaire pour la gestion des collisions dans la table de hachage;
- l'élément contenant les informations sur un état suspect: sa valeur et son nombre d'occurrence.

A.2.2 Interface des objets

Nous décrivons dans ce paragraphe l'ensemble des méthodes privées et publiques des différents objets de *StateAnalyse*.

Objet *AnalyzeFiles*

- **Méthodes privées:**
  - *void analyzeCommandLine(int argc, char **argv);*
    Cette fonction à pour but d'initialiser l'objet *AnalyzeFiles* à partir de la ligne de commande.
  - *void readFiles(char **argv);*
    Cette fonction permet de remplir la structure de donnée (HTable) à partir de la liste de fichier passer en paramètre. Notons que le nombre de fichier que contient cette liste est connu après l'analyse de la ligne de commande.
  - *void crossAnalyzing();*
    Cette fonction construit l'analyse croisée des états suspects.
  - *void setOfStatePrinting();*
    Cette fonction la valeur des états suspects distincts et leur représentativité par rapport à l'ensemble de tous les états suspects atteints.
  - *void conjAnalysing();*
    Cette fonction n'est pas opérationnelle. Son but est d'extraire la valeur des états suspects communs à deux séquences de test.
  - *void exitError(int argc, char **argv);*
    Cette fonction permet de sortir du programme en cas d'erreur en affichant la ligne de commandes.
  - *bool noOption(char *s);*
    Cette fonction teste si la chaîne passée en paramètre est une option reconnue par le programme.
  - *int nbParam(int i, int argc, char **argv);*
    Cette fonction compte le nombre de paramètres relatif à une option: $argc$ est le nombre total de paramètre, $argv$ la liste des paramètres et i l'indice indiquant où commence la liste des paramètres.
  - *void initData();*
    Initialise l'objet avec les valeurs par défaut.
- **Méthodes publiques:**
  - *AnalyzeFiles();*
    Constructeur de l'objet *AnalyzeFiles* qui l'initialise avec les valeurs par défaut.
  - *AnalyzeFiles(int argc, char **argv);*
    Constructeur de l'objet *AnalyzeFiles* qui l'initialise avec les informations portées sur la ligne de commandes et remplie la structure de donnée avec les fichiers spécifiés dans la ligne de commande.

- ~ *AnalyzeFiles*();
  Destructeur de l'objet.
- *void init*(*int argc*, *char **argv*);
  Initialisation de l'objet *AnalyzeFiles* avec les informations portées sur la ligne de commandes et remplie la structure de donnée avec les fichiers spécifiés dans la ligne de commandes. Cette fonction est utile lorsque l'objet est construit avec la méthode "*AnalyzeFiles*()".
- *void printResult*();
  Retourne le résultat de l'analyse des états suspects. L'objet doit être préalablement initialisé avec le constructeur "*AnalyzeFiles*(*int argc*, *char **argv*)" ou la méthode "*void init*(*int argc*, *char **argv*)".
- *friend ostream &operator* << (*ostream &sortie*, *AnalyzeFiles af*);
  Cette fonction permet d'afficher sur les flots E/S standards de C^{++} les valeurs internes de l'objet. Utile pour la mise au point du programme.

Objet *HTable*

- **Méthodes privées:**
  - *unsigned int h*(*string s*);
    Fonction de hachage.
- **Méthodes publiques:**
  - *HTable*(*int s*, *string n*);
    Constructeur de la table de hachage qui l'initialise avec *s* éléments. La chaîne *n* sert à spécifier le nom de la table (en général, le nom du fichier auquel correspondent les valeurs contenues dans la table).
  - *HTable*();
    Constructeur de la table de hachage qui l'initialise avec les valeurs par défaut.
  - *void init*(*int s*, *string n*);
    Initialise la table de hachage avec *s* éléments. La chaîne *n* sert à spécifier le nom de la table (en général, le nom du fichier auquel correspondent les valeurs contenues dans la table). Cette fonction est utile lorsque l'objet est construit avec "*HTable*()".
  - ~ *HTable*();
    Destructeur de l'objet.
  - *void Ajouter*(*HTElement *e*);
    Ajoute l'élément *e* à la table de hachage. Dans le cas où l'élément *e* est déjà présent, seul le nombre d'occurrence de *e* est augmenter.
  - *bool EstPresent*();
    Retourne *vrai* si et seulement si le dernier élément ajouter était déjà présent.
  - *bool Chercher*(*HTElement *e*);
    Retourne *vrai* si et seulement si l'élément *e* est présent dans la table de hachage.

- *int TrouveOccurenceDans(HTable *ht_buff);*
  Retourne le nombre d'éléments communs à la table de hachage contenue dans l'objet et celle contenue dans "*ht_buff*".
- *void setNom(string n);*
  Modifie le nom de la table de hachage.
- *string getNom();*
  Retourne le nom de la table de hachage.
- *int donneNbElement(bool flagDistinct);*
  Retourne le nombre total d'éléments ajoutés si la valeur de "*flagDistinct*" est égale à *faux* . Dans le cas contraire, cette fonction retourne le nombre d'éléments distincts ajoutés.
- *void debut();*
  Se place au début de la liste des éléments pour un indice de la table de hachage. Cette fonction et les trois suivantes permettent de parcourir la table de hachage comme une liste chaînée.
- *void suivant();*
  Passe à l'élément suivant.
- *HTElement *courant();*
  Retourne l'élément courant.
- *bool fin();*
  Retourne *vrai* lorsque tous les éléments de la table ont été parcourus.
- *friend ostream &operator << (ostream &sortie, HTable ht);*
  Cette fonction permet d'afficher sur les flots E/S standards de C^{++} les valeurs internes de l'objet. Utile pour la mise au point du programme.

Objet *ListHTElement*

- **Méthodes publiques:**
  - *ListHTElement();*
    Constructeur qui initialise la liste des éléments avec ses valeurs par défauts.
  - *ListHTElement(ListHTElement &l);*
    Constructeur de recopie: permet d'obtenir une liste identique à "*l*".
  - *~ ListHTElement();*
    Destructeur de l'objet.
  - *ListHTElement operator = (ListHTElement &l);*
    Opérateur d'affectation.
  - *void AjouterElem2List(HTElement *e);*
    Cette fonction permet d'ajouter l'élément "*e*" à la liste courante.
  - *void Premier();*
    Se place sur le premier élément de la liste.

- _void Suivant();_
  Passe à l'élément suivant.
- _HTElement *Courant();_
  Retourne l'élément courant.
- _bool EstDernier();_
  Retourne _vrai_ lorsque l'élément courant est le dernier de la liste.
- _bool EstVide();_
  Retourne _vrai_ lorsque la liste est vide.
- _void NbElemHT(int nb);_
  Utilise pour faire descendre l'information sur le nombre total d'éléments contenus dans la table au niveau des listes d'éléments.
- _friend ostream &operator << (ostream &sortie, ListHTElement l);_
  Cette fonction permet d'afficher sur les flots E/S standards de C^{++} les valeurs internes de l'objet. Utile pour la mise au point du programme.

Objet _HTElement_

- **Méthodes publiques:**
  - _HTElement(char *d);_
    Constructeur qui initialise l'élément avec la valeur _d_. En général, il s'agit de la valeur de l'état suspect.
  - _HTElement();_
    Constructeur qui initialise l'élément avec ses valeurs par défaut.
  - _HTElement(const HTElement &e);_
    Constructeur de recopie: l'élément construit avec ce constructeur est identique à _e_.
  - ~ _HTElement();_
    Destructeur de l'objet.
  - _void AjouterOccurence();_
    Augmente de un le nombre d'occurrence rencontrée de l'élément.
  - _void AjouterElem(HTElement *e);_
    Cette fonction permet de chaîner l'élément _e_ à l'objet courant.
  - _HTElement *ElementSuivant();_
    Retourne l'élément suivant.
  - _int NbOccurence();_
    Retourne le nombre d'occurrence d'un élément.
  - _string DonneChaine();_
    Retourne la valeur de l'élément.
  - _bool operator == (HTElement e);_
    Opérateur de comparaison de deux éléments: test de l'égalité.

- *HTElement operator =(const HTElement &e);*
  Opérateur d'affectation entre deux éléments.
- *friend ostream &operator << (ostream &sortie, HTElement e);*
  Cette fonction permet d'afficher sur les flots E/S standards de C^{++} les valeurs internes de l'objet. Utile pour la mise au point du programme.

Annexe B

Définition des nœuds LUSTRE

Nous donnons dans cette annexe les principaux nœuds LUSTRE utilisés dans ce manuscrit. Les nœuds que nous présentons ici sont classés en trois parties:

1. les principaux opérateurs de logique temporelle utilisés pour l'écriture des propriétés de sûreté;
2. les opérateurs de modélisation et de manipulation de la température (voir l'exemple utilisé au paragraphe 5.10) pour le climatiseur;
3. le nœud modélisant le comportement du climatiseur.

Nous ne reprenons pas ici la modélisation en LUSTRE des services téléphoniques qui peut être trouvé dans la thèse de Nicolas Zuanon [Zua00b].

B.1 Opérateurs de logique temporelle

B.1.1 Opérateur $Implies(A, B)$

L'opérateur $Implies(A, B)$ n'est pas réellement un opérateur de logique temporelle étant donné qu'il représente l'implication logique, mais il est nécessaire pour définir certains opérateurs de logique temporelle:

★ **Nœud B.1** : *code* LUSTRE *du nœud Implies.*

```
node Implies(A, B) returns (imp: bool);
let
    imp = not A or B;
tel
```

B.1.2 Opérateur $Always_Since(A, B)$

L'opérateur $Always_Since(A, B)$ est *vrai* si la valeur de A a toujours été *vrai* depuis la dernière évaluation à *vrai* de la valeur de B. Cet opérateur s'assure que l'événement A s'est toujours produit depuis la dernière occurrence de l'événement B:

★ **Nœud B.2** : *code* LUSTRE *du nœud Always_Since.*

```
node Always_Since(A, B) returns (alw_snc: bool);
let
    alw_snc  =  if B then A else
                if After(B) then A and pre(alw_snc) else  tel
                true;
```

B.1.3 Opérateur $Once_Since(A, B)$

L'opérateur $Once_Since(A, B)$ est *vrai* si la valeur de A a au moins une fois été *vrai* depuis la dernière évaluation à *vrai* de la valeur de B. Cet opérateur s'assure que l'événement A s'est produit au moins une fois depuis la dernière occurrence de l'événement B:

★ **Nœud B.3** : *code* LUSTRE *du nœud Once_Since.*

```
node Once_Since(A, B) returns (onc_snc: bool);
let
    onc_snc  =  if B then A else
                if After(B) then A or pre(onc_snc) else
                true;
tel
```

B.1.4 Opérateur $Always_From_To(A, B, C)$

L'opérateur $Always_From_To(A, B, C)$ est *faux* si la valeur de A a été au moins une fois à *faux* entre les instants où B et C ont pris la valeur *vrai* . Cet opérateur sert à s'assurer que l'événement A s'est toujours produit entre les instants où l'événement B et l'événement C se sont produits:

★ **Nœud B.4** : *code* LUSTRE *du nœud Always_From_To.*

```
node Always_From_To(A, B, C) returns (alw_fr_to: bool);
let
   alw_fr_to = implies(After(B), Always_Since(A, B) or Once_Since(C, B))
tel
```

B.1.5 Opérateur $Once_From_To(A, B, C)$

L'opérateur $Once_From_To(A, B, C)$ est *faux* si la valeur de A a toujours été *faux* entre les instants où B et C ont pris la valeur *vrai* . Cet opérateur permet de s'assurer que l'événement A s'est produit au moins une fois entre les instants où l'événement B et l'événement C se sont produits:

★ **Nœud B.5** : *code* LUSTRE *du nœud Once_From_To.*

```
node Once_From_To(A, B, C) returns (onc_fr_to: bool);
let
    onc_fr_to = implies(After(B) and C, Once_Since(A, B))
tel
```

B.2 Manipulations des tableaux de température

Nous représentons la valeur de la température par un tableau de valeurs booléennes en utilisant un codage "*un parmi n*". La taille de ce tableau est définie par la constante $TAILLE_TEMP$[1].

B.2.1 Test de l'égalité de deux températures

Le nœud *egal* prend en entrées deux tableaux de température et retourne *vrai* s'ils sont égaux. Deux températures sont égales si tous les éléments du tableau sont égaux:

★ **Nœud B.6** : *code* LUSTRE *du nœud egal.*

```
node egal(const taille: int; T1, T2: bool^taille) returns (eq: bool);
let
     eq   =   with taille > 1 then
                  ((T1[0] and T2[0]) or not(T1[0] or T2[0])) and
                      egal(taille - 1, T1[1..taille-1], T2[1..taille-1])
              else
                   (T1[0] and T2[0]) or not(T1[0] or T2[0]);
tel
```

B.2.2 Test de la supériorité de deux températures

Le nœud *sup* prend en entrées deux tableaux de température et retourne *vrai* si la température $T1$ est supérieure à $T2$.

[1]Nous avons pris $TAILLE_T EMP = 5$ dans l'exemple du climatiseur (voir §5.10).

★ **Nœud B.7** : *code* LUSTRE *du nœud egal.*

```
node sup(const taille: int; T1, T2: bool^taille) returns (gt: bool);
let
        gt   =   with taille > 1 then
                      if (T1[0] and T2[0]) or not(T1[0] or T2[0]) then
                           sup(taille - 1, T1[1..taille-1], T2[1..taille-1])
                      else
                           T2[0]
                 else
                      false;
tel
```

B.2.3 Test de l'infériorité de deux températures

Le nœud *inf* prend en entrées deux tableaux de température et retourne *vrai* si la température $T1$ est inférieure à $T2$.

★ **Nœud B.8** : *code* LUSTRE *du nœud egal.*

```
node inf(const taille: int; T1, T2: bool^taille) returns (lt: bool);
let
        lt   =   with taille > 1 then
                      if (T1[0] and T2[0]) or not(T1[0] or T2[0]) then
                           sup(taille - 1, T1[1..taille-1], T2[1..taille-1])
                      else
                           T1[0]
                 else
                      false;
tel
```

B.2.4 Modélisation de l'augmentation de température

Le nœud *augmente* permet comparer deux températures et renvoie *vrai* si la seconde est supérieure ou égale à la première. Ce nœud est utile pour l'expression des contraintes d'environnement du climatiseur; il permet de contraindre l'évolution de la température en la faisant monter:

★ **Nœud B.9** : *code* LUSTRE *du nœud augmente.*

```
node augmente(T1, T1prime: bool^TAILLE_TEMP)returns(aug: bool);
let
      aug = egal(TAILLE_TEMP, T1, T1prime) or sup(TAILLE_TEMP, T1prime, T1);
tel
```

B.2.5 Modélisation de la diminution de température

Le nœud *diminue* est construit sur le même principe que le nœud *augmente*. Il permet de contraindre, dans la description de l'environnement, l'évolution de la température en la faisant baisser:

★ **Nœud B.10** : *code* Lustre *du nœud diminue.*

```
node diminue(T1, T1prime: bool^TAILLE_TEMP)returns(dim: bool);
let
    dim = egal(TAILLE_TEMP, T1, T1prime) or inf(TAILLE_TEMP, T1prime, T1);
tel
```

B.3 Le climatiseur

Le nœud *controleur* code le comportement de notre climatiseur. Nous ne redonnons pas ici la spécification des entrées et sorties du contrôleur (voir §5.10.1).

★ **Nœud B.11** : *code* Lustre *du nœud principal du climatiseur.*

```
node controleur (Marche, SoufflerieOff, SoufflerieOn, SoufflerieVeille: bool;
        TempCapteur, TempChoisie: bool^TAILLE_TEMP)
returns (ARRETE, INACTIF, CHAUD, FROID, SOFF, SVEILLE, SON: bool);
let
    ARRETE   =  SoufflerieOff;
    INACTIF  =  Marche and egal(TAILLE_TEMP, TempCapteur, TempChoisie);
    CHAUD    =  SoufflerieOn and inf(TAILLE_TEMP, TempCapteur, TempChoisie);
    FROID    =  SoufflerieOn and sup(TAILLE_TEMP, TempCapteur, TempChoisie);
    SOFF     =  between(edge(not Marche), SoufflerieOff or Marche);
    SVEILLE  =  between(edge(Marche and
                    egal(TAILLE_TEMP, TempCapteur, TempChoisie)),
                    not egal(TAILLE_TEMP, TempCapteur, TempChoisie) or
                    not Marche or SoufflerieVeille);
    SON      =  between(edge(Marche and
                    not egal(TAILLE_TEMP, TempCapteur, TempChoisie)),
                    not Marche or SoufflerieOn or
                    egal(TAILLE_TEMP, TempCapteur, TempChoisie));
tel
```

Liste des tableaux

3.1 Évaluation du nœud *after*. 35

4.1 Modes de fonctionnement de Lutess. 45
4.2 Vecteurs d'entrées valides pour le climatiseur dans l'état initial. 59

5.1 Vecteurs d'entrées pertinents calculés en fonction de la longueur du chemin. . . 73
5.2 Vecteurs d'entrées pertinents calculés par *"Semi Environnement"* 74
5.3 Vecteurs d'entrées pertinents calculés par *"Sans Environnement"* 74

6.1 Tableau de présentation des résultats. 110
6.2 Résultats pour $CFBL - CFBL$ 1 111
6.3 Comparaison des résultats pour la stratégie paresseuse en fonction de l'approximation 112
6.4 Résultats pour $CFBL - CFBL$ 2 113
6.5 Résultats pour $CFBL - CELL$ 1 114
6.6 Résultats pour $CFBL - CELL$ 4 115
6.7 Résultats pour $CFBL - CELL$ 5 116
6.8 Résultats pour $CELL - CW$ 1 117
6.9 Résultats pour $CELL - CW$ 2 117
6.10 Résultats pour $CELL - CW$ 4 118
6.11 Résultats pour $CELL - CW$ 7 119
6.12 Résultats pour $CELL - CW$ 8 119
6.13 Résultats pour $CELL - TWC$ 1 120
6.14 Résultats pour $CELL - TWC$ 4 121
6.15 Résultats pour $CELL - TWC$ 5 121
6.16 Résultats pour $CND - TCS$ 1 122
6.17 Résultats pour $CND - TCS$ 2 123
6.18 Résultats pour $CND - TCS$ 4 124
6.19 Résultats pour $CND - TCS$ 5 124
6.20 Résultats pour $CW - TCS$ 1 125
6.21 Résultats pour $CW - TCS$ 2 126
6.22 Résultats pour $CW - TCS$ 4 127
6.23 Résultats pour $CW - TCS$ 5 127

7.1 Vecteurs d'entrées pertinents calculés par *"Tout Environnement"* ($P_{out} = vrai$).139

Table des figures

2.1 Opérateurs des graphes causes-effets 22
2.2 Exemple de graphes causes-effets 22

3.1 Logiciel réactif 29
3.2 Logiciel synchrone 30
3.3 Comportement du nœud *edge* 32
3.4 Aspects temporels de l'opérateur $after(A)$ 34
3.5 Arbre des exécutions. 36
3.6 Automate de contrôle. 37
3.7 Programme de vérification. 37

4.1 Principe de Lutess 42
4.2 Le climatiseur et son environnement. 47
4.3 Simulateur d'environnement 50
4.4 Décomposition de Shannon: principe. 54
4.5 Arbre de Shannon pour f. 55
4.6 Élimination des nœuds redondants: étape 1. 55
4.7 Élimination des nœuds redondants: étape 2. 56
4.8 Partage des sous-arbres isomorphes: étape 3. 56
4.9 Schéma du bdd d'environnement. 56
4.10 Etiquetage du bdd. 57
4.11 Le bdd d'environnement du climatiseur. 58

5.1 Opérateur $Once_From_To$. 66
5.2 Représentations d'un état suspect. 69
5.3 Représentations du simulateur d'environnement. 75
5.4 Vecteurs d'entrées pertinents. 79
5.5 Environnement: choix non systématique des vecteurs d'entrées. 79
5.6 Fenêtre temporelle 81
5.7 Propriété alternante. 82
5.8 Le climatiseur et son environnement. 85

6.1 Modèle du réseau téléphonique. 100
6.2 Digramme de transition pour un téléphone 102

7.1 États environnement / système sous test 134

7.2 Simulateur d'environnement 138

8.1 Relation de dépendance des événements de $CW - TCS$ 2 147