Test fonctionnel statistique de logiciels spécifiés en Lustre ; application à la validation de services téléphoniques

Lydie Du Bousquet

▶ To cite this version:

Lydie Du Bousquet. Test fonctionnel statistique de logiciels spécifiés en Lustre ; application à la validation de services téléphoniques. Génie logiciel [cs.SE]. Université Joseph-Fourier - Grenoble I, 1999. Français. NNT : . tel-00004828

HAL Id: tel-00004828
https://theses.hal.science/tel-00004828

Submitted on 18 Feb 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

Lydie du Bousquet

le 29 septembre 1999

Test fonctionnel statistique de systèmes spécifiés en Lustre

Application à la validation de services téléphoniques

Composition du Jury :

Rapporteurs	G. Bernot
	M. Rueher
Examinateurs	F. Ouabdesselam
	J.-L. Richier
	R. Groz
	N. Halbwachs

A Philippe
A mes parents

Remerciements

Cette thèse s'est déroulée au sein du laboratoire Logiciels, Systèmes et Réseaux, de l'Institut IMAG. Je tiens à exprimer ma reconnaissance à son directeur Paul Jacquet, pour m'avoir accueillie et pour toute son action en faveur des doctorants.

Je tiens aussi à remercier :

Gilles Bernot, professeur à l'université Evry-Val d'Essone, et Michel Rueher, professeur à l'université de Nice Sophia-Antipolis, pour avoir accepté de juger ce travail,

Nicolas Halbwachs, directeur de recherches CNRS, et Roland Groz, responsable de département au CNET-France télécom, pour avoir accepté de faire partie de ce jury,

Farid Ouabdesselam et Jean-Luc Richier, mes directeurs de thèse, pour toute la disponibilité dont ils ont fait preuve, pour leurs conseils et leur soutien pendant ces trois ans,

Pierre-Claude Scholl, professeur à l'université Joseph Fourier, qui m'a fait découvrir l'informatique en DEUG, qui m'a conseillé plus d'une fois dans mon orientation, et sans qui je n'aurais probablement pas fait ce doctorat,

Jean-Marc Vincent, maître de conférence à l'université Joseph Fourier, et Pierre Berlioux, maître de conférence à l'INPG, pour les discussions fructueuses que nous avons eues,

l'équipe Lustre de Verimag, et notamment Pascal Raymond pour l'aide qu'ils m'ont si gentiment apportée dans l'utilisation de Lustre et de Lesar,

Paul Amblard, maître de conférence et Laurent Trilling, professeur à l'université Joseph Fourier, pour la sympathie qu'ils m'ont toujours exprimée,

l'équipe administrative et technique : Liliane, Martine, Solange Rachelle, Bernard et François, pour leur toujours grande disponibilité,

Ioannis, pour son travail précurseur, Nicolas, Jérôme et Rémi, pour leur énorme investissement personnel pendant le concours FIW'98, Patrice, pour les améliorations qu'il a apporté à Lutess,

Sophie, pour l'amitié qu'elle m'a accordée au cours de ces deux dernières années,

Philippe et mes parents, pour leur soutien et leur amour...

Table des matières

1 Introduction **11**

I Lutess : un outil de validation de systèmes synchrones **17**

2 Le langage Lustre **19**
2.1 Description du langage 19
2.2 Modèle d'exécution de Lustre 21
2.3 Une représentation du modèle d'exécution : notion de BDD 22

3 L'outil de test Lutess **25**
3.1 Description générale de Lutess 26
3.2 Le générateur de données de Lutess 30
3.3 Méthode de génération aléatoire 33
3.4 Méthode de génération guidée par des propriétés 40

4 Validation des méthodes de génération **45**
4.1 Description du système de contrôle de l'ascenseur 46
4.2 Résumé des principes des tests d'hypothèses 50
4.3 Analyse des données produites par la méthode de génération aléatoire 51
4.4 Analyse des données produites par la méthode guidée par des propriétés . . . 57
4.5 Bilan et conclusion 61

II Lutess : vers une méthode de génération statistique **63**

5 Notion de profil opérationnel **65**
5.1 Profil opérationnel : concept et usages 66
5.2 Profils opérationnels et Lutess 68
5.3 Notion de probabilité conditionnelle 71

6 Méthode de génération basée sur des probabilités conditionnelles **73**
6.1 Guidage par des probabilités conditionnelles 73
6.2 D'un profil opérationnel à des probabilités conditionnelles 79
6.3 De probabilités conditionnelles à un profil opérationnel 84

7 Validation de la méthode de génération basée sur des probabilités conditionnelles 89
7.1 Algorithme de génération 89
7.2 De probabilités conditionnelles à un profil opérationnel 96
7.3 Bilan et conclusion 99

III Validation de spécifications de services téléphoniques 103

8 Problématique relative à la validation de services téléphoniques 105
8.1 Introduction 105
8.2 Notre approche pour la validation de services 108

9 Validation incrémentale de services téléphoniques 111
9.1 Introduction 111
9.2 Conception d'un modèle exécutable 112
9.3 Modélisation et validation d'un modèle du service de base 116
9.4 Modélisation et validation d'un modèle du service CFNR 123
9.5 Bilan de cette étude de cas 132

10 Détection d'interactions entre services téléphoniques 133
10.1 Introduction 133
10.2 L'énoncé du concours 134
10.3 Conception d'un modèle exécutable 137
10.4 Validation de services et détection d'interactions 143
10.5 Bilan de l'étude de cas avec Lutess 145
10.6 Travaux similaires 148

IV Réflexions et Comparaisons 151

11 Travaux comparables 153
11.1 Validation de systèmes synchrones 153
11.2 Lesar : un outil de vérification de programmes Lustre 156
11.3 Validation de protocoles 158

12 Bilan et perspectives 161

Bibliographie 165

Annexes 173

A Description des sémantiques citées 175
A.1 Définition formelle de Lustre 175
A.2 Définition formelle d'un noeud testeur 177

A.3 Opérateurs Lustre utilisés 178

Chapitre 1

Introduction

Contexte et Motivations

Quel que soit le mode de construction d'un produit logiciel, de la création complète à partir d'un cahier des charges à l'intégration de logiciels existants, l'activité de validation est indispensable pour s'assurer que le produit à tous ses stades de développement est bien conforme à ses objectifs. Toutes les études ont montré que cette activité est très coûteuse. Le test, qui constitue une des techniques maîtresses pour la validation quand elle est opérée sur un code exécutable, participe pour une très large part à ce coût.

La validation est d'autant plus importante que le système est critique, c'est-à-dire soumis à des contraintes de sûreté particulières. Dans de très nombreux cas, un dysfonctionnement d'un tel système peut mettre en jeu des vies humaines et/ou des sommes d'argent colossales. Parmi les systèmes critiques, sont souvent cités les systèmes de production et distribution d'énergie, les systèmes de pilotage de véhicules (avions, métros,...), les systèmes téléphoniques, etc.

Pour l'ingénierie de ce type de systèmes, l'application des méthodes formelles est d'une utilité bien reconnue. D'une part, elles ne peuvent être appliquées avec profit que si une grande rigueur de description et de travail est observée; cette dernière garantit une qualité de résultat. D'autre part, elles demeurent le seul moyen. Cette automatisation est surtout envisagée à travers des outils de preuve (déductive ou par évaluation sur des modèles), en *vérifiant* que le système est conforme à sa specification. Il faut noter au passage que les opérations de *vérification* constituent un sous-ensemble de celles de validation, car elles ne servent qu'à établir *une conformité avec un énoncé*. La validation a une plus grande portée : elle vise aussi à révéler des erreurs par rapport à des *spécifications implicites ou impossibles à formuler*, et pour toutes les parties non vérifiées, *à établir un niveau de confiance* (éventuellement quantifié) dans les comportements futurs du système.

Les méthodes de preuve restant fortement limitées dans leurs applications par les problèmes d'explosion combinatoire des espaces de recherche, le recours au test est nécessaire. Dans un cadre formel, le test peut servir à révéler des erreurs, à établir une conformité, et/ou plus rarement, à déterminer un niveau de confiance.

Cette thèse porte sur l'automatisation du processus de test de systèmes critiques spécifiés

dans un langage formel (Lustre) en vue de révéler des erreurs dans ces systèmes. L'approche retenue est bien fondée. L'enrichissement que nous en proposons vient accroître la démonstration de la pertinence du test dans la validation de systèmes requérant des méthodes formelles.

Systèmes réactifs

Nombre de systèmes critiques sont *réactifs* [55]. On désigne sous ce nom, les systèmes informatiques dont le rôle est de réagir continûment à leur environnement physique. Ils diffèrent donc des systèmes *transformationnels*, qui disposent de toutes leurs entrées à l'initialisation et délivrent leurs sorties à leur terminaison. Un système réactif réagit à une vitesse déterminée par son environnement. Il diffère donc aussi d'un système interactif, qui interagit continûment avec son environnement, mais à sa vitesse propre (comme, par exemple, un système d'exploitation).

Selon [55], les programmes réactifs sont généralement difficiles à décrire.

Lorsqu'ils doivent contrôler l'évolution d'un phénomène continu (trajectoire, vitesse, température), toutes les évolutions significatives de ce phénomène doivent pouvoir être prises en compte. Ceci impose des contraintes temporelles strictes, aussi bien pour le rythme d'acquisition des données, que pour le temps de réponse du programme.

D'autre part, ils sont souvent intrinsèquement complexes. Par exemple, pour contrôler la réaction nucléaire dans un réacteur, il faut maîtriser de nombreux paramètres tels que la température du cœur, l'évolution du combustible, les éventuelles défaillances des capteurs. Le programme de contrôle doit pouvoir gérer l'ensemble des combinaisons possibles, ce qui représentent des milliers de situations différentes.

Systèmes synchrones

Pour la conception et l'implantation de systèmes réactifs, des langages séquentiels (tels que C, Fortran), des utilitaires temps-réel ou de langages parallèles (CSP, Occam, ADA) ont été utilisés. Mais ces outils ne se sont pas révélés adaptés pour assurer la sûreté requise par beaucoup de ces systèmes. Les raisons en sont : le bas niveau d'abstraction des méthodes, la difficulté d'écrire et de valider les programmes, l'éventuel surcoût à l'exécution [8, 48].

C'est pourquoi, des langages synchrones ont été introduits. Ces langages, simples d'utilisation et définis formellement, fournissent des primitives "idéales" qui permettent de raisonner comme si le programme avait un temps de réaction nul aux événements externes. En pratique, cette *hypothèse de synchronisme* revient à s'assurer que le programme réagit assez vite pour percevoir tous les événements externes dans le bon ordre.

G. Berry décrit l'hypothèse de synchronisme comme une démarche qui consiste à considérer comme instantané ce qui prend du temps, lorsque cette quantité de temps est négligeable par rapport à "l'environnement". Par exemple, le son a un temps de propagation non nul. Pourtant, ce temps de propagation sera considéré comme nul, par toutes les personnes réunies dans une salle.

La famille des langages synchrones est constituée essentiellement du langage impératif Esterel [13], des langages flot de données Lustre [20] et Signal [64], des langages graphiques

Argos [68] (proche des Statecharts [55]), et SAGA/SAO+ [82] (un langage graphique se traduisant en Lustre). Dans la suite, nous ne considérerons que le langage Lustre.

Valider un système réactif

La grande difficulté de la validation des systèmes réactifs réside dans la définition même de ces systèmes : il faut veiller à ce que les contraintes temporelles soient toujours respectées, et s'assurer que les comportements du systèmes ne mènent jamais à des situations dangereuses (aspect sûreté). Or, un système réactif agit de façon continue avec son environnement et une infinité de comportements sont à étudier.

Une petite simplification est possible : la validation d'un programme réactif peut se faire en tenant compte de l'environnement dans lequel il va évoluer. Par exemple, il est physiquement impossible de décrocher deux fois de suite le combiné d'un téléphone sans, entre temps, l'avoir raccroché. La validation d'un programme gérant les services associés à un téléphone se fait en tenant compte de cette contrainte, ce qui limite le nombre de comportements à étudier.

Principe général du test

D'une manière générale, tester un logiciel consiste à l'exécuter en ayant une totale maîtrise des entrées, tout en vérifiant que le comportement du logiciel est bien celui attendu. Cette définition met en évidence deux tâches distinctes dans le processus de test : la description des données d'entrée et celles des sorties attendues du logiciel.

En pratique, l'activité de test repose sur un ensemble de cas de test. Chaque cas de test est un couple : donnée de test à fournir à l'implantation, comportement attendu (résultat) de l'implantation.

Un test est *complet* [75] si l'ensemble de cas de test associé permet la détection de toutes les erreurs de l'implantation. Le test alors est un outil de vérification.

D'une façon générale, il n'est pas possible de produire l'ensemble de tous les cas de test pour réaliser un test complet. Il faut donc **choisir** un nombre raisonnable de cas de tests. Ce choix se fait selon un ou plusieurs *critères* susceptibles de révéler un maximum d'erreurs [75].

Idéalement, un ensemble de critères permet de définir un ensemble de cas de test de taille réaliste, dont le pouvoir de détection des erreurs est équivalent à celui d'un test complet [75]. On classe les critères selon deux types [93] : les critères de sélection et les critères d'adéquation. Un critère de sélection est utilisé pour choisir a priori un ensemble de cas de test, selon un but. Un critère d'adéquation contrôle a posteriori si l'ensemble des cas de test choisi satisfait ce but.

Génération des cas de test : approche générale

L'analyse des cas de test doit permettre de déceler les comportements erronés (défaillances) du logiciel. C'est pourquoi, pour chaque entrée, il faut déterminer si les sorties

du logiciel sont correctes. La détermination des sorties est un problème délicat connu sous le nom du problème de l'*oracle*, qui dépend du type de spécification à notre disposition. Si la spécification est une description précise des couples d'entrée et de sortie admissibles[1] la détermination des sorties est immédiate. Si la spécification est sous forme logique, il faut calculer l'ensemble des valeurs admissibles à partir des formules.

La génération des cas de test dépend essentiellement du type de test envisagé : test fonctionnel ou structurel.

Le test structurel (ou test "boîte blanche") repose sur la couverture d'un (ou plusieurs) élément constitutif de la structure du programme [75]. Dans le secteur industriel, ce type de test est considéré comme une procédure obligatoire de validation à laquelle est attachée une notion purement qualitative de niveau minimum de confiance dans la qualité du produit. L'expérience a montré que le test structurel peut révéler des erreurs en particulier par un choix judicieux des données d'entrée à la suite d'une analyse de la structure à couvrir et des domaines d'entrée. Le test structurel ne fait l'objet d'aucun développement dans cette thèse.

Le *test fonctionnel* (ou test "boîte noire") a pour but de mettre en évidence les situations dans lesquelles le programme ne se comporte pas comme les spécifications l'imposent, sans se préoccuper de la structure du programme [75]. Le test fonctionnel est dérivé des spécifications : l'ensemble des données d'entrée possibles et des sorties associées sont déterminées à partir des spécifications.

On a vu plus haut que si l'ensemble des entrées possibles est de trop grande taille, le test exhaustif sur les entrées est impossible à réaliser. Les données de test sont alors sélectionnées soit de manière aléatoire, soit en appliquant des heuristiques.

Les heuristiques sont surtout utilisées dans le cadre d'une sélection déterministe des données de test. Elles s'appuient sur des hypothèses raisonnables de probabilité de détecter une erreur. Par exemple, les valeurs proches des bornes d'un domaine d'entrée correspondant à un intervalle ont des probabilités différentes de détecter des erreurs [75]. Donc, chacune de ces valeurs doit être prise en compte dans la construction d'un ensemble de cas de test. Les données d'entrée correspondant aux autres valeurs de l'intervalle sont supposées avoir la même probabilité de révéler les (mêmes) erreurs; choisir un seul cas de test les représentant toutes peut être suffisant.

Le choix de données aléatoires relève du *test statistique*, qui consiste à engendrer des données aléatoirement selon une distribution probabiliste du domaine d'entrée [70]. Un cas particulier de ce type de test est représenté par l'utilisation d'une distribution uniforme. Le test fonctionnel statistique répond essentiellement à deux besoins : révéler le plus grand nombre d'erreurs étant guidé par les spécifications, et/ou détermier la fiabilité du logiciel. Le test fonctionnel statistique est largement utilisé dans l'industrie [74]. Par exemple, pour la compagnie IBM, il constitue une des principales inovations de la méthode de développement appelée "cleanroom" [34]. Il est utilisé avec différents objectifs de validation : en particulier, appliqué en conjonction avec des modèles de fiabilité du logiciel, il sert à déterminer le niveau de qualité obtenu.

1. Par exemple, si la spécification est exprimée sous forme d'automate.

Contribution de la thèse

Nos travaux s'inscrivent totalement dans la construction de l'environnement de test Lutess. Dans le cadre de sa thèse, Ioannis Parissis [79] a ouvert la voie du test fonctionnel de systèmes spécifiés en Lustre, et réalisé un premier prototype de Lutess. Il a en particulier posé les fondements de cette approche du test dont la principale caractéristique est de permettre l'automatisation de la génération de données de test à partir de spécifications formelles.

Notre contribution comporte plusieurs points. Dans un premier temps, nous avons validé le prototype de manière formelle et expérimentale, considérant qu'un outil de test se devait d'être validé par le test. A cette occasion, nous avons procédé à certaines améliorations. Puis nous avons poursuivi l'approfondissement de cet axe de recherche en introduisant de nouvelles techniques de test statistique qui s'inspirent des profils opérationnels. Ce travail a été réalisé en préservant un cadre monolangage pour Lutess : Lustre est le seul langage de description pour le testeur, et un formalisme unique est utilisé pour préciser les fondements de toutes les techniques intégrées dans Lutess. Enfin, nous avons mis en œuvre nos techniques en élargissant le champ d'application de Lutess à d'autres types de logiciels (logiciels de télécommunication, par exemple) et à une pratique du test au niveau des spécifications initiales d'un logiciel. Cette application qui a inclus deux études de cas industrielles de taille importante a permis de valider le bien-fondé de notre approche.

Plan de la thèse

Ce manuscrit est composé de quatre parties. La première partie est consacrée à la présentation du cadre de la thèse : le langage Lustre et l'outil Lutess. Elle s'achève avec la description détaillée de la validation que nous avons effectuée des techniques de test implantées dans Lutess antérieurement à nos travaux.

La deuxième partie développe les notions de profil opérationnel et de probabilité conditionnelle. Nous voulons voir ces notions comme complémentaires pour faciliter le test dans des situations dont la fréquence d'occurrence peut être modulée. Les algorithmes permettant leurs mises en œuvre et la validation de leurs implantations complètent cette partie 2.

Les services téléphoniques ont été identifiés comme un terrain pertinent d'application de nos techniques. Nous développons deux études de cas sur ce thème. Chacune nous procure l'occasion d'une démonstration en vraie grandeur de la puissance des techniques que nous avons implantées. La seconde correspond à un concours international d'outils pour la détection d'interactions, que Lutess a remporté.

La quatrième partie regroupe les comparaisons et conclusions.

Première partie

Lutess : un outil de validation de systèmes synchrones

Chapitre 2

Le langage Lustre

2.1 Description du langage

Lustre est un langage synchrone, qui peut être considéré à la fois comme une logique temporelle [81] du passé et comme un langage de programmation. C'est un langage flot de données. Un flot est une séquence de valeurs couplée à une horloge qui indique à quel moment certaines valeurs apparaissent. Une équation sur un (ou des) flot(s) peut être vue comme un invariant. Un programme Lustre (dit *nœud*) est un ensemble d'équations (au sens mathématique du terme : pas d'ordre entre les équations et substitutivité des deux membres d'une équations), spécifiant des identités entre les flots.

Par exemple, la déclaration ci-dessous définit un compteur simple. Le nœud Lustre Compteur a une entrée booléenne (raz) et une sortie entière (s). Le corps du nœud (entre les mots clefs let et tel) est constitué d'une équation définissant la valeur de la sortie s; on donne la signification de cette équation après la présentation des opérateurs du langage.

```
node Compteur (raz : bool)
returns (s : int)
let
    s = 0 ⇔ if raz then 0 else pre(s) + 1
tel
```

a) Variables, horloges, équation, opérateur de données

Une variable (ou une expression) Lustre dénote un flot. Ainsi, chaque variable a une horloge, et sa n-ième valeur est disponible au n-ième top de son horloge. Un programme a un comportement cyclique qui définit son horloge de base. Une variable associée à l'horloge de base est telle que sa n-ième valeur est disponible au n-ième cycle du programme. D'autres horloges plus lentes peuvent être définies.

Chaque variable est définie par une équation : si X est une variable et E une expression, l'équation $X = E$ est définie comme étant la séquence $(x_0 = e_0, x_1 = e_1, \dots x_n = e_n, \dots)$, où $(e_0, e_1, \dots e_n, \dots)$, est la séquence des valeurs de E. De plus, l'équation $X = E$ définit

l'horloge de X comme étant la même que celle de E.

Les expressions sont construites à partir des variables, des constantes (séquence infinie constante sur l'horloge de base) et des opérateurs. Les opérateurs usuels (booléens, arithmétiques, conditionnels) opèrent sur des séquences. Par exemple, l'expression **if** $X > Y$ **then** $X \Leftrightarrow Y$ **else** $Y \Leftrightarrow X$ dénote la séquence de la valeur absolue de $X \Leftrightarrow Y$. X et Y doivent avoir la même horloge, qui sera celle du résultat.

b) Opérateurs sur les séquences

En plus des opérateurs sur les données, Lustre est pourvu d'opérateurs sur les séquences.

- L'opérateur pre est introduit pour mémoriser les valeurs d'une expression d'un cycle sur l'autre. Si X correspond au flot $(x_0, x_1, ..., x_n, ...)$ alors pre(X) dénote le flot $(\perp, x_0, x_1, ..., x_{n-1}, ...)$, où $\perp$ est une valeur indéfinie, comparable à une valeur non initialisée dans les langages impératifs. L'horloge de pre(X) est celle de X.
- L'opérateur "⇔>" (*suivi de*) est utilisé pour l'initialisation des variables. Si X et Y dénotent respectivement les flots $(x_0, x_1, ..., x_n, ...)$ et $(y_0, y_1, ..., y_n, ...)$ alors X⇔> Y dénote le flot $(x_0, y_1, ..., y_n, ...)$.
- les opérateurs when et current permettent d'échantillonner des flots sur des horloges booléennes de la façon suivante :

B	*faux*	*vrai*	*faux*	*vrai*	*faux*	*faux*	*vrai*	*vrai*
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
$y = x$ when B		x_1		x_3			x_6	x_7
current y	*nil*	x_1	x_1	x_3	x_3	x_3	x_6	x_7

Dans l'exemple du compteur ci-dessus, les variables raz et s sont associées à l'horloge de base. La valeur s est égale à zéro à l'état initial et lorsque raz est vrai, sinon elle est incrémentée de 1 par rapport à sa valeur au cycle précédent.

Dans la suite de ce document, nous considérons des programmes n'ayant qu'une seule horloge. Les opérateurs when et current ne sont donc pas utilisés. Ce choix correspond aux mêmes simplifications opérées pour le vérificateur de modèle Lesar [85]. Par ailleurs, le fait qu'un programme Lustre avec plusieurs horloges se ramène à un programme avec une seule horloge est un résultat établi pour la compilation des programmes Lustre [86].

c) Tableaux et noeuds récursifs

Les tableaux et les nœuds récursifs ont été introduits comme une facilité syntaxique. Ils n'augmentent pas le pouvoir de description du langage. Ainsi, les tableaux sont expansés en autant de variables que la taille du tableau le nécessite; et les nœuds récursifs sont instanciés en nœud classiques. La taille d'un tableau et la profondeur maximale d'une récursion doivent donc être connues à la compilation.

d) Assertions

Une *assertion* définit une propriété invariante relative aux variables d'un programme. A l'origine, les assertions ont été introduites afin de simplifier et optimiser le processus de compilation, en restreignant les relations entre les entrées et les sorties. Elles ont montré aussi une grande utilité pour la vérification de programme Lustre en restreignant le domaine des entrées.

e) Opérateurs temporels

Les nœuds sont réutilisables. Ils peuvent servir à définir un ensemble d'opérateurs temporels. Par exemple, l'opérateur temporel "toujours" est donné ci-dessous. En annexe, le lecteur trouvera la définition des opérateurs temporels utilisés dans ce manuscrit.

```
- - toujours : un booléen → un booléen
- - { toujours(x) ⇔ x est vrai depuis le début de la séquence d'entrée }
node toujours ( x : bool ) returns (res : bool)
let
   res = x ⇔> pre res and x ;
tel
```

2.2 Modèle d'exécution de Lustre

Nous reportons ici et en annexe A.1 quelques éléments de la définition formelle de Lustre. L'ensemble de ces éléments est détaillé dans la thèse de Christophe Ratel [85].

La définition du comportement d'un programme Lustre est basée sur la suite des valeurs prises par ces variables. Soit $\mathcal{P}$ un programme Lustre. Soient I l'ensemble de ses identificateurs et V l'ensemble des valeurs prises par ces identificateurs. On nomme *mémoire* de $\mathcal{P}$, toute fonction σ de I dans V. Une mémoire est une instanciation des variables d'un programme; elle définit un *état* du programme. Le comportement du programme dans une exécution est décrit comme la suite de ses mémoires. On appelle *trace d'exécution* Σ une séquence non vide, finie ou infinie de mémoires.

La valeur d'une expression Lustre se calcule à partir d'une trace d'exécution finie ou infinie. Soit E une expression et $\Sigma = (\sigma_0, \sigma_1, ..., \sigma_n)$ une trace finie. On note $\Sigma \vdash E|v$, le fait que l'expression E est évaluée à v après l'exécution de Σ. La sémantique de Lustre basée sur des traces finies est donnée en annexe A.1.1.

A partir de la sémantique sur les traces d'exécution, il est possible de construire une sémantique plus simple, ne reposant que sur les deux dernières instanciations des variables. Cette sémantique est appelée *sémantique opérationnelle* de Lustre. Elle est obtenue en restreignant l'application de l'opérateur pre aux identificateurs (et non plus à des expressions complexes). Ceci n'est pas restrictif, car à toute expression dans un programme Lustre,

on peut substituer une variable (principe de substitution). Grâce à cette hypothèse, il est possible d'évaluer toutes les expressions Lustre sur les deux dernières mémoires de la trace (l'avant dernière étant nécessaire et suffisante pour l'évaluation de toutes les expressions $\mathsf{pre}(x)$, x étant une variable).

La sémantique de Lustre reposant sur les deux dernières mémoires d'une trace est reportée en annexe A.1.2. Cette sémantique permet de définir la compatibilité d'une mémoire σ' avec un programme $\mathcal{P}$ sachant qu'à l'instant précédent, la mémoire était σ. L'évolution du programme est ainsi complètement traduite par l'évolution des mémoires. Par exemple, étant donné une expression x, si $(\sigma_0, ..., \sigma, \sigma') \vdash \mathsf{pre}\ (x)|v$ et $(\sigma_0, ..., \sigma) \vdash x|v$, alors σ' et σ sont compatibles pour x. Cette compatibilité est notée :

$$\sigma, \sigma' \vdash \mathcal{P}$$

A partir de la sémantique opérationnelle reposant sur les deux dernières mémoires, un système de transition sur les mémoires d'un programme $\mathcal{P}$ est défini. La relation de transition $\rightarrow$ est définie comme suit :

$$\forall \sigma, \forall \sigma', \sigma \rightarrow \sigma' \Longleftrightarrow \sigma, \sigma' \vdash \mathcal{P}$$

Le système de transition ainsi obtenu est infini. Pour se ramener à un système fini, une abstraction booléenne du programme Lustre est construite. Cette abstraction est dite booléenne car elle correspond à l'automate de contrôle généré par le compilateur, qui ignore les aspects relatifs aux variables non booléennes. De ce fait, un *modèle d'exécution* d'un programme Lustre repose sur la notion de machine d'états finis, étendue pour tenir compte des assertions.

Dans la suite, pour tout ensemble X de variables booléennes, on note V_X l'ensemble des valeurs possibles des variables de X. $x \in V_X$ est une affectation de valeurs de toutes les variables de X. x est un vecteur de valeurs.

Définition 1 *Le modèle d'un programme Lustre n'ayant que des variables booléennes est une machine d'états finis*
$M = (Q, q_{init}, E, S, \alpha, s, t)$ *où*

- Q *est un ensemble fini d'états,*
- $q_{init} \in Q$ *est l'état initial,*
- E *est l'ensemble des variables d'entrée du programme,*
- S *est l'ensemble de ses variables de sorties,*
- $\alpha : Q \times V_E \rightarrow \{vrai, faux\}$ *est la fonction d'assertion,*
- $s : Q \times V_E \rightarrow V_S$ *est la fonction de sortie,*
- $t : Q \times V_E \rightarrow Q$ *est la fonction de transition (éventuellement partielle).*

2.3 Une représentation du modèle d'exécution : notion de BDD

Dans la suite, nous allons utiliser la notion de diagrammes de décision binaire (*Binary Decision Diagrams* ou BDD), en particulier pour l'implantation de notre outil. Ce mode de

B1	B2	B3	f	B1	B2	B3	f
0	0	0	1	1	0	0	1
0	0	1	1	1	0	1	0
0	1	0	1	1	1	0	0
0	1	1	0	1	1	1	0

1 représente la valeur *vrai* et 0 la valeur *faux*

TAB. 2.1 – *Table de vérité de la fonction f*

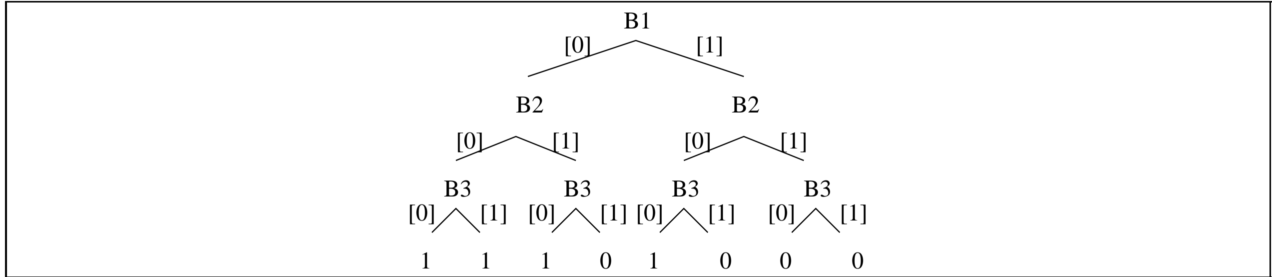


FIG. **2.1** – Arbre de Shannon de la fonction f

représentation des machines d'états finis avait déjà été retenu dans la thèse de Christophe Ratel [85] pour implanter des algorithmes de vérification efficaces.

Le modèle d'exécution est basé sur l'utilisation de fonctions booléennes. Pour représenter et manipuler ces fonctions booléennes, les BDD sont utilisés. Ils ont aussi la propriété de faciliter les calculs sur ces fonctions. Ce paragraphe donne un bref aperçu des BDD.

Ces diagrammes de décision binaire ont été introduits par Sheldon B. Akers en 1978 dans [4], et réutilisés en 1986 par Bryant dans [18].

Un diagramme de décision binaire permet de définir, d'analyser et de tester des fonctions booléennes de grande taille. Un tel diagramme est un graphe orienté, acyclique, et étiqueté.

Pour illustrer le principe des BDD, nous nous intéressons à la fonction booléenne ternaire $f(B1, B2, B3)$. Sachant que f est vraie quand au plus un de ses arguments est vrai, une première façon de représenter f consiste à établir sa table de vérité (table 2.1). Cette table peut se représenter sous la forme d'un arbre de Shannon (figure 2.1). Chaque nœud interne d'un arbre de Shannon représente une variable. Chaque branche issue d'un nœud x représente une instanciation possible de la variable booléenne x (chaque nœud a donc deux arcs "sortants"). A chaque feuille, on trouve une valeur booléenne de la fonction f pour l'instanciation des variables définie par le chemin de la racine à cette feuille.

Soit une fonction booléenne g à n variables $(x_1, x_2, ..., x_n)$. La construction de l'arbre de Shannon repose sur l'application successive de la formule d'expansion de Shannon [88] :

$$g(x_1, x_2, ..., x_n) = (x_1 \wedge g(1, x_2, ..., x_n)) \vee (\neg x_1 \wedge g(0, x_2, ..., x_n))$$

Lorsque l'application successive de la formule d'expansion se fait selon l'ordre des variables, l'arbre obtenu est dit *canonique*.

La taille de la table de vérité d'une fonction booléenne de n variables est 2^n. Sa représentation sous forme d'arbre possède 2^n feuilles. Ces deux représentations sont de taille exponentielle et ne sont donc pas des représentations efficaces.

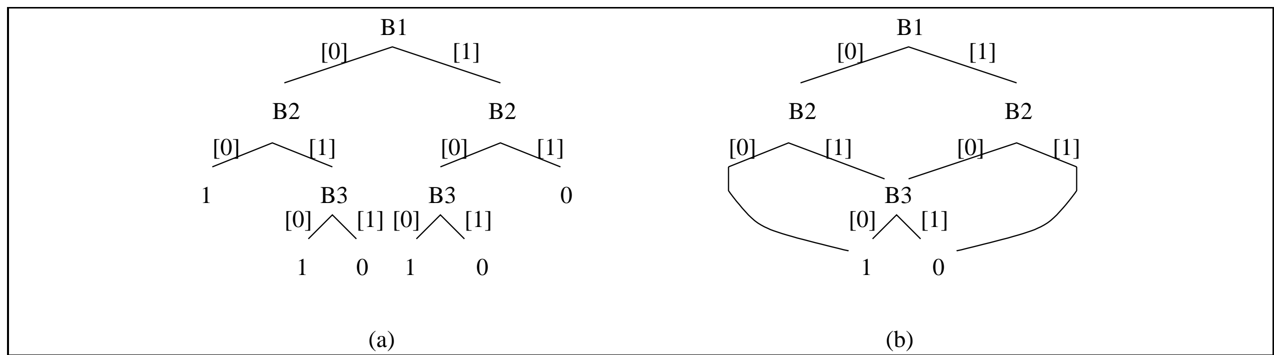


FIG. **2.2** – Simplifications de l'arbre de Shannon

Akers a proposé de réutiliser la structure des arbres de Shannon et de la simplifier par :

1. l'élimination des nœuds redondants,
2. et le partage des sous arbres isomorphes.

La première simplification consiste à éliminer les nœuds qui mènent aux mêmes résultats. Par exemple, pour la fonction f, lorsque $B1$ et $B2$ ont la valeur fausse (resp. vraie), la fonction f a la valeur vraie (resp. fausse). La valeur de f est donc indépendante de la valeur de $B3$ dans ces deux conditions. On peut donc simplifier l'arbre de Shannon comme indiqué par la figure 2.2(a).

La seconde simplification consiste à éliminer un des deux sous-arbres isomorphes et à partager le second. Sur le diagramme 2.2(b), on peut constater le résultat de cette simplification sur les deux sous-arbres de racine $B3$.

Lorsque les deux simplifications sont appliquées sur un arbre de Shannon, on obtient un BDD.

Chapitre 3

L'outil de test Lutess

Random values make a good source of data for testing effectiveness of computer programming
- Donald E. Knuth dans
'The art of computer programming'

Introduction

Lutess est un outil dont le but est de permettre de valider les programmes synchrones essentiellement par un test de type test fonctionnel (boîte noire). Il a été conçu par Ioannis Parissis pendant sa thèse [79], qui en a réalisé un premier prototype.

Le principe général du fonctionnement de Lutess consiste à reproduire (de façon dynamique) les échanges entre un système sous test et le milieu dans lequel il sera utilisé, c'est-à-dire son *environnement*.

Lutess a été mis au point pour permettre une dérivation *entièrement automatique* des données de test à partir de la description de cet environnement. L'automatisation du processus de test a pour but de faciliter la production de données de test. L'objectif implicite est d'augmenter le nombre de données utilisées pour le test et ainsi d'augmenter la confiance que l'on peut avoir envers le système sous test, faute de pouvoir réaliser un test exhaustif.

Lutess est spécialisé pour le test des système réactifs synchrones dont les entrées et sorties sont de type booléen. Ce choix a été fait pour deux raisons. D'une part, le travail sur les booléens exclusivement permet de définir des algorithmes de génération efficaces, d'autre part, il semble que beaucoup de systèmes critiques satisfont ce critère.

La production des données de test repose sur deux méthodes de génération de données différentes, une purement aléatoire et une autre guidée par des propriétés.

Dans la première section, on introduit en particulier les éléments constitutifs de Lutess et son fonctionnement que l'on illustre sur un exemple.

La section suivante est dédiée au moteur de l'outil : son générateur de données. On explique son mode de construction. On y trouve aussi les fondements théoriques du générateur. La présentation est une adaptation simplifiée et personnelle de la proposition originale de

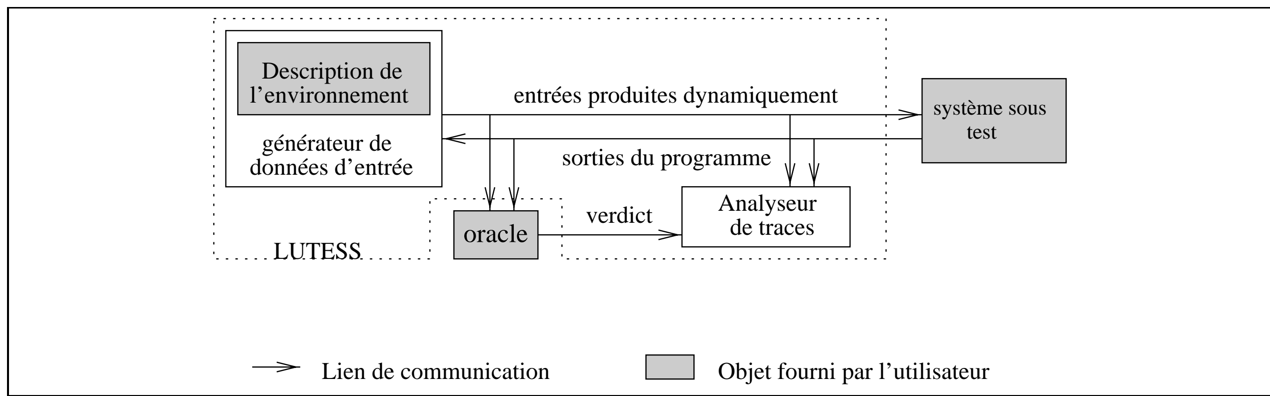


FIG. **3.1** – Architecture de Lutess

Ioannis Parissis. Cette simplification facilite la prise en compte de nouvelles techniques de test dans le même cadre théorique.

La troisième et la quatrième section sont consacrées aux méthodes de sélection de données de test. On y propose des descriptions informelles, algorithmiques, et formelles des deux méthodes de sélection des données décrites dans la thèse de Ioannis Parissis [79]. Dans ces sections, on s'attache à montrer que les algorithmes satisfont leurs spécifications à l'aide d'éléments d'analyse. Ce travail original a demandé une reformulation des algorithmes de sélection proposés dans [79].

3.1 Description générale de Lutess

Lutess fonctionne à partir de quatre éléments qui doivent être fournis par l'utilisateur (fig. 3.1) :

1. le système sous test (que l'on nomme indifféremment programme),
2. un oracle (ou observateur),
3. une description de contraintes que doit respecter l'environnement,
4. deux entiers : un germe pour initialiser le générateur aléatoire et la longueur de la séquence de test souhaitée (i.e. le nombre de cycles voulus).

Dans un premier temps, Lutess produit un *générateur de données* à partir de la description de l'environnement. Puis, Lutess procède de manière cyclique. A chaque cycle (pas de test), il produit un vecteur d'entrée pour le système sous test. Ce vecteur respecte les contraintes d'environnement. La réaction du système sous test est récupérée par le générateur, pour déterminer le vecteur suivant, etc. En parallèle, un programme (l'oracle) observe les échanges entre le simulateur et le système sous test, et émet un verdict à chaque pas de test. L'ensemble des données échangées et des verdicts sont récupérées et stockées sous forme de traces. L'utilisateur peut ensuite visualiser ces traces grâce au module *"analyseur de traces"* [32].

Le système sous test doit être exécutable, réactif, synchrone. Puisque l'on effectue un test fonctionnel, seul *l'exécutable* du programme est utilisé. La forme exécutable a été choisie

pour offrir à l'utilisateur une grande souplesse d'utilisation. Ainsi, le programme peut être conçu à partir de n'importe quel langage de programmation, du moment que les impératifs liés aux aspects synchrones sont satisfaits.

On rappelle que le programme testé doit avoir une interface "booléenne", c'est-à-dire que ses entrées et sorties doivent être de type booléen.

L'oracle est un programme dont le but est d'observer les échanges entre le générateur et le système sous test. Pour ce faire, les variables d'entrée de l'oracle sont les variables d'entrée et les variables de sortie du système sous test. L'oracle a une unique sortie de type booléenne appelée *verdict*. Comme pour le système sous test, l'oracle doit être fourni sous forme d'un exécutable.

On peut distinguer plusieurs utilisations de l'oracle. Il peut être conçu pour indiquer si les comportements du programmes sous tests sont valides, c'est-à-dire s'ils satisfont une certaine spécification. Il peut aussi être utilisé pour détecter des comportements intéressants à observer en détail.

Un oracle peut être construit de différentes façons. On peut réutiliser un programme existant, il suffit alors de s'assurer que son interface est cohérente avec celle attendue de l'oracle. On peut aussi le construire à partir de propriétés sur les entrées et sorties du programme. Un programme Lustre peut alors être construit de façon systématique, à partir de l'expression Lustre des propriétés. Soient E_1, E_2,.., E_n l'expression en Lustre de ces propriétés. La valeur de la sortie de l'oracle est calculée comme étant la conjonction des propriétés E_i.

```
node Oracle (les_entrées_du_programme; les_sorties_du_programme)
return (res: un booléen)
let
    res = E1 and E2 and ... and En ;
tel
```

Le fait que l'oracle exprime fidèlement et complètement les caractéristiques du système à tester est un problème important qui dépasse le cadre de cette thèse. Néanmoins, puisque l'oracle est un programme réactif, on peut utiliser Lutess pour s'assurer qu'il possède ces propriétés de correction et complétude. Ce procédé a été utilisé à de multiples reprises lors des études de cas (voir chapitres 9 et 10), pour s'assurer de la correction de certains oracles. Il consiste à mettre en œuvre Lutess en plaçant l'oracle en position de système sous test. L'analyse des échanges entre ce système sous test et son environnement est alors réalisée manuellement.

La description de l'environnement doit être fournie en Lustre. Le langage Lustre et sa sémantique ont été étendus (dans Lutess) par un ensemble de mots-clefs. En annexe A.2.1, on peut trouver l'extension de la sémantique de Lustre pour Lutess.

La description de l'environnement se fait sous la forme d'un nœud Lustre spécial, dit nœud testeur (*testnode*). Le nœud testeur est utilisé pour la construction du générateur de données. Il a pour entrées, les sorties du programme sous test et pour sorties, les entrées du programme sous test. Les contraintes d'environnement sont décrites à l'aide d'un opérateur

spécial dit *environment*, dont l'argument est une liste d'expressions Lustre. Généralement, ces contraintes décrivent des restrictions sur les valeurs possibles du vecteur d'entrée du programme à tester. Soient C_1, C_2,..,C_m l'expression Lustre de contraintes d'environnement. Un nœud testeur se structure alors ainsi :

```
testnode environment (les_sorties_du_programme)
return (les_entrées_du_programme )
var
    éventuellement_des_variables_locales
let
    environment (C1, C2, ..., Cm);
    calcul_de_la_valeur_des_variables_locales;
tel
```

Un nœud testeur diffère d'un nœud Lustre classique par son aspect non-déterministe. Dans un programme Lustre, la valeur de toutes les variables locales et de sortie est calculée de façon déterministe à chaque cycle du programme. Le nœud testeur permet à l'utilisateur de définir un ensemble de comportements possibles de l'environnement. Cette définition est non-déterministe : le nœud testeur n'impose pas une définition déterministe des variables de sortie.

A partir de la description du nœud testeur, un générateur de données de test est construit automatiquement. Ce générateur est *aléatoire* et *contraint*. Il est contraint car il a la propriété de produire des données de test telles que les contraintes d'environnement (C_1...C_m ci-dessus) sont toujours vraies. Le générateur est *aléatoire* dans la mesure où lorsque que les contraintes n'imposent pas une valeur pour une variable, il utilise un générateur de nombres aléatoires (la fonction C *rand*) pour déterminer la valeur de cette variable. Le générateur aléatoire contraint est détaillé dans la section suivante.

Le germe est utilisé par Lutess pour initialiser le générateur de nombres aléatoires. En variant les germes, l'utilisateur fait varier les comportements de l'environnement. En utilisant deux fois le même germe avec un programme et une description de son environnement, on obtient les mêmes traces d'exécution. En effet, le programme sous test est déterministe et le générateur aléatoire (la fonction *rand*) produit la même suite de valeurs lorsqu'il est initialisé avec un même germe.

Lutess produit des *séquences* d'échanges entre le générateur contraint et le système sous test. En plus du germe, l'utilisateur doit fixer le nombre de pas de test souhaité pour une séquence et le nombre de séquences à produire. Sachant que la construction du générateur de données de test est souvent coûteuse (cf. section suivante), il a été prévu un mécanisme permettant d'enchaîner la production de plusieurs séquences de test à partir d'un générateur de données d'entrée. Pour chaque nouvelle séquence, le programme est réinitialisé mais pas le générateur de nombres aléatoires, ce qui permet l'obtention de plusieurs séquences différentes. L'inconvénient de cette méthode est qu'il n'est pas possible de reproduire les séquences séparément (exceptée la première).

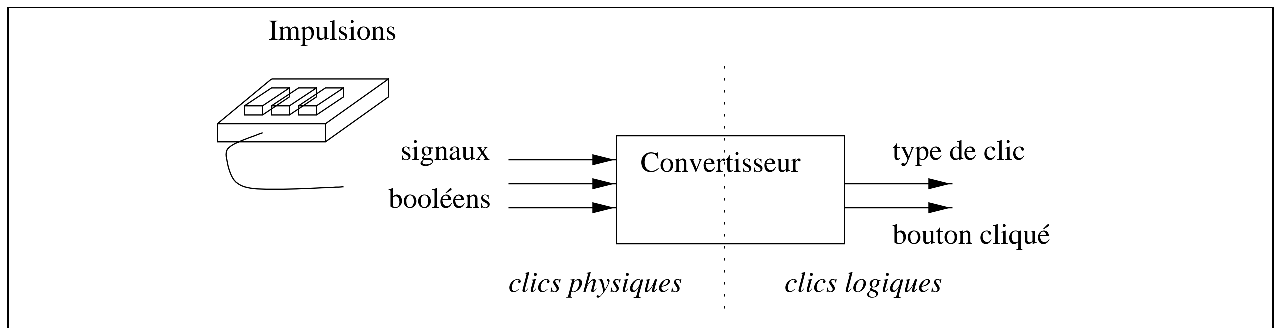


FIG. **3.2** – Convertisseur de clics physiques en clics logiques

Exemple d'un convertisseur de clics (une souris)

L'exemple que nous avons retenu pour une première illustration est un programme chargé d'interpréter une suite de clics physiques sur les boutons d'une souris en une suite de clics logiques. La souris considérée a trois boutons. (voir figure 3.2).

Un clic physique est représenté par une impulsion sur l'un des boutons de la souris. Un clic logique correspond à un simple, un double ou un triple clic sur un même bouton. On suppose qu'au plus un clic physique peut être émis à chaque cycle du programme. Soit T un entier définissant la vitesse du triple clic (en nombre de cycle de l'horloge de base du programme). Un triple clic logique doit être détecté lorsque le même bouton est cliqué trois fois de suite en moins de T cycle. Un double clic (resp. un simple clic) doit être détecté lorsqu'un même bouton est cliqué exactement deux fois de suite (resp. une seule fois) en moins de T cycles. Le triple clic est prioritaire sur le simple et le double clic. Le double clic est prioritaire sur le simple clic. Pour détecter un triple clic, il faut que T soit supérieur ou égal à trois cycles. Dans l'exemple ci-dessous, en supposant que $T = 4$, on doit successivement constater un triple clic (t), puis un simple (s) , puis un double (d).

top d'horloge	1	2	3	4	5	6	7	8	9	10	11	12	...
bouton cliqué	-	1	-	1	1	-	1	2	-	2	-	-	...
type de clic logique	-	-	-	-	t	-	-	s	-	-	d	-	...

Le programme de conversion a été réalisé sous la forme d'un programme réactif synchrone. L'interface du programme est booléenne; elle est constituées de 3 entrées (B1,B2,B3) et de 6 sorties (k1,k2,k3, r1,r2,r3). Les variables d'entrée représentent les trois boutons de la souris. Les variables de sortie caractérisent le clic logique : ki représente le type de clic (simple, double ou triple) et ri, représente le bouton cliqué.

La seule contrainte d'environnement que nous souhaitons observer est “au plus un bouton est cliqué à chaque instant”. Le nœud testeur correspondant sera donc décrit ainsi :

```
testnode EnvironnementConvertisseur (k1,k2,k3, r1,r2,r3: boolean)
return (B1,B2,B3 : boolean)
let
   environment( #(B1,B2,B3) );
      -- # est un opérateur Lustre prédéfini “AuPlusUn”,
      -- qui est vrai lorsqu'au plus un élément de sa liste d'arguments est vrai.
```

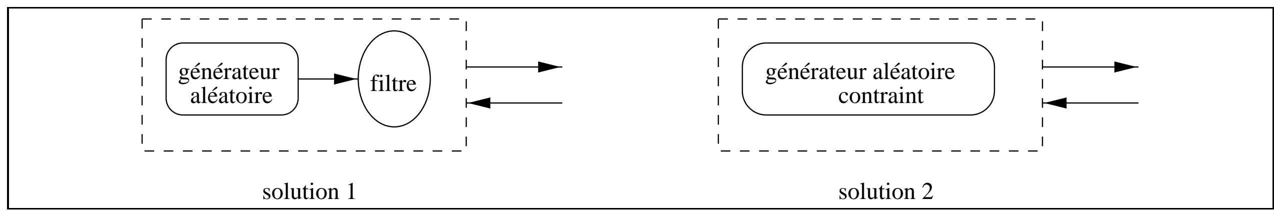


FIG. **3.3** – Deux architectures possibles de générateurs aléatoires contraints

tel

Quelques propriétés d'oracle simples que l'on peut vouloir observer sont :

- Un type de clic ne peut être annoncé sans que ne soit précisé le bouton cliqué et vice versa.
  (k1 or k2 or k3) = (r1 or r2 or r3)
- Un clic logique ne peut pas être de plusieurs types à la fois.
  #(k1, k2, k3)
- Un clic logique ne peut pas concerner deux boutons à la fois.
  #(r1, r2, r3)
- Il ne peut pas y avoir de clic logique si aucun bouton physique n'a été cliqué pendant T tops ($T = 4$).
  Soit tmp une variable locale indiquant si un bouton a été cliqué au cycle courant et tmp$i$ une variable locale indiquant si un bouton a été cliqué durant les i derniers cycles.
  $$\begin{cases} \text{tmp} = \text{not (B1 or B2 or B3)} \\ \text{tmp2} = \text{false} \Leftrightarrow \text{pre tmp and not (B1 or B2 or B3)} \\ \text{tmp3} = \text{false} \Leftrightarrow \text{pre tmp2 and not (B1 or B2 or B3)} \\ \text{tmp4} = \text{false} \Leftrightarrow \text{pre tmp3 and not (B1 or B2 or B3)} \\ \text{prop} = \text{tmp4} => \text{not (k1 or k2 or k3)} \end{cases}$$

3.2 Le générateur de données de Lutess

3.2.1 Description informelle

Examinons le fonctionnement de Lutess de façon plus précise. On peut distinguer essentiellement deux parties dans Lutess : une qui produit les données et une autre qui assure la connexion entre les différents éléments. La seconde partie, purement technique n'est pas décrite dans ce document. Ici, nous nous intéressons au principe de production des données.

D'une façon générale, il existe deux façons de procéder pour produire des valeurs respectant un ensemble de contraintes (figure 3.3). Soit les valeurs sont produites indépendamment des contraintes; les valeurs ne respectant pas les contraintes sont ensuite rejetées. Soit les valeurs sont produites de telle sorte qu'elles respectent les contraintes.

La première technique de génération (tirages avec rejet éventuels) est relativement facile à mettre en œuvre. Pour produire une donnée conforme, on utilise un générateur de données

aléatoires non contraint, on teste la donnée par rapport aux contraintes, et la procédure est recommencée jusqu'à obtenir une donnée satisfaisant les contraintes. La deuxième technique (tirages sans rejet), généralement plus difficile à mettre en œuvre, permet par contre un tirage de valeur en temps constant.

Lorsque le nombre de valeurs autorisées par les contraintes est petit par rapport au nombre des valeurs possibles (i.e. l'environnement est très contraint), la première technique peut coûter très cher à l'exécution, du fait d'un taux de rejet potentiellement important. C'est pourquoi la deuxième technique a été choisie.

La construction du générateur de données de test consiste en la compilation des contraintes d'environnement sous la forme d'un automate d'états fini. Cet automate est non déterministe, et est capable de reconnaître toutes les séquences d'entrées-sorties satisfaisant ces contraintes. La construction du générateur est décrite dans 3.2.3.

Par ailleurs, une propriété attendue du générateur est qu'il ne doit jamais se bloquer; quelles que soient les réactions du programme et l'état de l'environnement, le générateur doit être capable de fournir une entrée au programme. Cette propriété dépend essentiellement des contraintes d'environnement. Par exemple, un utilisateur peut définir des contraintes d'environnement contradictoires. A la construction du générateur, Lutess détecte donc certaines incohérences, telle que la présence d'états de l'environnement qui ne permettent pas de produire des données de tests. Cet aspect est abordé dans 3.2.2.

3.2.2 Aspect formel du générateur

Ce paragraphe présente les bases formelles du générateur aléatoire contraint. L'étape de formalisation permet d'identifier les limites théoriques du travail.

Un générateur aléatoire contraint est défini formellement comme un couple : une machine génératrice et un algorithme de sélection des données de test. Les algorithmes de sélection dépendent de la méthode de génération choisie par l'utilisateur. Elles font chacune l'objet d'une section particulière. Dans cette section, nous nous intéressons uniquement à la machine génératrice.

Les comportements d'un nœud testeur sont caractérisés formellement par une sémantique sur ses traces d'exécution analogue à la sémantique opérationnelle de Lustre, donnée en annexe A.2.1. A partir de cette sémantique opérationnelle, un modèle d'exécution du nœud testeur est défini.

Une machine génératrice est une machine d'entrée-sortie réactive qui communique avec le programme à tester. Les entrées (resp. les sorties) de la machine sont les sorties (resp. les entrées) du programme à tester. A l'origine, le modèle des machines d'entrée-sortie a été choisi pour décrire les machines génératrices car il s'agit d'un modèle plus général que les machines d'états finis.

Définition 2 *Une machine d'entrée-sortie M est un tuple $(Q, q_{init}, A, B, t, s)$ où*

- *Q est un ensemble fini d'état,*
- *$q_{init} \in Q$ est l'état initial,*
- *A est l'ensemble des entrées,*

- *B est l'ensemble des sorties,*
- *$t : Q \times V_A \times V_B \rightarrow Q$ est la fonction de transition,*
- *$s : Q \rightarrow V_B$ est la fonction de sortie.*

Définition 3 *Une machine d'entrée-sortie $M = (Q, q_{init}, A, B, t, s)$ est réactive si et seulement si :*

$$\forall q \in Q, \forall a \in V_A, \exists b \in B, \exists q' \in Q,\ q' = t(q, a, b)$$

Définition 4 *Une machine génératrice, c'est-à-dire une machine associée à un générateur contraint, est une machine d'entrée-sortie réactive $M_{env} = (Q, q_{init}, O, I, env, t_{env}, \phi_{env})$ où :*

- *I (resp. O) est l'ensemble des variables d'entrée (resp. de sortie) du système sous test,*
- *Q est l'ensemble des états de l'environnement; un état q est une instanciation des variables locales L du nœud testeur,*
- *$env \subseteq Q \times V_I$ représente les contraintes d'environnement,*
- *$t_{env} : Q \times V_O \times V_I \rightarrow Q$ est la fonction de transition contrainte par env, c'est-à-dire $t_{env}(q, o, i)$ est définie si et seulement si $(q, i) \in env$,*
- *$\phi_{env} : Q \backslash \{q | \nexists i, (q, i) \in env\} \rightarrow V_I$ est la fonction de sortie. Il s'agit d'une fonction non déterministe qui exhibe l'ensemble de toutes les entrées valides du système sous test dans un état q.*

Remarque 1: Les contraintes d'environnement ϕ_{env} peuvent être fournies par l'utilisateur sous forme de formules utilisant les variables de sorties. Mais à chaque cycle, ces contraintes ne peuvent dépendre que des valeurs de ces variables aux instants précédents.

~⊜ Par exemple, il n'est pas possible d'utiliser la contrainte "B2 => not (k1 or k2 or k3" indiquant que "cliquer sur le bouton B2 à un instant donné implique qu'il n'y a pas de clic logique à cet instant". ⊜~

Remarque 2: Les variables de L sont des variables intermédiaires créées pour l'évaluation des contraintes d'environnement. Par exemple, elles peuvent être utilisées pour mémoriser des valeurs de variables de sorties aux instants précédent.

Remarque 3: Pour déterminer globalement si *env* est génératrice, il faut calculer l'ensemble des états accessibles ou son complément (i.e. l'ensemble des états menant inévitablement à une violation de *env*) [79]. La méthode consiste à calculer un plus petit point fixe [49]. Plus précisément, soit *post* : $Q \rightarrow 2^Q$ la fonction permettant de calculer les états successeurs d'un ensemble d'états :

$$post : (q) = \{q' |\, \exists (o, i) \in V_O \times V_I, t(q, o, i) = q'\}$$

Soit $\widetilde{post}(q_{init})$ l'image de q_{init} sous la fermeture transitive de *post*. *env* est génératrice si $\forall q \in \widetilde{post}(q_{init}) \Rightarrow \exists (i, o, q'), q' = t(q, o, i) \wedge q' \in \widetilde{post}(q_{init})$.

Le calcul de $\widetilde{post}(q_{init})$ qui correspond à une analyse d'accessibilité, n'est pas toujours possible pratiquement. Toutefois, un générateur aléatoire contraint peut être construit sans cette étape de calcul préalable. Il suffit qu'à chaque pas de test, ce générateur s'assure qu'il

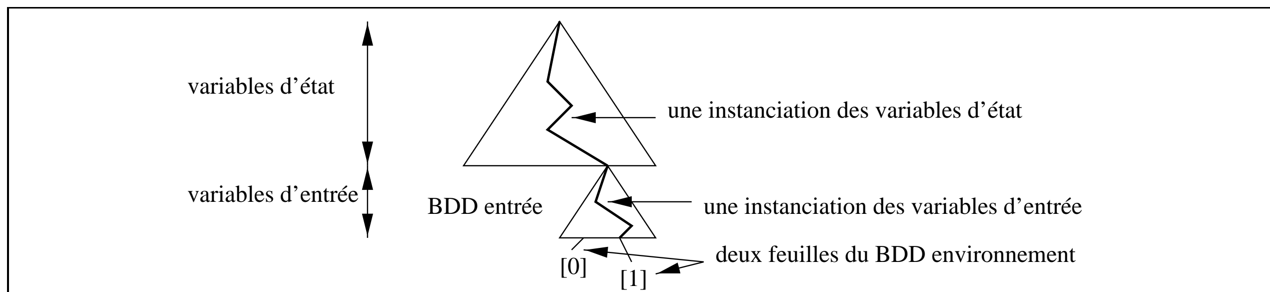


Fig. **3.4** – Stucture générale d'un BDD d'environnement

existe des entrées vérifiant *env*. Lutess est implanté de sorte qu'il peut toujours détecter un état de blocage. Il est de la responsabilité de l'utilisateur de réécrire les contraintes d'environnement lorsque le générateur détecte une telle situation.

3.2.3 Aspects techniques

Pour construire le générateur, on s'appuie sur les outils existants associés au langage Lustre. La construction se fait en quatre temps. Tout d'abord, le nœud testeur est transformé en un nœud Lustre "classique". Puis ce nœud est compilé pour obtenir un automate déterministe.

Cet automate est transformé en un automate non déterministe, puis il est enregistré sous la forme d'un BDD. Les états de cet automate représentent les états de l'environnement. L'état de l'environnement est défini par la valeur de toutes les variables d'état de l'automate.

Ces variables d'états forment la partie supérieure du BDD d'environnement (fig. 3.4). Un état de l'environnement est représenté par un chemin dans la partie supérieure du BDD environnement, de la racine à la première variable d'entrée du programme.

La partie inférieure est composée des variables de sorties du nœud testeur, c'est-à-dire les variables d'entrée du programme. L'ordre de ces variables dans le BDD est défini par l'ordre de déclaration des variables en sortie du nœud testeur. A chaque variable, un nombre représentant cet ordre est associé; ce nombre est appelé l'*indice* de la variable.

3.3 Méthode de génération aléatoire

3.3.1 Motivations et principe général

Le but de cette méthode est de permettre une première mise en œuvre simple de la validation du système sous test. Cette méthode simple ne demande pas d'autres efforts que celui de la spécification de l'environnement. En se basant sur cette seule spécification, le générateur n'a pas de critère pour choisir une donnée (satisfaisant les contraintes) plutôt qu'une autre. C'est pourquoi, la première méthode de Lutess est une méthode de sélection des données aléatoire selon une loi de probabilité équiprobable. Ainsi, dans chaque état de l'environnement, chaque vecteur d'entrée respectant les contraintes d'environnement a la même probabilité d'être choisi que les autres vecteurs d'entrée valides de cet état. Ainsi, si un état q définit n vecteurs d'entrée valides, ils ont chacun la probabilité $\frac{1}{n}$ d'être choisis.

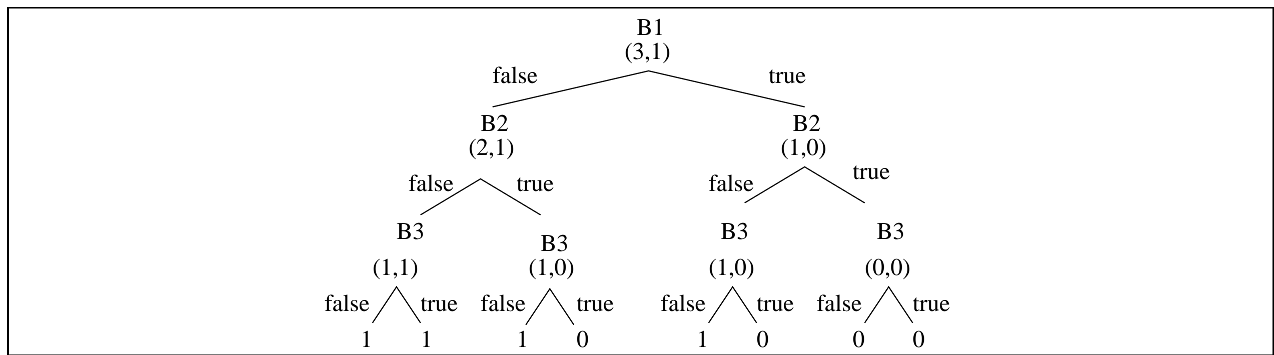


FIG. **3.5** – Arbre de Shannon étiqueté représentant l'environnement du programme de la souris

Les algorithmes présentés ci-dessous sont donnés selon deux versions, une première simplifiée sur un arbre de Shannon, puis une seconde sur la structure de BDD.

3.3.2 Algorithme de sélection des valeurs

Problématique

A chaque cycle, les variables d'états ont toutes une valeur et définissent l'état courant de l'environnement. Cet état désigne un BDD déterminant les vecteurs d'entrée valides pour ce cycle. Pour choisir le vecteur d'entrée, il faut parcourir un chemin de ce BDD jusqu'à une feuille étiquetée *vrai*. On procède en choisissant la valeur des variables les unes après les autres, dans l'ordre d'apparition dans le BDD. Le problème consiste donc, pour chaque variable, à choisir une valeur telle que le chemin final dans le BDD inférieur aboutisse à une feuille étiquetée *vrai*, c'est-à-dire choisir la valeur des variables de façon pertinente. L'idée consiste à associer à chaque nœud un couple d'entiers (v_0, v_1). v_0 (resp. v_1) représente le nombre de vecteurs d'entrée satisfaisant les contraintes lorsque la variable associée au nœud est affectée à faux (resp. vrai). Ce couple d'entiers est appelé *une étiquette*.

Etiquetage d'un arbre de Shannon

Pour calculer l'ensemble des étiquettes d'un arbre de Shannon, il suffit d'un unique parcours récursif de sa structure. Ci-dessous, on décrit l'algorithme d'étiquetage d'un arbre de Shannon de façon informelle.

CalculLabel : un ArbreDeShannon non vide $\rightarrow$ un ArbreDeShannon étiqueté

CalculLabel(A) =
si les sous-arbres de A sont des feuilles { *cas de base de la récursion* }
alors soit *fg* et *fd* sont les valeurs des sous-arbres gauche et droit de A
  { *fg et fd sont des booléens représentés par des entiers 0 (pour faux) et 1 (pour vrai)* }
  dans l'étiquette de A est $\langle fg, fd \rangle$.
sinon { *cas général de la récursion* }
  appliquer récursivement CalculLabel aux sous-arbres gauche et droit
  soit $\langle lgg, ldg \rangle$ et $\langle lgd, ldd \rangle$ sont les étiquettes des sous-arbres gauche et droit

dans l'étiquette de est $\langle lgg + ldg, lgd + ldd \rangle$

Etiquetage d'un BDD

Un algorithme similaire est utilisé pour l'étiquetage des BDD. On rappelle qu'un BDD est obtenu à partir d'un arbre de Shannon à partir des deux transformations (a) et (b) schématisées par la figure 2.2. La transformation (b) consiste en un partage des sous-arbres isomorphes. Cette transformation ne pose pas de problème : si les sous-arbres sont isomorphes, ils ont le même étiquetage.

La transformation (a) consiste à éliminer les nœuds redondants. Sur les chemins "simplifiés", les variables inutiles disparaissent (elles sont abstraites). Par exemple, pour la souris, lorsque B1 et B2 ont la valeur *faux*, la variable B3 est abstraite. Lorsque l'on instancie la variable B3, on obtient deux vecteurs possibles (0,0,0) et (0,0,1).

Intuitivement, une variable est abstraite lorsqu'elle peut être indifférement instanciée à *vrai* ou à *faux*. N variables consécutives définissent 2^N instanciations. Par rapport à l'algorithme d'étiquetage de l'arbre de Shannon, il faut donc multiplier la valeur ldg (resp. ldd) par le nombre d'instanciations défini par les variables abstraites entre la racine et le sous-arbre gauche (resp. droit) (cf. l'algorithme ci-dessous).

Le nombre de variables abstraites se calcule à partir de l'indice des variables dans le BDD (cf. page 33). Soient n_i et n_j deux nœuds consécutifs sur le BDD d'entrée. Soient x_i et x_j les variables portées par n_i et n_j, et I et J leurs indices respectifs. Si x_i et x_j sont consécutives alors $J \Leftrightarrow I = 1$.

Indice : un BDD non vide et non réduit à une feuille → un entier
{ *i = Indice(B) est l'indice de la variable portée par la racine de B* }
CalculLabelDuBDD : un BDD non vide, un entier → un BDD étiqueté
{ *CalculLabelDuBDD(A,N) calcule l'étiquetage de A, n est le nombre total de variables de A* }

CalculLabelDuBDD(A,N) =
{ *examen du sous-arbre gauche* }
si le sous-arbre gauche de A est une feuille
alors soit fg la valeur du sous-arbre gauche de A { *fg est un entier* }
  dans lg = N - Indice(A) { *nombre de variables abstraites* }
sinon
  appliquer récursivement CalculLabelDuBDD aux sous-arbres gauche et droit
  soit $\langle lgg, ldg \rangle$ et $\langle lgd, ldd \rangle$ les étiquette des sous-arbres gauche et droit
    $fg = lgg + ldg$, $fd = lgd + ldd$
    lg = Indice(sous-arbre gauche) - Indice(A) { *nombre de variables abstraites* }
    ld = Indice(sous-arbre droit) - Indice(A)
  dans l'étiquette de A est $\langle fg * 2^{lg}, fd * 2^{ld} \rangle$

Le BDD étiqueté représentant l'environnement du convertisseur de clics (~⊜) est donné figure 3.6.

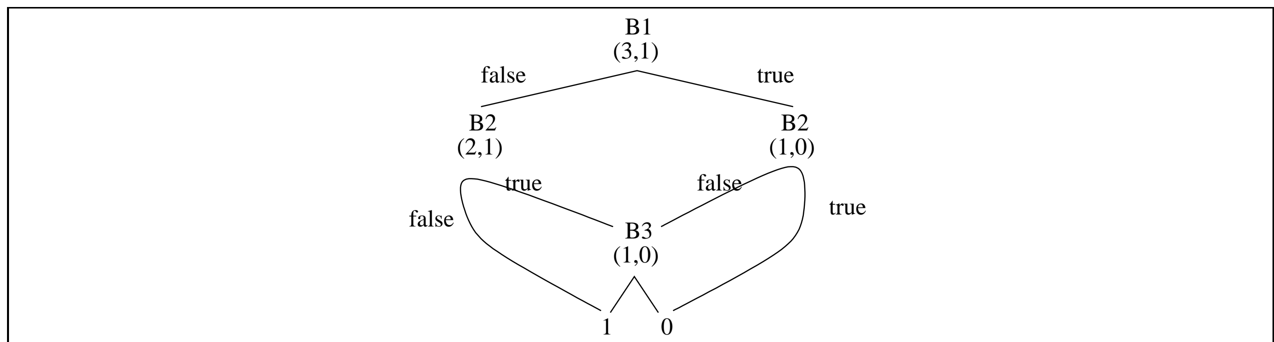


FIG. **3.6** – BDD étiqueté représentant l'environnement du convertisseur de clics

Algorithme de sélection pour un arbre de Shannon

A chaque cycle, Lutess choisit un vecteur grâce à son algorithme de sélection. L'instanciation des variables d'états définit l'état de l'environnement, et désigne un arbre de Shannon d'entrée. L'algorithme de sélection consiste en un parcours de cet arbre de sa racine à une feuille (étiquetée *vrai*). Dans un premier temps, on explique l'algorithme en s'appuyant sur un arbre de Shannon, puis on explique les modifications apportées à cet algorithme pour l'appliquer à des BDD.

Pour chaque nœud de l'arbre de Shannon, on procède de la même façon. Soit (x, lg, ld) l'étiquette portée par le nœud. Si lg (resp. ld) est nul, x sera affectée à vrai (resp. faux). Si lg et ld sont tous les deux non nuls, on choisit la valeur de x en fonction de la valeur *(ld/lg+ld)*.

```
tirage : un réel ∈ [0..1] → un booléen
{ tirage(i)la valeur vrai avec la probabilité i }

CalculVE : un ArbreDeShannon étiqueté non vide, un vecteur de booléen →
      un vecteur de booléen ou "erreur"

CalculVE(A,v) =
si A est une feuille alors
   si A=faux alors "erreur"
   sinon v
sinon                              { A n'est pas une feuille }
   soit ⟨lg, ld⟩ l'étiquette de la racine de A
       i l'indice de la variable portée par la racine de A
   dans
      si lg = 0
      alors si ld = 0
           alors "erreur"
           sinon { v[i] = vrai; CalculVE(D,v) }
      sinon si ld = 0
           alors { v[i] = faux; CalculVE(G,v) }
           sinon v[i] = tirage(ld/ld+lg)
                si v[i] = vrai alors CalculVE(D,v) sinon CalculVE(G,v)
```

Algorithme de sélection pour un BDD

La transformation de l'algorithme pour le travail sur les BDD est similaire à celle de l'algorithme d'étiquetage. Comme précédemment, la transformation (b) - partage des sous-arbres isomorphes - ne pose pas de problème car si les sous-arbres isomorphes ont le même étiquetage.

L'algorithme de choix sur les BDD doit instancier les variables abstraites. La stratégie consiste à affecter ses variables à vrai avec une probabilité 0.5.

```
CalculVE' : un BDD étiqueté non vide, un vecteur de booléen, 2 entiers →
        un vecteur de booléen ou "erreur"
{ CalculVE'(A,v,i,N) calcule une instanciation de v pour le BDD A;
   N est le nombre de variables de la fonction A, et i est le numéro de la
   variable courante à instancier. Les variables sont numérotées de 0 à N-1.}

CalculVE'(A,v,i,N) =
   si A est une feuille
      si A = faux alors "erreur"
      sinon pour k = i à N-1   { v[k] = tirage(0.5) }
   sinon                               { A n'est pas une feuille }
      soit ⟨lg,ld⟩ l'étiquette de la racine de A
           j l'indice de la variable portée par la racine de A
      dans
           pour k = i à j-1   { v[k] = tirage(0.5) }
           si lg = 0
           alors si ld = 0
                 alors "erreur"
                 sinon { v[i] = vrai; CalculVE'(D,v,i+1,N) }
           sinon si ld = 0
                 alors { v[i] = faux; CalculVE'(G,v,i+1,N) }
                 sinon { v[i] = tirage(lg/(ld+lg));
                         si (v[i] = vrai )
                         alors CalculVE'(D,v,i+1,N)
                         sinon CalculVE'(G,v,i+1,N)
```

Dans le paragraphe suivant, on montre que cet algorithme permet de choisir les vecteurs de donnée de façon équiprobable.

L'algorithme général de sélection des données est le suivant :

Lexique
- $M_{env} = (Q, q_{init}, S, E, env, t_{env}, \phi_{env})$: la machine d'entrée-sortie associée au noeud de test
- { *ϕ_{env} est représenté par un BDD et $\forall q \in Q$, $\phi_{env}(q)$ détermine un BDD d'entrée* }
- q : un état de M_{env}
- vs : une instanciation de S (vs $\in V_S$)
- ve : une instanciation de E (ve $\in V_E$)

A : un BDD
Génération :
q = q_{init}
à chaque pas de test faire :
A = ϕ_{env}(q); *sélection du BDD d'entrée*
vs = CalculVE'(A,vs);
envoyer(vs); *vers le système sous test*
récupérer(ve); *produit par le système sous test*
q = t_{env}(q,vs,ve); *calcul du nouvel état*

3.3.3 Formalisation

Un générateur est défini formellement comme un couple : une machine génératrice et un algorithme de sélection des données.

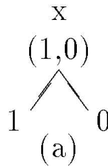
Le générateur $\mathcal{G}$ associé à cette méthode est donc $\mathcal{G} = (M_{env},$ Génération$)$ où $M_{env} = (Q, q_{init}, S, E, env, t_{env}, \phi_{env})$ est la machine d'entrée-sortie associée au nœud de test et Génération est l'algorithme défini précédemment.

Analyse de l'algorithme de sélection

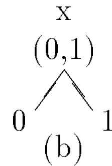
Dans cette partie, on veut montrer que l'algorithme de sélection produit une distibution uniforme des vecteurs d'entrée valides.

On suppose que l'algorithme d'étiquetage est correct et que Random est un générateur de données aléatoire et uniforme sur [0..1]. La propriété à montrer est : $\mathcal{P}(n)$: *l'algorithme de sélection des données est équiprobable pour un BDD représentant une fonction de n variables : l'ensemble des vecteurs d'entrée possibles définis par f ont la même probabilité; on suppose que les fonctions étudiées définissent au moins un vecteur valide.*

On montre cette propriété par récurrence. Tout d'abord, la propriété est vraie pour $n = 1$. Soit une fonction à un paramètre $f(x)$. Il existe 3 possibilités pour f.



(a)



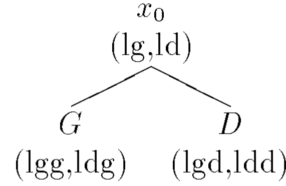
(b)

(c)

Pour (a) (resp. (b)), l'algorithme de sélection produit x=*false* (resp. x=*true*) avec une probabilité égale à 1. Pour ces deux cas, le domaine de f est réduit à un vecteur, $\mathcal{P}$ est trivialement vraie. Pour (c), l'algorithme de sélection produit x=*false* (resp. x=*true*) avec une probabilité égale $\frac{1}{1+1} = \frac{1}{2}$. Les deux instanciations possibles de x ont la même probabilité. L'analyse montre que $\mathcal{P}(1)$ est satisfaite.

On suppose maintenant $\mathcal{P}(n)$, et l'on montre $\mathcal{P}(n+1)$. Soient $(x_0, x_1..., x_n)$ les paramètres de f. Soit A le BDD canonique représentant $f(x_0, x_1..., x_n)$. Deux cas se présentent : (1) A a pour racine x_0, (2) A a pour racine $x_i, i > 0$.

Dans le cas (1). Soient G et D les sous-arbres gauche et droit de A, et (lgg,ldg), (lgd,ldd) l'étiquette de leur racine respective.



G et D représentent chacun une fonction de n paramètres : $f(0, x_1, ..., x_n)$ et $f(1, x_1, ..., x_n)$. Sachant que $\mathcal{P}(n)$ est vraie, on peut en conclure que :

- $lg = lgg + ldg$ est le nombre de vecteurs valides pour G et chacun a la probabilité $\frac{1}{lg}$ d'être choisi,
- $ld = lgd + ldd$ est le nombre de vecteurs valides pour D et chacun a la probabilité $\frac{1}{ld}$ d'être choisi.

Comme pour le cas de base, on peut distinguer 3 cas.

- $lg = 0$ et $ld \neq 0$
  L'algorithme de sélection produit x=*true*. A définit ld instanciations possibles. Chacune a la probabilité $\frac{1}{ld}$ d'être choisi.
- $lg \neq 0$ et $ld = 0$
  L'algorithme de sélection produit x=*false*. A définit lg instanciations possibles. Chacune a la probabilité $\frac{1}{lg}$ d'être choisi.
- $lg \neq 0$ et $ld \neq 0$
  L'algorithme de sélection produit x=*false* avec la probabilité $\frac{lg}{lg+ld}$ et x=*true* avec la probabilité $\frac{ld}{lg+ld}$.
  A définit $lg + ld$ instanciations possibles. Les lg instanciations possibles pour lesquelles $x_0 = false$ ont la probabilité $\frac{lg}{lg+ld} * \frac{1}{lg} = \frac{1}{lg+ld}$ d'apparaître. De même, ld instanciations possibles pour lesquelles $x_0 = true$ ont la probabilité $\frac{ld}{lg+ld} * \frac{1}{ld} = \frac{1}{lg+ld}$ d'apparaître.
  Ainsi, chacune des $lg + ld$ instanciations ont la probabilité $\frac{1}{lg+ld}$ d'être choisie.

Dans le cas (2), la variable x_0 est abstraite. L'algorithme de sélection choisit l'instanciation $x_0 = true$ (resp. $x_0 = false$) avec la probabilité 0.5. Si cette variable a été abstraite, c'est que $f(1, x_1, ..., x_n)$ et $f(0, x_1, ..., x_n)$ sont égales, par définition des BDD. L'algorithme s'applique récursivement pour instancier les n variables restantes. Comme $\mathcal{P}(n)$ est vraie, l'ensemble des vecteurs valides définis par $f(1, x_1, ..., x_n)$ (et ceux définis par $f(0, x_1, ..., x_n)$) ont la même probabilité. Soit p cette probabilité. Tous les vecteurs définis par f ont la probabilité $0.5 * p$ d'apparaître.

3.4 Méthode de génération guidée par des propriétés

3.4.1 Motivations et principe général

Une propriété de sûreté est une propriété qu'un logiciel doit toujours satisfaire. La validation d'un programme doit permettre de s'assurer que toutes les propriétés de sûreté sont bien respectées.

Soit PS une propriété de sûreté définie sur les entrées E et les sorties S du programme. La valeur de PS peut être déterminée par la sémantique des traces des programmes Lustre (cf. page 21).

A chaque pas de test, on peut caractériser les entrées du programme en deux ensembles : celles qui sont pertinentes par rapport à PS et celles qui ne le sont pas. Une donnée d'entrée est "pertinente" si elle met le programme en position de pouvoir *instantanément* (au même cycle) mettre en défaut une propriété de sûreté. Autrement dit, une entrée e_i est pertinente (Pertinente()), s'il existe une sortie s_j appartenant à l'espace des sorties possibles (V_S), telle que la propriété de sûreté PS peut être violée. Une entrée n'est pas pertinente dans le cas contraire.

$$\mathsf{Pertinente}(e_i, \Sigma, PS) \Longleftrightarrow \exists s_j \in V_S,\ (\sigma(E) = e_i) \wedge (\sigma(S) = s_j) \wedge (\Sigma.\sigma \vdash PS|\mathit{false})$$

Par exemple, soit la propriété $R(e, s) : e \Rightarrow s$, où e est une variable d'entrée booléenne et s une variable de sortie booléenne du programme. L'instanciation ($e = \mathit{faux}$) n'est pas pertinente car R sera vraie, quelle que soit la valeur de s. Par contre, l'instanciation ($e = \mathit{vrai}$) est pertinente car R peut potentiellement être violée si le programme produit ($s = \mathit{faux}$).

Nous insistons sur le fait que la définition de la fonction Pertinente repose sur une évaluation *instantanée* de la propriété considérée. Soit la propriété $R'(e, s)$: pre e and $e \Rightarrow s$, qui signifie intuitivement que si e était vrai à l'instant précédent, et l'est encore à l'instant courant, et, l'implantation doit réagir à l'instant courant en positionnant s à vrai. Lorsque (pre $e = \mathit{vrai}$), alors la caleur vrai pour e est pertinente. On note que si (pre $e = \mathit{faux}$), la définition de Pertinente indique qu'il n'existe pas d'entrée telle que R' peut être violée par la réaction à l'instant courant du système.

Dans la suite, par abus de langage, on dit que l'on *teste une propriété* $\mathcal{P}$ lorsque l'on fournit au programme une donnée pertinente par rapport à $\mathcal{P}$.

Cette méthode de test s'inscrit dans le cadre d'une démarche suggérée dans [79] pour la définition d'un oracle : l'utilisateur peut décrire les propriétés que le logiciel doit satisfaire sous la forme de propriétés Lustre. L'oracle peut alors être construit comme une conjonction de ces propriétés. Cette méthode de test propose de réutiliser les propriétés d'oracle ainsi définies (ou d'autres) pour guider le test.

La première méthode de guidage est relativement simple à mettre en œuvre. Malheureusement, elle ne donne pas toujours complète satisfaction. En effet, elle ne permet pas toujours de *tester* des propriétés utilisées pour la construction de l'oracle. La méthode présentée ici

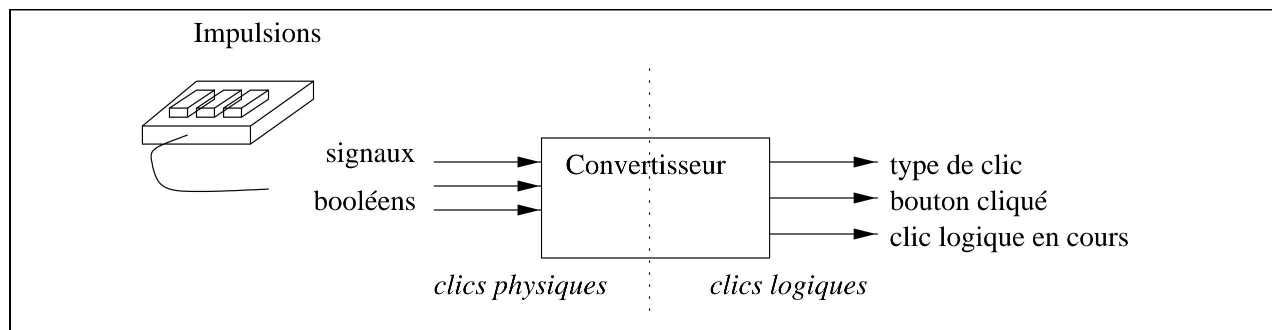


FIG. **3.7** – Nouveau convertisseur de clics physiques en clics logiques

choisit les données d'entrée de façon à favoriser *le test de ces propriétés*, et ne demande a priori pas d'effort de spécification supplémentaire.

A l'origine, cette seconde méthode de guidage a été conçue dans la perspective d'augmenter la sûreté. Les propriétés observées étaient donc naturellement les propriétés de sûreté. Toutefois, la définition de la fonction Pertinente est valable quel que soit le type de propriété, c'est-à-dire qu'elles soient des propriétés de sûreté ou non. C'est pourquoi, nous avons préferé l'expression "*génération guidée par des propriétés*" plutôt que de "*génération guidée par des propriétés de sûreté*".

3.4.2 Algorithme de génération

Pour permettre la déclaration des propriétés utilisées pour le guidage, un nouvel opérateur est ajouté à Lustre. Il s'agit de l'opérateur *safety*. Il s'agit d'un opérateur dont l'argument est une liste de propriétés Lustre utilisées pour le guidage. Comme l'opérateur *environment*, l'opérateur *safety* n'est utilisable que dans un nœud testeur.

Exemple du convertisseur de clics

Considérons un convertisseur de clics légèrement différent de celui étudié jusqu'à présent, c'est-à-dire possédant un petit voyant lumineux pour indiquer si un clic logique est en cours de calcul (voir figure 3.7).

On rappelle que T définit la vitesse du triple clic, en nombre de tops d'horloge. Le comportement général du convertisseur n'est pas modifié. En supposant que $T = 4$, on doit maintenant observer le comportement suivant.

top d'horloge	1	2	3	4	5	6	7	8	9	10	11	12	...
bouton cliqué	-	1	-	1	1	-	1	2	-	2	-	-	...
type de clic logique	-	-	-	-	t	-	-	s	-	-	d	-	...
clic logique en cours	*f*	*v*	*v*	*v*	*v*	*f*	*v*	*v*	*v*	*v*	*v*	*f*	...

L'unique contrainte d'environnement n'est pas modifiée. La propriété que l'on souhaite observer pour ce convertisseur est : *"lorsqu'il n'y a pas de clic logique en cours, un clic physique provoque le début d'un nouveau clic logique"*. Le nouveau nœud testeur est :

```
testnode EnvironnementConvertisseur (k1,k2,k3, r1,r2,r3, cl: boolean)
```

```
return (B1,B2,B3 : boolean)
let
   environment( #(B1,B2,B3) );
   safety( ((not pre cl) and (B1 or B2 or B3)) => cl );
tel
```

L'algorithme de génération consiste à utiliser deux fonctions booléennes, représentées sous forme de BDD. La première ϕ_{env} consiste en la description des contraintes d'environnement. La deuxième fonction ϕ_{per} décrit l'ensemble des instanciations satisfaisant les contraintes d'environnement et pertinentes pour le test des propriétés choisies par l'utilisateur.

L'instanciation d'un vecteur se fait alors en deux étapes. La première consiste à choisir laquelle des deux fonctions ϕ_{env} ou ϕ_{per} sera utilisée pour calculer l'instanciation. La fonction ϕ_{per} est choisie avec la probabilité 0.9 lorsqu'une donnée pertinente existe. On ne choisit pas systématiquement une donnée pertinente afin d'éviter des situations de blocage, consécutifs à une trop grande restriction de l'espace d'entrée. Par exemple, dans un état q, ϕ_{per} pourrait n'autoriser que des vecteurs d'entrée v tels que $t(q, v) = q$.

La deuxième étape consiste à appliquer la méthode de sélection équiprobable (définie page 37) sur la fonction choisie (i.e. le BDD choisi).

L'algorithme général de sélection des données est le suivant :

Lexique
- $M_{env} = (Q, q_{init}, S, E, env, t_{env}, \phi_{env})$: la machine d'entrée-sortie associée au noeud de test sans l'opérateur safety.
- $\phi_{per} : Q \times E \times \{vrai, faux\}$: la fonction décrivant les données pertinentes.
- { *ϕ_{per} est représenté par un BDD et $\forall q \in Q$, $\phi_{per}(q)$ détermine un BDD d'entrée* }
- tirage : un réel $\in [0..1] \rightarrow$ un booléen
- { *tirage(i) produit un booléen aléatoirement selon une distribution sur $[0, 1]$ centrée en i* }
- q : un état de M_{env}
- vs : une instanciation de S (v $\in V_S$)
- ve : une instanciation de E (v $\in V_E$)
- A : un BDD
- tmp : un temporaire de type booléen

Génération2 :
- q = q_{init}
- pour chaque pas de test faire :
  - A = ϕ_{per}(q);
  - si ((A = $faux$) ou (tirage(0.1) = $faux$)) — *pas de données pertinentes ou choix ϕ_{env}*
    - alors A = ϕ_{env}(q); — *sélection du BDD d'entrée*
  - vs = CalculVE'(A,vs);
  - envoyer(vs); — *vers le système sous test*
  - récupérer(ve); — *produit par le système sous test*
  - q = t_{env}(q,vs,ve); — *calcul du nouvel état*

3.4.3 Formalisation

Générateur associé à cette méthode

Comme il a été dit précédemment, un générateur est défini formellement comme un couple : une machine génératrice et un algorithme de sélection des données.

La machine génératrice est définie formellement par la sémantique opérationnelle des nœud testeur A.2.1 à laquelle est ajoutée une règle de compatibilité avec l'opérateur safety A.2.2.

La définition de cette machine repose notamment sur la notion de "pertinence" des données d'entrée. Cette notion est définie formellement de la façon suivante :

Définition 5 *Soit $M_{env} = (Q, q_{init}, S, E, env, t_{env}, \phi_{env})$ une machine génératrice et $P \subseteq Q \times V_E \times V_S$ un prédicat. Le fait qu'une donnée d'entrée (du système sous test) $e \in V_E$ est pertinente (pour le test) de P dans l'état $q \in Q$ est défini par* Pertinente(e, q, P) *:*

$$\mathsf{Pertinente}(e, q, P) \Longleftrightarrow \exists s \in V_S,\ P(q, e, s) = \mathit{false}$$

On définit alors la notion de machine guidée par des propriétés :

Définition 6 *Une machine guidée par des propriétés est un couple $M_p = (M_{env}, P)$ où*

- *$M_{env} = (Q, q_{init}, S, E, env, t_{env}, \phi_{env})$ est une machine génératrice,*
- *P est une conjonction de propriétés.*

Le générateur $\mathcal{G}_p$ associé à cette méthode est donc $\mathcal{G}_p = (M_p, \mathsf{Génération2})$ où M_p est une machine d'entrée-sortie guidée par des propriétés, et Génération2 est l'algorithme défini précédemment.

Chapitre 4

Validation des méthodes de génération

Introduction

Dans le chapitre précédent, nous avons décrit les méthodes de génération de Lutess. Ces méthodes possèdent des caractéristiques statistiques :

- la première méthode produit des données d'entrées (échantillons de valeurs) de façon *aléatoire* et *équiprobable*,

- la seconde méthode produit des données pertinentes pour le *test de propriétés*, générées aussi de façon *équiprobable*.

Notre objectif est de monter que l'implantation des méthodes de test de type statistique de Lutess fournit des données conformes à la définition de ces méthodes; c'est-à-dire que la distribution des variables d'entrées, qui est observée correspond à celle attendue.

Cette étape est importante, voire fondamentale, pour un outil destiné à la validation de systèmes critiques. En effet, la confiance dans le résultat de cette validation exige que l'outil lui-même présente une haute garantie de bon fonctionnement.

Un test aléatoire exige que les données de tests soient effectivement choisies de "façon aléatoire" [53], c'est-à-dire de façon non biaisée. A partir d'un échantillon de données aléatoires, on observe certains comportements, qui constituent alors vraiment une prédiction des comportements possibles du système pour la validation. Cette caractéristique fait que le test aléatoire est très différent des méthodes de test de couverture qui demandent une sélection spécifique des données, et qui par conséquent ne concernent que quelques comportements particuliers.

Les algorithmes de production de nombres "pseudo-aléatoire" ont été particulièment étudiés par Knuth [63]. Il suggère un ensemble de méthodes de type statistique pour analyser les données engendrées par de tels algorithmes.

Dans ce chapitre, nous nous inspirons de ce travail pour mettre en place une approche de validation des caractéristiques statistiques des méthodes de génération. Cette approche

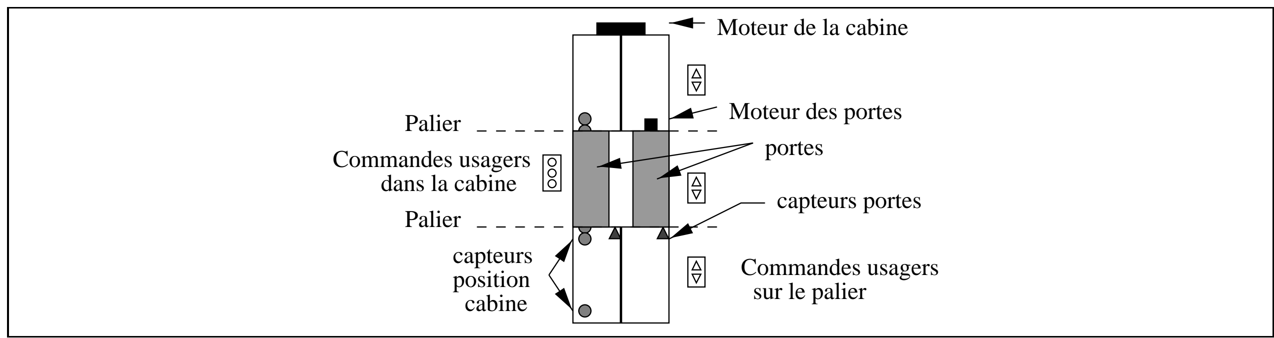


FIG. **4.1** – Un ascenseur

repose entièrement sur des *tests statistiques d'hypothèses* [67], basés sur les *intervalles de confiance* et le *khi-deux* (χ^2). L'approche de validation sera appliquée de nouveau au chapitre 7 pour valider une autre méthode de génération.

Nous illustrons cette approche à l'aide d'un exemple de système critique : un système de contrôle d'ascenseur. Dans ce chapitre, une large place est faite à la description de cet exemple (section 4.1). Au cours des trois sections suivantes, on expose le principe général des tests statistiques, puis on les applique aux données obtenues successivement avec la méthode de génération aléatoire et avec celle guidée par les propriétés. En conclusion, on dressera une liste des faiblesses de ces méthodes.

4.1 Description du système de contrôle de l'ascenseur

4.1.1 Description informelle

Soit un ascenseur composé des éléments suivants (figure 4.1) :

- une cabine,
- des portes,
- des commandes à la disposition des utilisateurs (boutons d'appel dans la cabine et sur les paliers),
- des capteurs (pour suivre le mouvement des portes et de la cabine) ,
- des signaux lumineux,
- deux moteurs (un pour actionner les portes et un autre pour déplacer la cabine).

Le logiciel que nous allons étudier commande les moteurs et les signaux lumineux de cet ascenseur.

Les portes sont ouvertes et fermées par le logiciel avec les commandes *ouvrir_portes* et *fermer_portes*. Deux capteurs *portes_ouvertes* et *portes_fermées* indiquent si les portes sont complètement ouvertes ou fermées. Notons que les portes ne peuvent pas être à la fois ouvertes et fermées, mais elles peuvent être dans une position intermédiaire, c'est-à-dire ni ouvertes, ni fermées.

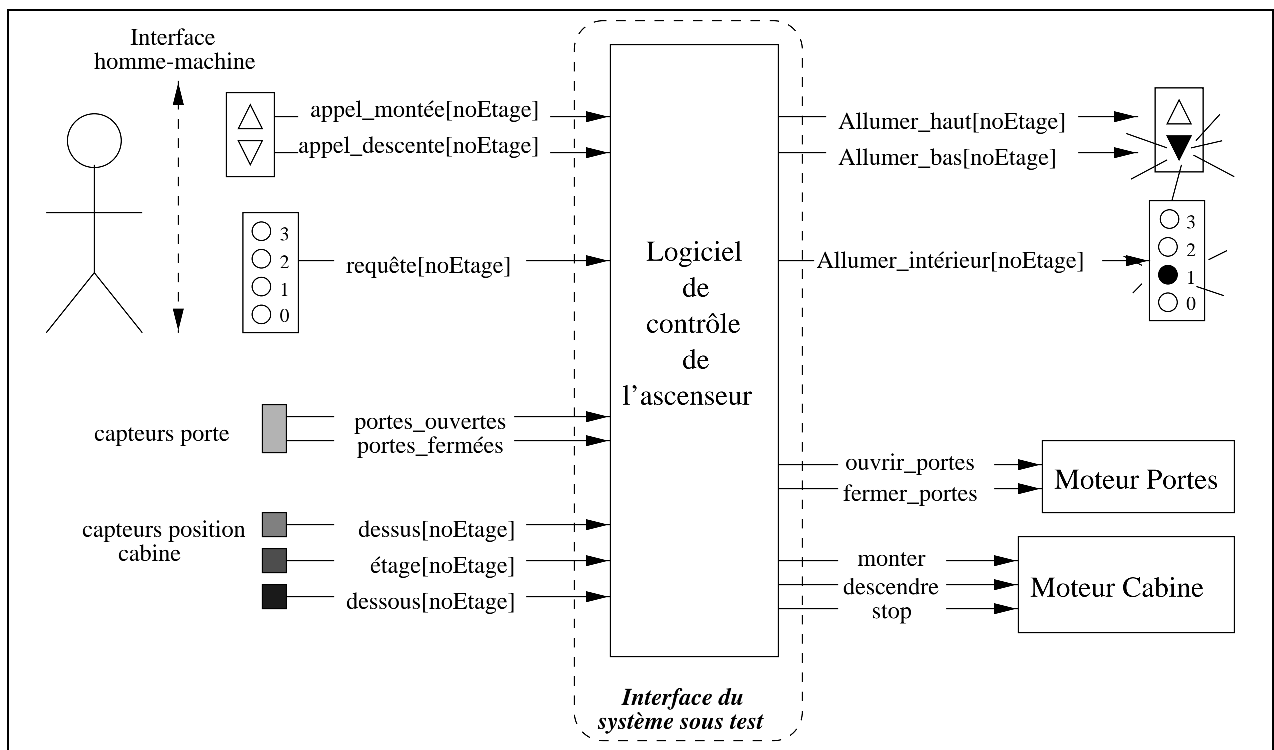


FIG. **4.2** – Interface du logiciel de gestion de l'ascenseur

Sur les paliers, l'usager dispose de deux boutons pour appeler l'ascenseur, un pour monter et un pour descendre (*appel_montée[noEtage]* et *appel_descente[noEtage]*). Le bouton d'appel pour descendre n'existe pas au rez-de-chaussée; celui pour monter n'existe pas au dernier étage.

Les boutons d'appel s'allument et restent allumés entre le moment où un usager appelle l'ascenseur et le moment où il est "servi". Une demande de descente (resp. de montée) est "servie" lorsque l'ascenseur s'arrête au palier correspondant, ouvre puis ferme ses portes et descend (resp. monte) ensuite. Le logiciel gère l'éclairage des différents boutons via les commandes *allumer_haut[noEtage]* et *allumer_bas[noEtage]*.

Dans la cabine, l'usager dispose d'autant de bouton qu'il y a d'étage. Les boutons s'allument et restent allumés (*Allumer_intérieur[noEtage]*) entre le moment où un usager effectue une requête (*requête[noEtage]*), et le moment où il est "servi". Une requête est servie lorsque l'ascenseur s'arrête à l'étage correspondant et ouvre ses portes.

La cabine est déplacée dans la cage grâce à un moteur commandé par le logiciel. Celui-ci dispose de trois commandes *monter*, *descendre*, et *stop*. Un ensemble de capteurs permettent de repérer la position de la cabine dans sa cage: *dessus[noEtage], dessous[noEtage]*, et *étage[noEtage]*. Lorsque l'ascenseur monte, seuls les capteurs *dessous[noEtage]* sont actifs. Lorsqu'il descend, seuls les capteurs *dessus[noEtage]* sont actifs. Le capteur *étage[noEtage]* n'est actif que lorsque l'ascenseur est arrêté à l'étage correspondant.

```
const NBETAGES = 4;

node Ascenseur(
        -- signaux de requêtes émises à l'intérieur de la cabine
        requete : bool^NBETAGES;

        -- signaux indiquant l'état des portes
        portes_ouvertes, portes_fermees : bool;

        -- signaux relatifs à la position de la cabine
        etage, dessus, dessous : bool^NBETAGES;

        -- signaux des appels de l'ascenseur depuis les étages
        appel_montee, appel_descente : bool^NBETAGES;)
returns(
        -- allumage des boutons à l'intérieur de la cabine
        allumer_interieur : bool^NBETAGES;

        -- commandes d'ouverture et de fermeture des portes
        ouvrir_portes, fermer_portes : bool;

        -- commandes du moteur
        monter, descendre, stop : bool;

        -- allumage des boutons d'appel des étages
        allumer_haut, allumer_bas : bool^NBETAGES
);
```

FIG. **4.3** – Interface en Lustre du programme de l'ascenseur

4.1.2 Mise en oeuvre de la validation

Le programme

L'interface du nœud Lustre est donné figure 4.3. Comme on peut le constater sur cette figure, les variables entrées et de sorties présentes à chaque étage sont codées sous forme de tableaux de booléens, de taille *NBETAGES*. Cette constante donne à ce programme un aspect générique.

La description de l'environnement

L'ascenseur est un objet "réel" sur lequel s'exercent des contraintes physiques. Par exemple, il est impossible que l'ascenseur soit à deux endroits en même temps. Cette réalité physique a des conséquences implicites sur l'environnement. Ainsi, si les capteurs de positions de la cabine fonctionnent correctement, il y a exactement un capteur de position signalant

la cabine à chaque instant. La liste retenue des contraintes de cet environnement est :

1. les portes ne peuvent pas être à la fois ouvertes et fermées;
   not (*portes_ouvertes* and *portes_fermées*)

2. le signal *appel_descente* (resp. *appel_montée*) est inactif au au rez-de-chaussée (resp. au dernier étage);
   not *appel_descente[0]* and not *appel_montée[NBETAGES-1]*

3. il n'y a pas de capteur *dessous* au rez-de-chaussée, ni de capteur *dessus* au dernier étage;
   not *dessous[0]* and not *dessus[NBETAGES-1]*

4. la cabine est à un seul endroit à la fois;
   LIN_OR*(NBETAGES-1,dessus[0..NBETAGES-2])* or LIN_OR*(NBETAGES-1], dessous[1..NBETAGES-1])* or LIN_OR*(NBETAGES,étage[0..NBETAGES-1])*
   #(dessus[0], dessus[1], dessus[2], dessous[1],..., étage[NBETAGES-2], étage[NBETAGES-1])

5. les portes ne s'ouvrent (resp. ferment) pas sans recevoir l'ordre adéquat;
   always_from_to(*portes_fermées*, *portes_fermées*, pre *ouvrir_portes*) and
   always_from_to(*portes_ouvertes*, *portes_ouvertes*, pre *fermer_portes*)

6. la cabine ne quitte pas spontanément un étage;
   pour $i \in [0..NBETAGE \Leftrightarrow 1]$, always_from_to*(étage[i], étage[i],* pre *(monter* or *descendre))*

7. la cabine monte (descend) seulement après réception d'un signal *monter* (*descendre*);

   pour $i \in [0..NBETAGE \Leftrightarrow 2]$, once_from_to(pre *monter,* pre *étage[i], dessous[i+1])*
   pour $i \in [1..NBETAGE \Leftrightarrow 1]$, once_from_to(pre *descendre,* pre *étage[i], dessus[i-1])*

8. la cabine ne s'arrête pas spontanément : elle ne s'arrête que suite à une commande *stop*;
   pour $i \in [0..NBETAGE \Leftrightarrow 1]$ always_from_to (not *étage,* not *étage,* pre *stop)*

9. après émission de la commande *stop*, la cabine s'arrête au premier étage suivant sa position courante;
   pour $i \in [1..NBETAGE \Leftrightarrow 2]$, once_from_to*(* pre *étage[i],* pre *(stop* and *dessous[i]), dessous[i+1])*
   pour $i \in [1..NBETAGE \Leftrightarrow 2]$, once_from_to*(* pre *étage[i],* pre *(stop* and *dessus[i]), dessus[i-1])*

10. le mouvement de la cabine est continu : si la cabine quitte une position, c'est pour passer à la position immédiatement au dessus ou au dessous.
    pour $i \in [1..NBETAGE \Leftrightarrow 2]$, always_from_to*(étage[i], étage[i], dessus[i-1]* or *dessous[i+1])*
    always_from_to*(étage[0], étage[0], dessous[1])*
    always_from_to*(étage[NBETAGE-1], étage[NBETAGE-1], dessus[NBETAGE-2])*

Propriétés attendues de l'ascenseur

Intuitivement, les propriétés attendues du logiciel sont la sécurité des usagers et le bon fonctionnement général de l'ascenseur. On peut éventuellement attendre d'un ascenseur qu'il soit performant; ce point toutefois n'est pas examiné dans ce chapitre. On s'intéresse ici

tout particulièrement aux aspects relatifs aux comportements sûrs de l'ascenseur et au bon fonctionnement de l'ascenseur.

- *Les comportements sûrs*
  Le logiciel sûr ne doit pas déplacer la cabine lorsque les portes sont ouvertes, il ne doit pas arrêter la cabine entre les étages, il ne doit ouvrir les portes que lorsque l'ascenseur est arrêté à un étage.

- *Le bon fonctionnement de l'ascenseur*
  On attend d'un ascenseur qu'il réagisse aux requêtes des usagers et que les boutons s'allument et s'éteignent au bon moment. Lorsqu'un usager est dans un ascenseur et qu'il a demandé à se rendre à un étage, il s'attend à ce que l'ascenseur s'arrête à cet étage et ouvre ses portes. De même, lorsqu'un usager est sur un palier et demande l'ascenseur pour descendre, il s'attend à ce que l'ascenseur s'arrête, ouvre, puis ferme ses portes et descende ensuite. Un usager doit pouvoir sortir d'un ascenseur arrêté à un étage (en appuyant sur le bouton de l'étage correspondant). De même, un usager doit pouvoir entrer dans un ascenseur arrêté à un étage (en appuyant sur l'un des boutons d'appel).

4.2 Résumé des principes des tests d'hypothèses

Un test d'hypothèse consiste à définir une règle de décision concernant la validité d'une hypothèse portant sur une loi de distribution dans une population dont on observe un (ou plusieurs) échantillon(s) aléatoire(s) [63, 67]. La procédure générale d'un test comprend les éléments suivants :

- caractériser une propriété X attendue d'une population et le type de loi de X pour cette population;
- définir une hypothèse $\mathcal{H}_0$ dites *hypothèse nulle* et son alternative $\mathcal{H}_1$ relativement à la population considérée;
- déterminer une variable de décision,
- choisir une valeur α pour le risque de refuser $\mathcal{H}_0$ alors qu'elle serait vraie;
- définir les conditions expérimentales de constitution d'un échantillon,
- énoncer la règle de décision (rejet ou acceptation de l'hypothèse $\mathcal{H}_0$) selon le résultat obtenu;
- procéder au test : produire l'échantillon, calculer la valeur de la variable de décision, et conclure (rejet ou acceptation).

Accepter $\mathcal{H}_0$ ne signifie pas que cette hypothèse est *vraie*, mais seulement que les observations disponibles ne sont pas incompatibles avec cette hypothèse et que l'on a pas de raison suffisante de lui préférer l'hypothèse $\mathcal{H}_1$, compte tenu des résutats expérimentaux obtenus sur l'échantillon.

La probabilité de rejeter $\mathcal{H}_0$ alors qu'elle est vraie s'appelle le *risque de première espèce*, qui par construction du test, a la valeur α. Le *niveau de confiance* envers un test est $1 \Leftrightarrow \alpha$.

Les résultats des tests pratiqués sont d'autant plus significatifs que les échantillons sont de grandes tailles. Lorsque les résultats d'un test sont proches de α mais qu'ils ne permettent

top	requête	appel montée	appel descente	position cabine	état des portes
1	R[1]	M[2]	D[1] D[2]	*étage[1]*	
2	R[0], R[2], R[3]	M[2]		*étage[1]*	Ouvertes
3	R[1], R[3]	M[1] M[2]	D[1] D[3]	*étage[1]*	Fermées
4	R[0], R[2], R[3]		D[2	*étage[1]*	Ouvertes
5	R[0]	M[2]	D[1] D[2] D[3]	*étage[1]*	Fermées
6	R[0], R[1], R[2], R[3]	M[0] M[1]	D[1] D[2]	*étage[1]*	Ouvertes
7	R[1], R[2], R[3]	M[1] M[2]	D[1] D[2] D[3]	*étage[1]*	Fermées
8	R[0], R[1], R[2]	M[1] M[2]	D[1] D[3]	*étage[1]*	Ouvertes

A chaque top, si R[*i*] alors le signal *requête[i]* est vrai, si M[*i*] alors *appel_montée[i]* est vrai, si D[*i*] alors le signal *appel_descente[i]* est vrai.

TAB. 4.1 – *Extrait de la séquence de test produite avec le germe 11111*

pas d'accepter une hypothèse, on peut augmenter la taille de l'échantillon et calculer la nouvelle valeur de la variable de décision. Si l'on constate une évolution significative vers la satisfaction de $\mathcal{H}_0$, alors on peut accepter l'hypothèse, sinon on la rejette.

Dans la suite, les propriétés que l'on cherche à observer sont des propriétés relatives à la distribution ou l'indépendance de variables. Pour chacun des tests, on explicite l'hypothèse $\mathcal{H}_0$ (la population satisfait la propriété). $\mathcal{H}_1$ est implicitement la négation de $\mathcal{H}_0$.

Pour l'ensemble des tests, on choisit un niveau de confiance égal à 99%. Cela signifie que, en moyenne, si l'on analyse 100 échantillons alors 1 d'entre eux peut conduire à rejeter $\mathcal{H}_0$ alors qu'elle est vraie.

Les variables d'entrée d'un programme booléen (i.e. les variables de sorties du générateur) sont assimilées à des variables aléatoires. Les conditions expérimentales de constitution d'échantillons consistent à utiliser Lutess pour générer cinq séquences de tests (traces) pour le programme de l'ascenseur. L'effectif (la taille) des échantillons est généralement de 500 éléments. La table 4.1 est un extrait d'une séquence de test produite avec le germe 11111.

La table 4.2 représente le nombre d'instances où les variables d'entrées *requête[0..3]*, *appel_montée[0..2]* et *appel_descente[1..3]* ont été affectées à vrai.

4.3 Analyse des données produites par la méthode de génération aléatoire

Nous procédons en deux temps. Tout d'abord, nous étudions les propriétés relatives à une variable d'entrée, indépendamment des autres. Nous nous assurons que sa distribution empirique (observée sur les traces) correspond à la distribution théorique spécifiée. Nous validons cette hypothèse à l'aide de tests statistiques basés sur les intervalles de confiance (test sur la valeur moyenne) ou basés sur les statistiques du χ^2 (test de la distribution) [63, 67].

Séquence (germe) $\rightarrow$ $\downarrow$ Nb d'occurences de ...	S1 (11111)	S2 (47347)	S3 (92375)	S4 (48603)	S5 (10256)
requête[0] (X_0)	258	247	263	262	272
requête[1] (X_1)	250	251	269	264	250
requête[2] (X_2)	247	251	236	256	226
requête[3] (X_3)	244	250	260	266	236
appel_montée[0] (X_4)	264	262	244	255	269
appel_montée[1] (X_5)	280	267	261	248	253
appel_montée[2] (X_6)	257	258	259	270	255
appel_descente[1] (X_7)	268	245	242	263	259
appel_descente[2] (X_8)	247	258	269	259	250
appel_descente[3] (X_9)	237	236	261	259	247

TAB. 4.2 – *Effectifs obtenus pour 5 séquences de 500 pas*

Dans un deuxième temps, nous étudions les corrélations entre les différentes variables générées. On utilise alors un test d'indépendance [63, 67].

4.3.1 Test de l'hypothèse d'uniformité de la distribution d'une variable par intervalles de confiance

Intervalle de confiance : résumé

Prenons l'exemple de la variable booléenne *requête[0]*, pendant une séquence de 500 pas. Si l'on observe que cette variable a pris la valeur *vrai* 258 fois et la valeur *faux* 242 fois, peut-on en déduire la probabilité réelle (p) avec laquelle *requête[0]* est affectée à vrai ?

On peut évidemment calculer la fréquence d'apparition de l'événement sur l'expérience ($\hat{p} = \frac{258}{500}$). Mais, dans la plupart des cas, il y a une probabilité nulle que la valeur $\hat{p}$ soit précisément égale à p. En effet, si l'expérience consiste en 500 tirages, il y a 2^{500} séquences de résultats possibles. Toutes ces séquences ont la même probabilité, puisque chaque tirage est supposé être indépendant des précédents. Une séquence particulière contenant autant de valeur *requête[0] = vrai* que de *requête[0] = faux* est aussi probable que n'importe qu'elle autre séquence.

Il est donc souhaitable de donner une majoration de l'erreur commise ($|\hat{p} \Leftrightarrow p|$). Pour cela, au lieu de donner une estimation ponctuelle de $\hat{p}$, on donne une estimation par intervalle: $p = \hat{p} \pm \varepsilon$.

Il faut déterminer ε tel que :

$$\mathbb{P}(p \in [\hat{p} \Leftrightarrow \varepsilon, \hat{p} + \varepsilon]) \geq (1 \Leftrightarrow \alpha).$$

où $1 \Leftrightarrow \alpha$ est le niveau de confiance. Dans le cas où une variable aléatoire ne peut prendre

que deux valeurs (population binomiale) [67], l'intervalle de confiance p est donné par :

$$\hat{p} \pm z_c \sigma_{\hat{p}} \text{ avec } \sigma_{\hat{p}} = \sqrt{\frac{\hat{p}(1 \Leftrightarrow \hat{p})}{n}}$$

où n est la taille de l'échantillon et z_c est le *coefficient de confiance*. La valeur de z_c se détermine à partir du niveau de confiance choisi et de la table de la loi normale réduite [67]. Lorsque l'on fixe le niveau de confiance à 99% ($(1 \Leftrightarrow \alpha) = 0.99$), on obtient $z_c = 2.58$.

Analyse des données produites par Lutess

Soient $X_0..X_9$ les variables aléatoires booléennes représentant les entrées du programme *requête[0]..appel_montée[3]*. On étudie la valeur de $\hat{p}$ uniquement pour les valeurs extrêmes, c'est-à-dire *appel_montée[1]* (X_5) dans la séquence S1 et *requête[2]* (X_2) dans S5. Par ailleurs, nous avons augmenté la taille des échantillons S1 et S5 en réexécutant Lutess sur 5 000 pas de test :

- la population : X_i une variable booléenne,
- l'échantillon : 500 ou 5 000 valeurs tirées,
- 2 classes : $c_1 = vrai$ et $c_2 = faux$,
- hypothèse de test : $\mathcal{H}_0$: X_i a une probabilité de 0.5,
- niveau de confiance : 99%.

var. aléatoire	X_5, S1	X_2, S5	X_5, S1'	X_2, S5'
taille de l'échantillon	500		5 000	
Observation	280	226	2510	2486
$\hat{p}$	0.56	0.45	0.502	0.497
ε	0.06	0.06	0.007	0.007

Ces résultats nous permettent accepter l'hypothèse $\mathcal{H}_0$ avec un niveau de confiance de 99%.

4.3.2 Test de l'hypothèse d'uniformité de la distribution d'une variable par la méthode du χ^2

Méthode du χ^2 : résumé

On s'intéresse ici à l'étude de la distribution d'un échantillon par rapport à des ensembles de valeurs dits *classes*. La méthode du khi-deux (χ^2) permet de déterminer si distribution observée de l'échantillon sur ces classes correspond à celle attendue.

Soit N le nombre de classes dans lesquelles sont réparties les valeurs de l'échantillon. L'hypothèse d'une certaine distribution de l'échantillon sur les classes définies nous permet de calculer l'effectif attendu x_i pour chaque classe i. Soit X_i, l'effectif observé pour chaque classe. Pour chacune des classes, on peut calculer l'*écart quadratique réduit*. Par définition, c'est la quantité :

Niveau de confiance	50%	90%	95%	97.5%	99%	99.5%	99.9%
α	0.5	0.1	0.05	0.025	0.01	0.005	0.001
$\chi^2_{\nu=1}$	0.455	2.706	3.84	5.02	6.63	7.88	10.83
$\chi^2_{\nu=3}$	2.37	6.25	7.81	9.35	11.34	12.84	16.27

TAB. 4.3 – *Extrait de la table du* χ^2

$$\frac{(X_i \Leftrightarrow x_i)^2}{x_i}$$

Khi-deux observé (χ^2_{obs}) ou *l'indicateur d'écart* est par définition, la somme des écarts quadratiques réduits.

$$\chi^2_{\text{obs}} = \sum_{i=1}^{N} \frac{(X_i \Leftrightarrow x_i)^2}{x_i}$$

Pour la variable *requête[0]*, il y a deux classes c_1 (*requête[0]=vrai*) et c_2 (*requête[0] = faux*). Pour chacune des classes, l'effectif attendu est $x_i = 250$, (hypothèse: la loi de distribution est uniforme). Si les effectifs observés sont $X_1 = 296$ et $X_2 = 204$, on obtient une valeur de $\chi^2_{\text{obs}} > 16.9$.

Une fois l'indicateur d'écart calculé, on utilise une table pour évaluer la validité de l'hypothèse. Pour un *degré de liberté* fixé ($\nu = N \Leftrightarrow 1$), la table donne la valeur maximum c de χ^2 en fonction du niveau de confiance. Si la valeur observée du χ^2 est supérieure à c, on rejette l'hypothèse [63, 67].

La probabilité de faire l'erreur "décider que le générateur est mauvais alors qu'il est bon" est alors égale à la probabilité qu'une variable aléatoire de loi du χ^2 à ν degrés de liberté prenne une valeur supérieure à c. Un extrait de cette table pour un χ^2 à 1 et à 3 degrés de liberté est donné table 4.3.

Pour *requête[0]*, le degré de liberté (ν) est égal à 1. La table du khi-deux indique que la probabilité d'obtenir une séquence avec un khi-deux supérieur 16.9 est de moins de 0.001.

$$\mathbb{P}(\chi^2_{\nu=1} \geq 16.9) \leq 0.0000390$$

On constate que cette expérience ne permet pas d'accepter l'hypothèse "*requête[0]* est produit avec la probabilité 0.5" avec un niveau de confiance supérieur à 99.9%.

Notons que pour les cas où les valeurs de l'échantillon sont réparties en deux classes (comme c'est le cas dans cet exemple), la technique du χ^2 est équivalente à l'estimation de la probabilité d'appartenance à une de ces classes.

Analyse des données produites par Lutess

Soient $X_0..X_9$ les variables aléatoires booléennes représentant les entrées du programme *requête[0]..appel_montée[3]*.

– la population: X_i une variable booléenne,

- l'échantillon : 500 ou 5 000 valeurs tirées,
- 2 classes : $c_1 = vrai$ et $c_2 = faux$,
- nombre de degrés de liberté : v =2-1=1,
- hypothèse $\mathcal{H}_0$: X_i a une probabilité de 0.5,
- niveau de confiance : 99%.

Comme précédemment, on étudie uniquement les valeurs extrêmes, c'est-à-dire *appel_montée[1]* (X_5) pour la séquence S1 et *requête[2]* (X_2) pour S5.

var. aléatoire	X_5, S1	X_2, S5	X_5, S1'
taille de l'échantillon	500		5 000
Observation	280	226	2510
Attendu	250	250	2500
χ^2	7.20	4.61	0.08

Ce tableau indique que l'hypothèse $\mathcal{H}_0$ serait rejetée si l'on ne considère que le test S1. Nous avons augmenté la taille de cet échantillon. Soit S1' la séquences obtenues pour 5 000 pas de tests. S1' et S5 nous permettent d'accepter $\mathcal{H}_0$ avec un niveau de confiance supérieur à 99%.

4.3.3 Test d'indépendance des valeurs affectées pour une chaque variable

On cherche à montrer que les données engendrées successivement par Lutess pour chaque variable d'entrée du système sous test sont indépendantes les unes des autres, lorsque la variable n'est pas contrainte par des propriétés de l'environnement.

Démarche

On procède en analysant la suite des paires consécutives de valeurs. Dans le cas d'une variable booléenne, il existe quatre paires ({(0,0), (0,1), (1,0), (1,1)}) et chacune à la probabilité $\frac{1}{4}$ d'apparaître si les tirages successifs sont indépendants les uns des autres. Il s'agit ici d'un test de 2-uniformité. Ce test se généralise pour tester la k-uniformité [67]. Dans l'absolu, pour montrer l'indépendance, il faudrait effectuer tous les tests de k-uniformité. Ce n'est évidement pas possible. Dans cette partie, nous nous contenterons de montrer la 2-uniformité, à l'aide d'un test du χ^2.

Analyse des données produites par Lutess

Soient $X_0..X_9$ les variables aléatoires booléennes représentant les entrées du programme *requête[0]..appel_montée[3]*.

- la population : couple de valeurs consécutives d'une variable booléenne X_i,
- l'échantillon : 250 couples de valeurs tirés,

	S1	S2	S3	S4	S5	S3'
c_0	60	58	74	52	72	634
c_1	65	60	49	70	64	588
c_2	68	73	67	70	66	658
c_3	57	59	60	58	48	620

Résultats pour X_2

	S1	S2	S3	S4	S5	S1'
c_0	52	48	62	64	60	606
c_1	58	76	50	63	55	661
c_2	58	61	65	61	72	617
c_3	82	65	73	62	63	616

Résultats pour X_5

TAB. 4.4 – *Effectifs observés pour X_2 et X_5*

	S1	S2	S3	S4	S5
taille de l'échantillon	250				
χ^2 pour X_2	1.20	2.42	5.52	3.93	5.10
χ^2 pour X_5	8.58	6.48	4.42	0.10	2.48

TAB. 4.5 – *χ^2 pour les paires de valeur des variables X_2 et X_5*

- 4 classes : $c_0 = (0,0)$, $c_1 = (0,1)$, $c_2 = (1,0)$ et $c_3 = (1,1)$,
- nombre de degrés de liberté : v =4-1=3,
- hypothèse $\mathcal{H}_0$: les valeurs de X_i sont 2-uniformes
- niveau de confiance : 99%.

On choisit d'illustrer le test de $\mathcal{H}_0$ avec les variables X_2 et X_5. La table 4.4 représente la répartition des échantillons pour ces variables pour les séquences S1 à S5. La table 4.5 donne la valeur de χ^2 pour ces cas là. Cette dernière table nous permet de conclure que l'hypothèse $\mathcal{H}_0$ est vérifiée, c'est-à-dire que les valeurs successives d'une variable d'entrée sont produites de façon indépendantes.

4.3.4 Test d'indépendance des variables entre elles

A chaque pas de test, Lutess produit un ensemble de données. Ces données sont produites les unes après les autres. Nous savons qu'elles respectent les contraintes d'environnement explicites (données par l'utilisateur). Mais, on souhaite s'assurer que des variables indépendantes (c'est-à-dire non liées par des contraintes d'environnement), le sont effectivement.

On procède comme dans le paragraphe précédent : on fait un test du χ^2, mais sur des paires de variables. Les variables étudiées sont *requêtes[0..3]*, *appel_montée[0..2]* et *appel_descente[1..3]*. Ces variables sont indépendantes comme on peut le constater sur la des-

Taille de l'échantillon : 500									
(X_0, X_i)	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9
c_0	124	122	124	119	105	103	109	123	126
c_1	118	120	118	123	137	139	133	119	137
c_2	126	131	131	117	115	139	122	129	116
c_3	132	127	127	141	143	119	136	129	121
χ^2	0.800	0.592	0.720	2.880	7.744	7.296	3.600	0.576	1.936

TAB. 4.6 – *Effectifs et χ^2 pour la séquence S1*

cription de l'environnement donnée section 4.1.2, page 48.

- la population : (X_i, X_j) un couple de variables booléennes,
- taille de l'échantillon : 500 couples,
- 4 classes : c_0 = (0,0), c_1 =(1,0), c_2 = (0,1) et c_3 = (1,1),
- nombre de degrés de liberté : v =4-1=3,
- hypothèse $\mathcal{H}_0$: X_i et X_j sont indépendantes
- niveau de confiance : 99%.

On choisit d'illustrer le test de $\mathcal{H}_0$ avec les 9 paires $(X_0, X_{i,1 \leq i \leq 9})$ pour la séquence S1 avec 500 échantillons. La table 4.6 nous indique que l'on peut accepter hypothèse $\mathcal{H}_0$. Ainsi, on peut considérer que l'affectation des valeurs de variables indépendantes se fait de façon indépendante.

4.4 Analyse des données produites par la méthode guidée par des propriétés

Une des propriétés que l'on souhaite assurer est que l'ascenseur ne reste pas bloqué. Il existe plusieurs phases pendant lesquels un ascenseur peut se bloquer. On étudie en particulier le cas où l'ascenseur est à un étage, portes ouvertes, et qu'il n'y a aucune demande pour cet étage. Le comportement attendu de l'ascenseur est qu'il ferme ses portes. Cette propriété s'écrit

P(i) = pre étage[i] and portes_ouvertes and not (appel_descente[i] or requête[i] or appel_montée[i])
    => fermer_porte
P = $\vee_{i=0}^{3}$ P(i)

On utilise deux oracles consistant respectivement en la conjonction des propriétés O(i) et Q(i) suivantes :

Q(i) = pre étage[i] and portes_ouvertes
Q = $\vee_{i=0}^{3}$ Q(i)
O(i) = Q(i) and not (appel_descente[i] or requête[i] or appel_montée[i])

Séquence (germe) → ↓ Nb d'occurences de ...	S1 (11111)	S2 (47347)	S3 (92375)	S4 (48603)	S5 (10256)
O(i)	229	261	277	229	251
Q(i)	1593	1638	1597	1548	1573
Méthode 1					
O(i)	1206	1236	1219	1242	1233
Q(i)	1315	1331	1321	1346	1334
Méthode 2					

TAB. 4.7 – *Résultats pour les 5 séquences produites*

$\mathsf{O} = \bigvee_{i=0}^{3} \mathsf{O(i)}$

La propriété Q est vraie lorsqu'il existe une donnée d'entrée pertinente pour le test de la propriété P. Le propriété O est vraie lorsqu'une donnée d'entrée pertinente a été choisie pour le test de P.

Pour comparer les méthodes de génération, on produit deux nœuds testeurs. Le premier ne décrit que les contraintes d'environnement. Le second décrit ces contraintes et les propriétés de guidage (une par étage).

Ces nœuds testeurs ont été utilisés pour générer des données. Pour chacun on a produit 5 séquences de 5 000 pas. Les résultats sont donnés table 4.7. En première approximation, on peut constater que la seconde méthode permet d'augmenter considérablement le nombre de fois où une donnée pertinente est choisie.

4.4.1 Estimation de la probabilité du choix d'une donnée pertinente

Dans le chapitre précédent, nous avons indiqué que l'instanciation d'un vecteur se faisait en deux étapes. La première consiste à choisir entre ϕ_{env} ou ϕ_{per} pour calculer l'instanciation. La deuxième étape consiste à appliquer la méthode de sélection équiprobable sur la fonction choisie. Nous avons indiqué que la fonction ϕ_{per} est choisie avec la probabilité 0.9 lorsqu'une donnée pertinente existe. Dans ce paragraphe, on cherche à mesurer avec quelle probabilité réelle une donnée pertinente est choisie.

Parmi les données générées, on choisit d'étudier les cas de test où Q est vraie. Ces cas représentent les états de l'environnement pour lesquels il existe une donnée pertinente pour le test de P.

Soit Tr(j) la sous-trace de Sj ($1 \leq j \leq 5$, table 4.7) où ne figure que les cas de test où Q est vraie. Les Tr(j) représentent les expériences que nous allons étudier.

Pour chacun des Tr(j), on observe si O est vraie. O est notre variable aléatoire. En utilisant la méthode des intervalles de confiance (cf. §4.3.1), on estime la probabilité $\hat{p}$ pour laquelle une donnée pertinente est choisie (i.e. O est vraie).

Soient $X_1..X_5$ les variables aléatoires booléennes représentant O pour les 5 échantillons Tr(j).

- la population : X_k une variable booléenne,
- taille de l'échantillon : selon k, entre 1315 et 1346.
- 2 classes : $c_1 = vrai$ et $c_2 = faux$,
- niveau de confiance : 99%.

On étudie la valeur de $\hat{p}$ pour toutes les séquences.

var. aléatoire	X_1	X_2	X_3	X_4	X_5
taille de l'échantillon	1315	1331	1321	1346	1334
Observation	1206	1236	1219	1242	1233
$\hat{p}$	0.92	0.93	0.92	0.92	0.92
ε	0.02	0.02	0.02	0.02	0.02

A première vue, ces résultats obtenus ci-dessus pourraient surprendre. En effet, on s'attend à une probabilité de $\hat{p}$=0.9, or on obtient une valeur de $\hat{p}$ supérieure. Ceci s'explique par le fait que choisir ϕ_{env} (en moyenne, une fois sur 10) n'empêche pas de produire une donnée pertinente.

4.4.2 Equidistribution d'une variable aléatoire

Quelle que soit la fonction ϕ_{env} ou ϕ_{per} choisie, la méthode de sélection des données s'appuie sur une génération équiprobable. Le but de ce paragraphe est de s'assurer que les données sont effectivement uniformément distribuée.

Nous avons produit 5 séquences de données (S1, S2, ..., S5) de 50 000 pas. La table 4.8 présente les valeurs obtenues. En particulier, ce tableau comptabilise le nombre de fois où les propriétés Q(i) sont vraies et la répartition des entrées *requête[i]* par rapport aux Q(i).

Dans cette section, on souhaite vérifier deux points. Tout d'abord, lorsqu'une variable n'est pas contrainte et que sa valeur n'est pas pertinente, elle est instanciée à vrai de façon équiprobable (i.e. avec la probabilité 0.5). C'est le cas par exemple, des variables (*requête[1..3]*) lorsque Q(0) est vraie.

Dans un deuxième temps, on souhaite montrer que la probabilité qu'une variable non contrainte soit instanciée à vraie est modifiée lorsque sa valeur est pertinente.

Distribution d'une variable jugée non pertinente

Soient $X_1..X_3$ les variables aléatoires booléennes représentant *requête[i]* sachant que Q(0) est vraie. Les hypothèses du test sont :

- la population : X_i une variable booléenne,
- taille de l'échantillon : nombre d'instance où Q(0) est vraie,
- 2 classes : $c_1 = X_i = vrai$ et $c_2 = X_i = faux$,

Séquence (germe) → ↓ Nb d'occurences de ...	S1 (11111)	S2 (47347)	S3 (92375)	S4 (48603)	S5 (10256)
Q(0)	2080	2220	2231	2403	2266
Q(1)	4564	4564	4337	4226	4355
Q(2)	4315	4242	4434	4507	4489
Q(3)	2304	2399	2167	2123	2144
Q(0) and *requête[1]*	1021	1095	1167	1186	1158
Q(0) and *requête[2]*	1051	1117	1151	1169	1151
Q(0) and *requête[3]*	1038	1105	1119	1175	1124
Q(1) and *requête[0]*	2302	2288	2182	2198	2236
Q(1) and *requête[2]*	2300	2297	2206	2122	2182
Q(1) and *requête[3]*	2280	2267	2267	2057	2184
Q(2) and *requête[0]*	2209	2157	2220	2330	2211
Q(2) and *requête[1]*	2188	2140	2253	2280	2275
Q(2) and *requête[3]*	2172	2107	2216	2263	2283
Q(3) and *requête[0]*	1181	1201	1073	1064	1091
Q(3) and *requête[1]*	1181	1215	1072	1085	1096
Q(3) and *requête[2]*	1148	1205	1079	1080	1081
Q(0) and *requête[0]*	724	782	776	859	771
Q(1) and *requête[1]*	1562	1515	1508	1478	1557
Q(2) and *requête[2]*	1510	1556	1609	1568	1569
Q(3) and *requête[3]*	816	830	788	771	751

TAB. 4.8 – *Effectifs des séquences S1 à S5 de 50 000 pas*

- nombre de degré de liberté : v =2-1=1,
- hypothèse $\mathcal{H}_0$: X_i a une probabilité de 0.5,
- niveau de confiance : 99%.

La table 4.9 donne les résultats du calcul du χ^2 pour $X_1..X_3$. De cette table, on peut accepter l'hypothèse $\mathcal{H}_0$. Ainsi, lorsque qu'une variable n'est pas contrainte et que sa valeur n'est pas pertinente, elle est instanciée à vrai de façon équiprobable (i.e. avec la probabilité 0.5).

Séquence	S1	S2	S3	S4	S5
Echantillon	2080	2220	2231	2403	2266
X_i attendu	1040	1110	1115	1201	1133
χ^2 observé pour X_1	0.694	0.405	4.850	0.374	1.103
χ^2 observé pour X_2	0.232	0.088	2.324	1.705	0.571
χ^2 observé pour X_3	0.007	0.045	0.028	1.125	0.142

TAB. 4.9 – χ^2 *observé pour requête[1..3] sachant que* Q(0) *est vraie*

Séquence	S1	S2	S3	S4	S5
X_0	0.35 ± 0.03	0.35 ± 0.03	0.35 ± 0.03	0.36 ± 0.03	0.34 ± 0.03
X_1	0.34 ± 0.02	0.33 ± 0.02	0.35 ± 0.02	0.34 ± 0.02	0.36 ± 0.02
X_2	0.35 ± 0.02	0.37 ± 0.02	0.36 ± 0.02	0.35 ± 0.02	0.35 ± 0.02
X_3	0.35 ± 0.03	0.37 ± 0.03	0.36 ± 0.03	0.36 ± 0.03	0.35 ± 0.03

TAB. 4.10 – *Estimation de la probabilité que "requête[i] soit vraie sachant que* Q(i) *est vraie"*

Distribution d'une variable jugée pertinente

La valeur *requête[i]* = *faux* est pertinente lorsque Q(i) est vraie. On s'attend à ce que la probabilité qu'une variable "*requête[i]* soit vraie sachant Q(i) vraie" soit strictement inférieure à 0.5. Pour chaque variable, on estime cette probabilité avec la méthode des intervalles de confiance (cf. figure 4.10).

Soient $X_0..X_3$ les variables aléatoires booléennes représentant *requête[0..1]* sachant que Q(0)...Q(3) sont vraies respectivement. Les hypothèses du test sont :

- la population : X_i une variable booléenne,
- taille de l'échantillon : nombre d'instance où Q(i) est vraie,
- 2 classes : $c_1 = X_i =$ *vrai* et $c_2 = X_i =$ *faux*,
- nombre de degré de liberté : v =2-1=1,
- hypothèse $\mathcal{H}_0$: "X_i sachant Q(i) vraie" a une probabilité inférieure à 0.5,
- niveau de confiance : 99%.

Les résultats obtenus table 4.10 nous permettent d'accepter l'hypothèse $\mathcal{H}_0$. Ainsi, la distribution d'une variable jugée pertinente est modifiée par la méthode de génération basée sur les propriétés, de telle sorte que les valeurs pertinentes soient plus probables.

4.5 Bilan et conclusion

Les deux sections précédentes nous ont permis de constater que les deux méthodes de générations de Lutess sont conformes à leurs spécifications. Nous avons montré que, sur l'exemple de l'ascenseur, les échantillons générés vérifient empiriquement les propriétés d'uniformité et d'indépendance, avec les restrictions liées aux tests statistiques utilisés [63].

Dans la pratique des tests statistiques, les valeurs les plus courantes pour le niveau de confiance sont 90%, 95% et 99%. Choisir un niveau de confiance élevé aboutit à n'abandonner $\mathcal{H}_0$ que dans des cas rarissimes et à accepter $\mathcal{H}_0$ alors qu'elle est fausse (*risque de deuxième espèce*) avec une probabilité élévée [67]. Pour la validation par le test consistant en la recherche d'erreurs, nous avons estimé qu'un niveau de confiance de 99% était suffisant. Si Lutess est utilisé pour évaluer la fiabilité du système sous test, le niveau de confiance doit être ajusté pour diminuer le risque de deuxième espèce.

Au cours de nos tests, nous avons aussi voulu nous assurer que le générateur aléatoire contraint produisait des valeurs de "façon aléatoire". Knuth note à ce sujet que lorsqu'un générateur produit des effectifs ayant une faible probabilité, il est possible qu'il ne soit pas aléatoire. Or, la probabilité que les effectifs observés soient exactement ceux attendus est

faible. C'est pourquoi, on peut être amené à rejeter un générateur s'il produit ce type d'effectif (i.e. s'il est "trop bon", il n'est peut être pas aléatoire). Par exemple, pour un test de χ^2 avec un risque $\alpha = 0.01$, il est suggéré de rejeter l'hypothèse "le générateur est aléatoire" pour des valeurs de χ^2 telle $\mathbb{P}(\chi^2 \leq v) \leq \alpha$. Pour un test dont le degré de liberté est égal à 1 (resp. 3), on a $v < 0.001$ (resp. $v = 0.072$). Au cours des tests précédents, les valeurs de χ^2 ont toujours été supérieures à ces valeurs limites. Nous considérons donc que notre générateur est très probablement aléatoire.

Au cours de ce chapitre, nous avons utilisé un exemple de spécification d'un ascenseur. Cet exemple a été largement utilisé pour illustrer l'utilisation de méthodes formelles telles que B [2], Z [38], SMV [83]. Pourtant, les approches sont difficilement comparables sur cet exemple : les spécifications développées sont différentes et les estimations des efforts sont rarement données sous forme quantitative. Ainsi, la spécification de l'ascenseur en SMV proposée dans [83] est plus abstraite que celle que nous avons développée, mais comporte plusieurs fonctionnalités supplémentaires (spécification du comportement de l'ascenseur lorsqu'il est vide, à $\frac{2}{3}$-plein ou en surcharge). Cette spécification (de 120 lignes de SMV) a été validée par un vérificateur de modèles, mais le temps nécessaire à la vérification n'est pas indiqué.

Nous n'avons pas estimé les temps de développement de notre spécification (350 lignes de Lustre), de son environnement et des propriétés. Par contre, nous avons effectué des mesures de temps d'exécution des tests avec Lutess. Ainsi, sur une station Sparc Ultra-1 avec 128 Mo de RAM, la construction du générateur pour la méthode de génération aléatoire, pour cet exemple, a été estimé à moins d'une minute (environ 40 secondes CPU). La seule production de 10 000 pas de test prend un peu plus d'une minute (environ 80 secondes CPU). La génération de 50 000 pas de test (incluant la construction du générateur) s'effectue en moins de 7mn.

Pour la méthode guidée par les propriétés, le temps de construction du générateur dépend du nombre et de la complexité des propriétés. Pour la production des séquences utilisées pour les études statistiques ci-dessus, le temps de construction du générateur est de 15mn (900 secondes CPU). Le temps de génération des données de test n'est pas modifié.

La méthode de génération de base présente un gros avantage : grâce à l'hypothèse d'uniformité, tous les vecteurs valides dans un état sont accessibles au bout d'un certain temps. On garantit ainsi l'équité de la simulation de l'environnement que l'on définit comme : "si un état q de la machine génératrice est accédé un nombre infini de fois, alors toute entrée i satisfaisant $env(q, i)$ sera exhibée un nombre infini de fois" [79]. Cette définition s'inspire de la propriété d'équité énoncée par Pnueli pour les systèmes réactifs [84].

Les deux méthodes présentent un inconvénient : la distribution des données d'entrées n'est pas paramétrable. Ainsi, il n'est pas possible de la modifier afin de favoriser des événements rares (au sens probabilistique). De plus, il est difficile d'introduire des corrélations entre des variables définies comme indépendantes dans les contraintes de l'environnement. Par exemple, avec la méthode de génération aléatoire, il n'est pas possible d'imposer que la probabilité d'appuyer sur le bouton *requête[i]* dépend de l'étage auquel se trouve l'ascenseur. La méthode de génération guidée par des propriétés permet d'établir une telle corrélation, mais il faut l'exprimer par une propriété comme par exemple `pre` *étage[i]* => `not` *requête[i]*.

Deuxième partie

Lutess : vers une méthode de génération statistique

Chapitre 5

Notion de profil opérationnel

Introduction

Le test fonctionnel (boîte noire) a pour vocation d'augmenter la confiance que l'on peut avoir dans une application. La confiance maximum ne peut être obtenue qu'en pratiquant un test exhaustif. Pour un programme transformationnel, cela consiste à démontrer que toutes les valeurs possibles du domaine d'entrée du programme mènent à une exécution réussie. Le lecteur aura déjà deviné que la tâche n'est pas souvent réalisable. La quantité astronomique de données à tester et le peu de temps en rapport rendent une telle démonstration impossible pour la plupart des cas.

Ce qui est quasiment impossible pour les programmes transformationnels, l'est encore plus pour les systèmes réactifs. En effet, pour un système réactif, un test exhaustif doit montrer que toutes les *séquences* d'entrée possibles mènent à une exécution réussie.

Dès lors que l'objectif de réaliser un test exhaustif, synonyme de vérification, est abandonné, le test fonctionnel vise à révéler le plus grand nombre de d'erreurs possible du système sous test. Il faut alors choisir un sous ensemble de données de test, représentatif des divers cas à tester, suffisamment riche pour avoir une forte capacité à révéler des erreurs, et suffisamment limité en taille pour être gérable. En l'absence de toute spécification fonctionnelle du système sous test, les cas à tester se résument aux divers usages (ou conditions d'utilisation) de ce système.

La première méthode que nous avons présentée repose sur une sélection purement aléatoire des données de tests. Bien que cette méthode offre une garantie d'équité et que son efficacité soit reconnue [33, 54], elle peut ne pas satisfaire un testeur. En effet, pour observer des comportements peu probables, on peut être amené à produire une très grande quantité de données.

Pour parvenir à une meilleure détection d'erreur, un utilisateur peut souhaiter s'assurer en priorité que les comportements de son application satisfont une propriété. C'est dans cette optique que I. Parissis a introduit dans Lutess une méthode de génération favorisant la sélection de données significatives pour le test de propriétés jugées intéressantes.

Par ailleurs, un utilisateur peut vouloir réunir en priorité les données de tests correspondant à l'utilisation la plus fréquente (ou à un usage très rare) sans affecter l'efficacité du test

du système. Ce souhait correspond à une démarche classique pour la validation des systèmes qui consiste à répartir l'effort de test en fonction des modes d'utilisations. Cette information est généralement décrite à l'aide de *profils opérationnels* [74].

Nous avons dans l'idée d'offrir aux utilisateurs de Lutess un moyen de favoriser l'apparition de certaines séquences de test, via l'utilisation de profils opérationnels ou de procédés similaires. L'objet de cette deuxième partie est la mise en œuvre et la validation d'une méthode de test correspondant à ces critères. Dans ce chapitre, nous décrivons la notion de profil opérationnel, nous évaluons les modifications à effectuer dans Lutess par rapport aux besoins d'un utilisateur de Lutess.

5.1 Profil opérationnel : concept et usages

Un profil opérationnel décrit de façon quantitative le mode d'utilisation d'un système. Selon Musa [74], le principal intérêt des profils opérationnels est de s'assurer que les fonctions les plus utilisées auront reçu le maximum d'attention, même si la validation du produit n'est pas terminée au moment de la livraison.

Un profil opérationnel définit des ensembles de vecteurs d'entrées disjoints et une probabilité d'apparition pour chacun de ces vecteurs. Par exemple, pour le convertisseur de clics (∼⊜), on peut distinguer deux ensembles d'entrées correspondant respectivement à "clic physique" et "aucun clic physique". On peut aussi distinguer quatre ensembles correspondant respectivement à chacune des entrées possibles (⊜∼).

Une propriété importante d'un profil est que la somme des probabilités fournies est égale à 1. Les profils sont généralement présentés sous forme de table ou de graphe. Un profil peut-être plus ou moins précis en fonction de ce que pour quoi il va être utilisé, et du temps que l'on peut consacrer à son élaboration.

Diverses utilisations d'un profil opérationnel

Les profils opérationnels peuvent être utilisés dans différentes phases de l'élaboration du produit logiciel. Ils peuvent suggérer les priorités de développement, permettant ainsi d'offrir aux clients les fonctionnalités les plus utilisées au plus tôt. Ils focalisent aussi la rédaction du manuel d'utilisation sur les parties à développer plus particulièrement.

Les profils opérationnels peuvent être vus comme un moyen de communication entre clients et développeurs car ils permettent de cerner les besoins de façon plus précise.

Ils peuvent enfin servir pour des études de performance. En particulier, on peut analyser le comportement d'un système en augmentant la fréquence des différentes opérations.

Construction d'un profil opérationnel

L'élaboration d'un profil opérationnel se fait généralement par une démarche descendante : de l'étude du type de client, aux opérations possibles. A chaque étape, on quantifie combien chaque élément est utilisé. Selon une pratique maintenant admise en analyse des

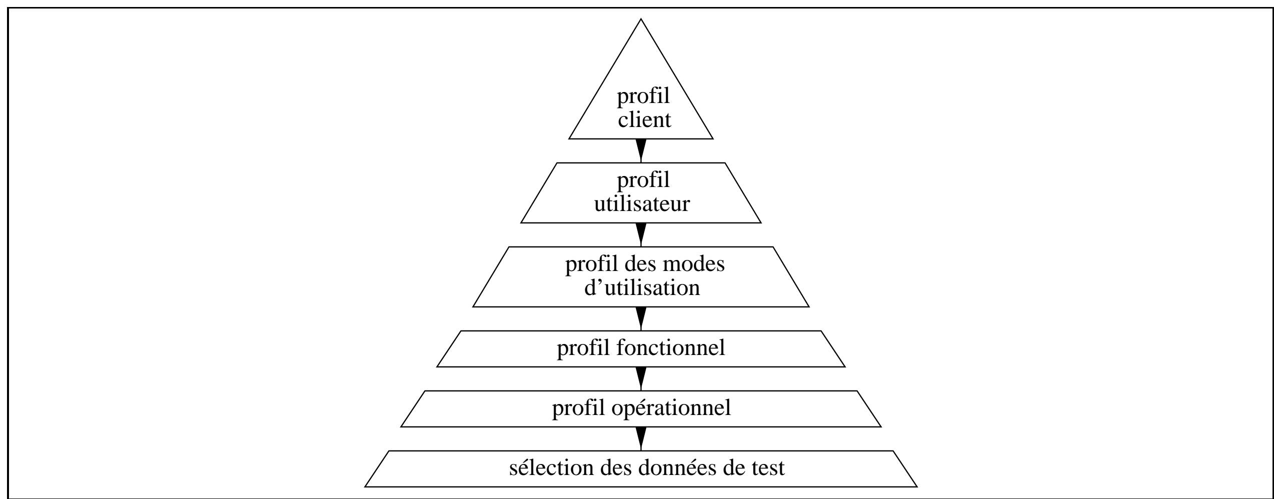


FIG. **5.1** – Développement d'un profil opérationnel

besoins (en génie logiciel), Musa distingue plusieurs étapes pour la construction d'un profil opérationnel. Les cinq qu'il identifie mènent à la détermination successives des profils des (fig. 5.1) :

- clients,
- utilisateurs,
- modes d'utilisations,
- fonctionnalités,
- et opérations (profil opérationnel).

Le client est une personne, un groupe, une institution qui fait l'acquisition du logiciel. La façon d'utiliser le produit dépend des activités du client. Par exemple, une calculatrice n'est pas utilisée de la même façon si elle est achetée par une institution ou par un particulier.

Un utilisateur est une personne, un groupe, une institution qui utilise le logiciel. On distingue le client des utilisateurs effectifs du système; le mode d'utilisation ne dépend pas forcément que du type de client. Par exemple, la calculatrice achetée par une mère de famille sera utilisée différemment par ses enfants, selon qu'ils sont au collège ou à l'université.

Un mode d'exécution est un ensemble de fonctions ou d'opérations qui ont été regroupées pour des facilités d'exécution. Une calculatrice type possède souvent un mode de calcul basic (+,-,*,/), un mode mathématique avancé (fonctions exp, log...), un mode statistique (fonctions calculant la moyenne, l'écart type...).

Pour chacune des étapes, on peut construire un "profil" adapté. Par exemple, on peut estimer la répartition des différents types de clients.

La construction d'un profil opérationnel s'appuie alors sur les profils intermédiaires. La probabilité de l'opération '+' est déterminée à partir de la probabilité des modes d'exécutions. Elle même est calculée à partir de la répartition des types d'utilisateurs, etc...

Méthode de description des profils

L'étape suivante consiste à décomposer chaque mode en un ensemble de fonctions. Pour la calculatrice, il s'agit des fonctions mathématiques citées ci-dessus.

Enfin, il convient d'élaborer les profils opérationnels. On peut exprimer les profils de façon explicite ou implicite. Expliciter un profil consiste à décrire tous les vecteurs d'entrées possibles. Un profil implicite consiste à ne décrire que l'ensemble des valeurs de chaque variable. Le profil implicite n'a vraiment de sens que dans le cas où les variables de la fonction sont indépendantes, bien que cette façon de faire soit courante dans l'industrie.

Soit, par exemple, une fonction à deux entrées x et y, qui ont respectivement 3 et 2 valeurs indépendantes ($x \in \{x_1, x_2, x_3\}$ et $y \in \{y_1, y_2\}$). On peut alors produire un profil opérationnel explicite, comme par exemple :
$\mathbb{P}(x_1, y_1)$=0.2, $\mathbb{P}(x_1, y_2)$=0.3, $\mathbb{P}(x_2, y_1)$=0.12, $\mathbb{P}(x_2, y_2)$=0.18, $\mathbb{P}(x_3, y_1)$=0.08, $\mathbb{P}(x_3, y_2)$=0.12, ou implicite, comme :
$\mathbb{P}(x_1)$=0.5, $\mathbb{P}(x_2)$=0.3, $\mathbb{P}(x_3)$=0.2, $\mathbb{P}(y_1)$=0.6, $\mathbb{P}(y_2)$=0.4.

5.2 Profils opérationnels et Lutess

Offrir aux utilisateurs de Lutess l'emploi de profils opérationnels nous a amené à proposer une définition formelle des profils opérationnels pour les systèmes réactifs, et à évaluer les modifications à apporter à Lutess pour l'implantation d'une méthode de test basée sur ces profils.

5.2.1 Définition formelle des profils opérationnels

Intuitivement, un profil opérationnel est une fonction de distribution qui attribue une probabilité d'occurrence à chaque vecteur d'entrée. La fonction de distribution est telle que la somme des probabilités d'occurrence des vecteurs d'entrée est égale à 1.

Définition 7 *Soit* $\mathcal{M}_{env} = (Q, q_{init}, O, I, env, t_{env}, \phi_{env})$ *une machine génératrice. On appelle* profil opérationnel *de* $\mathcal{M}_{env}$*, une fonction* $PO : Q \times V_I \rightarrow [0..1]$ *telle que :*

- $\forall q \in Q, \forall i \in V_I, \ (env(q,i) = vrai \Rightarrow PO(q,i) \in [0..1])$
- $\forall q \in Q, \forall i \in V_I, \ (env(q,i) = faux \Rightarrow PO(q,i) = 0)$
- $\forall q \in Q, \sum_{i \in V_I} PO(q,i) = 1$

Il existe un type de profils opérationnels remarquable : celui qui définit une distribution équiprobable.

Propriété 1 *Un profil opérationnel* PO *d'une machine génératrice* $\mathcal{M}_{env} = (Q, q_{init}, O, I, env, t_{env}, \phi_{env})$ *définit une distribution* équiprobable *si et seulement si :*
$\forall q \in Q, \forall i, i' \in V_I, ((env(q,i) = vrai \wedge env(q,i') = vrai) \Rightarrow (PO(q,i) = PO(q,i')))$

On note que par défaut, Lutess utilise un profil opérationnel équiprobable.

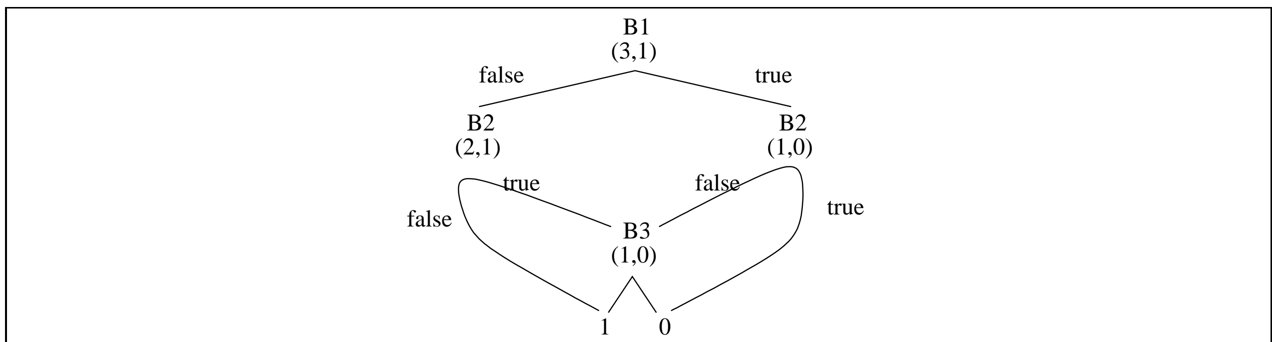


FIG. **5.2** – BDD étiqueté représentant l'environnement du convertisseur de clic : ECC

5.2.2 Description d'un profil

La définition d'un profil opérationnel s'appuie soit sur la description des domaines des variables (profil implicite), soit sur la définition de l'ensemble des vecteurs d'entrées possibles (profil explicite). La description la plus pratique pour l'utilisateur est évidemment la description implicite, puisque plus synthétique. Mais l'utilisation des profils implicites n'est possible que si les variables d'entrées sont indépendantes les unes des autres. Si l'on offre la possibilité de définir des profils aux utilisateurs de Lutess, il faut pouvoir offrir les deux modes de descriptions.

Définir un profil opérationnel explicite

Pour définir un profil opérationnel explicite, il faut connaître l'ensemble des vecteurs d'entrée valides. Pour Lutess, cet ensemble est représenté de façon synthétique à l'aide d'un BDD. Le travail de l'utilisateur consiste donc à parcourir tout le BDD, et donner une probabilité à chaque vecteur d'entrée. Lutess doit alors s'assurer que pour chaque état, la somme des probabilités des vecteurs d'entrée possibles est égale à 1 (par définition du profil) et signaler toute anomalie à l'utilisateur. On peut noter dès à présent que le travail demandé à l'utilisateur n'est pas réaliste a priori.

Exemple

Reprenons l'exemple du convertisseur de clic d'une souris, introduit chapitre 3. L'environnement du convertisseur est décrit par la fonction #(B1,B2,B3); de ce fait l'ensemble des vecteurs d'entrées possibles est $\{(0,0,0),(0,0,1),(0,1,0),(1,0,0)\}$.

Le BDD représentant cette contrainte d'environnement est décrit par la figure 5.2. Sur cette figure, l'étiquetage permet d'assurer une génération de donnée équiprobable. Lors d'un parcours de la racine vers une feuille, pour chaque variable (nœud), l'étiquetage (v_0, v_1) indique le nombre de chemins valides selon que la variable est instanciée à faux ou à vrai. La probabilité d'affecter la variable à vraie est $v_1/(v_0+v_1)$. Par exemple, la probabilité d'affecter $B1$ à vrai est $\frac{1}{3+1} = \frac{1}{4}$.

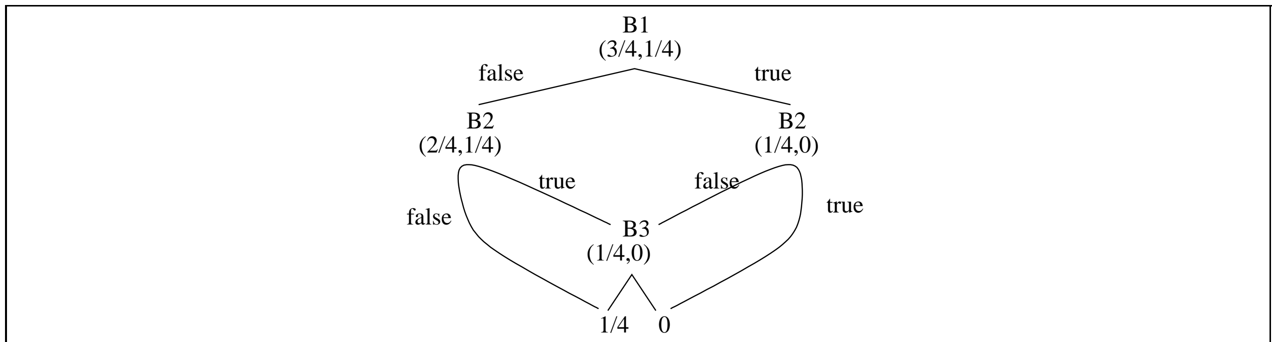


Fig. **5.3** – Etiquetage de ECC pour le profil opérationnel P_1

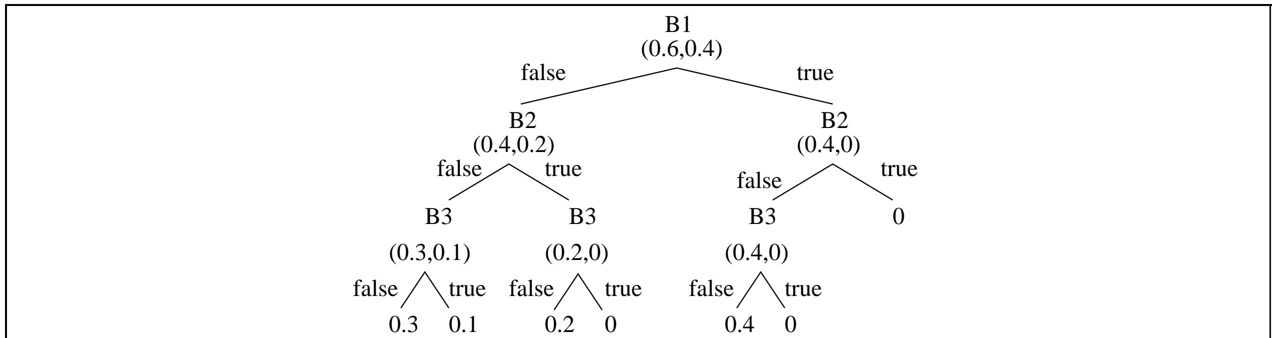


Fig. **5.4** – Etiquetage de ECC pour le profil opérationnel P_2

On souhaite guider la génération en fonction de ces profils opérationnels. Une idée simple consiste à redéfinir la notion d'étiquetage sur le modèle de la génération équiprobable. Aux feuilles valides, on fait apparaître la probabilité attendue du vecteur d'entrée correspondant. Puis pour chaque nœud interne, on définit un étiquetage (v'_0, v'_1) qui représente la distribution attendue des vecteurs "accessibles" selon que la variable est instanciée à faux ou à vrai. La probabilité d'affecter la variable à faux se calcule comme précédemment : $v'_0/(v'_0 + v'_1)$.

Soit un profil opérationnel $P1$ définissant une distribution équiprobable pour l'exemple : $\mathbb{P}(1,0,0) = \mathbb{P}(0,1,0) = \mathbb{P}(0,0,1) = \mathbb{P}(0,0,0) = \frac{1}{4}$. L'étiquetage attendu pour ce profil est donné par la figure 5.3. Avec cet étiquetage, on note que la probabilité d'affecter $B1$ à *vrai* est $\frac{\frac{1}{4}}{\frac{3}{4}+\frac{1}{4}} = \frac{1}{4}$, résultat qui concorde avec celui obtenu pour l'étiquetage de base.

Cette solution est simple mais malheureusement parfois inapplicable. En effet, soit un second profil opérationnel $(P2)$: $\mathbb{P}(1,0,0)=0.4$, $\mathbb{P}(0,1,0)=0.2$, $\mathbb{P}(0,0,1)=0.1$, $\mathbb{P}(0,0,1)=0.3$. Pour obtenir ce profil opérationnel, nous devons utiliser l'étiquetage donné par la figure 5.4. Comme le lecteur peut le constater, ce nouvel étiquetage a demandé une expansion du BDD pour différencier l'étiquetage des vecteurs.

5.2.3 Solution technique

Faut-il expanser des BDD?

Cette question est relativement délicate. La structure de BDD est une structure très compacte et efficace. Casser cette structure peut aboutir à une perte importante en terme de concision de représentation. On peut donc se demander si cette approche est à retenir. La question peut alors se reformuler comme suit :

- peut-on imaginer une autre solution pour représenter le profil opérationnel?
- est-ce que cette méthode correspond bien aux besoins de l'utilisateur?

Quelle(s) autre(s) solution(s)?

Parmi les autres solutions, on peut imaginer d'effectuer une étape de compression du BDD, s'appuyant sur les étiquettes définies par le profil opérationnel.

On peut aussi imaginer qu'à chaque nœud du BDD, on fait correspondre une liste d'étiquette. A chaque étiquette, on associe une condition définissant dans quels cas une étiquette doit être utilisée. A partir de la définition d'un profil opérationnel, il semble possible d'automatiser la construction des étiquettes et de leurs conditions associées. Notons que lorsqu'un profil opérationnel n'impose pas d'expansion de BDD, on retrouve l'étiquetage utilisé pour la méthode de base.

Il existe probablement d'autres méthodes. Mais avant de choisir, on souhaite évaluer dans quelle mesure la méthode des profils opérationnels peut satisfaire les utilisateurs de Lutess. Définir un profil opérationnel est un travail assez long, si l'on suit scrupuleusement la méthode décrite par Musa dans [74]. Pour chaque vecteur d'entrée possible, i.e. pour chaque feuille étiquetée *vrai* du BDD, il faut affecter une probabilité. Pour l'étude de cas de l'ascenseur, la description de l'environnement propose plus de 2^{35} vecteurs d'entrée valides. Cela demande donc un gros travail pour définir un profil opérationnel pour cet exemple, qui est pourtant un exemple simple.

5.3 Notion de probabilité conditionnelle

On s'est interrogé sur les besoins réels d'un utilisateur de Lutess, lorsque celui-ci cherche à valider des spécifications selon un estimation de la fréquence d'utilisation du futur système.

Un utilisateur peut vouloir décrire des comportements réalistes de l'environnement, mais, il ne souhaite pas forcément décrire *tous* les comportements réalistes. De même, il ne souhaite pas forcément décrire *exactement et précisément* ces comportements. La durée de test du logiciel (comprenant la spécification des comportements réalistes) doit être largement inférieure au temps nécessaire pour prouver le programme. En effet, à effort égal, la preuve d'une propriété apporte une confiance plus grande que le test, et il devient plus judicieux de l'utiliser.

Par ailleurs, dans un contexte de test de spécification, on ne peut pas demander à l'utilisateur de décrire de façon précise, quelque chose de potentiellement incomplet.

C'est pourquoi, il me semble que l'utilisation des profils opérationnels tels qu'ils ont été définis par Musa, ne convient pas pour les utilisateurs de Lutess. Nous proposons donc une autre méthode de description des comportements de l'environnement fondée sur l'utilisation de probabilités conditionnelles.

Lutess construit automatiquement un profil opérationnel qui assure l'équiprobabilité des vecteurs d'entrées. Nous pensons que pour un utilisateur, il est pratique de s'appuyer sur ce profil pour définir une nouvelle distribution. Nous lui proposons un méthode qui consiste à permettre des modifications *ponctuelles* de cette distribution de base.

Par ailleurs, définir une probabilité pour chaque vecteur possible dans un état est difficile. L'utilisateur ne connaît a priori ni les différents états de l'environnement, ni l'ensemble des vecteurs associés à ces états. Bien sûr, il est possible de lui fournir cette information : il suffit par exemple d'afficher le BDD d'environnement. Mais la quantité d'information alors proposée est potentiellement énorme et donc inexploitable (cf. les 2^{35} vecteurs valides du BDD d'environnement de l'ascenseur).

Par contre, l'utilisateur connaît bien l'ensemble des variables d'entrée du programme et les contraintes existant entre ces variables. De plus, le nombre des variables d'entrée est généralement restreint. Le choix que nous avons fait consiste à offrir à l'utilisateur la possibilité d'associer aux variables d'entrée différentes probabilités d'occurrences de la valeur *vrai* selon les états de l'environnement spécifiés par des prédicats (*conditions*). Les données d'entrée seront ensuite produites en fonction de ces *probabilités conditionnelles*.

Dans le chapitre suivant, on décrit la méthode de génération des données basée sur les probabilités conditionnelles et les algorithmes de traduction des probabilités conditionnelles en profils opérationnels, et vice versa.

Chapitre 6

Méthode de génération basée sur des probabilités conditionnelles

Ce chapitre est consacré à la présentation de la méthode de génération guidée par des probabilités conditionnelles, que nous avons implantée dans Lutess. Elle est accompagnée de méthodes de traduction des profils opérationnels en probabilités conditionnelles et vice et versa. Ces moyens systématiques de passage d'un mode d'expression à l'autre sont rendus nécessaires par le fait que les profils opérationnels sont un moyen d'expression plus fréquemment utilisé que les probabilités conditionnelles.

6.1 Guidage par des probabilités conditionnelles

6.1.1 Motivations et principe général

Cette méthode de génération a été mise en œuvre pour permettre à l'utilisateur de modifier localement la distribution statistique équiprobable de l'environnement. Par exemple, pour le convertisseur de clics (∼⊜), il est possible d'augmenter la probabilité de cliquer sur le premier bouton de la souris, ce qui correspond à une utilisation fréquente de ce type de périphérique (⊜∼).

Pour modifier la distribution des entrées, l'utilisateur doit fournir une liste de *probabilités conditionnelles associées aux variables d'entrées.* Cette liste est utilisée par Lutess pour déterminer les données de test.

Définition 8 *Soit $M_{env} = (Q, q_{init}, O, I, env, t_{env}, \phi_{env})$ une machine génératrice. Une liste de probabilité conditionnelle associée à M_{env} est une liste de triplets $[t_0, t_1, ..., t_n]$. Pour chaque triplet $t_i = \langle e, p, f_{pc} \rangle$, e est une variable d'entrée ($e \in I$), p est une probabilité ($p \in [0..1]$), et f_{pc} est un prédicat (une condition) tel que ($f_{pc} : Q \times V_O \times V_I \rightarrow \{vrai, faux\}$). p dénote la probabilité que la variable e prenne la valeur vrai lorsque la condition f_{cp} est vraie.*

On rappelle que dans Lutess le processus de génération des données est cyclique. A chaque cycle, le générateur exhibe une instanciation du vecteur d'entrée et la fournit au système sous test. Cette instanciation est construite en déterminant successivement une valeur pour chaque variable du vecteur d'entrée.

Pour chaque variable, soit l'état de l'environnement impose une valeur à la variable, soit il la laisse libre. Dans le deuxième cas, le générateur choisit une valeur en deux étapes. Tout d'abord, il calcule la probabilité avec laquelle cette variable doit être affectée à *vrai*. Puis, il effectue un tirage aléatoire en fonction de la probabilité précédemment calculée. Dans le cas de la méthode de base, la probabilité avec laquelle une variable est affectée à vraie est calculée en s'appuyant sur l'étiquetage de base.

Pour la méthode que nous proposons, Lutess calcule cette valeur selon l'algorithme suivant :

Soit $\mathcal{C}$ une liste de probabilités conditionnelles
Soit $\mathcal{C}(e_i)$ la liste de probabilités conditionnelles associées à la variable e_i
$\mathcal{C}(e_i) = \{\langle e, p, c\rangle \in \mathcal{C} | e = e_i\} = \{\langle e_i, p_1, ce_1\rangle, \langle e_i, p_2, ce_2\rangle, ..., \langle e_i, p_r, ce_r\rangle\}$

$\mathbb{P}(e_i = vrai) =$ si ce_1 alors p_1 sinon
si ce_2 alors p_2 sinon
....
si ce_r alors p_r sinon $\frac{v_1}{v_0+v_1}$ *avec v_1 et v_0 étiquetage de base*
$\mathbb{P}(e_i = faux)=$ (1 - $\mathbb{P}(e_i = vrai)$)

Le lecteur notera que les triplets sont évalués dans l'ordre d'occurrence de leur déclaration dans $\mathcal{C}$. Cela n'a pas d'influence si les conditions pour une variable sont exclusives. Par ailleurs, si l'utilisateur ne définit pas de probabilité conditionnelle, on retrouve l'algorithme de base. Le paragraphe suivant détaille cet algorithme.

6.1.2 Algorithme de sélection des valeurs

Cet algorithme s'appuie sur l'algorithme de sélection de base.

ProbaCond : Le type : $\langle$ nom_variable, réel $\in [0..1]$, expression booléenne $\rangle$

tirage : un réel $\in [0..1] \rightarrow$ un booléen
{ *tirage(i) produit la valeur 'vrai' avec la probabilité i* }
CalculVariable : un entier, une liste de ProbaCond, un réel, 2 vecteurs de booléen
$\rightarrow$ un booléen
{ *CalculVariable(i,L,p,E,v) : retourne l'instanciation de la variable i, calculée à partir de L et p. Les conditions de L sont évaluées à partir de la valeur des variables d'état et des valeurs de v[0..i-1].* }
CalculVariable(i,L,p,E,v) =
Parcourir la liste de probabilitées conditionnelles L du premier au dernier élément
s'il existe un triplet $\langle i, q, c\rangle$, tel que EvalCond(c,E,v) = *vrai*
alors tirage (q)
sinon tirage (p)

EvalCond : une expression booléenne, 2 vecteurs de booléens $\rightarrow$ un booléen
{ *EvalCond(c,E,v) est la valeur de c compte tenu de la valeur des variables d'état E et des variable d'entrée courantes v* }

```
CalculVE3 : un BDD étiqueté non vide, un vecteur de booléens, 2 entiers, une liste
        de ProbaCond, un vecteur de booléens → un vecteur de booléens ou "erreur"
{ CalculVE3(A,v,i,N,L,E) calcule une instanciation de v pour le BDD A; N est le
    nombre de variables d'entrée de A, et i est l'indice de la variable courante à
    instancier. L est la liste de probabilités conditionnelles. E est l'état courant de
    l'environnement }
CalculVE3(A,v,i,N,L,E) =
    si A est une feuille alors
        si A = faux alors "erreur"
        sinon pour chaque variable abstraite d'indice k=i à N-1,
            v[k] = CalculVariable(k,L,0.5,E,v)
    sinon                                    { A n'est pas une feuille }
        soit ⟨lg,ld⟩ l'étiquette de la racine de A
             j l'indice de la variable portée par la racine de A
        dans
             pour chaque variable abstraite d'indice k=i à j-1,
                    v[k] = CalculVariable(k,L,0.5,E,v)
             si lg = 0
             alors si ld = 0
                    alors "erreur"
                    sinon { v[i] = vrai ; CalculVE3(D,v,i+1,N,L,E) }
             sinon si ld = 0
                    alors { v[i] = faux ; CalculVE3(G,v,i+1,N,L,E) }
                    sinon { v[i] = CalculVariable(k,L,(ld/lg+ld),E,v) } ;
                             si (v[i] = vrai )
                             alors CalculVE3(D,v,i+1,N,L,E)
                             sinon CalculVE3(G,v,i+1,N,L,E)
```

Le lecteur aura remarqué le problème de l'interprétation des conditions (dans l'expression EvalCond(c,E,v)). Pour évaluer une condition il faut pouvoir évaluer les différentes variables. Cela a deux conséquences. Tout d'abord, chacune de ces variables, utilisée autre qu'une variable d'entrée, doit figurer dans l'ensemble des variables d'état de l'environnement (notée E ci-dessus). De plus, comme il n'est pas possible d'évaluer la valeur des toutes les variables du vecteur d'entrée courant, on impose des restrictions, qui sont formalisées dans le paragraphe suivant. On illustre ce problème avec l'exemple du convertisseur de clics (~⊜). Le programme comporte 3 variables d'entrée dont l'ordre est (B1,B2,B3). Le vecteur d'entrée est produit en instanciant une valeur pour chacune des variables dans l'ordre imposé par le programme : on affectera B1, puis B2, puis B3. Par conséquent, une condition associée à B1 ne doit pas porter sur la valeur de B2 ou B3 au pas courant, puisque ces variables ne sont pas encore affectées.

La méthode de génération basée sur les probabilités conditionnelles est compatible avec celle basée sur les propriétés. En effet, même si cette dernière construit deux BDD d'environnement pour représenter les fonctions ϕ_{env} et ϕ_{per}, le générateur choisit l'un des deux

et applique la méthode de sélection équiprobable. Or la méthode de génération basée sur les probabilités conditionnelles est applicable sur n'importe quel BDD. La compatibilité des deux méthodes a donc été facilement assurée, en appliquant l'algorithme de sélection présenté ci-dessus, sur le BDD choisi par la méthode basée sur les propriétés.

L' algorithme général de sélection des données prenant en compte la compatibilité des méthodes est le suivant :

Lexique
  $M_p = (M_{env}, P)$ une machine éventuellement guidée par des propriétés.
    { $M_{env} = (Q, q_{init}, S, E, env, t_{env}, \phi_{env})$ }
  $\phi_{per} : Q \times E \times \{vrai, faux\}$: la fonction décrivant les données pertinentes.
    { *ϕ_{per} est représenté par un BDD et $\forall q \in Q$, $\phi_{per}(q)$ détermine un BDD d'entrée* }
  $\mathcal{L}$ une liste de probabilités conditionnelles
  q : un état de M_{env}
  vs : une instanciation de S (v $\in V_S$)
  ve : une instanciation de E (v $\in V_E$)
  A : un BDD
  N : un entier
  tmp : un booléen
Génération3 :
  N = $|E|$ — *nombre de variable d'entrée*
  q = q_{init}
  pour chaque pas de test faire :
    A = $\phi_{per}(q)$;
    si ((A = $faux$) ou (tirage(0.1) = $faux$)) — *pas de données pertinentes ou choix ϕ_{env}*
      alors A = $\phi_{env}(q)$; — *sélection du BDD d'entrée*
    vs = CalculVE3(A,vs,0,N,L,q);
    envoyer(vs); — *vers le système sous test*
    récupérer(ve); — *produit par le système sous test*
    q = $t_{env}(q, vs, ve)$; — *calcul du nouvel état*

6.1.3 Formalisation

Contraintes sur les probabilités conditionnelles

Définition 9 *Soit $M_{env} = (Q, q_{init}, O, I, env, t_{env}, \phi_{env})$ une machine génératrice (resp. $M_p = (M_{env}, P)$ une machine génératrice guidée par des propriétés), avec I=$(i_0, i_1, ..., i_n)$. Une probabilité conditionnelle $\langle x, p, f \rangle$ associée à M_{env} (resp. M_p) est bien formée si*

- *$x \in I$,*
- *$0 \leq p \leq 1$*
- *si $x = i_0$ alors $f : Q \to \{vrai, faux\}$*
- *si $x = i_j, j > 0$ alors $f : Q \times i_0 \times ... \times i_{j-1} \to \{vrai, faux\}$*

Générateur associé à cette méthode

Il s'agit d'un couple associant une machine génératrice et un algorithme de sélection des données. On a choisi de définir une machine génératrice assurant la compatibilité entre les

méthodes de génération.

Définition 10 *Soit $M_p = (M_{env}, P)$ une machine génératrice guidée par des propriétés. Une machine génératrice guidée par des probabilités conditionnelles est définie par le tuple suivant : $M_d = (M_p, \mathcal{L})$ où $\mathcal{L} = (pc_0, pc_1, ..., pc_k)$ est une liste de probabilités conditionnelles bien formées associée à M_{env}.*

Le générateur $\mathcal{G}_d$ associé à cette méthode est donc $\mathcal{G}_d = (M_d,$ Génération3) où M_d est une machine d'entrée-sortie guidée par des probabilités conditionnelles, et Génération3 est l'algorithme défini précédemment.

Analyse de l'algorithme

Pour un BDD bien étiqueté, représentant une fonction de n variables et définissant au moins un vecteur valide, et une liste $\mathcal{L}$ quelconque de probabilités conditionnelles, on veut montrer que $\mathcal{P}(n)$: *l'algorithme de sélection des données associé à un générateur $\mathcal{G}_d$ s'appuie sur un profil opérationnel,*

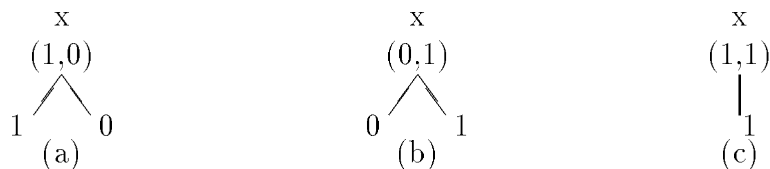
Montrer que l'algorithme de sélection de $\mathcal{G}_d$ produit un profil opérationnel revient à montrer que

- tous les vecteurs valides ont une probabilité,
- pour chaque état de l'environnement, la somme des probabilités des vecteurs est égale à 1.

Deux cas se présentent pour les propriétés désignées par P. Si P est *vrai*, on est ramené à analyser $\mathcal{P}(n)$ sur $\mathcal{G} = (M_{env},$ Génération). Dans le second cas, il faut analyser $\mathcal{P}(n)$ sur le BDD représentant ϕ_{env} et ϕ_{per}. Dans les deux cas, on est ramené à l'analyse de la propriété sur un BDD quelconque.

Lorsque $\mathcal{L}$ est vide, on est ramené à l'analyse menée en §3.3.3.

Lorsque $\mathcal{L}$ est non vide, on procède par récurrence. Analysons $\mathcal{P}(1)$
Soit une fonction à un paramètre $f(x)$. Il existe trois possibilités pour f.



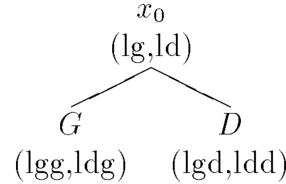
- Pour (a), (resp. (b)), la liste de probabilités conditionnelles n'est pas consultée, du fait des contraintes d'environnement. L'algorithme de sélection produit x=*false* (resp. x=*true*) avec une probabilité égale à 1. Pour ces deux cas, le domaine de f est réduit à un vecteur, $\mathcal{P}$ est trivialement vraie.

- Pour (c), $\mathcal{L}$ est consultée. Soit p la probabilité utilisée pour l'instanciation de la variable x. p est extraite de $\mathcal{L}$ ou de l'étiquetage de base. CalculVariable établit la valeur x=*true*

avec une probabilité égale p. La valeur x=*false* est établie avec la probabilité $(1 \Leftrightarrow p)$. La somme des probabilités des deux instanciations possibles de x est égale à 1. $\mathcal{P}(1)$ est vraie.

On suppose maintenant $\mathcal{P}(n)$, et on montre $\mathcal{P}(n+1)$. Soient $(x_0, x_1..., x_n)$ les paramètres de f. Soit A le BDD canonique représentant $f(x_0, x_1..., x_n)$. Deux cas se présentent : (1) A a pour racine x_0, (2) A a pour racine $x_i, i > 0$.

Cas (1) : soient G et D les sous-arbres gauche et droit de A, et (lgg,ldg) et (lgd,ldd) l'étiquette de leur racine respective.



G et D représentent chacun une fonction de n paramètres : $f(0, x_1, ..., x_n)$ et $f(1, x_1, ..., x_n)$. Sachant que $\mathcal{P}(n)$ est vraie, on peut en conclure que :

- $lg = lgg + ldg$ est le nombre de vecteurs valides pour G. Chaque vecteur valide i a une probabilité $p(i)$ d'être choisi, et la somme des probabilités de ces vecteurs est égale à 1.
- $ld = lgd + ldd$ est le nombre de vecteurs valides pour D . Chaque vecteur valide i a une probabilité $p(i)$ d'être choisi, et la somme des probabilités de ces vecteurs est égale à 1.

Comme pour le cas de base, on peut distinguer trois cas.

- $lg = 0$ et $ld \neq 0$
  L'algorithme de sélection produit x=*true* avec la probabilité 1, sans avoir à consulter $\mathcal{L}$. A définit lg vecteurs valides. Chaque vecteur valide i a la probabilité 1*p(i)=p(i) d'être choisi. Comme $\mathcal{P}$(n) est vraie, on obtient que $\mathcal{P}$(n+1) est vraie.
- $lg \neq 0$ et $ld = 0$
  L'algorithme de sélection produit x=*false* avec la probabilité 1, sans avoir à consulter $\mathcal{L}$. A définit ld vecteurs valides. Chaque vecteur valide i a la probabilité 1*p(i)=p(i) d'être choisi. Comme $\mathcal{P}$(n) est vraie, on obtient que $\mathcal{P}$(n+1) est vraie.
- $lg \neq 0$ et $ld \neq 0$
  $\mathcal{L}$ est consultée. Soit q la probabilité utilisée pour l'instanciation de la variable. q est extraite de $\mathcal{L}$ ou de l'étiquetage de base. CalculVariable établit la valeur x=*true* avec une probabilité égale q. La valeur x=*false* est établie avec la probabilité $(1 \Leftrightarrow q)$.
  - A définit $lg + ld$ instanciations possibles.
  - Chacune des lg instanciations possibles pour lesquelles $x_0 = false$ ont la probabilité $p(i) * (1 \Leftrightarrow q)$ d'apparaître. La somme des probabilités des vecteurs pour ce sous-arbre est $\sum_i (p(i) * (1 \Leftrightarrow q)) = (1 \Leftrightarrow q) * \sum_i p(i) = (1 \Leftrightarrow q)$, car $\mathcal{P}$(n) est vraie.

- De même, les ld instanciations possibles pour lesquelles $x_0 = true$ ont la probabilité $p(j) * q$ d'apparaître. La somme des probabilités des vecteurs pour ce sous-arbre est $\sum_j (p(j) * q) = q * \sum_j p(j) = q$, car $\mathcal{P}$(n) est vraie.

- On obtient que la somme des probabilités des instanciations possibles est $q + (1 \Leftrightarrow q) = 1$.

$\mathcal{P}$(n+1) est vraie.

Dans le cas (2), la variable x_0 est abstraite. Cela signifie que les BDD représentant $f(0, x_1, ..., x_n)$ et $f(1, x_1, ..., x_n)$ sont isomorphes.

L'algorithme de sélection choisit l'instanciation $x_0 = true$ (resp. $x_0 = false$) avec la probabilité q (resp. 1-q) à l'aide de la fonction CalculVariable. Comme $\mathcal{P}(n)$ est vraie,

- pour chaque vecteur i défini par $f(1, x_1, ..., x_n)$, on obtient une probabilité d'instanciation égale à $q * p(i)$ et $\sum_i p(i) * q = q$, et
- pour chaque vecteur j défini par $f(0, x_1, ..., x_n)$, on obtient une probabilité égale à $q * p(j)$ et $\sum_j p(j) * (1 \Leftrightarrow q) = (1 \Leftrightarrow q)$.
- La somme des probabilités des vecteurs valides définis par $f(x_0, x_1, ..., x_n)$ et $f(0, x_1, ..., x_n)$ est donc $q + (1 \Leftrightarrow q) = 1$

En conclusion, $\mathcal{P}(n+1)$ est vraie.

6.1.4 Aspects techniques

Les conditions associées aux probabilités conditionnelles sont compilées en même temps que la description des contraintes d'environnement. Ainsi, toutes les variables nécessaires à l'évaluation des conditions sont présentes dans le BDD d'environnement. Pour se faire, nous avons introduit des sorties fictives dans le nœud de test, pendant l'étape de transformation du noeud testeur en noeud Lustre (voir page 33). A chaque condition, nous avons associé une sortie fictive. Les variables d'état nécessaires à l'évaluation d'une condition sont alors présentes dans le BDD représentant l'environnement.

6.2 D'un profil opérationnel à des probabilités conditionnelles

La définition de distributions spécifiques pour les entrées d'un logiciel relève de l'activité connue de construction de profils opérationnels. Sur le plan pratique, comme les utilisateurs de Lutess ne peuvent pas utiliser directement des profils, nous proposons une méthode pour traduire un profil opérationnel en des probabilités (in)conditionnelles [26]. On illustre tout d'abord la méthode avec deux exemples concernant le convertisseur de clics. On établit ensuite l'algorithme général.

6.2.1 Exemple avec un profil opérationnel simple

L'algorithme de génération présenté ci-dessus utilise les probabilités conditionnelles à la place de l'évaluation des étiquettes pour déterminer la probabilité d'instancier une variable à vrai ou à faux. Ainsi, il faut (1) caractériser les nœuds du BDD et (2) associer une valeur aux nœuds ainsi caractérisés.

~☰ Reprenons l'exemple du convertisseur de clics. L'environnement du programme est défini par la contrainte “au plus un clic à chaque instant”. Cette contrainte s'exprime par l'arbre de Shannon présenté figure 6.1. Chaque nœud de l'arbre identifie une variable et une condition caractérisant le chemin aboutissant à ce nœud depuis la racine.

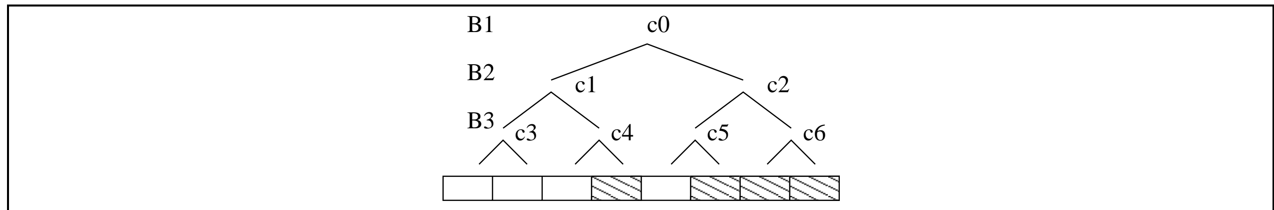


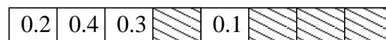
FIG. **6.1** – Coefficients de l'arbre de Shannon définissant l'environnement du convertisseur de clics

Sur la figure 6.1, un coefficient $c_i, 0 \leq i \leq 6$ a été associé à chacun des nœuds de l'arbre. On suppose que c_i est la probabilité d'instancier à vrai la variable portée par le nœud correspondant du BDD. On peut alors construire la liste de probabilités conditionnelles abstraites suivante :
[⟨ B1, c_0, true ⟩, ⟨ B2, c_1, not B1 ⟩, ⟨ B2, c_2, B1 ⟩, ⟨ B3, c_3, not B1 and not B2 ⟩,
⟨ B3, c_4, not B1 and B2 ⟩, ⟨ B3, c_5, B1 and not B2 ⟩, ⟨ B3, c_6, B1 and B2 ⟩]

Le problème consiste alors à déterminer la valeur des coefficients c_i selon le profil opérationnel choisi. Pour ce faire, il suffit de résoudre le système d'équations suivant.

$$\left\{\begin{array}{ll}
(1 \Leftrightarrow c_0).(1 \Leftrightarrow c_1).(1 \Leftrightarrow c_3) & = \mathbb{P}(\neg \text{ B1} \wedge \neg \text{ B2} \wedge \neg \text{ B3}) \\
(1 \Leftrightarrow c_0).(1 \Leftrightarrow c_1).c_3 & = \mathbb{P}(\neg \text{ B1} \wedge \neg \text{ B2} \wedge \text{ B3}) \\
(1 \Leftrightarrow c_0).c_1.(1 \Leftrightarrow c_4) & = \mathbb{P}(\neg \text{ B1} \wedge \text{ B2} \wedge \neg \text{ B3}) \\
(1 \Leftrightarrow c_0).c_1.c_4 & = \mathbb{P}(\neg \text{ B1} \wedge \text{ B2} \wedge \text{ B3}) \\
c_0.(1 \Leftrightarrow c_2).(1 \Leftrightarrow c_5) & = \mathbb{P}(\text{B1} \wedge \neg \text{ B2} \wedge \neg \text{ B3}) \\
c_0.(1 \Leftrightarrow c_2).c_5 & = \mathbb{P}(\text{B1} \wedge \neg \text{ B2} \wedge \text{ B3}) \\
c_0.c_2.(1 \Leftrightarrow c_6) & = \mathbb{P}(\text{B1} \wedge \text{ B2} \wedge \neg \text{ B3}) \\
c_0.c_2.c_6 & = \mathbb{P}(\text{B1} \wedge \text{ B2} \wedge \text{ B3})
\end{array}\right.$$

S'il l'on considère le profil opérationnel suivant,



0.2	0.4	0.3		0.1			

on obtient :

$$\left\{\begin{array}{l}(1 \Leftrightarrow c_0).(1 \Leftrightarrow c_1).(1 \Leftrightarrow c_3) = 0.2\\(1 \Leftrightarrow c_0).(1 \Leftrightarrow c_1).c_3 = 0.4\\(1 \Leftrightarrow c_0).c1.(1 \Leftrightarrow c_4) = 0.3\\(1 \Leftrightarrow c_0).c1.c_4 = 0\\c_0.(1 \Leftrightarrow c_2).(1 \Leftrightarrow c_5) = 0.1\\c_0.(1 \Leftrightarrow c_2).c_5 = 0\\c_0.c_2.(1 \Leftrightarrow c_6) = 0\\c_0.c_2.c_6 = 0\end{array}\right. \Rightarrow \left\{\begin{array}{l}c_0 = 0.1\\c_1 = 1/3\\c_2 = 0\\c_3 = 2/3\\c_4 = 0\\c_5 = 0\\c_6 = \text{-}\end{array}\right.$$

On note que le système équationnel ne définit pas de valeur unique ou significative pour c_6. Comme on peut le constater figure 6.1, c_6 est associé à un sous-BDD pour lequel il n'y a aucun vecteur d'entrée valide.

On propose une seconde méthode pour établir la valeur des coefficients c_i. Intuitivement, on utilise un algorithme d'étiquetage similaire à la première méthode de génération. Ici, on utilise un couple de réels (f_0, f_1) qui représente la proportion de chemins valides (plutôt que le nombre) si la variable est instanciée à faux ou à vrai. L'algorithme de calcul de l'étiquette (donné page 34) permet d'établir f_0 et f_1 et $c_i = f_1/(f_0 + f1)$. La figure 6.2 illustre le résultat après l'application de l'algorithme d'étiquetage.

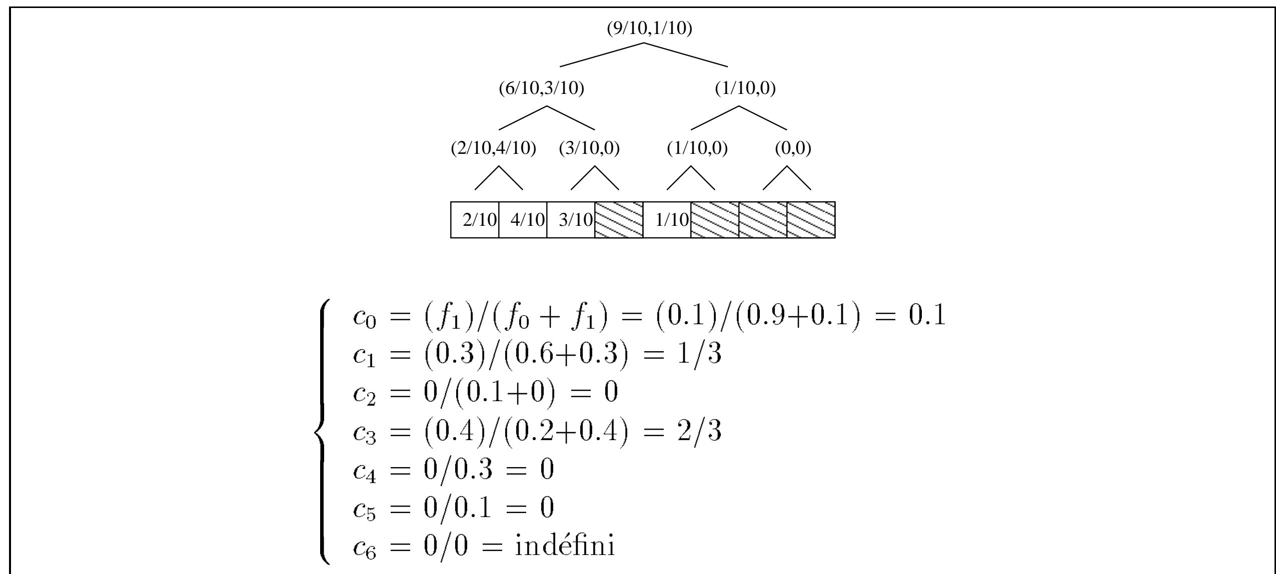


FIG. **6.2** – Illustration de l'algorithme de calcul des coefficients c_i

Après application de l'algorithme d'étiquetage, on peut construire la liste suivante :
$\mathcal{K}$ = [⟨ B1, 0.1, true ⟩, ⟨ B2, 1/3, not B1 ⟩, ⟨ B2, 0, B1 ⟩, ⟨ B3, 2/3, not B1 and not B2 ⟩, ⟨ B3, 0, not B1 and B2 ⟩, ⟨ B3, 0, B1 and not B2 ⟩, ⟨ B3, 0, B1 and B2 ⟩]

Cette liste peut être simplifiée en examinant les contraintes d'environnement. Tout d'abord, il est inutile de conserver la probabilité conditionnelle associée à c_6 pour les raisons invoquées ci-dessus (aucun vecteur d'entrée valide).

De même, on peut éliminer les triplets associés à c_2 et c_5. En effet, l'environnement impose B2=*faux* et B3=*faux* lorsque B1=*vrai*. Or, lorsque l'environnement impose une valeur, la liste des probabilités conditionnelles n'est pas consultée. Le même raisonnement peut être fait pour c_4.

Enfin, les conditions des triplets associés à c_1 et c_3 peuvent être simplifiées. Prenons l'exemple du triplet associé à c_1. Compte tenu des contraintes d'environnement et des remarques précédentes, le seul moment où l'on évalue une probabilité conditionnelle pour B2 est lorsque B1 a été affectée à faux. Or, dans ce cas, la condition "not B1" est vraie. On peut donc réduire la condition associée à c_1 à *true*.

L'ensemble des simplifications permettent d'obtenir la liste suivante :

$\mathcal{K}' = [\langle$ B1, 0.1, *true* $\rangle$, $\langle$ B2, 1/3, *true* $\rangle$, $\langle$ B3, 2/3, *true* $\rangle]$

⌨∼

6.2.2 Exemple avec un profil opérationnel multiple

Dans [94], Whittaker a remarqué combien il était utile de définir une distribution multiple pour chacun des modes d'utilisation du programme. Nous allons montrer comment on peut établir un ensemble de probabilités conditionnelles pour définir une distribution multiple.

∼⌨ Prenons un nouvel exemple pour le convertisseur de clics. Nous voulons faire varier la probabilité de cliquer selon qu'un clic logique est en cours ou non. Par exemple, on souhaite exprimer le fait que "lorsque l'on vient de cliquer sur un bouton de la souris, la probabilité de cliquer sur ce bouton une nouvelle fois (et faire un double ou un triple clic) est plus grande que la probabilité de cliquer sur un autre bouton".

Soit M une variable mémorisant le dernier bouton cliqué pendant la durée de construction d'un clic logique. M est égale à zéro lorsque le programme n'est pas dans une période de construction de clic logique. Ci-dessous, on propose un profil opérationnel multiple, en fonction de la valeur de M (au top précédent).

	pre (M=1)	pre (M=2)	pre (M=3)	pre (M=0)
$\mathbb{P}$(B1=*true*)	0.6	0.1	0.1	0.2
$\mathbb{P}$(B2=*true*)	0.1	0.6	0.1	0.2
$\mathbb{P}$(B3=*true*)	0.1	0.1	0.6	0.2
$\mathbb{P}$(Aucun)	0.2	0.2	0.2	0.4

Pour élaboration de la liste de probabilités conditionnelles, on procède en deux temps. Tout d'abord, on applique l'algorithme de calcul des coefficients, basé sur l'étiquetage de l'arbre de Shannon des entrées. L'étiquetage obtenu est présenté dans la figure 6.3. Dans cette figure, les triangles symbolisent l'arbre de Shannon supérieur et caractérisent les sous-arbres des entrées. Dans ces triangles, seule la valeur de M apparaît, puisqu'elle seule intervient dans la caractérisation des sous arbres.

Ensuite, on cherche à construire la liste de probabilités conditionnelles. Pour chaque coefficient, on établit la condition le caractérisant dans le sous-arbre des entrées. Puis, on l'étend pour prendre en compte la condition sur M. On obtient alors,

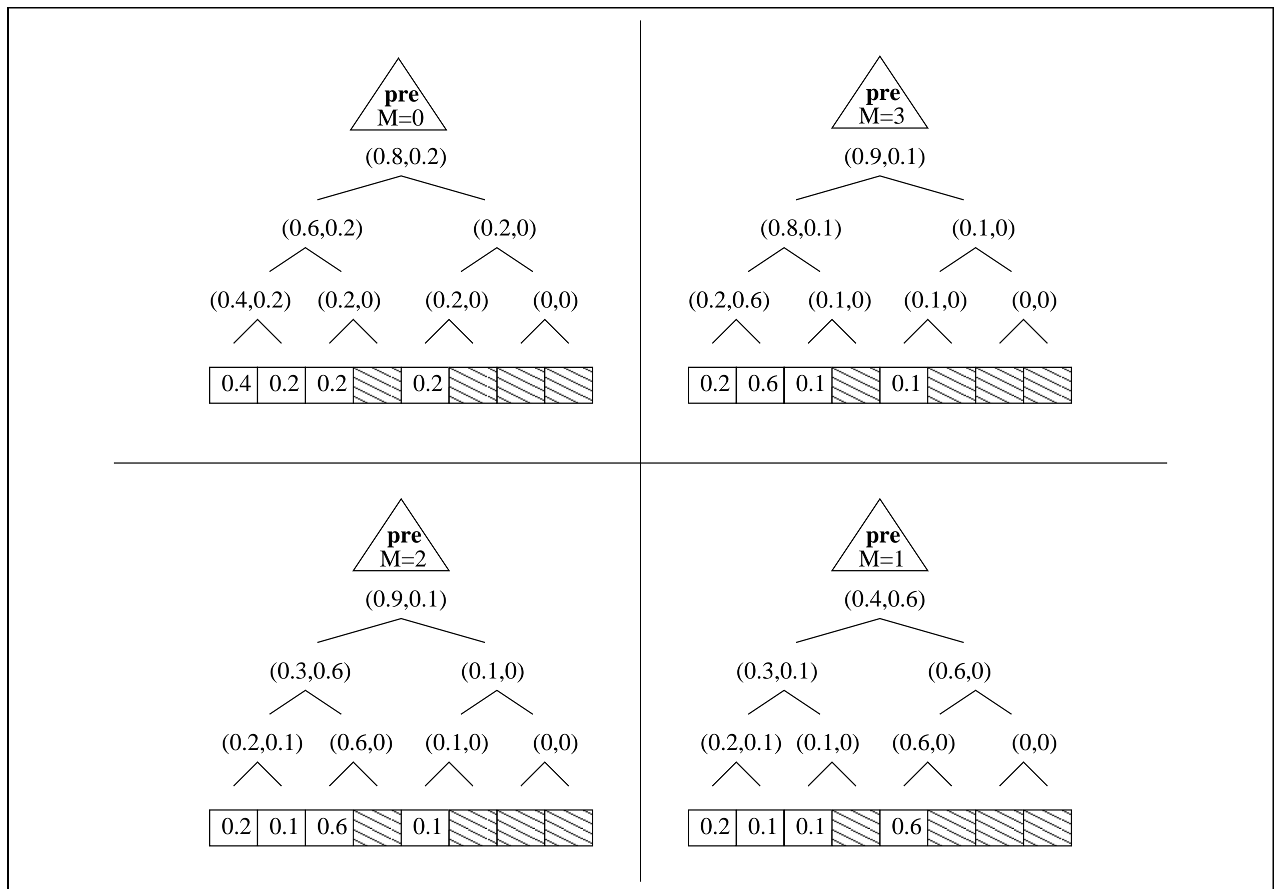


FIG. **6.3** – Profil opérationnel multiple

[⟨ B1, 0.2, pre (M=0) ⟩, ⟨ B1, 0.6, pre (M=1) ⟩, ⟨ B1, 0.1, pre (M=2) or pre (M=3) ⟩,
⟨ B2, 1/4, not B1 and pre (M=0) ⟩, ⟨ B2, 1/4, not B1 and pre (M=1) ⟩,
⟨ B2, 2/3, not B1 and pre (M=2) ⟩, ⟨ B2, 1/9, not B1 and pre (M=3) ⟩,
⟨ B3, 1/3, not B1 and not B2 and not pre (M=3) ⟩, ⟨ B3, 3/4, not B1 and not B2 and pre (M=3)⟩]

On peut simplifier les conditions des triplets de la liste ci-dessus, de la même manière que précédemment. On obtient alors :
$\mathcal{L}'$ = [⟨ B1, 0.2, pre (M=0) ⟩, ⟨ B1, 0.6, pre (M=1) ⟩, ⟨ B1, 0.1, pre ((M=2) or (M=3)) ⟩,
⟨ B2, 1/4, pre (M=0) ⟩, ⟨ B2, 1/4, pre (M=1) ⟩, ⟨ B2, 2/3, pre (M=2) ⟩,
⟨ B2, 1/9, pre (M=3) ⟩, ⟨ B3, 1/3, not pre (M=3) ⟩, ⟨ B3, 3/4, pre (M=3)⟩]

6.2.3 Algorithme général

Algorithme de construction des probabilités conditionnelles :
  Construire l'arbre de Shannon correspondant aux contraintes d'environnement
  Etablir le profil opérationnel (simple ou multiple)
  Instancier les feuilles de l'arbre avec la probabilité de chaque vecteur

Appliquer l'algorithme d'étiquetage *(cf. page 35)*
Pour chaque noeud significatif, constuire un triplet $\langle x, p, f \rangle$
(i.e. ici, un noeud significatif est un noeud permettant 2 instanciations de la variable)
Déterminer la variable associée au noeud : on obtient x
Calculer le coefficient à partir de l'étiquette, on obtient p
Etablir la condition associée : on obtient f
Eventuellement, effectuer des simplifications de conditions

6.3 De probabilités conditionnelles à un profil opérationnel

La possibilité d'associer des probabilités conditionnelles aux variables d'entrées est bien utile pour un utilisateur qui ne veut que partiellement décrire un profil opérationnel ou qui doit décrire un profil opérationnel implicite. Cette possibilité est aussi utile pour celui qui préfère manipuler les variables séparément.

La visualisation du profil opérationnel obtenu à partir de probabilités conditionnelles est alors indispensable pour celui qui souhaite s'assurer que ce qu'il a défini correspond à ce qu'il avait espéré.

Pour répondre à ce besoin, nous avons construit un programme d'affichage d'un profil opérationnel défini par une liste de probabilités conditionnelles. Le principe de ce programme consiste en un parcours BDD d'environnement, pendant lequel la probabilité de chaque instanciation est calculée, en se basant sur l'étiquetage de base et la liste de probabilités conditionnelles. Lorsqu'une feuille valide est atteinte, on affiche l'instanciation des variables d'état, du vecteur d'entrée et la probabilité d'instanciation associée à ce dernier.

Le principe de l'algorithme ci-dessus est simple mais ne peut être mis en pratique directement. En effet, si la taille du BDD d'environnement est très importante, l'affichage naïf de chaque vecteur peut s'avérer illisible. Par exemple, le programme de l'ascenseur comporte 26 variables d'entrées dont 10 au moins ne sont pas contraintes. Ainsi, pour chaque état de l'environnement, il faudrait afficher au moins $2^{10} = 1024$ vecteurs correspondant aux variables non contraintes. La quantité d'information proposée à l'utilisateur dans de telles situation est bien trop importante pour être pertinente.

Nous avons donc recherché un affichage concis. Cette concision réside en deux points. Tout d'abord, pour chaque état, on regroupe les vecteurs d'entrée associés. Le vecteur d'état est alors affiché une seule fois pour l'ensemble des vecteurs.

Ensuite, nous avons appliqué le principe de simplification utilisés pour les BDD. Dans un BDD, les nœuds redondants sont éliminés, i.e. les variables qui n'influencent pas la définition de la fonction booléenne, ne figurent pas explicitement dans le BDD. On dit que ces variables sont abstraites. De la même manière, on choisit de présenter les vecteurs de façon synthétique. Les variables d'états abstraites n'apparaissent pas pour l'affichage des conditions et les variables d'entrées abstraites sont figurées par une étoile ('*') lors de l'affichage du vecteur

d'entrée.

Remarquons qu'il n'est pas toujours possible d'effectuer ces simplifications. En effet, si une variable est abstraite, c'est qu'elle peut prendre indifféremment la valeur vraie ou fausse. Or, pour évaluer une condition portant sur cette variable, il faut fixer une valeur à cette variable. C'est pourquoi, on a choisi de n'abstraire que les variables non utilisées pour exprimer les conditions des triplets.

~≣ Cet algorithme a été implanté dans Lutess. Ici, on illustre son application sur l'exemple du convertisseur de clics.

▹ Tout d'abord, on observe le résultat de l'algorithme d'affichage sans probabilités conditionnelles, i.e. on s'attend à obtenir une distribution équiprobable des vecteurs d'entrées. La contrainte d'environnement "au plus un bouton" définit 4 vecteurs d'entrées valides, et donc, comme attendu, la table 6.1 donne le profil opérationnel où chaque vecteur a une probabilité de 0.25.

(B1B2B3)	$p(i)$
100	0.25
010	0.25
00*	0.25

Tab. 6.1 – *Profil opérationnel calculé par Lutess sans probabilités conditionnelles*

▹ Avec le profil opérationnel inconditionnel donné précédemment (§ 6.2.1), l'application de l'algorithme de traduction des profils opérationnels nous a permis d'établir la liste de probabilités conditionnelles suivantes: $\mathcal{K}' = [\langle B1, 0.1, true\rangle, \langle B2, 1/3, true\rangle, \langle B2, 2/3, true\rangle]$

La table 6.2 donne le résultat obtenu par Lutess conformes au résultat attendu.

(B1 B2 B3)	$p(i)$
100	0.1
010	0.3
001	0.4
000	0.2

Tab. 6.2 – *Profil opérationnel calculé par Lutess avec* $\mathcal{K}'$

▹ Enfin, nous avons utilisé le profil opérationnel multiple défini section 6.2.2 et de sa traduction en probabilités conditionnelles ($\mathcal{L}'$). Pour traduire les conditions portant sur M, nous avons introduit 3 variables d'état, qui correspondent respectivement à (pre M=1), (pre M=2) et (pre M=3).

La table 6.3 reproduit les résultats fournis pas Lutess. On pouvait s'attendre à ce que Lutess explore seulement 4 états de l'environnement. En fait, Lutess explore 8 états, qui correspondent aux différentes instanciations des variables d'états. Intuitivement, certains de ces états ne seront jamais atteints, puisque a priori, les conditions (pre M=1), (pre M=2) et (pre M=3) sont exclusives, Mais Lutess ne peut le savoir.

Si l'on observe les 4 états que l'on avait initialement prévus, on retrouve exactement le profil opérationnel multiple initialement défini.

condition	(B1B2B3)	$p(i)$
sv0 and sv1 and sv2		
	100	0.6
	010	0.1
	001	0.225
	000	0.075
sv0 and sv1 and not sv2		
	100	0.6
	010	0.1
	001	0.225
	000	0.075
sv0 and not sv1 and sv2		
	100	0.1
	010	0.6
	001	0.225
	000	0.075
sv0 and not sv1 and not sv2		
	100	0.1
	010	0.1
	001	0.6
	000	0.2
not sv0 and sv1 and sv2		
	100	0.6
	010	0.1
	001	0.1
	000	0.2
not sv0 and sv1 and not sv2		
	100	0.6
	010	0.1
	001	0.1
	000	0.2
not sv0 and not sv1 and sv2		
	100	0.1
	010	0.6
	001	0.1
	000	0.2
not sv0 and not sv1 and not sv2		
	100	0.2
	010	0.2
	001	0.2
	000	0.4

sv0 = ((pre M) = 3);
sv1 = ((pre M) = 1);
sv2 = ((pre M) = 2);

Tab. 6.3 – *Profil opérationnel calculé par Lutess avec* $\mathcal{L}'$

Chapitre 7

Validation de la méthode de génération basée sur des probabilités conditionnelles

Dans le chapitre précédent, nous avons proposé une méthode de génération de Lutess , basée sur les probabilités conditionnelles. Nous avons alors présenté une analyse de l'algorithme de génération qui nous a permis de nous assurer que les aspects fonctionnels attendus étaient bien remplis.

Dans ce chapitre, on propose de valider, de manière expérimentale, le fait que l'algorithme fournit des données respectant les répartitions statistiques attendues. Pour cela, on s'appuie sur les méthodes utilisées dans le chapitre 4.

7.1 Algorithme de génération

7.1.1 Utilisation de probabilités inconditionnelles

Pour montrer que Lutess génère les données de tests conformément à une distribution inconditionnelle définie par l'utilisateur, on reprend l'exemple de l'ascenseur et on produit une liste de triplets incluant différentes valeurs de probabilités.
On rappelle que l'interface du programme de l'ascenseur est :

```
testnode Environment(   allumer_interieur : bool^NBETAGES;
                        ouvrir_portes, fermer_portes : bool;
                        monter, descendre, stop : bool;
                        allumer_haut, allumer_bas : bool^NBETAGES)
returns(   portes_ouvertes, portes_fermees : bool;
           etage, dessus, dessous : bool^NBETAGES;
           requete : bool^NBETAGES;
           appel_montee, appel_descente : bool^NBETAGES );
```

Séquence →	S1	S2	S3	S4	S5	résultats
↓ Nb d'occurences de ...	11111	47347	92375	48603	10256	attendus
appel_montée[0] (X_0)	9024	8998	8993	9037	9078	9000
appel_montée[1] (X_1)	7058	7087	7113	7049	7081	7000
appel_montée[2] (X_2)	5041	5061	5111	5051	5035	5000
appel_descente[1] (X_3)	4982	4969	5022	5180	4941	5000
appel_descente[2] (X_4)	3141	3021	3038	3019	3117	3000
appel_descente[3] (X_5)	958	1015	1009	1087	1002	1000
requête[0] (X_6)	835	777	797	791	791	800
requête[1] (X_7)	621	586	642	630	604	600
requête[2] (X_8)	317	329	329	318	319	300
requête[3] (X_9)	111	114	131	114	117	100

TAB. 7.1 – *Effectifs observés pour 5 séquences de longueur 10 000 pas générées avec* $\mathcal{Q}$

On choisit la liste de probabilités suivantes :
$\mathcal{Q}$ = [⟨*appel_montée[0]*, 0.9, *true*⟩, ⟨*appel_montée[1]*, 0.7, *true*⟩,
⟨*appel_descente[1]*, 0.5, *true*⟩, ⟨*appel_descente[2]*, 0.3, *true*⟩, ⟨*appel_descente[1]*, 0.1, *true*⟩,
⟨*requête[0]*,0.08, *true*⟩, ⟨*requête[1]*,0.06, *true*⟩, ⟨*requête[2]*,0.03, *true*⟩, ⟨*requête[3]*,0.01, *true*⟩]

Nous avons utilisé Lutess pour générer cinq séquences de 10 000 pas de tests. La table 7.1 dresse un décompte des occurrences de la valeur *vrai* de chacune des variables d'entrée figurant dans $\mathcal{Q}$. Pour chaque variable et chaque séquence, nous avons effectué un test du χ^2.

- la population : X_i une variable booléenne,
- l'échantillon : 10 000 valeurs tirées,
- 2 classes : $c_1 = vrai$ et $c_2 = faux$,
- nombre de degré de liberté : v =2-1=1,
- hypothèse $\mathcal{H}_0$: X_i a une probabilité fixée par les probabilités conditionnelles,
- niveau de confiance : 99%.

Les résultats sont donnés dans la table 7.2. Sur les 50 tests de χ^2 effectués, 45 permettent d'accepter $\mathcal{H}_0$ avec un niveau de confiance égal à 99%.

Les 5 cas problématiques sont *appel_descente[2]* pour la séquence 1, *requête[3]* pour la séquence 3, *appel_descente[1]* et *appel_descente[3]* pour la séquence 4, et *appel_montée[0]* pour la séquence 5.
En augmentant la taille de l'échantillon observé pour les séquences 1, 3, 4 et 5, on a obtenu respectivement :

- $\chi^2 = 0.08$ pour *appel_descente[2]* pour S1 sur 20 000 pas,
- $\chi^2 = 7.28$ pour *requête[3]* pour S3 sur 50 000 pas,
- $\chi^2 = 2.64$ pour *appel_descente[1]* et $\chi^2 = 4.48$ pour *appel_descente[3]* pour S4 sur 50 000 pas, et

	S1	S2	S3	S4	S5
X_0	0.640	0.004	0.054	1.521	6.760
X_1	1.601	3.604	6.080	1.143	3.124
X_2	0.672	1.488	4.928	1.040	0.490
X_3	0.129	0.384	0.193	12.960	1.392
X_4	9.467	0.210	0.687	0.171	6.518
X_5	1.960	0.250	0.090	8.410	0.004
X_6	1.664	0.718	0.012	0.110	0.110
X_7	0.781	0.347	3.127	1.595	0.021
X_8	0.993	2.890	2.890	1.113	1.240
X_9	1.222	1.979	9.707	1.222	2.919

TAB. 7.2 – χ^2 *pour les variables* X_i *des séquences S1 à S5*

– $\chi^2 = 0.18$ pour *requête[0]* pour S5 sur 20 000 pas.

On observe une diminution du χ^2, mais qui n'est pas suffisante pour accepter $\mathcal{H}_0$ pour *requête[3]* de S3.

On rappelle que choisir un niveau de confiance de 99% signifie que si l'on analyse 100 échantillons, alors un des échantillons peut conduire à rejeter $\mathcal{H}_0$ alors qu'elle est vraie. C'est pourquoi ici, bien qu'il existe un échantillon ne permettant pas d'accepter $\mathcal{H}_0$, on accepte “globalement” $\mathcal{H}_0$ en se basant sur les 49 autres tests.

7.1.2 Utilisation de probabilités conditionnelles

Le but de ce paragraphe est de s'assurer que Lutess génère des données de tests conformément à une distribution multiple définie par l'utilisateur. On étudie deux exemples : l'ascenseur et le convertisseur de clics. Dans le premier cas, on observe directement la probabilité effective des données d'entrées et on la compare à celle attendue. Dans le second cas, on observe la probabilité effective des données de sorties.

Exemple de l'ascenseur

Soit $\mathcal{R}$ la liste de probabilités conditionnelles suivantes :

⟨*appel_montée[0]*, 0.9, pre *ouvrir_portes*⟩, (pour X_0)
⟨*appel_montée[1]*, 0.5, pre *ouvrir_portes*⟩, (pour X_1)
⟨*appel_montée[2]*, 0.3, pre *ouvrir_portes*⟩, (pour X_2)
⟨*appel_descente[1]*, 0.5, *étage[1]* ⟩, (pour X_3)
⟨*appel_descente[2]*, 0.3, *étage[2]* ⟩, (pour X_4)
⟨*appel_descente[3]*, 0.1, *étage[3]* ⟩, (pour X_5)
⟨*requête[0]*, 0.7, *étage[0]* and pre *ouvrir_portes*⟩, (pour X_6)
⟨*requête[1]*, 0.5, *étage[1]* and pre *ouvrir_portes*⟩, (pour X_7)
⟨*requête[2]*, 0.3, *étage[2]* and pre *ouvrir_portes*⟩, (pour X_8)
⟨*requête[3]*, 0.1, *étage[3]* and pre *ouvrir_portes*⟩ (pour X_9)

Cette liste a été établie afin d'étudier plusieurs types de probabilités conditionnelles. Les conditions associées aux signaux *appel_montée[i]* sont exprimées en fonction d'une sortie du programme à l'instant précédent. Celles associées aux signaux *appel_descente[i]* dépendent d'une entrée du programme. Les conditions portant sur les probabilités des signaux *requête[i]* sont exprimées en fonction d'une entrée et d'une sortie du programme.

Nous avons utilisé Lutess pour générer cinq séquences de 50 000 pas de tests. La table 7.3 dresse un décompte des occurrences de la valeur *vrai* de chacune des variables d'entrée figurant dans $\mathcal{R}$. Pour chaque variable et chaque test, nous avons effectué un test du χ^2.

- la population : X_i une variable booléenne,
- l'échantillon : fonction de la variable et de la séquence,
- 2 classes : $c_1 = vrai$ et $c_2 = faux$,
- nombre de degré de liberté : v =2-1=1,
- hypothèse $\mathcal{H}_0$: X_i a une probabilité fixée par les probabilités conditionnelles,
- niveau de confiance : 99%.

Les résultats sont donnés dans la table 7.4. Les 50 tests effectués permettent d'accepter $\mathcal{H}_0$ avec un niveau de confiance égal à 99%.

Le convertisseur de clics

On reprend l'exemple du programme de conversion de clics. On rappelle qu'il existe un entier T définissant la vitesse du triple clic, exprimé en nombre de top d'horloge. Un triple clic logique doit être détecté lorsqu'un même bouton est cliqué trois fois de suite en moins de T tops. On étudie le cas où $T = 4$. Le délai entre un clic physique et le clic logique correspondant est appelé *phase de construction d'un clic.*

Dans cet exemple, on reprend la distribution définie fig. 6.3, page 83. Afin de rendre l'observation plus facile, on suppose que le programme de conversion est correct.

On cherche à analyser la distribution des clics logiques. Cette distribution peut être calculée en utilisant la théorie des chaînes de Markov [6, 76]. La probabilité de cliquer sur un bouton dépend de l'histoire du programme. Trois informations sont nécessaires : il faut savoir

- si un clic logique est en construction ou non (M$\neq$0 ou M=0);

Séquence → ↓ Nb d'occurences de ...	S1 11111	S2 47347	S3 92375	S4 48603	S5 10256
pre OV	19506	19532	19464	19516	19491
étage[0]	8121	8296	8358	8146	8250
étage[1]	17113	16529	16620	16825	16618
étage[2]	14416	14744	14382	14572	14722
étage[3]	4856	4963	5105	4974	4902
pre OP and *étage[0]*	3603	3692	3718	3616	3665
pre OP and *étage[1]*	7641	7353	7387	7499	7391
pre OP and *étage[2]*	6292	6461	6268	6371	6443
pre OP and *étage[3]*	1970	2026	2091	2030	1992
M[0] and pre *OV* (X_0)	17567	17605	17491	17632	17465
M[1] and pre *OV* (X_1)	9852	9811	9757	9880	9804
M[2] and pre *OV* (X_2)	5855	5855	5888	5928	5925
D[1] and *étage[1]* (X_3)	8614	8285	8331	8485	8335
D[2] and *étage[2]* (X_4)	4338	4413	4440	4480	4478
D[3] and *étage[3]* (X_5)	451	524	541	490	500
R[0] and pre *OP* and *étage[0]* (X_6)	2535	2610	2600	2545	2593
R[1] and pre *OP* and *étage[1]* (X_7)	2309	2190	2172	2264	2212
R[2] and pre *OP* and *étage[2]* (X_8)	630	672	675	661	651
R[3] and pre *OP* and *étage[3]* (X_9)	975	984	1059	1058	1013

OP : ouvrir_portes, M : appel_montée, D : appel_descente, R : requête

TAB. 7.3 – *Effectifs observés pour les 5 séquences de 50 000 pas générées avec* $\mathcal{R}$

	S1	S2	S3	S4	S5
X_0	0.082	0.094	0.403	2.601	3.371
X_1	2.009	0.414	0.128	3.050	0.702
X_2	0.003	0.005	0.582	1.307	1.474
X_3	0.786	0.101	0.106	1.249	0.162
X_4	0.064	0.033	5.206	3.839	1.219
X_5	2.648	1.717	2.024	0.122	0.217
X_6	0.223	0.845	0.008	0.250	0.982
X_7	0.180	0.163	1.253	0.129	0.018
X_8	0.001	1.153	0.018	0.996	0.077
X_9	0.203	1.660	0.348	3.643	0.580

TAB. 7.4 – χ^2 *obtenus pour les variables* X_i *des séquences S1 à S5*

- le nombre de tops écoulés depuis le début de la phase de construction du clic logique courant (t=1, t=2, t=3);

- le nombre de clics physiques déjà effectués pendant la phase de construction courant.

La distribution définie fig. 6.3. est symétrique : le numéro des boutons n'est pas significatif. Ainsi, on ne différencie pas les cas où M$\neq$0. La figure 7.1 décrit la chaîne de Markov correspondant au profil opérationnel étudié. Une méthode de construction de cette chaîne est donnée dans [94] et utilisée dans [26].

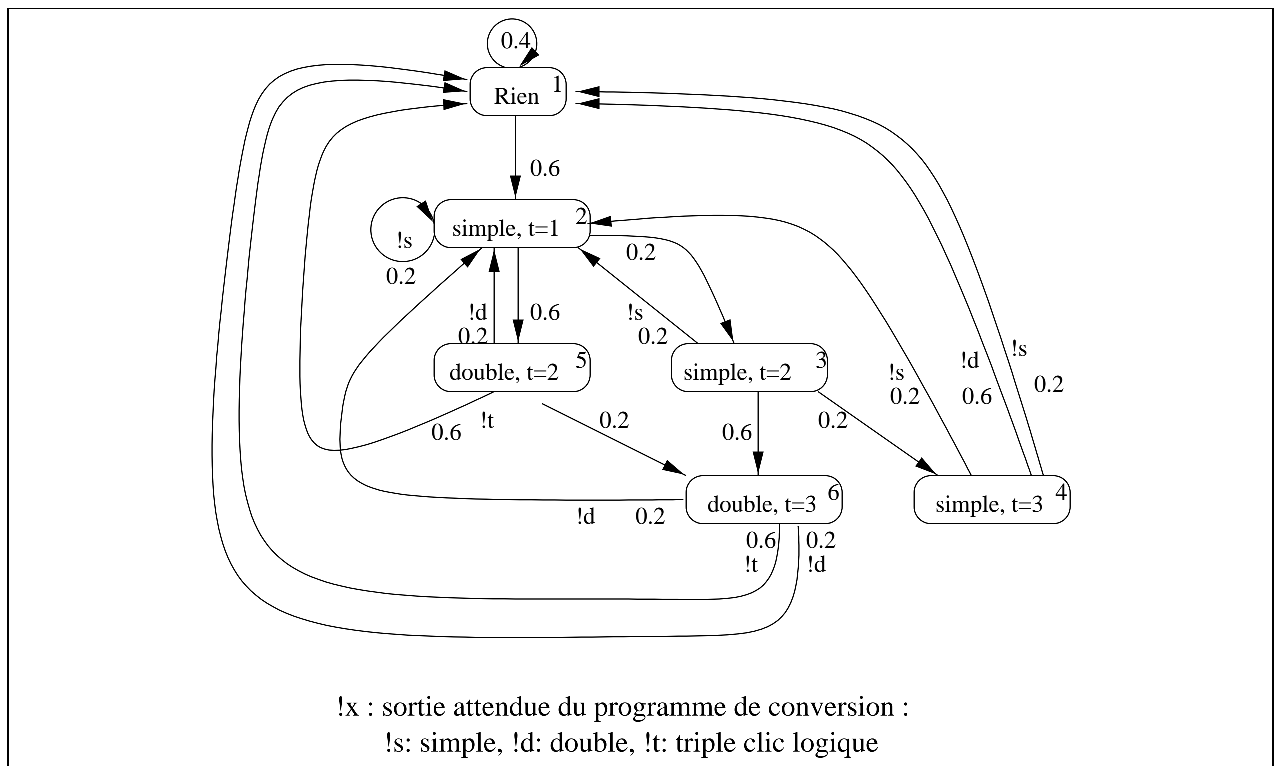


FIG. **7.1** – Chaîne de Markov obtenue pour $T = 4$

La chaîne de Markov de la figure 7.1 est homogène car la probabilité de transition d'un état à un autre ne dépend pas du temps. Par conséquent, elle peut être représentée par la matrice de transition $\mathcal{T}$ ci-dessous. Cette matrice indique la probabilité d'effectuer une transition d'un état à un autre en un pas de test.

$$\mathcal{T} = \begin{pmatrix} 0.4 & 0.6 & 0 & 0 & 0 & 0 \\ 0 & 0.2 & 0.2 & 0 & 0.6 & 0 \\ 0 & 0.2 & 0 & 0.2 & 0 & 0.6 \\ 0.8 & 0.2 & 0 & 0 & 0 & 0 \\ 0.6 & 0.2 & 0 & 0 & 0 & 0.2 \\ 0.8 & 0.2 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	valeurs attendues	S1 (11111)	S2 (47347)	S3 (92374)	S4 (48603)	S5 (10256)
SIMPLE	840	804	840	767	857	819
DOUBLE	780	818	809	755	811	778
TRIPLE	1 650	1686	1642	1708	1653	1663
RIEN	6 730	6692	1642	6770	6679	6740
Résultats des tests du χ^2		3.63	0.76	9.56	1.55	0.64

TAB. 7.5 – *Effectifs observés et χ^2 pour S1 à S5*

Comme cette chaîne est apériodique[1] et irréductible[2], elle va converger vers un état stationnaire (au sens des probabilités) [6, 76]. Soit $\pi = (\pi_1, \pi_2, ..., \pi_6)$ la distribution statistique des états du système. π_i représente la probabilité d'être dans l'état i du système. π est l'unique solution du système linéaire $\pi.\mathcal{T} = \pi$, avec $\sum_{i=1}^{6} \pi_i = 1$.

$$\pi = (0.319,\ 0.328,\ 0.066,\ 0.013,\ 0.197,\ 0.077)$$

A partir de π, on peut calculer la probabilité de chaque clic logique de la façon suivante :

$$\begin{cases} \mathbb{P}(\text{SIMPLE}) = 0.2^*\mathbb{P}(\text{state}=2) + 0.2^*\mathbb{P}(\text{state}=3) + 2^*0.2^*\mathbb{P}(\text{state}=4) = \\ \qquad 0.2^*\pi_2 + 0.2^*\pi_3 + 2^*0.2^*\pi_4 = 0.084 \\ \mathbb{P}(\text{DOUBLE}) = 0.6^*\mathbb{P}(\text{state}=4) + 0.2^*\mathbb{P}(\text{state}=5) + 2^*0.2^*\mathbb{P}(\text{state}=6) = \\ \qquad 0.6^*\pi_4 + + 0.2^*\pi_5 + 2^*0.2^*\pi_6 = 0.078 \\ \mathbb{P}(\text{TRIPLE}) = 0.6^*\mathbb{P}(\text{state}=5) + 0.6^*\mathbb{P}(\text{state}=6) = \\ \qquad 0.6^*\pi_5 + 0.6^*\pi_6 = 0.165 \end{cases}$$

Nous avons produit 5 séquences 10 000 pas. La table 7.5 indique la distribution des clics logiques attendue et celle obtenue pour les 5 séquences. Nous avons utilisé un test du χ^2 (cf. chapitre 4) pour comparer ces résultats.

- la population : X_i le type de clic logique produit par le programme de conversion,
- l'échantillon : 10 000 valeurs tirées,
- 4 classes : c_0 pas de clic logique, c_1 simple clic, c_2 double clic et c_3 triple clic,
- nombre de degré de liberté : v =4-1=3,
- hypothèse $\mathcal{H}_0$: *la distribution des clics logiques correspond à celle extraite de la chaîne de Markov donnée figure 7.1,*
- niveau de confiance : 99% ($\chi^2 \leq 11.34$).

La table 7.5 indique les résultats des tests de χ^2. L'hypothèse $\mathcal{H}_0$ est acceptée avec un niveau de confiance égal à 99%.

1. Une chaîne est périodique s'il existe un entier $d > 1$ tel que tout chemin d'un état i à un état j est de longueur $k * d$, où k est un entier. Elle est apériodique sinon.
2. Une chaîne est irréductible s'il existe un chemin de tous les états vers tous les autres.

po	*pf*	*étage[0..3]*	*dessus[0..3]*	*dessous[0..3]*	*R[0..3]*	*M[0..3]*	*D[0..3]*	probabilité
1	0	1000	0000	0000	****	***0	0***	0.000244141
1	0	0100	0000	0000	****	***0	0***	0.000244141
1	0	0010	0000	0000	****	***0	0***	0.000244141
1	0	0001	0000	0000	****	***0	0***	0.000244141

'0' et '1' représent les booléens 'faux' et 'vrai'.

po et *pf* représentent les entrées *portes_ouvertes* et *portes_fermées*.

R, *M* et *D* représentent *requête*, *appel_montée* et *appel_descente*.

TAB. 7.6 – *Profil opérationnel de l'état initial avec la méthode de génération aléatoire équiprobable.*

7.2 De probabilités conditionnelles à un profil opérationnel

Dans cette partie, nous examinons les résultats de l'algorithme permettant de traduire les probabilités conditionnelles en profils opérationnels (présenté section 6.3). Pour ce faire, on reprend l'exemple de l'ascenseur et on étudie plusieurs distributions : la distribution équiprobable de la méthode de base, une distribution basée sur des probabilités inconditionnelles et une distribution basée sur des probabilités conditionnelles. Pour chaque cas, on étudie l'état initial du profil opérationnel calculé par l'algorithme de traduction.

Lorsque NBETAGES = 4, le générateur de test doit produire des valeurs pour les 26 variables d'entrées du programme, qui respectent les contraintes d'environnement. Examinons l'état initial de l'environnement. Les contraintes d'environnement indiquent que dans cet état :

- *appel_montée[3]* et *appel_descente[0]* sont toujours faux.
- Il n'y a pas de contraintes sur les autres boutons de l'interface homme-machine : *requête[0..3]*, *appel_montée[0..2]*, et *appel_descente[1..3]*.
- Dans l'état initial, les portes sont ouvertes,
- et la cabine se trouve à l'un des étages.
- Les variables *dessus[i]* et *dessous[i]* sont fausses (l'ascenseur ne peut être qu'à un seul endroit à la fois).

Méthode de génération de base

On observe tout d'abord le profil opérationnel pour l'état initial, dans le cas de la méthode de génération de base. Le résultat du programme de calcul du profil opérationnel est donné par la table 7.6. Chaque ligne de ce profil correspond à une position particulière de la cabine à l'instant initial. Pour chacun des cas, il y a 2^{10} vecteurs d'entrée possibles. Chaque vecteur a la probabilité 0.000244141 d'être choisi. La somme des probabilités des vecteurs est égale à 1 à 10^{-5} près. ($0.000244141 * 2^{10} * 4 = 1.000001536$ et $0.00024414 * 2^{10} * 4 = 0.99999744$).

Probabilités inconditionnelles

Soit une liste de probabilités inconditionnelles $\mathcal{L}$ très simple, consistant en la modification de la probabilité des signaux *appel_montée[i]*:

$$\langle\ \textit{appel_montée[0]},\ 0.01,\ \textit{true}\ \rangle,$$
$$\langle\ \textit{appel_montée[1]},\ 0.01,\ \textit{true}\ \rangle,$$
$$\langle\ \textit{appel_montée[2]},\ 0.01,\ \textit{true}\ \rangle$$

On observe le profil opérationnel obtenu pour l'état initial (table 7.7). On peut constater que les probabilités inconditionnelles n'ont pas modifié le domaine des entrées.

On se propose de vérifier que la probabilité de chaque vecteur d'entrée correspond aux probabilités définies ci-dessus. Pour chaque étage, on peut énumérer les combinaisons possibles des valeurs des signaux *appel_montée[i]*. Il y a 8 combinaisons possibles car *appel_montée[3]* est toujours faux. On rappelle que ces signaux sont indépendants.

$$\begin{cases}
\mathbb{P}(1110) = \mathbb{P}(\textit{M[0]}\ \mathsf{and}\ \textit{M[1]}\ \mathsf{and}\ \textit{M[2]}) & = 0.01*0.01*0.01 = 0.000001 \\
\mathbb{P}(1100) = \mathbb{P}(\textit{M[0]}\ \mathsf{and}\ \textit{M[1]}\ \mathsf{and\ not}\ \textit{M[2]}) & = 0.01*0.01*0.99 = 0.000099 \\
\mathbb{P}(1010) = \mathbb{P}(\textit{M[0]}\ \mathsf{and\ not}\ \textit{M[1]}\ \mathsf{and}\ \textit{M[2]}) & = 0.01*0.99*0.01 = 0.000099 \\
\mathbb{P}(1000) = \mathbb{P}(\textit{M[0]}\ \mathsf{and\ not}\ \textit{M[1]}\ \mathsf{and\ not}\ \textit{M[2]}) & = 0.01*0.99*0.99 = 0.009801 \\
\mathbb{P}(0110) = \mathbb{P}(\mathsf{not}\ \textit{M[0]}\ \mathsf{and}\ \textit{M[1]}\ \mathsf{and}\ \textit{M[2]}) & = 0.99*0.01*0.01 = 0.000099 \\
\mathbb{P}(0100) = \mathbb{P}(\mathsf{not}\ \textit{M[0]}\ \mathsf{and}\ \textit{M[1]}\ \mathsf{and\ not}\ \textit{M[2]}) & = 0.99*0.01*0.99 = 0.009801 \\
\mathbb{P}(0010) = \mathbb{P}(\mathsf{not}\ \textit{M[0]}\ \mathsf{and\ not}\ \textit{M[1]}\ \mathsf{and}\ \textit{M[2]}) & = 0.99*0.99*0.01 = 0.009801 \\
\mathbb{P}(0000) = \mathbb{P}(\mathsf{not}\ \textit{M[0]}\ \mathsf{and\ not}\ \textit{M[1]}\ \mathsf{and\ not}\ \textit{M[2]}) & = 0.99*0.99*0.99 = 0.970299
\end{cases}$$

On calcule la probabilité attendue pour chaque instanciation possible du vecteur d'entrée lorsque *étage[0]* est vrai :

$$\begin{cases}
\mathbb{P}(10100000000000{*}{*}{*}{*}\mathbf{1110}0{*}{*}{*}) = 1/4 * 1/2^7 * 0.000001 = 1.953 * 10^{-9} \\
\mathbb{P}(10100000000000{*}{*}{*}{*}\mathbf{1100}0{*}{*}{*}) = 1/4 * 1/2^7 * 0.000099 = 1.933 * 10^{-7} \\
\mathbb{P}(10100000000000{*}{*}{*}{*}\mathbf{1010}0{*}{*}{*}) = 1/4 * 1/2^7 * 0.000099 = 1.933 * 10^{-7} \\
\mathbb{P}(10100000000000{*}{*}{*}{*}\mathbf{1000}0{*}{*}{*}) = 1/4 * 1/2^7 * 0.009801 = 1.9142 * 10^{-5} \\
\mathbb{P}(10100000000000{*}{*}{*}{*}\mathbf{0110}0{*}{*}{*}) = 1/4 * 1/2^7 * 0.000099 = 1.933 * 10^{-7} \\
\mathbb{P}(10100000000000{*}{*}{*}{*}\mathbf{0100}0{*}{*}{*}) = 1/4 * 1/2^7 * 0.009801 = 1.9142 * 10^{-5} \\
\mathbb{P}(10100000000000{*}{*}{*}{*}\mathbf{0010}0{*}{*}{*}) = 1/4 * 1/2^7 * 0.009801 = 1.9142 * 10^{-5} \\
\mathbb{P}(10100000000000{*}{*}{*}{*}\mathbf{0000}0{*}{*}{*}) = 1/4 * 1/2^7 * 0.970299 = 0.00189
\end{cases}$$

On constate que les probabilités obtenues table 7.7 sont bien celles qui sont attendues.

Probabilités conditionnelles

Soit une liste de probabilités conditionnelles $\mathcal{M}$ consistant en la modification de la probabilité des signaux *requête[i]*:
$\mathcal{M} = [\langle$*requête[0], 0.1, étage[0]*$\rangle$, $\langle$*requête[1], 0.1, étage[1]*$\rangle$, $\langle$*requête[2], 0.1, étage[2]*$\rangle$, $\langle$*requête[3], 0.1, étage[3]*$\rangle]$

La table 7.8 est le résultat donné par Lutess pour l'état initial. On peut constater que cela correspond bien à un profil opérationnel. La somme des probabilités des vecteurs est

po	*pf*	*étage[0..3]*	*dessus[0..3]*	*dessous[0..3]*	*R[0..3]*	*M[0..3]*	*D[0..3]*	probabilité
1	0	1000	0000	0000	****	1110	0***	$1.95313 * 10^{-9}$
1	0	1000	0000	0000	****	1100	0***	$1.93359 * 10^{-7}$
1	0	1000	0000	0000	****	1010	0***	$1.93359 * 10^{-7}$
1	0	1000	0000	0000	****	1000	0***	$1.91426 * 10^{-5}$
1	0	1000	0000	0000	****	0110	0***	$1.93359 * 10^{-7}$
1	0	1000	0000	0000	****	0100	0***	$1.91426 * 10^{-5}$
1	0	1000	0000	0000	****	0010	0***	$1.91426 * 10^{-5}$
1	0	1000	0000	0000	****	0000	0***	0.0018951
1	0	0100	0000	0000	****	1110	0***	$1.95313 * 10^{-9}$
1	0	0100	0000	0000	****	1100	0***	$1.93359 * 10^{-7}$
1	0	0100	0000	0000	****	1010	0***	$1.93359 * 10^{-7}$
1	0	0100	0000	0000	****	1000	0***	$1.91426 * 10^{-5}$
1	0	0100	0000	0000	****	0110	0***	$1.93359 * 10^{-7}$
1	0	0100	0000	0000	****	0100	0***	$1.91426 * 10^{-5}$
1	0	0100	0000	0000	****	0010	0***	$1.91426 * 10^{-5}$
1	0	0100	0000	0000	****	0000	0***	0.00189512
1	0	0010	0000	0000	****	1110	0***	$1.95313 * 10^{-9}$
1	0	0010	0000	0000	****	1100	0***	$1.93359 * 10^{-7}$
1	0	0010	0000	0000	****	1010	0***	$1.93359 * 10^{-7}$
1	0	0010	0000	0000	****	1000	0***	$1.91426 * 10^{-5}$
1	0	0010	0000	0000	****	0110	0***	$1.93359 * 10^{-7}$
1	0	0010	0000	0000	****	0100	0***	$1.91426 * 10^{-5}$
1	0	0010	0000	0000	****	0010	0***	$1.91426 * 10^{-5}$
1	0	0010	0000	0000	****	0000	0***	0.00189512
1	0	0001	0000	0000	****	1110	0***	$1.95313 * 10^{-9}$
1	0	0001	0000	0000	****	1100	0***	$1.93359 * 10^{-7}$
1	0	0001	0000	0000	****	1010	0***	$1.93359 * 10^{-7}$
1	0	0001	0000	0000	****	1000	0***	$1.91426 * 10^{-5}$
1	0	0001	0000	0000	****	0110	0***	$1.93359 * 10^{-7}$
1	0	0001	0000	0000	****	0100	0***	$1.91426 * 10^{-5}$
1	0	0001	0000	0000	****	0010	0***	$1.91426 * 10^{-5}$
1	0	0001	0000	0000	****	0000	0***	0.00189512

'0' et '1' représent les booléens 'faux' et 'vrai'.
po et *pf* représentent les entrées *portes_ouvertes* et *portes_fermées*.
R, *M* et *D* représentent *requête*, *appel_montée* et *appel_descente*.

TAB. 7.7 – *Profil opérationnel donné pour l'état initial obtenu pour* $\mathcal{L}$

po	pf	étage	dessus	dessous	R[0..3]	AM[0..3]	AD[0..3]	probabilité
1	0	1000	0000	0000	1***	***0	0***	$4.88281 * 10^{-5}$
1	0	1000	0000	0000	0***	***0	0***	0.000439453
1	0	0100	0000	0000	*1**	***0	0***	$4.88281 * 10^{-5}$
1	0	0100	0000	0000	*0**	***0	0***	0.000439453
1	0	0010	0000	0000	**1*	***0	0***	$4.88281 * 10^{-5}$
1	0	0010	0000	0000	**0*	***0	0***	0.000439453
1	0	0001	0000	0000	***1	***0	0***	$4.88281 * 10^{-5}$
1	0	0001	0000	0000	***0	***0	0***	0.000439453

TAB. 7.8 – *Profil opérationnel donné pour l'état initial obtenu pour $\mathcal{M}$*

égale à 1, à 10^{-6} près. $(4.88281*10^{-5}+0.000439453)*4*2^9 = 0.9999996$ et $(4.88281*10^{-5}+0.0004394531)*4*2^9 = 1.0000001$.

De plus, les résultats donnés sont cohérents. A l'état initial, l'ascenseur a la probabilité 0.25 d'être à un étage. Etudions le cas où l'ascenseur est à l'étage 0. Il y a alors 2^{10} vecteurs possibles (10100000000000*******00***). La probabilité que chacun des signaux *appel_montée[0..2]*, *appel_descente[1..3]* et *requête[1..3]* soit vrai est de 0.5 dans ce cas là. La probabilité de *requête[0]* est de 0.1. Ainsi, on obtient :

$$\begin{cases} P(10100000000000\mathbf{1}{*}{*}{*}{*}{*}{*}00{*}{*}{*}) = 0.25 * 0.1 * (0.5^9) = 4.88281 * 10^{-5} \\ P(10100000000000\mathbf{0}{*}{*}{*}{*}{*}{*}00{*}{*}{*}) = 0.25 * 0.9 * (0.5^9) = 0.0004394531 \end{cases}$$

Ce qui correspond aux résultats produits par Lutess. Nous attirons l'attention du lecteur sur le fait que nous avons réalisé ces calculs avec une calculatrice Unix simple précision et n'avons pas pratiqué d'arrondis. Toutefois, si l'on s'attache à interpréter les probabilité proposées par l'outil au sens probabilistique du terme, seul les ordres de grandeurs sont significatifs. Ainsi dans la table 7.7, on peut lire que la probabilité du vecteur "1 0 0001 0000 0000 **** 1110 0***" est de $1.95313 * 10^{-9}$. La longueur des séquences considérées est telle que seul l'approximation $2 * 10^{-9}$ est significative.

7.3 Bilan et conclusion

Dans ce chapitre, nous avons souhaité valider la méthode de génération basée sur les probabilités conditionnelles, que nous avons proposée. Pour cela, nous avons utilisé deux programmes : l'ascenseur et le convertisseur de clics.

Nous avons effectué des mesures de performance pour ces deux exemples, avec et sans probabilités conditionnelles, sur une station Sparc Ultra-1 avec 128 Mo de RAM. Pour le convertisseur de clics (exemple avec un profil opérationnel multiple donné pages 80 ou 92), la construction du générateur prends environ 2 secondes avec et sans les probabilités conditionnelles. Sans les probabilités conditionnelles, il faut environ 35 secondes pour produire 10 000 pas de test; la production de 50 000 pas de test (avec la construction du générateur) prend moins de 3 minutes. Avec les probabilités conditionnelles, il faut 40 secondes pour la

seule production de 10 000 pas de tests, et moins de 4 minutes pour produire 50 000 pas de test (dont la construction du générateur).

Pour l'ascenseur, on rappelle que sans probabilité conditionnelles, il faut moins d'une minute (environ 40 secondes CPU), (environ 80 secondes CPU) pour la construction du générateur, un peu plus d'une minute pour la génération de 10 000 pas de test et moins de 7mn pour produire 50 000 pas de test (construction du générateur inclue). Avec les probabilités conditionelles correspondant au profil opérationnel $\mathcal{Q}$, il faut environ 20 mn pour produire les 50 000 pas de test, dont 15mn pour la construction du générateur.

Dans ce chapitre, nous avons mis un soin tout particulier à étudier chaque programme proposé d'un point de vue statistique. Nous n'avons présenté ici que les résultats finaux, qui sont tous satisfaisants. Toutefois, au cours de notre étude, ces résultats n'ont pas toujours été aussi bons. Dans cette conclusion, nous nous attardons sur trois exemples de problèmes révélés au cours de cette étude statistique.

Lorsque nous avons voulu valider la méthode de génération avec des probabilités conditionnelles (cf. page 91), nous avons essayé plusieurs types de conditions : des conditions dépendant des entrées, des sorties à l'instant précédent... Les calculs de χ^2 se sont alors révélés mauvais pour des triplets comme ⟨*requête[0],p,étage[0]*⟩. Après avoir constaté que cela correspondaient à des conditions portant sur des variables d'entrées non contraintes, nous avons identifié un problème d'implantation de l'algorithme. Nous avons corrigé le problème et refait l'ensemble des calculs, qui cette fois ont été satisfaisants. L'étude statistique aura permis de corriger un problème résiduel.

Lorsque nous avons étudié le convertisseur de clics avec la distribution multiple (fig. 6.3, page 83), nous avons commis une erreur pendant la traduction du profil opérationnel en probabilités conditionnelles. L'étude statistique des données produites a montré une divergence entre la distribution des données attendue et celle obtenue. Nous avons alors utilisé la méthode d'affichage des profils opérationnels et avons détecté le problème de traduction.

Cette expérience nous a conforté dans l'idée que l'automatisation des traductions est utile. Pour l'instant, seule la traduction des probabilités conditionnelles vers les profils opérationnels a été entièrement automatisée. Mais les résultats proposés par le programme d'affichage ne sont pas toujours exploitables. Par exemple, pour le convertisseur de clics, Lutess propose 8 états au lieu de 4 dans l'exemple donné page 87. Pour l'ascenseur, nous n'avons jamais eu la patience d'explorer l'ensemble des états proposés (plus de 2^{35} vecteurs d'entrées valides). L'utilisateur gagnerait à ce qu'il puisse sélectionner par avance les états qu'il souhaite explorer.

Enfin, nous avons rencontré un autre problème pendant l'étude du convertisseur de clics, en comparant la distribution des sorties (type de clic logique) obtenue avec celle attendue. Le calcul de la distribution théorique repose sur l'élaboration de la chaîne de Markov correspondant au comportement probabiliste du programme et de son environnement (cf. page 94). La comparaison des distributions des sorties attendues et obtenues est suggérée par Whittaker dans sa thèse [94] pour détecter d'éventuelles erreurs de programmation.

Nous avons utilisé cette technique pour valider le générateur et avons supposé le programme correct. La première étude statistique a révélé un problème. Nous avons alors remis en cause successivement le modèle de l'environnement, la traduction du profil opérationnel, le modèle de la souris (fig. 7.1) et le calcul de la chaîne de Markov, sans résultat. Et c'est alors que nous avons remis en cause le programme et découvert une erreur, ainsi que l'avait prédit Whittaker dans sa thèse ! Après correction de l'erreur du programme, nous avons facilement pu valider les hypothèses statistiques faites.

Il est clair que l'étude de la distribution statistique des sorties serait profitable pour l'utilisateur. Ainsi, celui-ci pourrait aisément détecter des erreurs grossières dans son programme. Malheureusement, comme l'a remarqué Whittaker, cette méthode demande l'établissement de la distribution théorique des sorties, ce qui est rarement possible lorsque les programmes sont de taille conséquente.

Troisième partie

Validation de spécifications de services téléphoniques

Chapitre 8

Problématique relative à la validation de services téléphoniques

Nous abordons ici l'application de Lutess, en particulier des extensions que nous avons introduites, à un domaine en plein essor : la création de services téléphoniques.

Nous avons retenu ce domaine pour plusieurs de ses caractéristiques. Les spécifications formelles y sont très largement employées, en particulier les logiques temporelles. Plusieurs des méthodes de validation des spécifications de services fondées sur des outils de preuve (souvent par évaluation des modèles) ont montré leurs limites dans le traitement de cas de taille importante (cf. chapitre 10). La validation avant l'implantation est essentielle car l'interaction entre services téléphoniques peut avoir des conséquences désastreuses.

Cette partie est structurée en trois chapitres. Dans celui-ci, nous présentons les services téléphoniques et les enjeux relatifs à leur validation. Dans les chapitres suivants, nous présentons deux études de cas qui nous ont permis de montrer l'intérêt et l'efficacité de Lutess, pour la validation de spécifications de services téléphoniques.

8.1 Introduction

Evolution et compatibilité

Au cours des années, les réseaux téléphoniques ont beaucoup évolué, bénéficiant des progrès techniques. Les connexions entre abonnés ont d'abord été faites manuellement (par *"les demoiselles du téléphone"* en France). Puis, les commutateurs électroniques ont été introduits. Ils sont désormais progressivement remplacés par des réseaux numériques, appelés *Réseaux Intelligents.*

L'un des soucis majeurs lorsqu'on fait évoluer une technique est conserver une compatibilité, ce qui autorise de remplacer une partie du système en continuant d'utiliser une autre partie. En téléphonie, ce principe a été suivi dès le début : les téléphones du début du siècle sont ainsi encore compatibles avec le réseau actuel (moyennant quelques "règles" de branchement).

Parallèlement à l'évolution des réseaux, des services ont peu à peu été proposés aux clients que nous sommes. Le premier service offert aux usagers est le service de base dit POTS (*Plain Old Telephone Service*). Ce service permet d'établir, de maintenir, de clôturer et de facturer une communication téléphonique. Les autres services proposés sont appelés des *services supplémentaires*, car ils s'appuient généralement sur le service de base. Un service supplémentaire peut ajouter des fonctionnalités au service de base ou en modifier son comportement. Par exemple, un service de filtrage permet d'interdire l'établissement d'une communication sous certaines conditions.

Validation des services téléphoniques

Aujourd'hui, la plupart des services s'appuient sur les réseaux numériques et la réalisation d'un service est avant tout logicielle. La criticité du système impose des impératifs de sûreté sur les nouveaux services : leur introduction ne doit pas perturber le fonctionnement général du réseau téléphonique ou celui des autres services.

Le problème de l'introduction de services téléphoniques dans un réseau est un problème d'évolution et de maintenance d'un système informatique de grande taille. Aussi, la conception, l'implantation et la validation de services téléphoniques sont considérées dans le cadre de méthodes de génie logiciel classiques, comme par exemple des méthodes orientées objet [42, 62, 91] et des méthodes formelles.

L'application de méthodes formelles à ce domaine est particulièrement explorée du fait de l'usage et du succès assez anciens de ces méthodes aux protocoles de communication. Diverses logiques temporelles [9, 41, 23], SDL [23] et LOTOS [39] sont ainsi utilisés pour modéliser et spécifier un réseau et des services téléphoniques.

Pour toutes ces approches la validation s'effectue par des méthodes également classiques : animation [23], vérification de modèles [23], preuves déductives [41, 72] ou test [44]

Les niveaux de validation les plus courants (pour les équipes de recherche) se situent très tôt dans le cycle de développement du logiciel, c'est-à-dire au niveau de la spécification. En effet, se placer à un tel niveau permet de raisonner sur des modèles abstraits de tailles raisonnables pour l'application de méthodes formelles. De plus, valider les spécifications est une étape essentielle avant le développement d'un service, car plus tôt des erreurs seront détectées dans le cycle de développement, moins elles seront coûteuses à corriger.

Interactions entre les services téléphoniques

Parmi les problèmes intrinsèques de validation de services, un problème particulier a attiré l'attention de la communauté des chercheurs. Il s'agit du problème de l'*interaction entre services téléphoniques*.

Certains services sont conçus pour interagir avec d'autres. Par exemple, le service de *transfert d'appel explicite* utilise le service de *mise en attente*. Le service de transfert explicite (ECT) permet à un usager de mettre en communication ses deux correspondants. Ce service fonctionne de la façon suivante. Soit une secrétaire ($\mathcal{A}$) qui reçoit un appel d'un client ($\mathcal{B}$) qui souhaite parler au directeur de $\mathcal{A}$ ($\mathcal{C}$). Pour transférer l'appel, $\mathcal{A}$ doit mettre $\mathcal{B}$ en attente,

puis doit composer le numéro de $\mathcal{C}$. La communication entre $\mathcal{B}$ et $\mathcal{C}$ peut ensuite être établie en invoquant le service ECT. L'interaction entre le service de transfert d'appel explicite et le service de mise en attente d'un usager est voulue. Elle est spécifiée dans le document [37].

D'autres services ont parfois des comportements contradictoires. Par exemple, un service de transfert d'appel peut devoir établir une connexion alors qu'un service de filtrage peut devoir interdire cette même connexion. Il peut arriver que plusieurs services proposent simultanément des actions contradictoires pour un même appel. Le conflit qui en résulte est une interaction de service. On parle parfois d'interaction *néfaste* [47].

Les interactions néfastes de services font l'objet de beaucoup d'attention pour deux raisons. Tout d'abord, une interaction non prévue entre deux services peut perturber fortement un réseau téléphonique et aboutir à une panne. L'indisponibilité, même momentanée, du réseau téléphonique est à proscrire. Trop de services critiques, tels que les services d'urgences, dépendent du téléphone. De plus, lorsqu'un service est proposé au client, il doit accomplir les fonctionnalités pour lesquelles il a été vendu. L'introduction d'un nouveau service ne doit pas modifier les services existants, faute de quoi les clients manifestent leur mécontentement. Dans la suite, l'expression *interactions de services* désignera les interactions *néfastes* de services.

Les interactions entre les services sont connues pour constituer un des obstacles les plus sérieux au développement du domaine [14]. En effet, ce sont des centaines d'interactions potentielles qu'il faut pouvoir analyser pour tout nouveau service. Il est donc devenu critique de procéder à l'analyse des interactions à la création aussi bien que lors de l'exécution des services. L'importance du problème a donné lieu à de nombreuses publications, en particulier dans le cadre d'une conférence qui lui est consacrée [12, 22, 61].

Les travaux sur les interactions portent à la fois sur leur analyse et sur leur prise en compte.

L'analyse des différentes interactions connues permet de mieux les caractériser. Dans [19], les auteurs proposent une classification des interactions selon deux dimensions : la nature et les causes de l'interaction. Par exemple, on peut distinguer les interactions selon qu'elles impliquent un ou plusieurs utilisateurs, plusieurs instances du même service ou de différents services. Par ailleurs, l'architecture du réseau ou le nombre de composants du réseau mis en œuvre pour exécuter les services peut influer sur la présence d'une interaction.

La prise en compte des interactions relève de trois types de travaux dont les objectifs sont respectivement d'éviter les interactions, de les détecter, et de résoudre les conflits qui peuvent en résulter [12].

Certaines approches pour éviter l'introduction des interactions ont pour but de développer des environnements permettant de concevoir, de développer et d'implanter des services en minimisant les risques d'interactions [92, 21]. Par exemple, [92] propose d'appliquer des concepts simples utilisés pour le développement de systèmes distribués. D'autres approches proposent une aide à la création et au développement de services sous forme de guides ou de processus de développement de services [12].

Une phase de détection d'interaction est souvent nécessaire, car il est presque impossible d'empêcher l'introduction d'interaction. Une interaction peut être introduite à différents moments du développement : par exemple, pendant la spécification, le développement ou le

déploiement. On peut chercher à détecter les interactions à chaque étape de développement (chaque étape du cycle de vie du logiciel). On distingue deux grands types de méthodes de détection selon qu'elles sont utilisées avant ou après le déploiement (méthodes *off-line* et méthodes *on-line*). Les mécanismes de détection off-line peuvent se présenter sous la forme d'outils de validation, de vérification [66, 23] ou de simulation. Les mécanismes de détection on-line peuvent se présenter sous la forme d'outils d'analyse de traces [12].

Une fois qu'une interaction est détectée, il faut pouvoir la résoudre. De façon similaire à la détection, la résolution peut s'effectuer de façon statique, i.e. pendant la conception ou le développement (résolution *off-line*) [60], ou de façon dynamique, i.e. pendant l'exécution (résolution *on-line*) [46].

L'ensemble des approches de prise en compte des interactions peut être résumé dans le tableau ci-dessous [12].

	off-line	on-line
éviter les interactions	environnement de développement aide à la création de services (guide) processus de développement de services ...	
détecter les interactions	vérification validation simulation ...	analyse de comportement ...
résoudre les interactions	reconception enquête clientèle ...	négociation gestionnaire d'interactions résolution basée sur les événements ...

8.2 Notre approche pour la validation de services

Nous avons choisi d'utiliser Lutess pour la validation de spécifications et la détection d'interactions de façon off-line. Pour nous, valider une spécification de service consiste à s'assurer que cette spécification satisfait un ensemble de propriétés attendues du systèmes. Nous avons exprimé les spécifications sous une forme exécutable à l'aide d'un modèle synchrone, et nous avons opté pour une validation des modèles par des méthodes de test de type fonctionnel. Nous pensons que ces deux choix sont bien adaptés.

L'utilisation d'un modèle synchrone nous paraît bien adapté dans ce contexte. En effet, l'application d'une approche synchrone permet d'exprimer le modèle avec un haut niveau d'abstraction en ne considérant que des événements significatifs.

De même, il nous a semblé que le test était très bien adapté car il a pour vocation de rechercher les défaut d'une implantation [75]. Or la validation de services et en particulier

la détection d'interactions peut être assimilée à une recherche de défauts. De plus, le test est fort utile quand la validation des spécifications peut être pratiquée par des méthodes d'animation. Le spécifieur dispose alors d'un moyen pour se convaincre que les spécifications qu'il observe se comportent comme il le pressentait.

La détection d'interaction nous paraît être un cas particulier de la validation d'un service. Bien que la notion d'interaction de services ne soit pas complètement caractérisée dans la plupart des approches développées pour la spécification des services, il existe un certain nombre de définition formelles. Nous avons utilisé celle proposée par P. Combes, souvent référencée [23]. Elle indique qu'il existe une interaction lorsque, pris séparément, deux services (ou instances de services) satisfont leurs propriétés respectives, et qu'au moins une propriété n'est plus satisfaite lorsque les services sont considérés ensemble.

Nous avons étudié le problème de la validation de services par le test au cours de deux études de cas. Chronologiquement, la première étude de cas fut celle proposée par le CNET. Elle avait pour but d'étudier plusieurs spécifications ETSI de services [35, 37], de les modéliser et de les *valider* avec Lutess. L'aspect recherche d'interaction n'a pas été abordé.

La deuxième étude de cas a été menée dans le cadre d'un concours organisé en marge de la conférence *Feature Interaction Workshop 1998 (FIW98)* [61]. Ce concours a été organisé avec le soutient de Bell Labs et visait à évaluer la capacité de différents outils et méthodes à détecter des interactions [45].

Les deux prochains chapitres sont dédiés à ces deux études de cas. L'étude de cas proposée par le CNET pose les bases d'une méthode de validation de services supplémentaires. Cette méthode repose sur les points suivants :

- établissement de modèles (exécutables) du service de base et de services ajoutés et de leurs propriétés,
- définition d'une opération de composition,
- utilisation de Lutess pour valider les services.

L'étude de cas de FIW98 aborde de façon relativement complète le problème de la détection d'interaction. En effet, cette étude de cas consiste en l'étude de l'interaction potentielle de 12 services supplémentaires. Le modèle utilisé pour cette étude ainsi que l'opération de composition se situe à un plus haut niveau d'abstraction que celle demandée par le CNET. En particulier, pour l'étude de cas FIW98, les communications (au sein d'un commutateur) sont considérées comme instantanées et on ne considère que le cas où les services sont souscrits et activés, contrairement à l'étude de cas du CNET, pour laquelle les temps de communications sont modélisés et les notions d'activation et de désactivation des services sont prises en compte.

Dans le chapitre 9, nous décrirons donc une partie de l'étude de cas réalisée pour le CNET, consacrée à la validation de spécifications services. Puis, dans le chapitre 10, nous aborderons le problème de la détection d'interaction au travers de l'étude de cas FIW. Enfin, Le chapitre 12 présente une réflexion générale, a posteriori, sur les modèles et la méthode de validation qui ont été utilisés et développés pour ces études de cas.

Chapitre 9

Validation incrémentale de services téléphoniques (sur une étude de cas fournie par le CNET)

9.1 Introduction

Dans ce chapitre, nous étudions le problème de la validation de la spécification d'un service supplémentaire introduit dans un réseau en complément au service de base. Nous avons adopté une démarche qui consiste à

- établir un modèle exécutable des spécifications d'un système comprenant le service de base et le service supplémentaire étudié,
- établir un modèle d'environnement et un ensemble de propriétés de ces spécifications,
- valider le service par le test en simulant le comportement de l'environnement pour déterminer la "validité" du service "par rapport à un ensemble de propriétés". La simulation de l'environnement est construite de façon à invoquer le service "suffisamment" souvent. La validation est opérée de manière incrémentale car les services sont introduits les uns après les autres et la définition de certains services est fonction de caractéristiques d'autres services.

L'étude de la validation d'un service supplémentaire est basée sur une partie de l'étude de cas que nous a proposé le CNET[1]. Elle concernait plusieurs spécifications de services normalisés par l'ETSI[2] : transfert d'appel sur non réponse (CFNR : *Call Forwarding No Reply*) [35], transfert d'appel explicite (ECT : *Explicit Call Transfert*) [37], et service de complétion d'appel (CCBS : *Completion of Calls to Busy Subscriber* [36]. On présente ici le travail que nous avons effectué pour le service de base et le service CFNR, sachant que la prise en compte d'autres services a confirmé le bien-fondé de l'approche [27].

1. dans le cadre du contrat n°957B043 entre le CNET-France Télécom et l'Université Joseph Fourier.
2. Institut européen des normes de télécommunication, qui a pour mission d'élaborer des normes européennes de télécommunication pour permettre d'interconnecter les réseaux et les services nationaux.

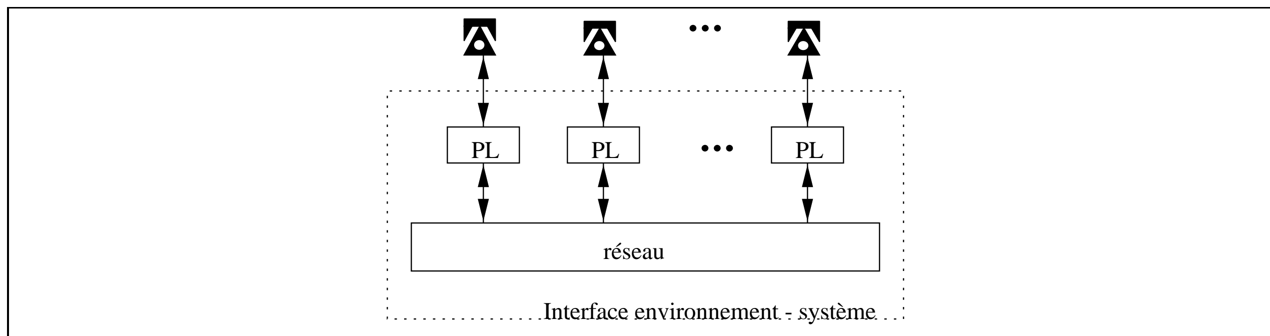


FIG. **9.1** – Architecture du modèle exécutable du système téléphonique

On présente le principe du modèle exécutable utilisé pour cette étude de cas en section 9.2. Puis, dans les sections 9.3 et 9.4, on étudie le service de base et le service de transfert d'appel sur non réponse (CFNR). On dresse enfin un bilan de cette étude de cas (section 9.5).

Cette étude de cas a constitué la première utilisation de la méthode de génération basée sur des probabilités conditionnelles (pages 120 et 129). Ce point particulier nous oblige à une description complète de l'interface du programme sous test, des contraintes d'environnement et des propriétés d'oracle utilisées. Cette présentation est fastidieuse. Nous nous en excusons d'avance auprès de nos lecteurs.

9.2 Conception d'un modèle exécutable

Pour un service supplémentaire, une norme ETSI décrit, par trois documents, du plus général au plus précis :

- le service du point de vue de l'utilisateur,
- les capacités fonctionnelles et les flots d'information nécessaires pour le service décrit dans le premier document, et
- les protocoles de communication et les fonctions de commutation nécessaires à l'implantation du service.

Les descriptions des services du point de vue utilisateur sont la base de notre travail. Chacune d'entre elles comprend des indications sur des procédures d'abonnement, de désabonnement, d'activation, de désactivation et d'invocation du service dans le cas général et dans des cas "exceptionnels" (tels que les tentatives de fraudes). Elles précisent aussi les interactions attendues avec certains services. Ces descriptions constituent des spécifications informelles.

9.2.1 Architecture du modèle du système exécutable

Nous avons construit un modèle selon le point de vue d'un utilisateur. L'environnement du système est constitué d'un nombre paramétrable d'appareils téléphoniques classiques. Chaque téléphone physique est commandé par une unité logicielle qui lui est propre : un poste logique (PL), qui gère les services proposés aux usagers.

Un poste logique a un comportement synchrone : il réagit instantanément aux sollicitations externes et internes. Un poste logique n'a pas connaissance de l'état des autres postes; il communique avec les autres à l'aide de messages, qui transitent par un réseau (fig. 9.1). La communication entre les postes logiques n'est donc pas instantanée. Un message émis sur le réseau au cycle t sera délivré à son destinataire au cycle $t+1$. Ce point de vue est proche des travaux de F. Boniol autour de la notion de synchronisme faible [11].

D'une façon plus précise, à chaque cycle, chaque poste logique lit ses entrées. Celles-ci proviennent de l'environnement (actions de l'utilisateur) et du réseau (messages de communication entre les postes logiques). Les postes logiques calculent ensuite leurs sorties vers l'environnement (contrôle du téléphone physique) et vers le réseau (message). Les messages sont pris en charge par le réseau, qui ne les délivre qu'au cycle suivant.

Il y a quatre types d'événements dans ce modèle[3] :

- Les *entrées externes* sont les actions des utilisateurs sur leurs téléphones.
- Les *sorties externes* sont les commandes produites par les postes logiques vers les téléphones. Ces commandes correspondent essentiellement à des tonalités.
- Les *entrées* et les *sorties internes* sont les messages produits par les postes logiques pour d'autres postes logiques. Ils transitent par le réseau. Ils ne sont pas visibles par les utilisateurs.

9.2.2 Structure d'un poste logique

Dans [15], les services sont modélisés par une pile $\mathcal{M}$ de machines à transitions d'états. La machine de plus bas niveau représente le service de base (nommé POTS). Les machines de niveaux supérieurs sont associées aux services supplémentaires (fig. 9.2). Les entrées de $\mathcal{M}$ sont aussi les entrées de la machine en sommet de pile. Ces entrées sont propagées de "haut en bas" en direction du POTS. A chaque niveau, la machine correspondante peut agir de trois façons différentes :

- elle peut transmettre l'entrée sans la modifier à la machine de niveau inférieur;
- elle peut la transmettre en la modifiant;
- elle peut la consommer et proposer une sortie.

Une sortie transite à travers les machines de "bas en haut". Une machine peut modifier une sortie proposée à un niveau inférieur.

On propose une représentation d'une pile de machines à transition d'états en Lustre (fig. 9.3) : à chaque machine de cet ensemble, on associe un nœud Lustre. Les nœuds représentant les services supplémentaires sont structurés en trois parties:

- calcul des entrées à fournir à la machine de niveau inférieur,
- appel du nœud représentant la machine de niveau inférieur,
- calcul des sorties à fournir à la machine de niveau supérieur.

Le nœud représentant le POTS n'effectue que le calcul des sorties à fournir à la machine de niveau supérieur.

3. Le nom des événements ont été choisis en adoptant le point de vue d'un poste logique.

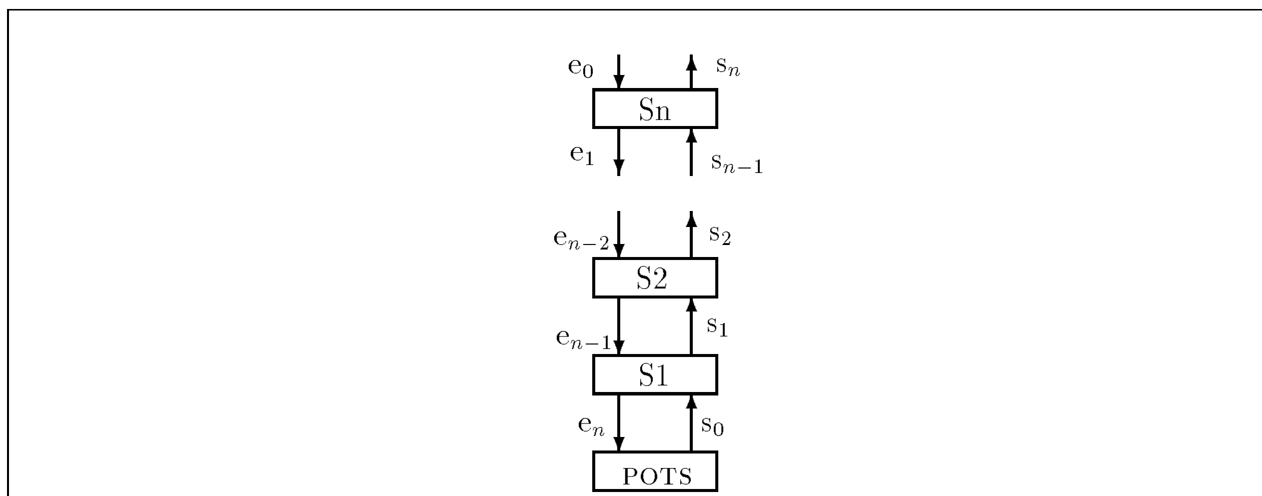


FIG. **9.2** – Composition des services

Tous les postes logiques sont identiques. Ainsi, un usager à un accès potentiel à tous les services. La liste des services auxquels un usager est abonné est une donnée interne de son poste logique. Cette liste est fixée statiquement, mais il serait possible de la modifier de façon dynamique.

9.2.3 Codage de l'information

A chaque type de messages, on associe un tableau structuré en sous-tableaux. Le nombre de sous-tableaux est égal au nombre maximum de paramètres qu'un message peut contenir. Prenons l'exemple de l'entrée externe *Dial(x,y)*. Cette entrée se décompose en trois sous-tableaux : le type du message *Dial*, l'émetteur x et une adresse y. Sachant qu'un téléphone physique et son poste logique communique directement, le sous-tableau codant le nom de l'émetteur du message est inutile. Il y a donc deux sous-tableaux pour les entrées externes.

La taille d'un sous-tableau dépend de la quantité d'information à coder. Par exemple, pour les messages de type "action utilisateur", la première partie (adresse de l'émetteur) doit pouvoir coder l'adresse de tous les utilisateurs. Nous avons choisi un codage "un parmi n" qui consiste à utiliser un booléen pour chaque utilisateur. Il nous faut donc 4 booléens pour coder une adresse.

Un tel codage présente deux avantages. Tout d'abord, l'absence d'information est directement codée. Lorsque toutes les variables d'un tableau codant un message sont fausses, cela signifie qu'il n'y a pas d'information à échanger. Dans la suite, on appellera ce code particulier "no_message". De plus, la cohérence des messages est trivialement testable. (Un message qui contient plus d'un booléen par sous-partie est trivialement incohérent).

Nous détaillons les messages au fur et à mesure de l'étude des services.

```
node ServiceN (eE: entrée_externe; eI: entrée_interne;
                LS: liste_de_souscription; N: Numéro_d'identification)
returns (sI: sortie_interne; sE: sortie_externe)
  var  eE': entrée_externe;
       eI': entrée_interne;
       sE': sortie_externe;
       sI': sortie_interne;
  let
       (eI',eE') = ModificationEntreeParServiceN(eE,eI,LS,N);
       (sI',sE') = ServiceN_1(eE',eI',LS,N);
       (sI,sE) = ModificationSortieParServiceN(sE',sI',LS,N);
  tel
                        ⋮
node Service1 (eE: entrée_externe; eI: entrée_interne;
                LS: liste_de_souscription; N: Numéro_d'identification)
returns (sI: sortie_interne; sE: sortie_externe)
  var  eE': entrée_externe;
       eI': entrée_interne;
       sE': sortie_externe;
       sI': sortie_interne;
  let
       (eI',eE') = ModificationEntreeParService1(eE,eI,LS,N);
       (sI',sE') = POTS(eE',eI',LS,N);
       (sI,sE) = ModificationSortieParService1(sE',sI',LS,N);
  tel

node POTS(eE: entrée_externe; eI: entrée_interne;
              LS: liste_de_souscription; N: Numéro_d'identification)
returns (sI: sortie_interne; sE: sortie_externe)
  var  State: variable_d'état;
       Party: numéro_du_correspondant;
  let
       (State,Party,sI,sE) = ...
  tel
```

FIG. **9.3** – Implantation du poste logique

9.3 Modélisation et validation d'un modèle du service de base

Le modèle de base choisi s'inspire de celui proposé par Nicolas Zuanon dans [95].

9.3.1 Modèle du service de base

Description des messages

- Un utilisateur peut effectuer trois actions : décrocher (*Off*), raccrocher (*On*), composer (*Dial*). Avec l'absence d'action, elles constituent les entrées externes.
- Un poste logique peut effectuer sept opérations de contrôle sur un téléphone physique :
  - connecter le microphone/haut-parleur *'OpenAudio'* (*OA*),
  - déconnecter le microphone/haut-parleur *'Mute'*,
  - déclencher la tonalité invitant à composer *'DialTone'* (*DTone*),
  - déclencher le signal indiquant la recherche du correspondant *'WaitingTone'* (*WTone*),
  - déclencher la sonnerie indiquant l'arrivée d'un appel *'RingingTone'* (*RTone*),
  - déclencher la sonnerie de retour perçue chez l'appelant lorsque le poste appelé sonne *'RingingBackTone'* (*RbTone*),
  - déclencher la tonalité "correspondant occupé" *'BusyTone'* (*BTone*).
- Il y a cinq types messages (entrées et sorties internes) échangés :
  - le correspondant décroche *'distantOff'* (*dOff*),
  - le correspondant raccroche *'distantOn'* (*dOn*),
  - demande de connexion *'distantDial'* (*dDial*),
  - notification de la réussite de la tentative de connexion *'connectSuccess'* (*cSucc*),
  - notification de l'échec de la tentative de connexion *'connectFailure'* (*cFail*).

La figure 9.4 détaille le codage des entrées et des sorties externes.

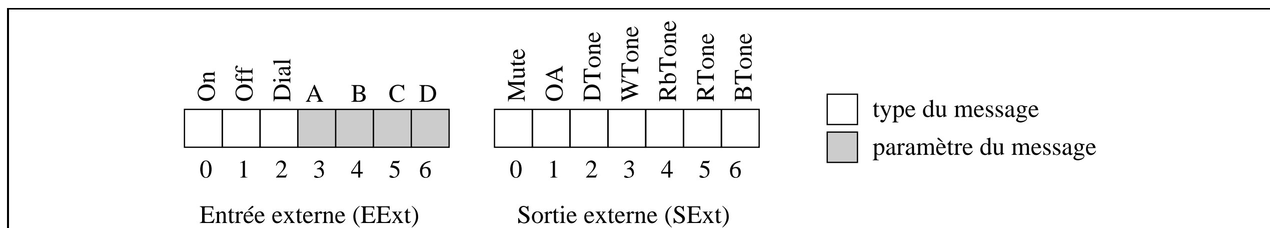


FIG. **9.4** – Codage des messages pour le POTS

Comportement du service de base

Ce comportement est représenté schématiquement sous la forme d'un automate (voir figure 9.5). Cet automate a 7 états : *Idle, Dialing, Waiting, Alerting, Talking, Ringing, Exception.*

Afin de ne pas surcharger ce schéma, nous n'avons pas représenté la production du message *cFail*. Ce message est produit dans tous les états autres que *Idle* lorsque le message interne *dDial* est reçu.

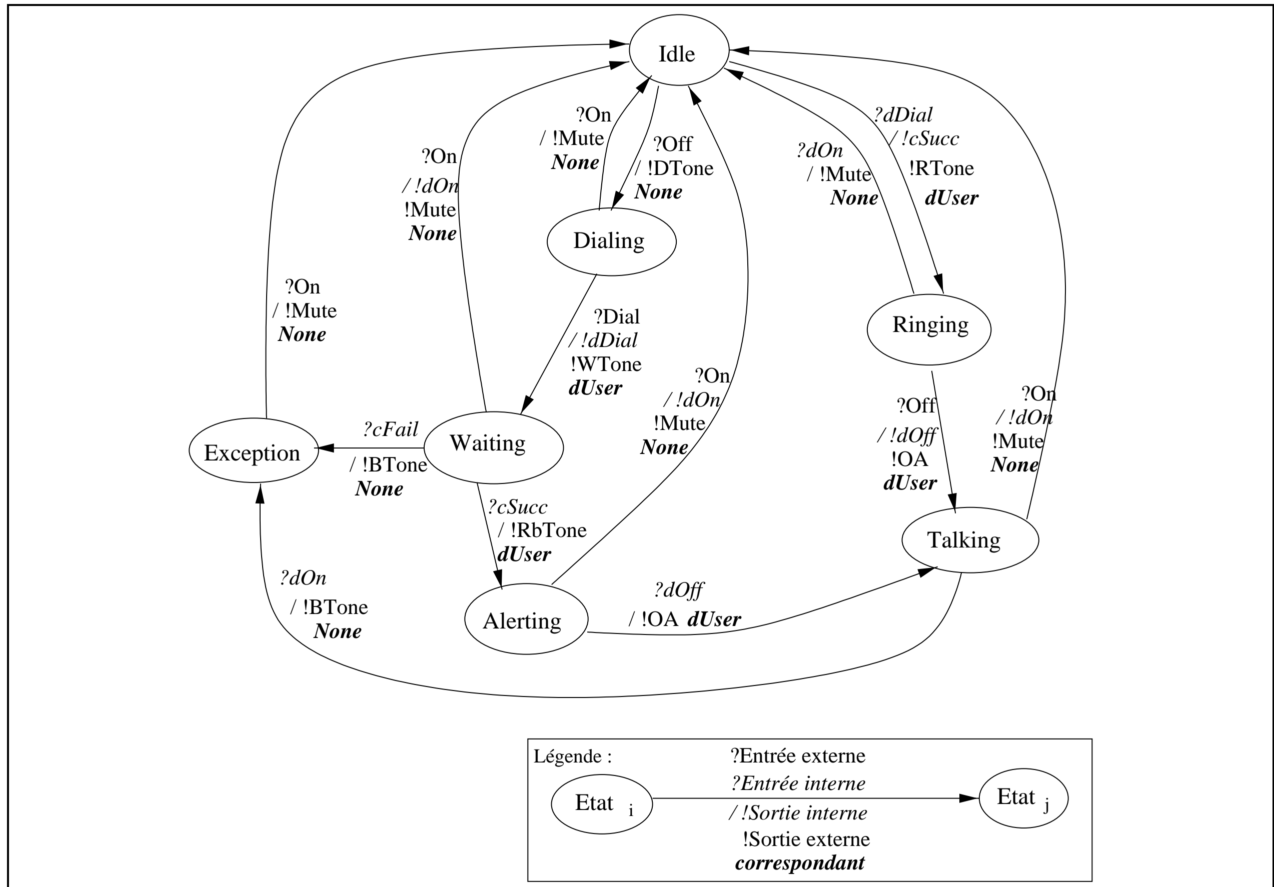


FIG. **9.5** – Automate du service de base

9.3.2 Validation du modèle du service de base

Pour faciliter la validation avec Lutess du modèle du service de base, nous avons rendu possible la consultation de la valeur de deux variables internes à partir de l'environnement. Il s'agit de l'état d'un poste logique (*Etat[0..6]* = { *Idle, Dialing, Waiting, Alerting, Talking, Ringing, Exception* }), et du numéro de son éventuel correspondant (*Party[0] : pas de correspondant; Party[i],* $i \neq 0$*: le correspondant est le poste i*).

Dans toute notre expérience, nous avons choisi un modèle de système téléphonique contenant quatre postes logiques. Les entrées et sorties externes concernant un poste logique i,

son état et le numéro de son correspondant sont notés respectivement $EExt_i$, $SExt_i$, $Etat_i$, et $Party_i$.

Pour valider le service de base, nous avons choisi de générer trois séquences enchaînées (cf. page 28) de 10 000 pas de tests (à partir du germe 11111).

Oracle

Les principales propriétés du modèle du service du base, que nous avons définies sont :

1. la variable *Party* est bien formée : elle indique un unique correspondant ou pas de correspondant;
   $ExactementUn(5, Party_i[0..4])$

2. les états d'un poste logique sont exclusifs;
   $ExactementUn(7, Etat_i[0..6])$

3. un poste logique produit au plus une sortie externe à chaque instant;
   $AuPlusUn(7, SExt_i[0..6])$

4. les sorties externes produites sont cohérentes avec l'état du poste logique (*Mute* => Idle, *DialTone* => *Dialing* ...):
   $SExt_i[0] => Etat_i[0]$, $SEx_it[2] => Etat_i[1]$, ...

Contraintes d'environnement

1. Contraintes d'environnement stipulant que les entrées sont bien formées :

   (a) chaque événement a au plus un type (*On*,*Off*,*Dial*) :
   $AuPlusUn(3, EExt_i[0..2]))$

   (b) le message *Dial* ($EExt[2]$) a un paramètre (une adresse);
   $EExt_i[2] => (ExactementUn(4, EExt_i[3..6]))$

   (c) les messages *On* et *Off* ($EExt_i[0]$ et $EExt_i[1]$) n'ont pas de paramètres;
   $(EExt_i[0]$ or $EExt_i[1]) =>$ not $LIN_OR(4, EExt_i[3..6])$

2. Contraintes d'environnement décrivant les comportements valides des utilisateurs :

   (a) on ne peut jamais effectuer deux fois de suite une action décrocher (resp. raccrocher) sans effectuer entre temps une action raccrocher (resp. décrocher);
   $always_from_to(EExt_i[0],$ pre $EExt_i[1], EExt_i[1])$,
   $always_from_to(EExt_i[1],$ pre $EExt_i[0], EExt_i[0])$.

   (b) un usager ne peut raccrocher qu'après avoir décroché au moins une fois;
   $EExt_i[0] => after(EExt_i[1])$

   (c) un usager ne peut composer que lorsqu'il entend le *DialingTone*;
   $EExt_i[2] =>$ pre $SExt_i[2]$.

Séquence → ↓ Nombre d'occurrences de	S1	S2	S3
On	8 574	8 463	8 518
Off	8 576	8 467	8 520
Dial	6 440	6 363	6 390
erreurs	1 007	1 042	950
cas où un poste est dans l'état D (tous postes confondus)	9 622	9 392	9 576
cas où un poste est dans l'état T (tous postes confondus)	486	526	492
où un poste est dans l'état E (tous postes confondus)	2 388	2 487	2 525

TAB. 9.1 – *Synthèse des données produites par la méthode de aléatoire pour POTS*

Génération aléatoire équiprobable

Une première étape de validation consiste à utiliser la méthode de génération aléatoire (cf. chapitre 3). Cette étape permet d'obtenir une première validation du modèle du service, à l'aide d'une mise en œuvre simple.

La table 9.1 a été obtenue par comptage des actions des utilisateurs et des états des postes logiques pour les 3 séquences de tests. Les erreurs observées correspondent toutes à des pertes de messages consécutives à la saturation du réseau. Celui-ci est saturé lorsque deux messages sont destinés à un poste logique au même instant. La saturation du réseau perturbe le fonctionnement du service de base qui n'a pas été conçu pour être tolérant aux pertes de messages.

Il est possible de diminuer le nombre de message perdus en imposant la contrainte "au plus une action par cycle". On obtient alors 700 occurences d'erreurs pour la séquence S1.

L'étude des traces indique que cette méthode de génération produit des données telles que :

- un usager compose son numéro une fois sur quatre;
- dans l'état *Idle*, un usager décroche puis raccroche immédiatement après;
- dans l'état *Exception*, un usager attend lontemps avant de raccrocher;
- dans l'état *Dialing*, un usager attend lontemps avant de composer.

Ces comportements sont valides et intéressants à observer. Toutefois, la *fréquence* avec laquelle ces cas apparaissent n'est pas justifiée par rapport à la faible diversité des comportements qu'ils représentent; on considère en fait implicitement que ces comportements appartiennent à une même de classe d'équivalence. Ainsi, nous allons favoriser les comportements considérés comme les plus intéressants sans interdire les autres, grâce à la méthode de génération poposée dans le chapitre 6.

Génération à partir de profils opérationnels

Pour produire les profils opérationnels, on identifie tout d'abord les entrées externes possibles dans chaque état de l'automate (cf. fig. 9.6a).

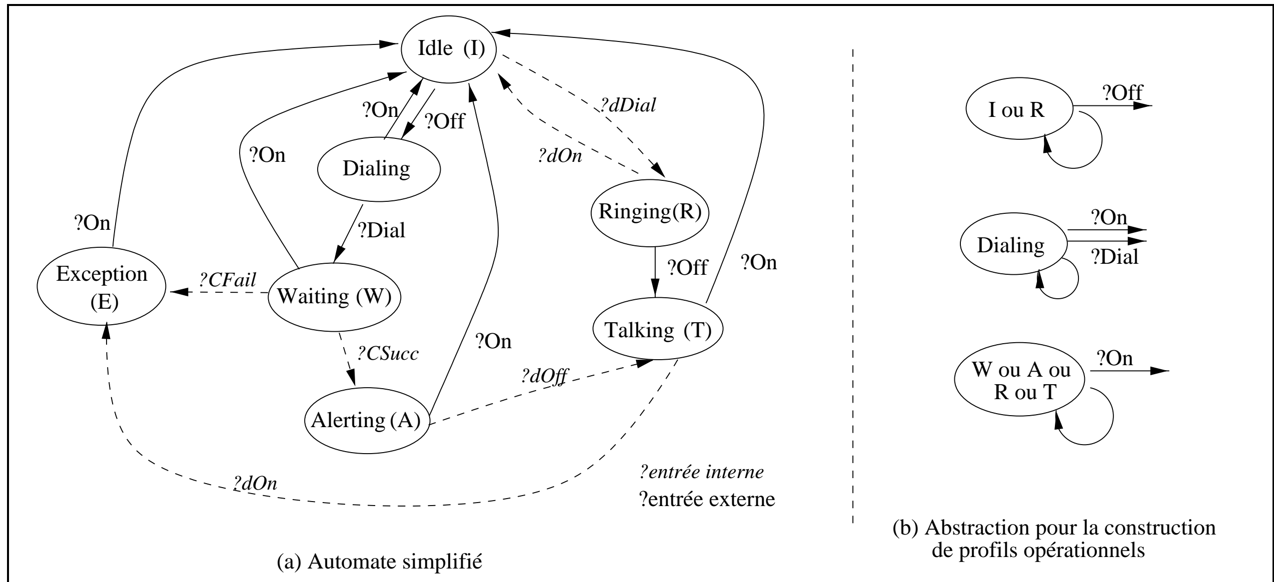


FIG. **9.6** – Détermination des éléments constitutifs des profils opérationnels

Puis, pour chaque état, nous établissons un profil opérationnel. Par exemple, pour les états *Idle*, *Dialing*, et *Exception*, nous avons choisi de façon empirique :

Idle (Etat[0])	
Off	0.1
rien	0.9

Exception (Etat[6])	
On	0.9
rien	0.1

Dialing (Etat[1])	
On	0.1
Dial	0.7
rien	0.2

avec $\begin{cases} On & EExt[0] \\ Off & EExt[1] \\ Dial & EExt[2] \end{cases}$

Ces profils opérationnels fournissent une description partielle de l'environnement. Par ailleurs, plusieurs profils opérationnels pourraient être établis pour différents postes, traduisant ainsi des comportements de différents utilisateurs. Par exemple, en diminuant la probabilité de décrocher dans l'état *Idle*, on obtient un utilisateur téléphonant moins que celui décrit ci-dessus.

A partir de ces profils opérationnels, nous avons utilisé l'algorithme présenté section 6.2 pour obtenir des probabilités conditionnelles (figure 9.7). Pour chaque utilisateur i, nous obtenons alors la liste de probabilités conditionnelles suivante, qui précise la probabilité d'occurence de chaque action :

$\mathcal{L}_i = [\langle EExt_i[0], 0.9, \textsf{pre}\ Etat_i[6] \rangle, \langle EExt_i[1], 0.1, \textsf{pre}\ Etat_i[0] \rangle, \langle EExt_i[0], 0.1, \textsf{pre}\ Etat_i[1] \rangle, \langle EExt_i[2], 7/9, \textsf{pre}\ Etat_i[1] \rangle]$

Nous souhaitons maintenant diminuer la fréquence avec laquelle un utilisateur s'appelle. Il faut agir sur le paramètre *adresse* quand *Dial* est émis. La seule contrainte d'environnement concernant *adresse* dans ce cas est “ExactementUn” (cf. contrainte d'environnement

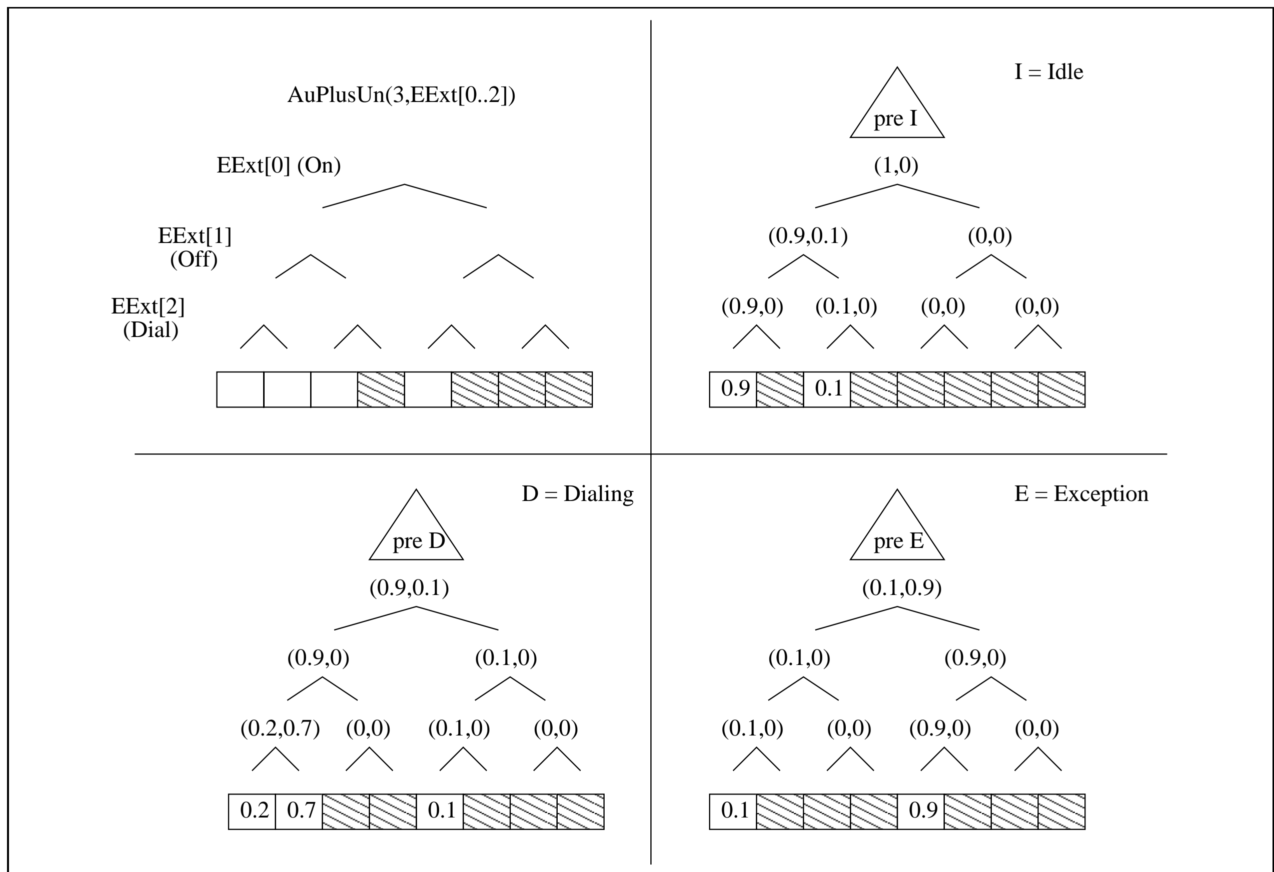


FIG. **9.7** – Détermination de la valeur des probabilités conditionnelles pour $\mathcal{L}_i$

1b). En appliquant l'algorithme donné section 6.2 à cette contrainte, on obtient une liste de probabilités pour chaque poste i. La figure 9.8 permet d'établir cette liste de façon générique :
$\mathcal{L}'_i = [\langle EExt_i[3]$, c1, $true\rangle$, $\langle EExt_i[4]$, c3/(c2+c3), $true\rangle$, $\langle EExt_i[5]$, c5/(c4+c5), $true\rangle$] [4]

Lorsque l'on propose le profil opérationnel suivant, pour modifier la distribution des adresses :

	$adresse_i = 1$ $EExt_i[3]$	$adresse_i = 2$ $EExt_i[4]$	$adresse_i = 3$ $EExt_i[5]$	$adresse_i = 4$ $EExt_i[6]$
PO($i = 1$)	0.01	0.33	0.33	0.33
PO($i = 2$)	0.33	0.01	0.33	0.33
PO($i = 3$)	0.33	0.33	0.01	0.33
PO($i = 4$)	0.33	0.33	0.33	0.01

on obtient pour chaque poste, les listes de probabilités conditionnelles suivantes :
$\mathcal{L}'_1 = [\ \langle EExt_1[3]$, 0.01, $true\rangle$, $\langle EExt_1[4]$, 33/99=1/3, $true\rangle$, $\langle EExt_1[5]$, 33/66=0.5, $true\rangle$]
$\mathcal{L}'_2 = [\ \langle EExt_2[3]$, 0.33, $true\rangle$, $\langle EExt_2[4]$, 1/67, $true\rangle$, $\langle EExt_2[5]$, 33/66=0.5, $true\rangle$]

4. avec, pour le poste i, c1 la probabilité de composer le numéro du poste 1, c3 la probabilité de composer le numéro du poste 2, c5 la probabilité de composer le numéro du poste 3 et c4 la probablité de composer le numéro du poste 4, comme indiqué figure 9.7.

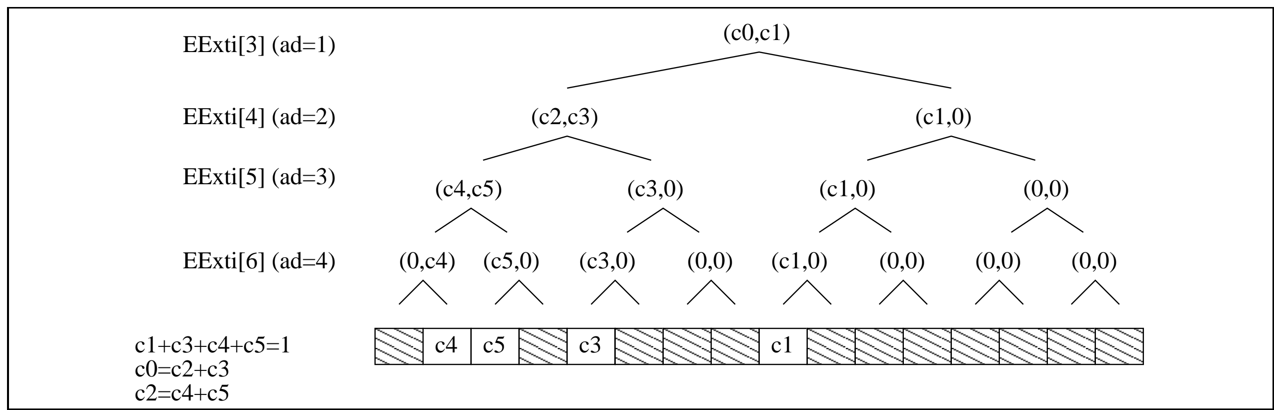


FIG. **9.8** – Détermination de la valeur des probabilités conditionnelles pour $\mathcal{L}'_i$

Séquence → ↓ Nombre d'occurrences de	S1	S2	S3
On	3 717	3 759	3 786
Off	3 717	3 759	3 786
Dial	2 555	2 575	2 593
erreurs	186	211	207
cas où un poste est dans l'état D (tous postes confondus)	3 628	3 579	3 677
cas où un poste est dans l'état T (tous postes confondus)	742	708	652
où un poste est dans l'état E (tous postes confondus)	889	936	900

TAB. 9.2 – *Synthèse des données produites par la méthode guidée par les profils opérationnels pour le POTS*

$\mathcal{L}'_3 = [\ \langle EExt_3[3],\ 0.33,\ true\rangle,\ \langle EExt_3[4],\ 33/67,\ true\rangle,\ \langle EExt_3[5],\ 1/34,\ true\rangle\ ]$
$\mathcal{L}'_4 = [\ \langle EExt_4[3],\ 0.33,\ true\rangle,\ \langle EExt_4[4],\ 33/67,\ true\rangle,\ \langle EExt_4[5],\ 33/34,\ true\rangle\ ]$

La table 9.2 a été obtenue par comptage des actions des utilisateurs et de l'état des postes logiques pour les trois séquences générées. Les erreurs observées correspondent là encore à des pertes de messages consécutives à la la saturation du réseau. Le nombre d'erreurs est plus faible que précédemment car le nombre d'actions effectuées par les utilisateurs a aussi décru.

Conclusion

L'utilisation des probabilités conditionnelles a permis la génération de données de test, pour une meilleure analyse du service de base. En effet, par ce biais, nous avons pu diminuer le nombre d'erreurs dues à des pertes de messages (186 erreurs à analyser au lieu de 1007, pour la séquence S1). L'analyse des erreurs restantes a été ainsi facilitée.

L'élaboration des probabilités conditionnelles avait pour but de diminuer la fréquence

d'apparition de certains comportements que l'on peut considérer comme étant de même nature (appartenant à une même classe d'équivalence en quelques sortes). C'est le cas des séquences d'actions *décrocher-raccrocher*, ou les attentes dans les états *Dialing* ou *Exception*. De ce point de vue, les données générées satisfont pleinement les objectifs fixés. Ainsi, pour la séquence S1, on peut constater que

- le nombre de cas où les postes sont dans l'état *Dialing* a diminué de 9 622 à 3 628;
- le nombre de cas où les postes sont dans l'état *Exception* a diminué de 2 388 à 889;
- le nombre d'actions *décrocher* et *raccrocher* a diminué de 8 574 à 3 717, et parallèlement, le nombre de cas où les postes sont dans l'état *Talking* a augmenté de 486 à 742, ce qui suggère que le nombre de communications ayant abouti a augmenté.

Par ailleurs, les mesures de performances effectuées indiquent que sur une station Sparc Ultra-1 avec 128 Mo de RAM, la génération de 10 000 pas de test à l'aide de la première méthode prend 5 minutes environ dont 4 secondes CPU pour la construction du générateur. A l'aide de la méthode guidée par les probabilités conditionnelles (avec 16 triplets), la génération de 10 000 pas de test prend 12 minutes dont 5 secondes CPU pour la construction du générateur. Notons que le temps supplémentaire passé dans la génération des nouvelles séquences a largement été compensé par le gain de temps dans l'analyse des erreurs restantes.

9.4 Modélisation et validation d'un modèle du service CFNR

9.4.1 Description informelle du service

Le service de transfert d'appel sur non réponse (*Call Forwarding No Reply (CFNR)*) est défini par la norme ETSI [35]. Ce service permet à un abonné de faire transférer ses appels vers un usager de son choix. Un appel doit être transféré lorsqu'il arrive au poste de l'abonné si celui-ci n'a pas répondu à cet appel après un délai fixé. Le laps de temps offert à l'abonné pour sa réponse est fixé par l'opérateur. Dans la suite, on le note ce laps de temps τ.

Afin d'éviter l'établissement d'une boucle de transfert, la norme fixe un nombre maximum de transferts pour un même appel. Cette limite est à la discrétion des opérateurs mais ne doit pas dépasser 5 transferts (quelsque soient les services de transfert invoqués).

Le principe du service est décrit par le schéma 9.9. Lorsqu'il est fourni à l'utilisateur, le service est inactif. L'abonné peut activer le service en composant un numéro de code, suivi du numéro d'un usagé (*CFNRon(adresse)*), où *adresse* est le numéro de l'usager qui recevra les appels de l'abonné. Avant l'activation du service, le numéro *adresse* peut être vérifié. Si ce numéro est jugé invalide, l'activation est refusée (*CFNR ActivKO*); elle est acceptée sinon (*CFNR ActivOK*). Par exemple, lorsqu'un abonné indique son propre numéro pour *adresse*, l'activation est jugée incorrecte. Lorsque le service est actif, un abonné peut modifier le destinataire de son transfert en utilisant la même commande (*CFNRon(adresse)*).

L'usager peut aussi désactiver le service (*CFNRoff*). La désactivation peut être acceptée ou non (*CFNR DesacOK, CFNR DesacKO*).

Le résultat des procédures d'activation et de désactivation doit être notifié à l'abonné.

On dit que le service de transfert d'appel sur non réponse est invoqué (*Invocation*) lorsqu'un appel aboutit chez cet abonné et que le délai de réponse τ est expiré. Un service est transféré lorsque l'invocation est acceptée (*InvocationOK*), c'est-à-dire lorsque le service est actif et que le nombre limite de transfert n'est pas atteint.

En sus de la description de l'utilisation normale du service, la spécification ETSI décrit aussi le comportement du service en cas de tentatives d'utilisation "anormale" ou frauduleuse. Par exemple, les usagers non abonnés peuvent utiliser les codes d'activation et de désactivation des services (*CFNRon, CFNRoff*). Dans ces cas, le service doit rester inactif. On ne détaillera pas plus le document [35].

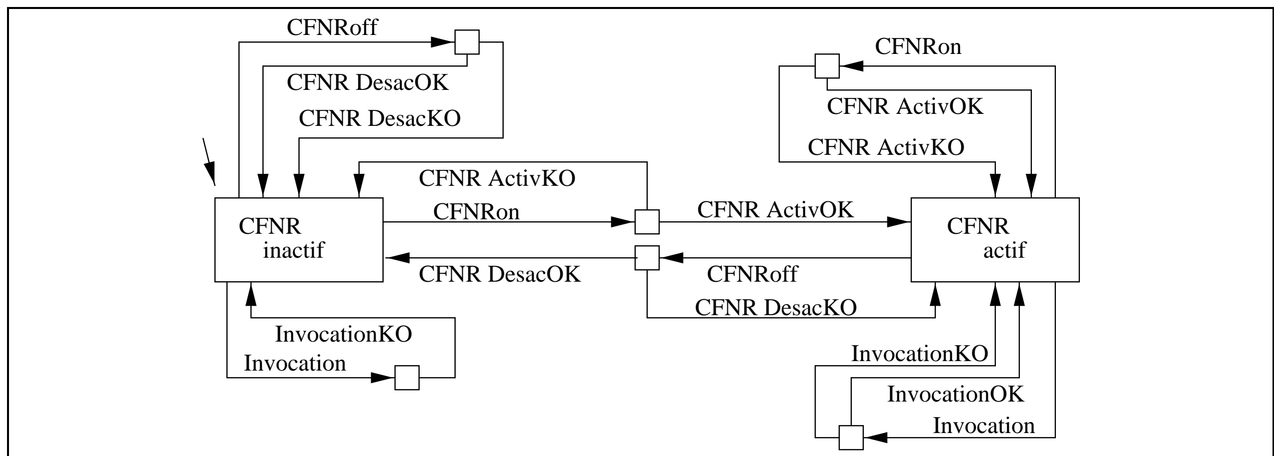


FIG. **9.9** – Principe de fonctionnement du service CFNR

9.4.2 Le modèle du service

Description des messages

- Un utilisateur peut effectuer cinq actions : les trois actions de base, activer le service CFNR (*CFNRon*) et le désactiver (*CFNRoff*).
- Un poste logique peut effectuer onze opérations de contrôle sur un téléphone physique :
  - les sept opérations proposées par le service de base,
  - notification que l'activation du service est réussie (*CFNR ActivOK*),
  - notification que l'activation du service a échoué (*CFNR ActivKO*),
  - notification que la désactivation du service est réussie (*CFNR DesacOK*),
  - notification que la désactivation a échoué (*CFNR DesacKO*).
- Il y a huit types de messages échangés :
  - les cinq définis par le service de base,
  - la demande de transfert d'appel (*CallForward*),

- notification de la réussite du transfert d'appel (*CallForwardSucc*),
- notification de l'échec du transfert d'appel (*CallForwardFail*).

La figure 9.10 détaille le codage des entrées et des sorties externes.

FIG. **9.10** – Codage des messages pour le service CFNR

Description de l'automate du service

Cet automate est une extension de l'automate du service de base en l'enrichissant par de nouvelles transitions (dans la figure 9.11, les transitions représentant le service de base sont figurées en pointillé).

FIG. **9.11** – Automate du service CFNR

L'introduction de CFNR crée deux nouvelles actions *CFNRon* et *CFNRoff* à prendre en compte. Ces actions ne sont autorisées que dans l'état *Dialing*. Chacune de ces actions donne lieu à deux types de réactions de la part du système : une notification d'acceptation ou d'échec. Aussi, dans l'état *Dialing*, il y a deux transistions par action.

Lorsque le service est invoqué (l'abonné est dans l'état *Ringing*) et que le transfert est autorisé, l'appel est transféré. Cela se traduit par l'émission du message interne *!CallForward*. Le poste de l'abonné revient à l'état *Idle*.

Le message *CallForward* est envoyé à deux usagers : le destinataire du transfert et l'appelant. Ce message est utilisé par l'appelant pour mettre à jour sa variable *Party* (qui dénote le numéro de son correspondant). Ainsi, l'appelant peut envoyer les messages pour son correspondant directement au destinataire du transfert d'appel.

Lorsque le destinataire du transfert reçoit un message *CallForward*, deux situations se présentent : soit il est dans l'état *Idle* soit il ne l'est pas. Dans le premier cas, il émet le message *CallForwardSucc* à l'appelant; dans le second, il émet le message *CallForwardFail* (non figuré sur le schéma 9.11) et l'appelant passe de l'état *Alerting* à l'état *Exception*.

Afin de ne pas surcharger ce schéma, nous n'avons pas représenté la production du message *CallForwardFail*, qui est produit dans tous les états autres que *Idle* lorsque le message *CallForward* est reçu. De plus, nous n'avons pas fait figurer les trois variables suivantes : l'état d'abonnement du service CFNR, l'état du service (actif ou non) et le numéro du correspondant chez qui aboutit le transfert d'appel.

Pour la réalisation, nous avons limité le nombre de transferts d'appel possible à trois : il doit donc être impossible d'observer quatre transferts consécutifs pour un même appel. Le nombre de transferts qu'a subi un appel est indiqué dans le message *CallForward*.

9.4.3 Validation du service

La validation du service CFNR a été effectuée de la même manière que pour le service de base. Comme précédemment, pour faciliter l'étape de validation, nous avons rendu possible la consultation de la valeur de quatre variables internes à partir de l'environnement : l'état d'un poste logique, le numéro de son éventuel correspondant, l'état d'abonnement au service CFNR ($abCFNR_i$: le poste i est abonné), et le numéro de l'utilisateur destinataire du transfert (*CFNRparty[0]: le service est inactif; i*$\neq 0$, *CFNRparty[i]: le service est actif et le destinataire du transfert est le poste i*).

Comme pour le service de base, pour chaque méthode nous avons produit trois séquences enchaînées de 10 000 pas de test. Nous avons utilisé le même germe que précédemment.

Oracle

Les propriétés du système sont une extension des propriétés du POTS; on retient :

- propriétés d'oracle 1, 2 et 4 du POTS

  1. la variable *Party* est bien formée : elle indique un unique correspondant au plus; *ExactementUn(5,Party[0..4])*

2. les états d'un poste logique sont exclusifs;
   ExactementUn(7,Etat[0..6])
4. les sorties externes produites sont cohérentes avec l'état du poste logique;
   (*Mute* => Idle, *DialTone* => *Dialing* ...):
   $SExt_i[0] \Rightarrow Etat_i[0]$, $SEx_it[2] \Rightarrow Etat_i[1]$, ...

- modification de la propriété d'oracle 3 du POTS :

3. un poste logique produit au plus une sortie externe à chaque instant;
   AuPlusUn(11,SExt[0..10])

- ajout de propriétés spécifiques à CFNR :

5. la variable *CFNRparty* est bien formée : elle indique un unique destinataire ou pas de destinataire;
   AuPlusUn(5,CFNRparty[0..4]) and LIN_OR(5,CFNRparty[0..4])
6. un non-abonné au service ne peut pas obtenir l'activation du service;
   not *abCFNR* => not *CFNR_ActivOK*
7. les notifications et la variable *CFNRparty* sont cohérentes;
   (SExt[7..10] = CFNR_ActivOK, CFNR_ActivKO, CFNR_DesacOK, CFNR_DesacOK)
   abCFNR and *SExt[7]* => not *CFNRparty[0]*
   abCFNR and *SExt[8]* => *(*pre *CFNRparty = CFNRparty)*
   abCFNR and *SExt[9]* => *CFNRparty[0]*
   abCFNR and *SExt[10]* => *(*pre *CFNRparty = CFNRparty)*
8. les sorties externes produites par CFNR sont cohérentes avec l'état du poste logique;
   (*CFNR_ActivOK* or ... or *CFNR_DesacOK* => *Exception*:
   $LIN_OR(4,SExt_i[7..10]) \Rightarrow Etat_i[6]$

Contraintes d'environnement

1. Contraintes d'environnement stipulant que les entrées sont bien formées :

   - propriétés d'environnement 1a, 1b et 1c de l'environnement du POTS :

     (a) chaque événement a au plus un type;
     AuPlusUn(5,EExt[0..4])

     (b) le message *Dial* a un paramètre (une adresse);
     EExt[2] => ExactementUn(4,EExt[5..8])

     (c) les messages *On*, *Off* et *CFNRoff* n'ont pas de paramètres
     (EExt[0] or *EExt[1]* or *EExt[3]) =>* not *LIN_OR(4,EExt[3..7])*

   - ajout de contraintes spécifiques pour CFNR

     (d) le message *CFNRoff* n'a pas de paramètres
     EExt[3] => not *LIN_OR(4,EExt[3..7])*

Séquence → ↓ Nombre d'occurrences de	S1	S2	S3
On	2996	3009	2958
Off	2997	3010	2958
Dial	1016	1058	1036
CFNRon	1294	1267	1298
CFNRoff	255	249	245
transferts	70	50	57
triple transferts	0	0	0
erreurs	384	243	314
cas où *CFNR_actif[i]* (tous postes confondus)	22778	22532	22496
cas où les postes sont dans l'état R (tous postes confondus)	864	853	813

TAB. 9.3 – *Synthèse des données produites par la méthode de génération aléatoire équiprobable pour CFNR*

(e) le message *CFNRon* peut ou non avoir un paramètre;
EExt[4] => AuPlusUn(4,EExt[3..7])

2. Contraintes d'environnement décrivant les comportements valides des utilisateurs :

   – propriétés d'environnement 2a, 2b et 2c de l'environnement du POTS

   (a) on ne peut jamais effectuer deux fois de suite une action décrocher (resp. raccrocher) sans effectuer une action raccrocher (resp. décrocher);
   always_from_to(EExt[0], pre *EExt[1], EExt[1]),*
   always_from_to(EExt[1], pre *EExt[0], EExt[0])*

   (b) un usager ne peut raccrocher qu'après avoir décroché une fois;
   EExt[0] => after(EExt[1])

   (c) un usager ne peut composer un numéro que lorsqu'il entend le *DialingTone*;
   EExt[2] => pre *SExt[2].*

   – ajout de contraintes spécifiques pour CFNR

   (d) un usager ne peut composer un code CFNR que lorsqu'il entend le *DialingTone*;
   (EExt[3] or *EExt[4]) =>* pre *SExt[2].*

Génération aléatoire équiprobable

Nous avons utilisé un modèle comportant trois abonnés au service de transfert d'appels. La table 9.3 a été obtenue par comptage des actions des utilisateurs et de l'état des postes logiques pour les trois séquences générées.

Les erreurs observées ont été analysées. Elles sont toutes dues à des pertes de messages consécutives à la saturation du réseau. Le nombre d'erreur pour CFNR est plus faible que pour le service de base car le nombre d'actions des utilisateurs ne produisant pas de messages (dont *CFNRon* et *CFNRoff*) est plus important pour le service CFNR.

A partir des résultats de la table 9.3, on peut remarquer que le service CFNR est souvent actif, mais le nombre de transferts d'appel observés est faible. De plus, le nombre de triples transferts pour un appel est nul: les données ne permettent donc pas de s'assurer qu'un 4ème transfert n'est pas possible. On peut donc conclure que les données générées ne sont pas pertinentes (cf. page 40) pour tester le service de transfert d'appel sur non réponse.

Génération basée sur les profils opérationnels

Pour que le service soit invoqué plus souvent, nous avons spécifié des probabilités conditionnelles. Cela consiste en trois points :

- pour permettre l'invocation du service, un appelant doit rester suffisamment longtemps dans l'état *Ringing*;
- de même, pour permettre l'invocation du service, un abonné dont le service est actif ne doit pas décrocher souvent son téléphone, en particulier lorsqu'il est appelé;
- enfin, pour augmenter la probabilité d'un triple transfert d'appel (alors qu'il n'y a que 4 utilisateurs dont 3 abonnés), on peut favoriser l'établissement de boucles de transferts d'appel. Soit A, B, C les trois abonnés au service. On peut créer une boucle par exemple si A transfère ses appels vers B, et si B les transfère vers C et C les transfère vers A.

A partir des remarques ci-dessus, on propose les profils opérationnels suivants :

Profil opérationnel pour le poste non abonné au service CFNR
not *abCFNR*

Idle	
Off	0.1
rien	0.9

Exception	
On	0.9
rien	0.1

Dialing	
On	0.1
Dial	0.7
rien	0.2

Profil opérationnel pour les postes abonnés au service CFNR dans le cas où le service n'est pas actif
abCFNR and *CFNRparty[0]*

Idle	
Off	0.6
rien	0.4

Exception	
On	0.9
rien	0.1

Dialing	
On	0.1
Dial	0.2
CFNRoff	0.05
CFNRon	0.6
rien	0.05

Profil opérationnel pour les postes abonnés au service CFNR dans le cas où le service est actif
abCFNR and not *CFNRparty[0]*

Idle	
Off	0.1
rien	0.9

Exception	
On	0.9
rien	0.1

Ringing	
Off	0.01
rien	0.99

Ces profils opérationnels ont été transformés en un ensemble de probabilités conditionnelles. La table 9.4 dresse un comptage des actions pour les séquences de tests obtenues.

Sur une station Sparc Ultra-1 avec 128 Mo de RAM, la génération de 10 000 pas de test à l'aide de la méthode guidée par les probabilités conditionnelles (avec 42 triplets) prend environ 17 minutes dont 6 secondes CPU pour la production du générateur. Sans probabilité conditionnelles, il faut environ 6 minutes pour produire 10 000 pas de test, dont moins de 6 secondes CPU pour la production du générateur.

Conclusion

L'introduction de probabilités conditionnelles dans ce contexte avait pour but de favoriser l'apparition de transferts d'appels consécutifs à l'invocation du service CFNR. Sur ce point, les données obtenues réalisent l'objectif attendu. En effet, on obtient 591 transferts d'appels (pour S1) au lieu de 70 pour la méthode de génération équiprobable. Par ailleurs, on peut observer 41 triples transferts contre zéro pour la première méthode. L'augmentation du nombre de transferts est liée au nombre de cas où les postes sont dans l'état *Ringing* (3 095 au lieu de 864). L'obtention de ces résultats compense largement le temps supplémentaire demandé pour la génération des séquences en utilisant les probabilités conditionnelles.

En revanche, on peut constater que le nombre d'erreurs observées a particulièrement augmenté. L'analyse de ces erreurs a montré qu'elles étaient dues aux pertes de messages consécutives à la saturation du réseau, favorisée par l'augmentation du nombre de transferts d'appels.

Les tests effectuées n'ont révelé aucune erreur due à la non satisfaction des propriétés ajoutées pour CFNR.

Séquence → ↓ Nombre d'occurrences de	S1	S2	S3
On	2862	2866	2831
Off	2864	2868	2830
Dial	1471	1474	1474
CFNRon	915	912	868
CFNRoff	156	161	176
transferts	591	636	598
triple transferts	41	54	43
erreurs	1138	1447	1257
cas où *CFNR_actif[i]* (tous postes confondus)	28780	28487	28505
cas où un poste est dans l'état R (tous postes confondus)	3095	3542	3291

TAB. 9.4 – *Synthèse des données produites par la méthode de génération statistique pour CFNR*

pas	$EExt_i$(param)	$Party_{1..4}$	$Etat_{1..4}$	observation d'un transfert	$CNFRparty_{1..4}$	verdict
542	**Dial 4 (1)**	0 0 0 1	I I D W		**2** 1 1 0	OK
543	NRon 3 (1)	4 0 0 1	**R** I E W		2 1 1 0	OK
544	ON 3	4 0 0 1	**R** I I **A**		2 1 1 0	OK
545		4 0 0 1	**R** I I **A**		2 1 1 0	OK
546	**ON 4**	0 0 0 0	I I I I	**tr1**	2 1 1 0	OK
547	OFF 4	0 4 0 0	I **R** I D		2 1 1 0	OK
548	OFF 3	0 **0** 0 0	I **R** D D		2 1 1 0	OK

Au pas 542, l'usager 4 compose le numéro de l'abonné 1 (colonne 2). L'état du service de ce dernier est indiqué colonne 6 : le service est actif et les appels seront transférés vers l'usager 2.
Au pas 546, le service de 1 est invoqué et l'appel est transféré (colonne 5) vers le poste 2. Au même instant, l'usager 4 raccroche.
Au pas 547, le poste 2 reçoit l'appel transféré (son poste passe dans l'état *Ringing*).
Au pas 548, on peut constater un fait étrange : le numéro du correspondant du poste 2 est effacé mais le poste logique est toujours dans l'état *Ringing*.
Ce scénario illustre un problème de codage de CFNR. Une notification de fin de connexion (*dOn 4*) est parvenue au poste 2 au pas 548. Cette notification a été prise en compte pour le calcul du numéro du correspondant mais pas pour le calcul de l'état du poste logique.

TAB. 9.5 – *Commentaires sur un extrait de la séquence S1*

9.5 Bilan de cette étude de cas

Dans ce chapitre, nous avons étudié le problème de la validation d'un service supplémentaire. Nous avons adopté une démarche *incrémentale* de validation de spécifications de services.

Tout d'abord, les modèles des services sont ajoutés au fur et à mesure à celui du service de base (POTS). L'ajout de services implique des modifications d'interfaces, et par conséquent des modifications et des extensions de la définition de l'environnement et de l'oracle. La démarche de validation proposée consiste à utiliser la méthode de génération de base de Lutess, puis les méthodes guidées. Lorsqu'un nouveau service est inséré, les profils opérationnels précédemment utilisés sont modifiés pour prendre en compte les phases d'activation, de désactivation et d'invocation du service.

Les profils opérationnels ont ici démontré tout leur intérêt en permettant de focaliser l'analyse de chaque nouveau service sur les situations les plus intéressantes. Pour le service de base, l'utilisation des profils a permis de diminuer les situations de saturation de réseaux et a favorisé une analyse précise des erreurs restantes. Pour le service CFNR, l'observation de triples transferts d'appels a ainsi été possible; ce qui ne l'avait pas été avec la méthode de génération aléatoire.

Indépendemment des qualités respectives des méthodes de génération, cette étude de cas a confirmé que l'utilisateur devait être attentif lors de la construction des oracles. En effet, les services ont été validés par rapport à un ensemble de propriétés extraites d'une spécification informelle et établies "à la main".

A première vue, l'analyse des traces nous permet d'être satisfaits du modèle du service CFNR. En effet, aucune erreur n'est due à la non satisfaction des propriétés de CFNR, et le service est souvent invoqué (cf. le nombre de transferts réussis). On pourrait donc estimer le service valide.

Pourtant, certaines propriétés n'ont pas été établies. Par exemple, aucune propriété n'impose une cohérence entre le calcul du numéro du correspondant et l'état du poste logique. Une telle propriété pourrait être exprimé par la formule : "tout poste dans un état autre que *Idle* et *Exception* doit avoir un numéro de correspondant".

$$\mathsf{not}\ (Poste[0]\ \mathsf{or}\ Poste[6]) = LIN_OR(4,Party[1..4])$$

La table 9.5 illustre un problème, découvert après une analyse manuelle, qui peut être révélé à l'aide de cette propriété.

Enfin, après un examen a posteriori du travail effectué pour cette étude, nous pensons que ce modèle choisi est utilisable, mais qu'il ne présente pas de bonnes qualités pour un développement incrémental et séparé des services supplémentaires. En effet, notre adaptation Lustre du modèle proposé par Braithwaite et Atlee dans [15] nous a posé un problème majeur : il faut modifier les nœuds existants lorsque de nouvelles sorties sont ajoutées. Ainsi, pour chaque sortie externe ajoutée (*CFNR ActiveOK,..., CFNR DesactiKO*), il a fallu exhiber une équation dans le nœud codant le POTS. Cette constatation nous a poussé à choisir un autre mode de composition des services pour l'étude de cas suivante (voir chapitre 10).

Chapitre 10

Détection d'interactions entre services téléphoniques (sur une étude de cas fournie dans le cadre de conférence FIW'98)

10.1 Introduction

En marge de la conférence *Feature Interaction Workshop 1998* [61], un concours a été organisé, avec le support de Bell Labs, pour évaluer la capacité de différents outils et méthodes à détecter des interactions de services téléphoniques [45].

Ce concours portait sur l'analyse de douze services téléphoniques. Le concours s'est déroulé en deux phases : une première période de cinq mois (de février à juillet 1998) devait permettre d'analyser dix services et d'adapter éventuellement les méthodes au problème. Une deuxième phase plus courte (15 jours, en juillet 1998) devait permettre d'évaluer plus précisément les performances des outils, par l'analyse de deux nouveaux services ajoutés à l'ensemble existant.

L'objectif consistait à trouver le plus grand nombre d'interactions possibles, au niveau des spécifications, en étudiant tour à tour chaque paire de services possible. Une paire pouvait être constituée de deux instances d'un même service, ou d'instances de deux services distincts. Au total, 78 configurations étaient à considérer.

Les résultats à fournir à la fin de chaque phase devaient prendre la forme d'un ensemble de scénarios faisant apparaître chaque interaction. L'ensemble des services avait été préalablement analysé manuellement par le comité organisateur pour déterminer l'ensemble des interactions par rapport auquel les résultats de chaque participant devaient être évalués. Cette évalution prenait en compte le nombre d'interactions non existantes (non reconnues par le jury).

Dans ce chapitre, nous présentons le travail réalisé pour ce concours, en décrivant successivement les instructions du concours, la démarche de conception pour l'obtention d'un modèle exécutable, et l'utilisation de Lutess pour la validation de ce modèle et la détection des interactions [28, 30].

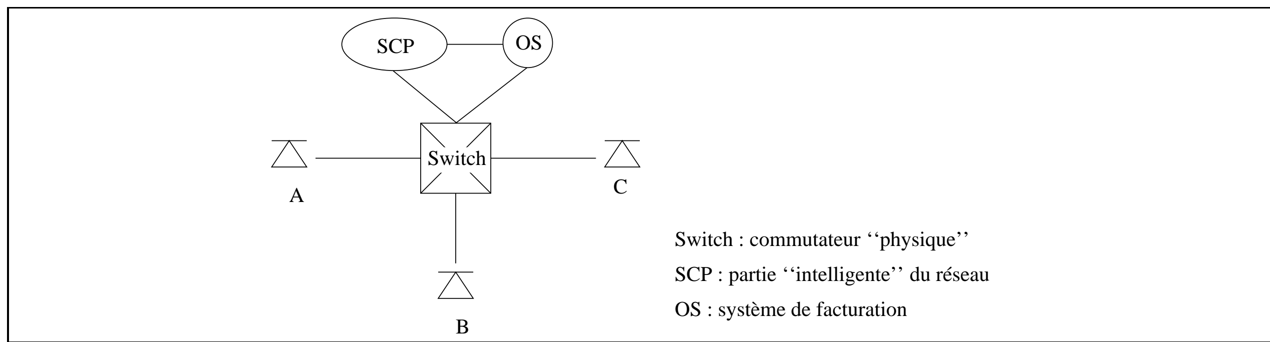


FIG. **10.1** – Architecture du système

10.2 L'énoncé du concours

Architecture du système

L'architecture du système de communication consiste en 4 éléments : l'ensemble des téléphones physiques des utilisateurs, un commutateur, un module gérant la partie intelligente du système appelé *point de contrôle de service* (SCP : *Service Control Point*) et un module de gestion de la facturation des communications (OS : *Operating System*).

Les liens de communication entre ces différents éléments sont représentés par la figure 10.1. Les messages transitant entre ces éléments sont précisément définis dans l'énoncé du concours.

Hypothèses simplificatrices

Six hypothèses simplificatrices ont été introduites :

1. Il n'existe pas d'autres messages que ceux définis par les instructions.
2. Les actions (messages) produites par un utilisateur sont traitées instantanément par le commutateur.
3. Une action *On-hook* (raccrocher) est immédiatement suivie d'un message *Disconnect*. Il n'y a pas de temps de latence pour prendre en compte la fin de la connexion.
4. Le réseau n'est jamais surchargé.
5. On ne tient pas compte des phases d'abonnement et de désabonnement, ni des phases d'activation et de désactivation d'un service. Les usagers sont ou non abonnés à un service et cela est défini statiquement.
6. Les téléphones sont équipés d'un bouton permettant d'exécuter l'action *flash*.

Formalisme de Spécification

Chaque service était décrit de manière informelle par quelques phrases en anglais, et plus formellement sous forme de *diagrammes de Chisel* [3], un langage spécialement développé pour la création de spécifications de services de télécommunication.

Un diagramme de Chisel est un graphe orienté, dont les nœuds sont des événements apparaissant aux interfaces. Un diagramme définit l'ensemble des séquences pour un appel. Chaque chemin (de la racine à une feuille) représente une séquence possible d'événements. Les séquences d'événements impliquant plusieurs appels peuvent être interfoliées et définissent alors l'activité du système global. La figure 10.2 illustre la définition formelle du service de base (*POTS : Plain Old Telephone Service*).

Chaque nœud du diagramme contient un ou plusieurs événements et/ou affectations de variables. Par ailleurs, chaque nœud est identifié par un entier. Les nœuds sont connectés entre eux par des arcs orientés. Lorsque plusieurs événements apparaissent dans un nœud, ils sont séparés par le symbole |||. Un nœud contenant plusieurs événements est équivalent au diagramme représentant toutes les séquences de ces événements ($A|||B$ signifie $\{AB, BA\}$).

Liste des services proposés

Phase I

- Call Forwarding Busy Line (CFBL) : transfert d'appel sur signal occupé. Ce service permet à l'abonné de faire transférer ses appels à un autre numéro lorsqu'il est occupé.

- Calling Number Delivery (CND) : présentation du numéro. Le souscripteur voit s'afficher sur son écran l'identité de l'appelant.

- IN Freephone Billing (FB) : c'est un service "intelligent" (IN), qui permet à l'abonné de payer les appels entrants (Numéro Vert, pour France Télécom) Le souscripteur est facturé en intégralité pour tous les appels lui étant destinés.

- IN Freephone Routing (FR) : c'est un service "intelligent" (IN), qui permet à l'abonné de faire transférer ses appels à un autre numéro, en fonction de l'heure ou du numéro de l'appelant.

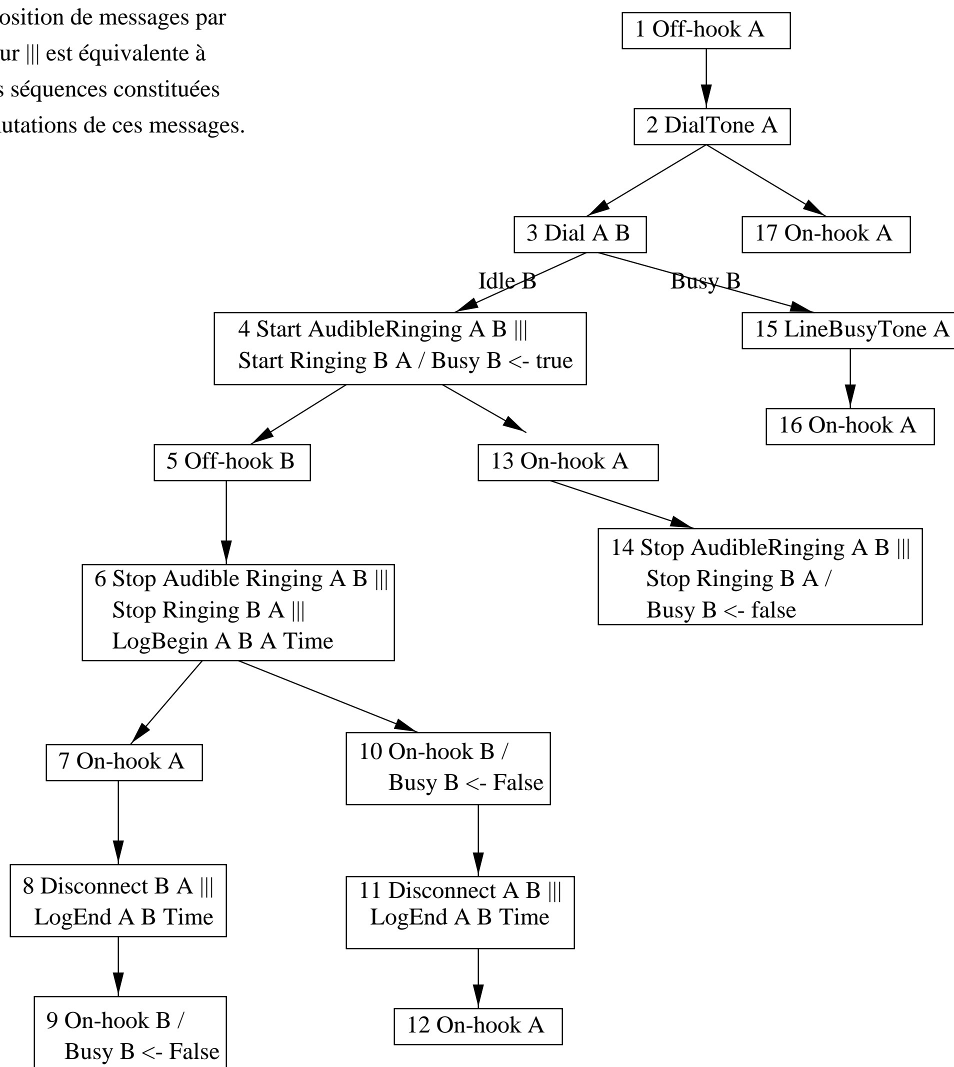
- IN Teen Line (TL) : c'est un service "intelligent" (IN), qui permet à l'abonné de bloquer les appels sortants en fonction de l'heure. Pendant les heures "filtrées" ou "interdites", il faut composer un code secret pour se servir du téléphone.

- Terminating Call Screening (TCS) : ce service permet à l'abonné de bloquer les appels entrants émis par les utilisateurs figurant sur une *liste noire*. La liste est définie par l'abonné. Ce service est un service de filtrage d'appel entrant.

- Three Way Calling (TWC) : c'est un service de conférence à trois. Ce service permet à son souscripteur de connecter un troisième usager à une communication déjà existante.

- IN Call Forwarding (CF) : c'est un service "intelligent" (IN), qui permet à l'abonné de faire transférer ses appels à un autre numéro de façon inconditionnelle.

- Call Waiting (CW) : signal d'appel. Le souscripteur de ce service est averti lors de l'arrivée d'un appel alors qu'il est déjà en communication. Il peut alors passer d'un correspondant à l'autre en utilisant un mécanisme de mise en attente.



Définition de variables associées au service

Busy A: true between an Off-hook A event and the next On-hook A event; between a Start Ringing A B event and the next Stop Ringing A B event, if no Off-hook A intervenes; or between a Start Ringing A B event and the next On-hook A.

Ringing A B: true between a Start Ringing A B event immediately following a Dial B A event and the next Stop Ringing A B event.

AudibleRinging A B: true between a Start AudibleRinging A B event immediately following a Dial A B event and the next Stop AudibleRinging A B event.

All of the Plain Old Telephone Service (POTS) event sequences start and end with Busy A = False (Idle A = True).

FIG. **10.2** – Exemple de description formelle : le service de base

- Charge Call (CC) : ce service permet à l'abonné de faire facturer ses appels à un autre abonné. Ce service est similaire à la "carte Pastel" de France Télécom.

Phase II

- Cellular (Cell) : il s'agit d'un service de facturation pour les téléphones mobiles. Le possesseur d'un téléphone mobile est facturé pour le temps passé en communication, qu'il soit à l'origine ou non de l'appel.
- Return Call (RC) : ce service permet à son souscripteur de rappeler automatiquement le dernier appelant dont l'appel n'a pas abouti.

Pour la plupart, ces services sont des simplifications de services réels offerts par les opérateurs de télécommunication. Parmi les services proposés, on peut constater 3 formes de transfert d'appel (CFBL, FR, CF), 3 types de refacturation (FB, CC, Cell), et 2 catégories de filtrage (TL, TCS).

La moitié des services proposés sont des services dits "intelligents" (FB, FR, TL, CF, CC, RC), qui utilisent les fonctionnalités du module SCP.

Chaque service a été développé de manière totalement indépendante des autres. Cette manière volontairement "naïve" de procéder a introduit de nombreuses possibilités d'interactions.

10.3 Conception d'un modèle exécutable

La phase de conception du modèle exécutable a comporté trois étapes distinctes. Dans un premier temps, nous avons produit un modèle général de système. Puis nous élaboré les modèles des services. Enfin, nous avons imaginé une méthode de composition des services. Dans cette section, nous décrivons successivement ces trois phases.

10.3.1 Modèle général du système sous test

Détermination du système sous test

Pour pouvoir appliquer Lutess, il faut déterminer les éléments composant le programme sous test et ceux constituant son environnement (figure 10.3(a)) :

- il est immédiat que le commutateur doit être considéré comme le cœur du système réactif et que les téléphones doivent faire partie de l'environnement,
- le système de facturation (OS) a été inclus dans l'environnement. En effet, il n'a aucune influence sur le fonctionnement du système, mais se borne à en recevoir des informations. Le concevoir hors du système permet de rendre visibles ces informations,
- le contrôleur de services (SCP) est placé dans le système sous test, car ses échanges avec le commutateur relèvent de la mise en oeuvre des communications.

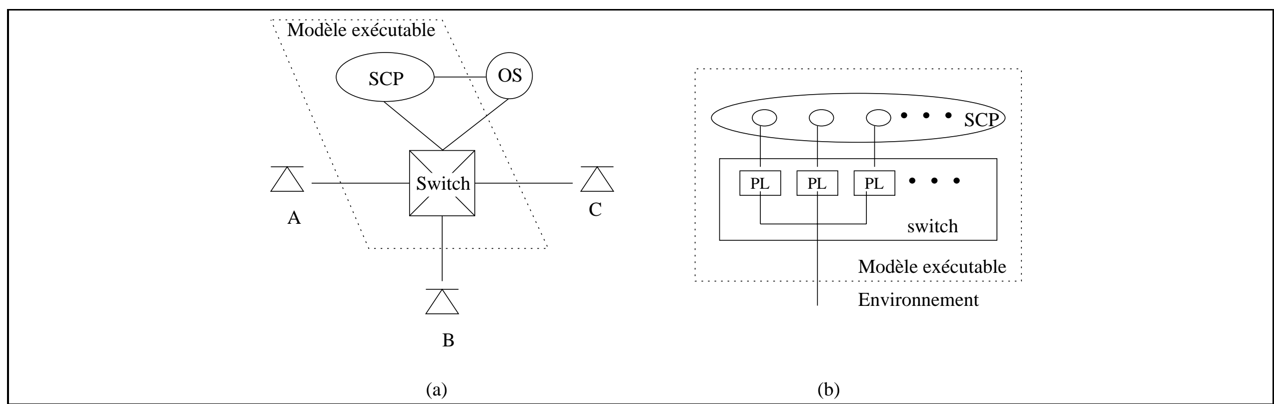


FIG. **10.3** – Système sous test

Architecture du modèle exécutable

L'architecture du modèle développé est basée essentiellement sur le commutateur. Comme pour l'étude de cas précédente, nous avons structuré le commutateur en *postes logiques* (fig. 10.3(b)). Ainsi, notre modèle de système téléphonique est facilement extensible (par rapport au nombre d'utilisateurs) et indépendant des services étudiés.

Par contre, nous avons opté pour une gestion "centralisée" du commutateur. Chaque poste logique connaît l'état des autres postes logiques à l'instant précédent, et reçoit tous les messages. Ainsi, les postes peuvent évoluer en parallèle sans communiquer, ce qui permet de produire les sorties instantanément, i.e. dans le cycle correspondant aux entrées.

Illustrons ce principe. Soient trois utilisateurs $\mathcal{A}, \mathcal{B}, \mathcal{C}$. Etudions le cas du service de base où $\mathcal{A}$ compose le numéro de $\mathcal{B}$. Le message *dial($\mathcal{A}$,$\mathcal{B}$)* est reçu par les 3 postes logiques. Le poste logique de $\mathcal{C}$ constate que ce message ne le concerne pas, et ne modifie donc pas son état.
Le poste logique de $\mathcal{A}$ se reconnaît comme étant l'appelant. Connaissant l'état du poste $\mathcal{B}$ à l'état précédent (inactif ou occupé), il calcule le nouvel état du poste logique et produit le message adéquat (*Start AudibleRinging($\mathcal{A}$, $\mathcal{B}$)* ou *LineBusyTone($\mathcal{A}$)*).
Parallèlement, $\mathcal{B}$ reçoit le message et s'identifie comme étant l'appelé. Selon son état (inactif ou occupé), il effectue les actions adéquates (changement d'état et production du message *Start Ringing($\mathcal{B}$, $\mathcal{A}$)* ou ne rien faire).

Le SCP et le commutateur échangent parfois des messages. Pour faire en sorte que cet échange soit instantané, nous avons codé le SCP sous la forme d'une fonction. Ainsi, lorsqu'un service fait appel au SCP, un poste logique calcule la réponse du SCP à l'aide de la fonction SCP associée au service. Cette façon de simuler présente l'avantage de ne pas prendre de temps. Ainsi, quel que soit le service, l'échange de messages entre un poste physique et le commutateur est fait dans le même cycle, qu'il y ait ou non appel au SCP.

Cette façon de simuler présente malheureusement l'inconvénient suivant : si le SCP est une ressource critique qui ne peut gérer plusieurs appels simultanément (ce n'est pas précisé dans l'énoncé du concours), nous pourrions alors "manquer" des interactions.

Cette modélisation a semblé acceptable. D'une part, la modélisation respecte les hypo-

thèses simplificatrices proposées par l'énoncé du concours. D'autre part, nous avons avons estimé que si notre conception du SCP masquait une interaction, celle-ci se manifesterait probablement d'une autre façon pendant l'appel.

Dans le commutateur, les postes logiques sont identiques. Ils comprennent le service de base et les services supplémentaires (au plus deux). Ces services sont conçus comme des fonctions autonomes et composées au sein du poste logique. Dans les paragraphes suivants, nous décrivons successivement l'obtention d'une fonction "codant" un service, et le mode de composition des services au sein du poste logique.

Codage de l'information

Nous avons utilisé le même principe de codage de l'information que pour l'étude de cas précédente (cf. §9.2.3).

Pour chaque type d'événements de l'énoncé, on associe un tableau. Chaque tableau est structuré en sous-parties. Le nombre de sous-parties est égal au nombre maximum de paramètres qu'un message peut contenir. La taille des sous-parties dépend de la quantité d'information à coder. Pour chaque sous-partie, on prévoit un tableau booléen et on utilise un codage "un parmi n": il y a au plus un booléen par sous-partie vraie à chaque instant. L'absence d'information est codée par un message (dit "no_message") qui est traduit par un tableau dont tous les éléments sont faux.

Contrairement au modèle proposé au chapitre 9, les messages impliquant les postes logiques contiennent un champ permettant d'identifier l'émetteur et le destinataire du message. Dans le cas précédent, ceci était inutile puisque chaque poste logique communiquait directement avec son poste physique (cf. fig. 9.1, page 112) alors que ce n'est plus le cas ici (cf. fig. 10.3, page 138).

10.3.2 Modèle d'un service

Les diagrammes de Chisel sont directement utilisés pour modéliser les services sous forme d'automates d'entrées/sorties, dont les entrées sont les actions possibles des usagers. Cette traduction se fait en trois étapes de manière systématique :

1. On identifie le nombre d'usagers impliqués dans le fonctionnement du service. Ce nombre permet de définir le nombre de rôles que peut avoir un poste pour un service donné, et donc le nombre de "modes de fonctionnement" du service. Le service de base permet d'identifier ainsi un rôle *appelant* et un rôle *appelé*. Pour un service de transfert d'appel, il existe un troisième rôle : la *cible du transfert*.

2. Le diagramme de Chisel du service est ensuite projeté sur chacun de ces rôles, de manière à déterminer quels événements concernent quels rôles. On obtient ainsi un automate par mode de fonctionnement. La figure 10.4 décrit les deux rôles du service de base.

3. Tous ces automates sont regroupés dans un même module qui constitue la modélisation du service. Chacun de ces automates évolue de manière indépendante. Il est possible

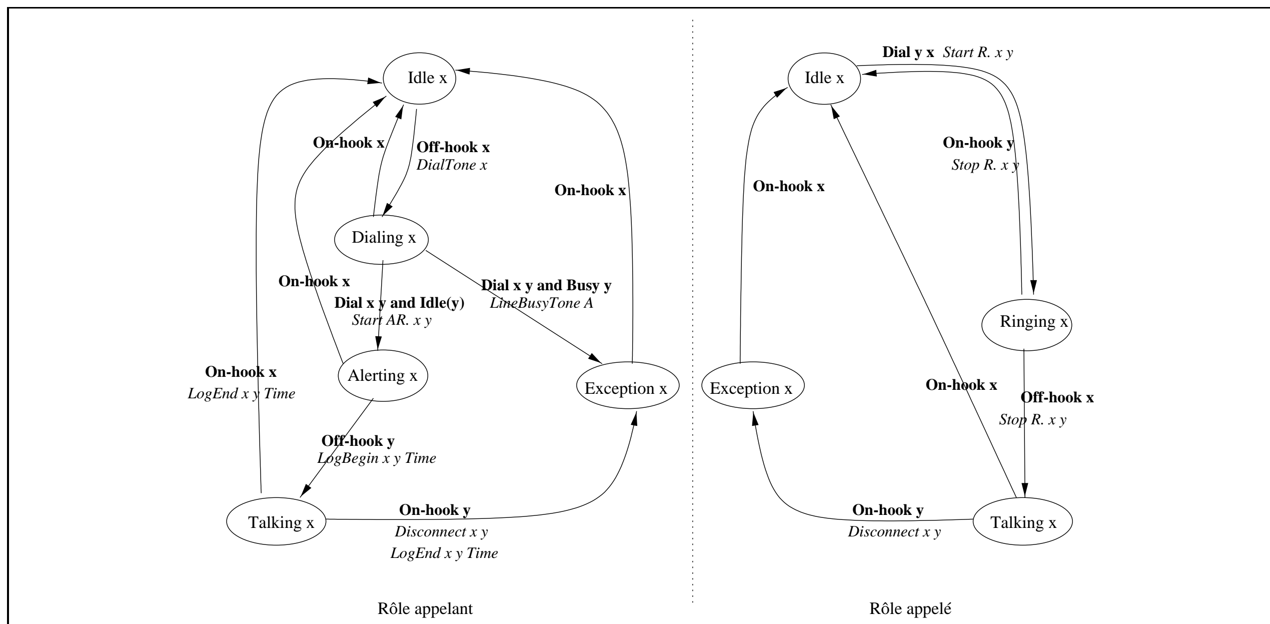


FIG. **10.4** – Automate du service de base En **gras** est représentée la condition que doit satisfaire l'entrée pour que la transition soit choisie ; en *italique* apparaissent les sorties associées à chaque transition.

pour un même usager d'occuper des rôles différents pour plusieurs instances d'un même service. En revanche, et c'est là une des limitations de notre modélisation, il n'est pas possible pour un usager d'avoir plusieurs rôles identiques pour un même service. Par exemple, un usager transférant un appel ne peut pas transférer d'autres appels tant que celui en cours n'est pas terminé.

Il aurait également été possible de ne pas recombiner les automates associés aux divers rôles d'un même service, et de les considérer chacun comme la spécification d'un service différent. Nous avons fait ce choix uniquement pour rendre plus clair le mode de composition des services.

La figure 10.4 décrit le service de base. Les états de cet automate correspondent à ceux du service de base utilisé précédemment (figure 9.5). Seul l'état *Waiting* n'est plus présent. Cet état correspondait à la phase d'établissement de la communication, qui est désormais instantanée.

10.3.3 Composition de services

L'énoncé du concours ne décrit pas précisément l'opération de composition des services. Nous avons donc choisi une composition qui nous a paru "raisonnable". Nous avons abandonné l'opération de composition précédente s'inspirant de [15] car elle ne permet pas vraiment de développer les services indépendamment des autres.

Description Informelle

Intuitivement, le principe de l'opération de composition est le suivant. Soit le service de base et un ou deux services supplémentaires. Ces services sont codés par des automates. L'état du poste logique est une variable globale du poste logique et chaque automate y a accès. A chaque cycle, chaque automate reçoit les entrées du poste logique et propose un nouvel état et un ensemble de sorties, en fonction de l'état courant du poste logique. L'opérateur de composition choisit et envoie l'une des réponses proposées, et modifie l'état du poste logique en conséquence.

Nous avons utilisé deux opérations de composition $\oplus_1$ et $\oplus_2$. La première opération introduit une priorité entre les services supplémentaires. La seconde opération n'en impose pas. Pour chacune des deux opérations, le service de base est moins prioritaire que les services ajoutés; i.e. lorsqu'un service ajouté propose une réponse, elle sera choisie quelle que soit la proposition du service de base.

Pour distinguer les cas où les services sont actifs des cas où ils ne le sont pas, chaque fonction codant un service supplémentaire f possède une sortie booléenne a_f. Cette sortie est vraie lorsque le service propose une sortie. Cette variable a été introduite pour distinguer les cas où (1) le service est inactif (et donc propose "no_message") des cas où (2) le service est actif et propose comme réponse "no_message".

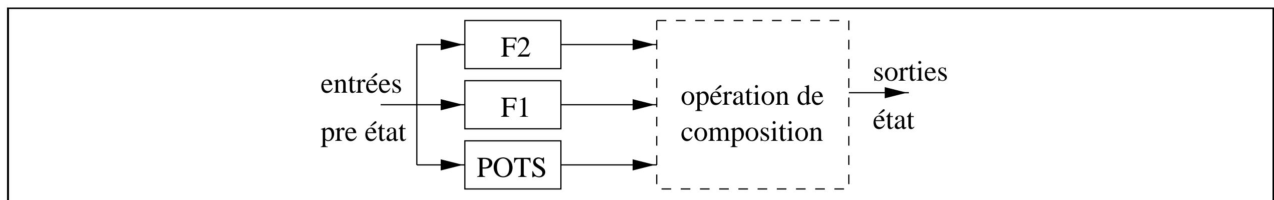


FIG. **10.5** – Principe de l'opération de composition

Formalisation

Le service de base et les services ajoutés sont représentés par des fonctions partielles. Soit σ_{pots} la fonction réalisant le service de base. Le profil de σ_{pots} est :

$$\sigma_{pots} : \ V_{SG} \times V_E \rightarrow V_{SP} \times V_S$$

où:

- E est l'ensemble des messages entrants (produits par l'environnement)[1],
- S est l'ensemble des messages sortants (émis vers l'environnement),
- SG est l'ensemble des variables globales $SG = SP_1 \times SP_2 \times ... \times SP_m$ où m est le nombre de poste logiques et SP_i est l'ensemble des variables globales du poste logique i. SG est l'état global du *Switch*.

Cette définition appelle trois commentaires :

- Le comportement du service de base (σ_{pots}) est le même pour tous les utilisateurs.

1. On rappelle que si X représente un ensemble de variables, V_X représente l'ensemble des instanciations possibles de ces variables.

- Pour un poste logique donné, SP est l'ensemble des variables communes au service de base et aux services ajoutés.
- Cette fonction prend en compte l'aspect centralisé du modèle exécutable.

Soit σ_f un modèle exécutable d'un service f. Le profil de σ_f est :

$$\sigma_f : V_{SG} \times V_{Sf} \times V_E \to V_{SP} \times V_{Sf} \times a_f \times V_S$$

où Sf est l'ensemble des variables (locales) associées au service f, a_f est le booléen indiquant si le service est actif, et E, S, SP et SG sont définis comme ci-dessus.

Soit $\frac{F_1 \oplus_1 \ldots \oplus_1 F_n}{POTS}$ la composition du service de base et de n services $F_1 \ldots F_n$. L'opération de composition $\oplus_1$ introduit une priorité entre les services. Ici, F_1 est plus prioritaire que les autres. F_n est le moins prioritaire. Le modèle exécutable σ_m résultant de cette opération est défini de la façon suivante :

$$\left\{ \begin{array}{l}
\sigma_m : V_{SG} \times \bigcup_{i=1}^{n} V_{sf_i} \times V_E \to V_{SP} \times \bigcup_{i=1}^{n} V_{sf_i} \times V_S \\
\\
\sigma_m(e_g, e_{f_1}, ..., e_{f_n}, ve) = \\
\qquad \text{soit} \\
\qquad\quad (e'_p, s') = \sigma_{pots}(e_g, ve) \\
\\
\qquad\quad \forall i \in \{1..n\}, (e''_{p_i}, e''_{f_i}, a_{f_i}, s''_i) = \sigma_{f_i}(e_g, e_{f_i}, ve) \\
\qquad \text{dans} \\
\qquad\quad \text{si } a_{f_1} \text{ alors } (e''_{p_1}, \bigcup_{i=1}^{n} e''_{f_i}, s''_1) \text{ sinon} \\
\qquad\quad .... \\
\qquad\quad \text{si } a_{f_n} \text{ alors } (e''_{pn}, \bigcup_{i=1}^{n} e''_{f_i}, s''_n) \text{ sinon} \\
\\
\qquad\quad (e'_p, \bigcup_{i=1}^{n} e''_{f_i}, s')
\end{array} \right.$$

A l'expérience, $\oplus_1$ présente deux défauts. D'un point de vue technique, l'introduction d'une priorité entre les services impose la vérification de toutes les combinaisons possibles de ces services. Par exemple, lorsque l'on considère les deux services F_1 et F_2, il faut observer $\frac{F_1 \oplus_1 F_2}{POTS}$ et $\frac{F_2 \oplus_1 F_1}{POTS}$

Le second défaut de $\oplus_1$ est que si un service est actif, il inhibe les autres qui sont moins prioritaires. Intuitivement et très grossièrement, on obtient que “tout service interagit avec tous les autres”. Prenons l'exemple des services CND et CF. Lorsque l'on observe $\frac{\text{CND} \oplus_1 \text{CF}}{POTS}$, on constate qu'un abonné aux deux services ne peut transférer ses appels. En effet, à chaque appel, CND est actif (présente le numéro) et empêche donc le transfert; il s'agit d'une interaction qui n'existe pas dans les spécifications du concours.

Lorsque l'on observe $\frac{\text{CF} \oplus_1 \text{CND}}{POTS}$, on constate que CND est inhibé à chaque transfert d'appel. Chez l'abonné le numéro de l'appelant ne s'affichera pas lorsque ses appels sont transférés. A priori, ce comportement paraît normal; mais il s'agit d'une interaction qui existe dans les spécifications du concours et qui devrait donc être observable.

Pour la seconde opération $\oplus_2$, soit $\frac{F_1 \oplus_2 ... \oplus_2 F_n}{POTS}$ la composition du service de base et des n services $F_1 \ldots F_n$. Le modèle exécutable σ_m résultant de cette opération est défini de la façon suivante :

$$
\left\{
\begin{array}{l}
\sigma_m : \; V_{SG} \times \bigcup_{i=1}^{n} V_{sf_i} \times V_E \rightarrow V_{SP} \times \bigcup_{i=1}^{n} V_{sf_i} \times V_S \\
\\
\sigma_m(e_g, e_{f_1}, ..., e_{f_n}, ee) = \\
\qquad \text{soit} \\
\qquad\quad (e'_p, s') = \sigma_{pots}(e_g, ee) \\
\\
\qquad\quad \forall i \in \{1..n\}, (e''_{p_i}, e''_{f_i}, a_{f_i}, s''_i) = \sigma_{f_i}(e_g, e_{f_i}, ee) \\
\qquad \text{dans} \\
\qquad\quad \text{si } \exists i, af_i = true, \text{ alors } (\bigvee_{j, af_j = true} e''_{p_j}, \bigcup_{i=1}^{n} e''_{f_i}, \bigvee_{j, af_j = true} s''_n) \text{ sinon} \\
\\
\qquad\quad (e'_p, \bigcup_{i=1}^{n} e''_{f_i}, s') \\
\\
\qquad \text{avec} \\
\qquad\quad \bigvee : \text{l'opérateur logique ``ou_bit_à_bit''}
\end{array}
\right.
$$

$\oplus_2$ a été conçu pour pallier les manques de $\oplus_1$. Lorsque deux services (ou plus) sont actifs, leurs réponses sont composées à l'aide de l'opérateur logique "ou_bit_à_bit" sur chaque type de message. Par exemple, lorsque deux services produisent "no_message", le résultat obtenu est "no_message" (fig 10.6a). lorsqu'un service produit "no_message" et l'autre un message m, le message obtenu est m (fig 10.6b). Enfin, lorsque les services produisent respectivement m_1 et m_2 différents, le résultat obtenu est un message "indéfini".

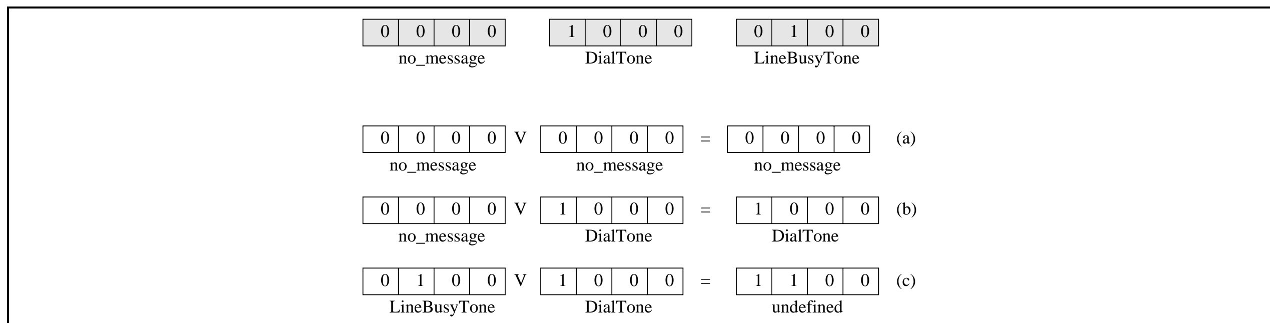


FIG. **10.6** – Composition de messages avec $\oplus_2$

Lorsque l'on utilise $\oplus_2$, l'apparition d'un message "indéfini" révèle une incohérence (une interaction) entre deux services. Il est nécessaire de définir précisément quels sont les messages "définis" et les messages "indéfinis". Le codage des messages ("Au_plus_un_bit" par sous-partie de message) est bien adapté à l'utilisation de $\oplus_2$.

10.4 Validation de services et détection d'interactions

Nous avons extrait les propriétés des services à valider à partir de leur description informelle. Par exemple, la phrase suivante décrivant le service TCS *"Calls from lines that appear*

on a screening list are redirected to a vague but polite message." a été traduite par :

$$\left\{\begin{array}{l}\triangleright \text{TCSsub(y) and ConnectRequest(x,y) and IsScreenedBy(x,y)} \Rightarrow \\ \qquad \text{AnnounceScreenedMsg(x)} \\ \text{où} \\ \{\ \text{TCSsubs(x)} \Leftrightarrow \text{x est abonné au service TCS.}\ \} \\ \{\ \text{IsScreenedBy(x,y)} \Leftrightarrow \text{x appartient à la liste noire de y (Screened(y)).}\ \} \\ \{\ \text{AnnounceScreenedMsg(x)} \Leftrightarrow \text{x obtient "un message vague mais poli".}\ \} \\ \{\ \text{ConnectRequest(x,y)} \Leftrightarrow \text{x demande une connexion vers y.}\ \} \end{array}\right.$$

10.4.1 De la détection d'interaction

Notion d'interaction

L'énoncé du concours ne définit pas la notion d'interaction. Informellement, on définit l'interaction de services de la façon suivante. Pour chaque service, on peut exprimer un ensemble de propriétés, décrivant par exemple les attentes des utilisateurs. Lorsqu'un service (ou une instance d'un service) fonctionne seul(e) avec le service de base, ces propriétés sont satisfaites.

Soient deux services $\mathcal{S}_1$ et $\mathcal{S}_2$ et leurs propriétés respectives P_1 et P_2. On dit qu'il y a une interaction lorsque pris séparément $\mathcal{S}_1$ et $\mathcal{S}_2$ satisfont leurs propriétés respectives et que, une des propriétés (au moins) n'est plus satisfaite lorsqu'ils sont considérés ensemble. Cette définition s'appuie sur les travaux de Combes et al. [23].

De façon formelle, on définit une interaction entre n services de la façon suivante, où $\oplus$ représente l'une des deux opérations de composition données ci-dessus.

$$\left\{\begin{array}{ll}\frac{F_i}{POTS} \models P_i, 1 \leq i \leq n & \text{(a)} \\ \frac{F_1 \oplus \ldots \oplus F_n}{POTS} \not\models P_1 \wedge \ldots \wedge P_n & \text{(b)}\end{array}\right. \qquad \text{(T)}$$

La relation de satisfaction $\models$ est celle utilisée pour l'étude de cas précédente, avec toute l'incertitude qu'elle comporte. Ainsi, c'est à l'opérateur humain (le testeur ou le client) de décider si un service (ou un ensemble de services) satisfait une propriété.

La définition ci-dessus doit être complètée pour prendre en compte le cas de l'interaction d'un service avec lui-même. En effet, lorsque pour un service donné F_j, (Ta) n'est pas satisfaite, il est possible que F_j ou P_j ne soient pas bien spécifiées ou que F_j ne satisfait pas P_j. Dans ce dernier cas, on peut en conclure que le service interagit avec lui-même.

Méthodologie de détection

La détection s'effectue de façon incrémentale, en deux étapes :

- Une première consiste à valider chaque service séparément, comme s'il était seul disponible en plus du service de base. Cette première étape permet de s'assurer que le service satisfait bien les propriétés qu'on en attend. C'est également l'occasion de corriger éventuellement la modélisation ou l'expression des propriétés attendues.

- La deuxième phase repose sur l'analyse de chaque paire de services et vise à identifier les problèmes d'interactions que peut poser leur coexistence. On s'appuie pour cela sur l'ensemble des propriétés attendues de chacun.

Le fait de ne pas trouver de mise en défaut d'un oracle donné ne signifie pourtant pas forcément que la situation soit exempte d'interactions, car il se peut qu'aucune des propriétés ne soit adaptée à révéler le problème.

Pour pallier cet inconvénient, le testeur peut remplacer les propriétés d'oracle par des observateurs, c'est-à-dire des propriétés *révélatrices de situation*. Grâce à ces dernières, le testeur peut concentrer son analyse sur certaines situations dignes d'intérêt.

Typiquement, une propriété de situation peut être construite à partir des prémisses d'une implication. Par exemple, à partir de la propriété (TCS) on peut construire l'oracle de situation :

$$\mathcal{R}(y,x) = \text{TCSsub}(y) \text{ and } \text{ConnectRequest}(x,y) \text{ and } \text{IsScreenedBy}(x,y).$$

$\mathcal{R}(y,x)$ permet de détecter les cas où un abonné à TCS est appelé par un usager figurant dans sa liste noire.

Une propriété de situation peut aussi décrire une situation relativement rare ou inattendue. Par exemple, dans le cadre de la modélisation du service de base, il est convenu que tout poste logique désigné comme correspondant dans une communication ait lui-même un correspondant ; si cette convention n'est pas vérifiée, il se peut que cela ne révèle pas de problème car certains services outrepassent cette règle sans qu'une interaction n'apparaisse. C'est au testeur humain de juger, le révélateur se bornant à localiser et à désigner les différents points d'apparition de la situation dans les traces.

10.5 Bilan de l'étude de cas avec Lutess

Effort de test

- L'étude de cas a requis un effort global de 136 hommes*jours qui peut se détailler comme suit :
  - 10 hommes*jours pour produire le modèle exécutable du système,
  - 3 hommes*jours pour produire la spécification exécutable de chaque service, soit 3*12 = 36 hommes*jours,
  - 1 homme*jour pour déterminer les propriétés de chaque service et produire l'oracle correspondant, soit 12 hommes*jours,
  - 1 homme*jour pour tester chaque paire de services, soit 78 hommes*jours.
- L'essentiel de l'effort de programmation a porté sur la réalisation du modèle et a requis la production de 5000 lignes de code Lustre.
- L'effort de validation quant à lui a consisté en la génération de 10 à 20 séquences de test de 1000 à 10000 pas de longueur par paire de services à tester. Sur l'ensemble, Lutess a été exécuté plus de 1500 fois et a produit en moyenne un million d'entrées par paire de services.

Performances de l'outil

La technologie d'implantation choisie (à base d'arbres de décisions binaires [4]) permet de calculer de manière statique l'espace des états de l'environnement. La phase de génération proprement dite est donc considérablement allégée. À titre d'illustration, le calcul de l'espace d'états requiert jusqu'à 30 minutes sur une station Sparc Ultra-1 avec 128 Mo de RAM, pour les contextes de test les plus complexes. L'exécution de 1000 pas de test (dont la génération de données ne constitue qu'une partie) s'effectue en 2 minutes.

Utilisation des méthodes de guidage

La méthode de génération de base a toujours été utilisée pour une première étape de validation. Les résultats de cette étape nous ont paru suffisant pour des paires de services simples, telles que CND+CND, CND+TCS, TCS+TCS ...

Pour des services plus complexes, les probabilités conditionnelles ont été utilisées, d'une part pour rendre la simulation réaliste et d'autre part, pour faciliter l'invocation des services. Par exemple, pour les services TL, CC, Cell, les probabilités conditionnelles ont été choisies de manière à ce que les abonnés cherchent à établir une connexion aussi souvent que possible. Pour les services CFBL, FR, CF, TCS, les probabilités conditionnelles ont été choisies de sorte que les utilisateurs appellent de préférence un abonné au service.

Les probabilités conditionnelles se sont révélées peu pratiques à utiliser pour provoquer l'invocation de services complexes tels que TWC ou CW. En effet, la longue succession d'actions nécessaire à leur invocation impliquait un grand nombre de triplets. Par ailleurs, la durée de génération des données en était affectée. Une autre méthode de test a été utilisée [29, 78].

La méthode de génération guidée par des propriétés n'a pas été utilisée. Il lui a été préféré la dernière méthode, plus puissante et plus facile à mettre en œuvre.

Résultats

Nous avons découvert 82 interactions [28, 30]. Ces interactions sont décrites complètement dans le rapport technique [31]. La table 10.1 résume les interactions trouvées.

Lutess a été déclaré “meilleur outil”, vainqueur de la compétition. Sur l'ensemble des 78 situations de test, notre outil a révélé environ 75% des interactions recensées par le jury. Un certain nombre d'interactions à détecter sont donc absentes de nos résultats, tandis que plusieurs autres ont été détectés. D'autres enfin se sont manifestées sous une forme différente de celle attendue.

En ce qui concerne les interactions non détectées, la différence de résultats provient d'une compréhension différente de ce qui constitue l'essence du service et l'attente de l'usager. En nous basant sur la seule description informelle des services, nous n'avons pas testé certaines fonctionnalités qui nous ont semblé négligeables pour mieux nous consacrer à l'étude de leurs caractéristiques *essentielles*. Le choix du jury a ainsi divergé plusieurs fois du nôtre.

	CFBL	CND	FB	FR	TL	TCS	3WC	CF	CW	CC	CELL	RC
CFBL	1	2	3,4,5	6,7,8	9	10,11	12	13,14,15	16,17	18	19,20	21, 22
CND	-	x	x	23	x	24	25	26	27	28	x	29
FB	-	-	x	30	x	x	31	32, 33	34	35	36,37	x
FR	-	-	-	38,39	x	40,41	42	43,44,45	46	47	48,49	50,51
TL	-	-	-	-	52	x	53	54	x	x	x	55
TCS	-	-	-	-	-	x	56	57, 58	59	60	x	61
3WC	-	-	-	-	-	-	62	63	64	65	66,67	68
CF	-	-	-	-	-	-	-	69	70	71	72,73	74,75
CW	-	-	-	-	-	-	-	-	x	76	77,78	79
CC	-	-	-	-	-	-	-	-	-	x	80,81	82
CELL	-	-	-	-	-	-	-	-	-	-	x	x
RC	-	-	-	-	-	-	-	-	-	-	-	x

Dans ce tableau, on examine chaque paire de service. On ne considère pas d'ordre de priorité entre les services. Les interactions trouvées sont numérotées de 1 à 82, et 'x' représentent les cas où aucune interaction n'a été trouvée.

TAB. 10.1 – *Synthèse des interactions trouvées pour le concours*

Par exemple, nous n'avons pas jugé essentiel de tester la bonne terminaison de chaque appel, alors que le jury a considéré cette notion comme primordiale.

Pour ce qui est des interactions non-recensées par le jury, elles sont de deux natures :

- Certaines ne peuvent pas se produire dans le modèle décrit par l'énoncé, mais peuvent exister dans la réalité [10]. Le fait de les avoir détectées avec notre outil résulte d'une interprétation légèrement différente du modèle. Leur détection n'était pas pénalisante pour le résultat du concours, du fait de leur réalisme.

- D'autres sont tout simplement inexistantes et résultent d'une propriété d'oracle injustifiée. Plus précisément, de telles détections ont résulté de la définition d'attentes de l'usager trop fortes par rapport à la description des services. C'est le cas par exemple pour le service TeenLine (service de restriction d'appel suivant l'heure de la journée). Une propriété attendue du service est d'empêcher que l'usager puisse initier un appel sous certaines conditions. On pourrait supposer que le souhait implicite du souscripteur est de ne pas avoir d'appel facturé sous ces mêmes conditions. Cette dernière propriété, qui a mené à la détection de plusieurs interactions, a été considérée comme trop forte par rapport à la description du service.

Enfin, dans plusieurs cas, nous avons détecté une interaction attendue, mais se manifestant de manière différente de ce que prévoyait le jury. Ceci concerne uniquement des services "intelligents" et s'explique par la simplification que nous avons effectuée relativement au SCP. Cette simplification "décale" l'apparition des interactions censées se produire au niveau du SCP. De telles interactions sont observables au niveau du commutateur, avec un certain retard.

Auteurs	Langage d'expression des propriétés	Langage d'expression des services
F.J. Lin et Y.-J. Lin [66]	Formules de logique temporelle (exprimées en Promella)	Promella
P. Combes et S. Pickin [23]	LTL	SDL
M. Faci et L. Logrippo [39]	LOTOS	LOTOS
B. Mermet et D. Méry [72]	B et TLA+	B
M. Plath et M. Ryan [83]	CTL	SMV

TAB. 10.2 – *Quelques approches à deux modèles pour la détection d'interactions*

10.6 Travaux similaires

La méthode de détection d'interaction que nous avons développée, repose sur les points suivants :

- une modélisation exécutable du système de télécommunication et des services qu'il contient, en adoptant un point de vue réactif et synchrone;
- une recherche automatisée des interactions grâce à Lutess.

10.6.1 Détection d'interactions et spécification de services

La détection d'interactions au niveau des spécifications des services est une approche largement étudiée, qui s'appuie en général sur des modèles et des techniques outillées de vérification-validation, sans lesquels la complexité du problème est impossible à maîtriser. Pour entièrement automatiser la détection, de rares travaux ont consisté soit à créer des langages dédiés pour la création et la validation de services [52, 90], soit à produire une définition formelle de l'interaction [9, 41]. Ces derniers ont abouti à un cadre théorique spécifique dans lequel la notion d'interaction est simplifiée; en conséquence de nombreuses interactions ne peuvent être découvertes.

Nous avons choisi une approche de détection d'interaction s'appuyant sur deux niveaux de spécification différents : les propriétés attendues du service et la description de ses fonctions. Cette approche a déjà été mise en œuvre avec succès dans [23, 66, 39, 72, 83]. Ces travaux se différencient les uns des autres surtout les choix de langages d'expression des propriétés et des spécifications fonctionnelles.

Un autre type d'approche consiste à ne travailler qu'avec un seul niveau de description pour exprimer toute les caractéristiques des services [9, 41, 15]. La détection d'interaction consiste alors généralement à rechercher des incohérences dans le modèle produit. Par exemple, dans [15], les services sont conçus comme des machines à transition d'états. La composition des services correspond à un empilement des machines. Les interactions sont détectées par analyse du graphe d'états de la machine composée.

Jusqu'à présent, aucun travail n'a montré la supériorité de l'une des approches ou de l'un des modèles pour la détection d'interactions. On aurait pu penser que le concours FIW'98

donnerait une indication sur ce point, mais ce ne fut pas le cas. Les interactions trouvées par les compétiteurs étaient similaires [10]. En fait, le concours était destiné à évaluer la *capacité des outils* à detecter des interactions et c'est la seule information qui peut en être retiré (en terme de performances).

10.6.2 Spécification de services et approche synchrone

L'utilisation de l'approche synchrone pour la spécification et l'implantation de services téléphoniques a été relatée dans [73, 56]. Ici, la notion de service téléphonique est prise au sens large. Dans [73], il s'agit du service de base, et dans [56], il s'agit d'une fonctionnalité d'un commutateur d'AT&T.

[73] décrit une expérience qui a consisté à réécrire le programme d'un protocole "terminal". Le programme initial était écrit en C. La spécification a été réécrite en ESTEREL, puis compilée en C à l'aide du compilateur ESTEREL. L'utilisation d'un langage synchrone pour coder un tel protocole ne semble pas avoir posé de problème particulier.

[56] décrit l'utilisation de ESTEREL pour le codage d'une fonctionnalité d'un commutateur d'AT&T. ESTEREL est utilisé comme un langage de description haut niveau. Le code final du programme a été produit automatiquement par le compilateur ESTEREL. En utilisant cette approche, les auteurs ont particulièrement apprécié les points suivants :

1. la sémantique formelle d'ESTEREL a permis la preuve formelle de certaines propriétés sur le programme,
2. il a été possible de raisonner à un haut niveau d'abstraction puis de générer automatiquement le code C du programme,
3. il a été possible de structurer, de raffiner et de prototyper le programme de façon simple et pratique.

10.6.3 Détection d'interactions et validation par le test

Dans [57], Jagadeesan et al. décrivent leur outil de test et relatent une expérience utilisant cet outil dans le cadre des systèmes téléphoniques. L'outil qui a été développé fonctionne sur le même principe que Lutess. Dans un premier temps, l'outil construit un exécutable à partir d'un programme réactif synchrone à tester, de la description des entrées-sorties, des propriétés de sûreté, et d'un oracle. Cet exécutable génére automatiquement des données, qui sont testées.

L'expérience décrite par Jagadeesan et al. dans [57] a consisté à utiliser leur outil de test pour valider plusieurs implémentations d'une fonctionnalitée d'un commutateur d'AT&T. En utilisant cet outil, les auteurs ont particulièrement apprécié la facilité d'utilisation de leur outil et son efficacité pour la détection de défaut.

L'approche proposée par J. C. Godskesen dans [44] définit formellement la notion de test permettant de conclure à la présence d'interaction. La démarche de l'auteur se décompose en deux étapes.

Tout d'abord, J. C. Godskesen décrit un cadre formel pour la détection d'interactions. L'auteur considère trois niveaux de description (les besoins, la spécification, l'implantation). A chaque niveau est associée une définition formelle de la notion d'interaction entre deux services.

Puis, l'auteur décrit une approche formelle pour tester la présence d'interaction au niveau implantation, dont le principe est le suivant. Le résultat de l'application d'un test à une implantation est décrite par une fonction booléenne indiquant si le test est réussi ou non. Un test est dit *valide* s'il n'échoue que lorsque l'implantation est défectueuse. Il est alors possible de conclure à la présence d'une interaction entre deux services lorsqu'un test valide échoue pour la combinaison des services.

Dans [59], Kelly et al. décrivent leur méthode pour la validation d'implantations. Leur démarche consiste tout d'abord à développer une classe de modèles abstraits, premettant de décrire les services. Puis, l'environnement des services est décrit. Il consiste essentiellement dans la description du modèle de base. Chaque nouveau service est ensuite modélisé et composé avec les autres. Le comportement du système obtenu est ensuite testé. Le test est orienté vers la détection des interactions potentielles.

Les auteurs indiquent que les méthodes de tests utilisées sont de type fonctionnel (boîte noire), mais ne les détaillent pas. En conclusion, il est indiqué que les résultats obtenus sont encourageants. En particulier, l'approche a permis de découvrir des interactions difficiles à prédire avant la simulation. De plus, il est souligné que l'outil d'animation, très interactif et visuel, a été particulièrement utile pour la phase d'expérimentation des modèles et de détection des interactions.

Quatrième partie

Réflexions et Comparaisons

Chapitre 11

Travaux comparables

La validation des programmes Lustre a suscité plusieurs travaux sur diverses méthodes de test et sur le recours à la preuve. Dans ce chapitre, nous comparons ces travaux avec le nôtre, et en particulier on compare les mérites respectifs de Lesar (évaluateur de modèles spécifiques à Lustre) et Lutess dans le cadre du concours FIW.

Nous étudions ensuite les travaux relatifs au test de protocoles, car ce domaine reste la référence en matière d'application réussie d'une approche formalisée du test à des logiciels industriels.

11.1 Validation de systèmes synchrones

Test fonctionnel

Comme Lutess, Lurette [50, 87] est un outil de test fonctionnel, qui repose sur le même principe général de fonctionnement (fig. 11.1). Les deux outils nécessitent trois éléments : une description de l'environnement, le programme à valider et des propriétés que doit satisfaire le programme.

Par contre, Lurette permet de tester des programmes dont les entrées-sorties sont de types plus riches puisque les valeurs entières et réelles sont prises en compte.

Lurette ne permet pas à l'utilisateur de guider la sélection des données mais il possède une méthode de test (appelée *test épais*), qui lui permet a chaque pas de test, de tester plusieurs transitions. Le principe consiste à produire plusieurs vecteurs de données pour un état du programme observer les différentes réactions du programme et à choisir la transition la plus *pertinente*[1]. Cette méthode est réalisée en mémorisant l'état interne du programme

1. Au sens où nous l'avons utilisé pour la méthode de génération guidée par des propriétés.

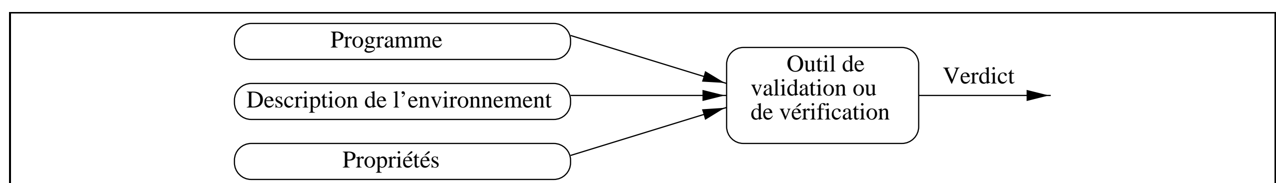


FIG. **11.1** – Principes des outils de validation de programme Lustre

à un pas de test, et à rétablir cette état pour chaque nouvel essai.

Dans [57] est décrit un outil de test fonctionnel pour les programmes écrits en Esterel. Cet outil s'inspire des travaux menés pour Lutess. Un générateur de données est construit à partir d'une description de l'environnement et des propriétés de sûreté données en logique temporelle. La méthode de génération de cet outil est comparable à la méthode de génération guidée par les propriétés de Lutess.

Contrairement à Lutess, cet outil compile le programme, l'oracle et le générateur en une seule entité qui effectue le test. La construction de cette entité s'avère être particulièrement coûteuse de l'avis des concepteurs et limite l'emploi de l'outil aux programmes écrits en Esterel.

Test structurel

On présente ici trois approches de test structurel de programmes Lustre. Ces trois approches se distinguent par le type de structure considérées. Le première approche présentée [70] opère sur la structure de l'automate produit après compilation du programme. La seconde [77] prend en compte le réseau d'opérateurs associé au programme. La dernière s'appuie directement sur le code Lustre [69].

L'approche de test structurel proposée dans [70] consiste à choisir les données de test de deux façons : d'une part, une sélection manuelle de données de test ayant une probabilité élevée de produire une défaillance, et d'autre part, une sélection automatique d'une grande quantité de données pour compenser les lacunes de la sélection manuelle.

Cette deuxième méthode de sélection des données de test s'appuie sur l'automate Lustre, engendré à la compilation. La sélection des données est faite de façon aléatoire, et repose sur le choix d'une distribution des probabilités d'entrées propre à satisfaire rapidement un critère de couverture choisi (tel que la couverture des transitions de l'automate). Cette approche est intéressante lorsque la compilation du programme Lustre produit des automates non triviaux (i.e. non réduit à un seul état) dont le contrôle est basé sur les mémoires booléennes. Mais ce point est aussi la faiblesse de cette approche puisqu'il n'est pas toujours possible de générer les automates non triviaux. A titre d'exemple, pendant l'expérience du concours FIW, nous n'avons jamais pu générer ces automates, faute de temps et de place mémoire.

Dans [77], F. Ouabdesselam et I. Parissis proposent une méthode de génération de données test basé sur le réseau d'opérateur défini par le programme Lustre.

Prenons l'exemple de l'opérateur Edge. Cet opérateur a une entrée et une sortie booléennes. La sortie est vraie à l'instant t lorsque son entrée était fausse à l'instant $t \Leftrightarrow 1$ et vraie à l'instant t. Le code de ce nœud est par exemple :

```
node Edge (X: bool) returns (res: bool)
   let
      res = x ⇔> (X and not pre (X));
   tel
```

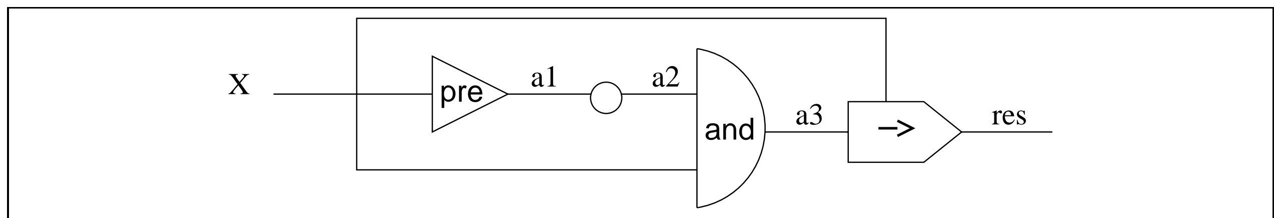


FIG. **11.2** – Réseau d'opérateur du programme Edge

(1)	s = vrai	s = faux	
(2)	a and pre b = vrai	a and pre b = faux	
(3)	a = vrai pre b = vrai	a = faux	a = vrai pre b = faux
(4)	$(\text{top} = i)$ a = vrai $(\text{top} = i \Leftrightarrow 1)$ b = vrai	$(\text{top} = i)$ a = vrai	$(\text{top} = i)$ a = vrai $(\text{top} = i \Leftrightarrow 1)$ b = faux

FIG. **11.3** – Principe d'élaboration des cas de test avec GATeL

Le réseau d'opérateur de ce nœud Lustre est donnée figure 11.2.

Le principe du test structurel tel qu'il est décrit dans [77] consiste à choisir des données de test pour couvrir les arcs ou les *chemins* du réseau d'opérateurs. Un chemin est une séquence d'arc successifs dans le réseau d'opérateur, le premier arc étant une entrée du réseau et le dernier une sortie. Pour le nœud Edge, 3 chemins peuvent être définis (X,res), (X,a1, a2, a3, res) et (X, a3, res) où a1=pre X, a2=not a1, et a3=a2 and X.

Pour générer les données, Lesar est utilisé. Il s'agit d'un model-checker pour les programmes Lustre (détaillé plus loin), qui exhibe des contre-exemples lorsqu'une propriété est fausse. Intuitivement, si l'on cherche à prouver que la négation de l'expression Lustre associée ce chemin est toujours vrai, Lesar fournit un contre-exemple activant le chemin.

GATeL [69] est un outil de sélection de séquences de test à partir de descriptions Lustre. Ces descriptions peuvent être le programme sous test (il s'agit alors de test structurel), ou une spécification de programme sous test (il s'agit alors de test fonctionnel). Que la description soit le programme ou sa spécification, les données sont sélectionnées à partir de la structure de la description Lustre, avec l'objectif d'avoir le plus petit nombre de pas de test possible.

Le principe général de l'outil est de construire pour chaque variable de sortie un ensemble de cas de test en considérant les différentes façon d'obtenir les valeurs possibles de cette variable. Soit l'expression booléenne s = a and pre b. Pour tester cette expression, il faut générer des cas de tests amenant s à vrai et à faux (étape (1), figure 11.3). La production des cas de test repose sur l'analyse de l'expression a and pre b (étapes (2) et (3)). Lorsque l'expression est vraie, l'opérateur and requiert que a et pre b soient vraies. Lorsque l'expression est fausse, on peut considérer deux cas possibles[2]. La longueur des tests est ensuite calculé et le cas de test est représenté par rapport à l'instant final (étape 4).

2. Si l'on veut générer plus de données, il faut considérer trois cas.

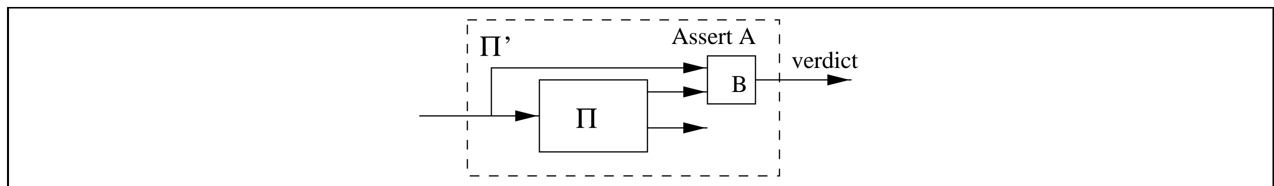


FIG. **11.4** – Structure du programme de vérification utilisé pour Lesar

11.2 Lesar : un outil de vérification de programmes Lustre

Les études de cas développées aux chapitres 9 et 10 ont montré, si cela était encore nécessaire, que le test n'offre aucune garantie d'absence de défaut dans un programme. Seule la preuve a cette qualité.

Au cours de cette thèse, nous avons utilisé Lesar (un outil de preuve par évaluation de modèles) pour évaluer la capacité de Lutess à découvrir des erreurs. Cette expérience a été menée sur l'étude de cas FIW [25].

Principes de fonctionnement de Lesar

Lesar [51, 85, 43] est un outil pour la vérification de programmes Lustre. Il s'agit d'un vérificateur de modèles (model-checker). Le principe de la vérification de modèle [71] consiste à construire un graphe d'état du programme. Pour chaque propriété P à vérifier, ce graphe est exploré pour déterminer dans chaque état si P est vraie.

Lesar fonctionne à partir d'un *programme de vérification* (fig. 11.4). Ce programme particulier rassemble trois éléments : le programme original à prouver (Π), une description de l'environnement (A) et la propriété à prouver (B). Ainsi, Lesar fonctionne sur le même principe que Lutess et Lurette (fig. 11.1, page 153).

La vérification est effectuée sur une abstraction booléenne Π'' sur programme Π'. Pour construire cette abstraction, les variables entières sont ignorées et les expressions booléennes dépendant de variables numériques telles que les comparaisons sont considérées comme non déterministes.

Utilisation de Lesar pour la détection d'interactions

L'idée était d'utiliser Lesar de s'assurer de l'absence effective d'interaction lorsque Lutess n'en trouvait pas.

Nous avons repris les programmes, les propriétés et les description d'environnement utilisés pendant le concours FIW. Nous avons construit les programmes de simulation Π' à partir de ces éléments. Quelques modifications ont dû être faites.

Tout d'abord, il a fallu éliminer les variables entières des programmes sous test. En effet, pour la plupart des services, la seule variable entière représentait état de l'automate. Cette

variable était utilisée en fait comme un type énuméré. Il a donc été facile de remplacer un type énuméré codé par un entier, par un type énuméré codé par un booléen.

Puis, nous avons simplifié l'expression de certaines propriétés d'oracle. En particulier, plutôt que de considérer une propriété $P = A \wedge B$, nous avons considéré les propriétés A et B, sachant que prouver P est équivalent à prouver A et prouver B.

Nous avons considéré les paires de services CND+(CND, TCS, CFBL, CF, CELL), TCS+(TCS, CFBL, CF, CELL), CELL+CELL, CFBL+CFBL, CF+CF. Ces paires ont été choisies dans cet ordre pour augmenter progressivement la taille du programme à prouver (la taille est évaluée à partir du nombre de nœuds de la description des services dans les diagrammes de Chisel).

Lesar a réussi à construire l'abstraction booléenne (Π''), pour 8 paires sur 12 précitées. Cette abstraction n'a pu être construite pour les 4 paires CF+CF, CF+CND, TCS+CFBL, TCS+CF par manque de place mémoire ou du fait d'un temps de calcul excessif (dépassant 12 heures). C'est pourquoi d'autres paires de services n'ont pas été prise en compte.

Pour les 12 paires de services considérées, 12 interactions ont été détectées par Lutess. Lesar n'a exhibé que 6 de ces interactions. Les 6 autres interactions concernent les 4 paires pour lesquelles Lesar n'a pu construire l'abstraction booléenne. A chaque fois que Lesar a produit un contre-exemple, il s'agissait d'une interaction découverte en utilisant Lutess.

Pour les paires de services les plus simples (CND+CND and CND+TCS) Lesar a exploré plus de 2 000 états et 17 000 transitions. Lesar a prouvé les propriétés sans même que la liste d'abonnement aux services n'ait été fixée. Dans ce cas, Lesar s'est montré bien plus puissant que Lutess, puisque ce dernier a toujours nécessité que la liste de souscription soit fixée.

Bilan de cette expérience

Pour l'étude de cas FIW, l'utilisation de Lesar ne nous a pas permis d'évaluer l'efficacité de Lutess à découvrir des erreurs ou à l'élaboration d'un critère arrêt du test. En effet, les programmes considérés se sont très vite révélés être de tailles trop importantes, et malgré des tentatives de simplification (réduction à 3 utilisateurs, par exemple), Lesar n'a pas toujours produit un verdict.

Toutefois, cette expérience nous a suggéré une démarche de validation combinant le test avec Lutess et la preuve avec Lesar. Cette démarche repose sur le fait que les deux outils utilisent les mêmes éléments et que le passage de l'un à l'autre ne demande pas un effort considérable. Tout d'abord, il faut utiliser Lutess comme outil d'animation, pour s'assurer que la description de l'environnement et des propriétés correspond bien à ce que l'on supposait. La seconde étape consiste en l'utilisation de Lesar pour prouver les propriétés attendues. Enfin, une troisième étape consiste à réemployer Lutess pour valider le programme lorsque la preuve n'a pas été possible ou lorsque le programme sous test contient des entiers.

Justifions chaque étape et leur ordre respectif. Il est nécessaire de valider la description de l'environnement les propriétés avant leur utilisation dans Lesar. Durant l'expérience d'utilisation de Lesar, nous avons introduit une erreur de description de l'environnement dans un programme de simulation. Nous avons alors prouvé une propriété, alors que Lutess avait trouvé un contre-exemple pour la propriété et la paire de services considérées. Après avoir

corrigé l'erreur introduite dans la description de l'environnement, Lesar a exhibé le même contre-exemple que Lutess. Ainsi, si la description de l'environnement ou les propriétés ne correspondent pas à l'idée que l'on s'en fait, les résultats produit par Lesar peuvent être mal interprétés.

L'utilisation de Lesar apporte une certitude que Lutess ne peut pas apporter. Lesar est de ce point de vue plus puissant que Lutess et il ne faut pas se priver des résultats qu'il peut apporter.

L'utilisation Lutess en dernier lieu permet, d'une part d'obtenir verdict partiel lorsque Lesar n'a pu en apporter, et d'autre part de s'assurer que le comportement du programme correspond bien a celui attendu. En effet, les propriétés que l'on cherche à prouver peuvent ne pas exhiber certaines défaillances.

Il existe évidemment d'autres outils de preuves de programmes synchrones. Dans les articles [58] et [5], les auteurs font référence à des vérificateurs de modèles pour les programmes Esterel .

Notons qu'il existe aussi d'autres approches pour la vérification. Par exemple, une approche de preuve déductive est étudiée pour les programmes Lustre [7]. Ce travail consiste à combiner Lustre et le démonstrateur PVS.

11.3 Validation de protocoles

La validation de protocoles repose essentiellement sur le test. Quand l'approche de test retenue est celle du *test de conformité*, elle s'apparente à de la vérification. Le test de conformité a pour but d'aider à répondre à la question : "une implantation I est-elle conforme à une spécification S ?".

La conformité, dans ce cadre, est définie formellement comme une relation, dite relation d'implantation, entre les objets modélisant les implantations et ceux modélisant les spécifications [80]. De nombreuses relations de conformité ont été définies selon ce que l'on veut vérifier et ce que l'on est en mesure d'observer [24, 1, 16, 65, 89, 80]. Par exemple, on peut vouloir vérifier qu'une implantation réalise au moins toute une spécification, seulement une partie de la spécification, exactement une spécification, ou encore qu'une implantation ne se bloque pas (pas de boucle de calcul infinie). Il existe des relations d'implantation pour chacune de ces notions.

Dans la majorité des travaux du domaine, les spécifications et les implantations sont définies de façon comportementale et exprimées sous forme d'automate ou de traces. L'expressivité du formalisme utilisé pour la description des implantations et des spécifications a une influence sur la relation d'implantation. Par exemple, il n'est pas possible de détecter un blocage de l'implantation si la notion de blocage n'est pas définie explicitement dans le formalisme utilisé.

Formellement, la relation d'implantation ne restreint pas les comportements à observer. Le nombre potentiellement infini de comportements nécessite un choix de données à tester, représenté par des *cas de test*. Un cas de test correspond à une suite d'échanges entre l'implantation et son environnement, menant explicitement à un verdict "succès" ou "échec", et

par défaut "pas de conclusion possible". Les cas de test sont regroupés dans un automate appelé *testeur*. Informellement, une implantation est conforme à une spécification si les interactions entre le testeur et l'implantation n'ont jamais conduit le testeur dans son état "échec".

Pour que quelques diagnostics suffisent à conclure que la relation d'implantation est satisfaite, il faut que les cas sélectionnés offrent la garantie d'être caractéristiques de tous les cas possibles. Cette conviction est bâtie sur l'énoncé d'*hypothèses de test* et la mise en évidence d'*objectifs de test* . Les hypothèse de test portent sur l'implantation et consiste par exemple :

- à supposer une limite au nombre d'états des automates modélisant les implantations (hypothèse de régularité),
- à tirer parti de la connaissance de la structure des cas (hypothèse d'indépendance),
- à partitionner les domaines d'entrée de l'implantation selon les erreurs à rechercher (hypothèse d'uniformité), ...

Les objectifs de test sont utilisés pour décrire des situations particulières que l'on veut observer. Dans [17, 89], un cadre formel est proposé pour définir des classes d'équivalence d'erreurs et de "bons" objectifs de tests, mais il est difficilement applicable. Un algorithme d'extraction de cas de test à partir d'un objectif de test formalisé est proposé dans [40].

En résumé, cette approche partage avec les méthode de test intégrées dans Lutess, plusieurs caractéristiques :

- il s'agit d'un test fonctionnel (en "boîte noire");
- les données de test sont produites à partir de spécifications formelles;
- le verdict de test est énoncé à partir de l'examen de spécifications d'actions observables (typiquement des traces d'échanges entre le système sous test et son environnement);
- l'examen des traces nécessite la création d'observateurs;
- on dispose d'un moyen pour caractériser des situations à observer (notion de schémas de comportements dans Lutess [29]).

Lutess se distingue principalement de l'approche mise en place pour le test de conformité par les points suivants :

- les cas de tests sont produits de façon dynamique, c'est-à-dire qu'il n'y a pas de "testeur" au sens des protocoles;
- la sélection des données de test pour les protocoles a été formalisée de longue date et standardisée[3] sous la forme d'une démarche. Elle comporte plusieurs étapes et le passage de l'une à l'autre (resp. d'un langage à un autre) est aussi formalisé dans le but de garantir la cohérence entre ces niveaux, selon une démarche classique dans les méthodes de vérification.

3. standard ISO IS-9646

Chapitre 12

Bilan et perspectives

Cette thèse représente une contribution à la la validation de systèmes réactifs spécifiés en Lustre. Un système réactif $\mathcal{R}$ spécifié en Lustre est caractérisé par deux descriptions : celle de l'environnement $\mathcal{E}$ dans lequel s'exécute le système, et les propriétés requises $\mathcal{P}$ pour ce système. Notre travail a consisté à spécifier, implanter et valider une technique de test fonctionnelle et statistique permettant de déterminer si $\mathcal{R}$ satisfait $\mathcal{P}$, à partir de données engendrées aléatoirement selon une distribution spécifiée respectant $\mathcal{E}$. Cette technique de test assure une génération automatique de ces données, et s'inspire des profils opérationnels, dont nous avons montré combien l'utilisation pouvait aider le testeur à révéler des erreurs.

Nous avons par ailleurs proposé une alternative de description des profils opérationnels, sous la forme de probabilités conditionnelles. Cette possibilité vise à simplifier le travail de description du testeur, lorsque celui-ci dispose d'un profil opérationnel équiprobable.

Bien que très connus, les profils opérationnels sont peu utilisés pour la génération automatique de données de tests. En ce sens, notre travail semble représenter un apport significatif.

Ce travail a pour cadre Lutess, un environnement de validation développé par notre équipe d'accueil, et que nous avons étendu.

Au cours des exemples traités, nous avons pu constater que la méthode de guidage par les probabilités conditionnelles apportait à Lutess une réelle souplesse d'utilisation. Ainsi, grâce à cette méthode, le testeur peut choisir les comportements qu'il souhaite valider en priorité. Cela a permis par exemple d'invoquer le service de transfert d'appel sur non réponse (CFNR) de telle sorte que les triples transferts d'appels ont pu être mis en évidence, alors que la méthode de génération équiprobable n'avait pu mener à leur observation (cf. chapitre 9).

La validation formelle et empirique que nous avons pratiquée a établi que les propriétés statistiques attendues des méthodes de génération de Lutess étaient satisfaites. L'utilisation de tests statistiques a montré par ailleurs son utilité quant à la détection d'erreur. En effet, grâce à ces tests, nous avons pu détecter une erreur dans le programme de conversion de clics (~⊜), alors que les propriétés du systèmes n'avaient pas été mises en défaut (cf. chapitre 7).

Enfin, nous avons appliqué la méthode de génération guidée par des probabilités conditionnelles à la validation de spécifications de services téléphoniques, au cours de deux études

de cas de taille conséquente. Lutess s'est révélé être un outil performant pour cette tâche, en partie grâce à cette méthode. En effet, Lutess a été déclaré vainqueur du concours d'outils pour la détection d'interaction de services téléphoniques, organisé en marge de la conférence FIW98.

Apports de l'outil

Dans une étude de cas aussi complète que celle de FIW98, Lutess a montré ses avantages :

- La majorité des erreurs sont révélées par des séquences relativement courtes.
- L'automatisation du test a soulagé le testeur humain, et lui a permis de se consacrer à des tâches plus importantes que la mise au point de cas de test. Cet avantage s'est avéré particulièrement intéressant dans le cadre du concours où le temps était limité.
- Du point de vue du spécifieur, l'utilisation de Lutess s'est montrée profitable pour la mise au point des diverses spécifications : oracles, modèles de service et contraintes d'environnement ont bénéficié favorablement d'une validation par l'outil. En effet, la possibilité de visualiser les traces permet à l'observateur humain (remplaçant ainsi l'oracle) d'évaluer l'adéquation des spécifications.
- La prise en compte de situations pouvant nécessiter plusieurs millions de pas de tests ne constitue pas un problème; l'efficacité de l'outil est entièrement déterminée par la complexité des descriptions de l'environnement.

Lutess semble donc bien adapté à la mise au point de services de télécommunication lors de leur définition et de leur spécification. Une perspective de valorisation serait de l'intégrer à un atelier logiciel de conception de services.

Outillage de la méthode de génération basée sur les probabilités conditionnelles

La méthode de génération guidée par des probabilités conditionnelles gagnerait à être mieux outillée, d'une part pour aider l'utilisateur à définir des profils opérationnels, et d'autre part, pour rendre l'affichage de ces profils plus exploitable.

Tout d'abord, nous pensons qu'il serait utile d'offrir à l'utilisateur un module automatisant le calcul des probabilités conditionnelles à partir d'un profil opérationnel partiel.

Il ne s'agit pas de demander à l'utilisateur de donner la probabilité de chaque vecteur valide déductible de la description de l'environnement. En effet, le nombre de vecteurs est souvent bien trop important. De notre expérience de spécification sur les études de cas (le convertisseur de clics, l'ascenseur et les modèles de services), nous faisons le constat que l'utilisation de probabilités conditionnelles se restreint souvent à des sous-ensembles de variables sur lesquels s'appliquent des contraintes simples (du type "AuPlusUn", "ExactementUn"). Par exemple, pour le convertisseur de clics, la contrainte est "au plus un bouton cliqué à chaque cycle".

A partir de cette constatation, on peut imaginer qu'une interface

- propose à l'utilisateur de sélectionner un sous-ensemble de variables,

- construise automatiquement les affectations valides pour ces variables,
- invite l'utilisateur à définir un profil opérationnel,
- et en déduise automatiquement les probabilités conditionnelles.

Par ailleurs, nous avons remarqué que l'affichage des profils opérationnels (calculés à partir des probabilités conditionnelles et des contraintes) est inexploitable lorsque le nombre de vecteurs et d'états est top grand. Il serait sans doute intéressant pour l'utilisateur de pouvoir sélectionner les états de l'environnement pour lesquels l'affichage du profil est utile. La sélection des états pourrait être basée sur une propriété Lustre; ne seraient affichés que les vecteurs d'entrée correspondant aux états pour lesquels la propriété est vraie.

Extensions de Lutess

Lutess possède déjà trois méthodes de génération de données : aléatoire et équiprobable, guidée par des propriétés et guidée par des probabilités conditionnelles. Les guidages proposés sont de type "instantané", c'est-à-dire qu'ils ne sont pas conçus pour amener le système dans un état qui ne serait accessible qu'après un nombre fini de cycles à partir de l'instant courant. Par exemple, considérons un système ayant pour entrées a et b et une sortie c, et la propriété P: *always_from_to(c,* pre *a,b).* Selon la méthode de guidage par propriétés, les données d'entrée pertinentes pour le test de P sont celles apparaissant dans l'intervalle entre pre a et b. Pour obtenir cette situation, il faut que pre a soit réalisé, c'est-à-dire qu'il faut produire a et *attendre un cycle*. Il s'agit là d'un guidage sur deux cycles. Ainsi, la recherche d'une violation instantanée n'est pas toujours satisfaisante pour la violation de propriétés temporelles et c'est pourquoi une méthode permettant la violation d'une propriété après n pas de test doit être envisagée, même si la méthode de génération aléatoire n'exclue pas une violation non instantanée.

Déjà, Nicolas Zuanon a intégré dans Lutess une méthode basée sur des scénarios ou "schémas comportementaux" [29], permettant un guidage sur plusieurs pas de tests. Un schéma comportemental est composé d'une succession de conditions portant alternativement sur des instants et des intervalles. Les conditions d'instant caractérisent les entrées que l'on souhaite voir engendrées, tandis que les conditions d'intervalle décrivent les invariants à préserver entre deux entrées successives. La méthode de génération consiste à générer les données de test pour progresser dans le schéma, en satisfaisant les conditions les unes après les autres.

Cette méthode de guidage pourrait apporter une solution au problème soulevé. Pour cela, il faudrait extraire des schémas comportementaux pour la violation d'une propriété directement à partir de la dite propriété. Le processus d'élaboration des schémas doit pouvoir s'appuyer sur les techniques de résolution de contraintes qui sont utilisées dans GATeL [69] pour la génération des séquences de données de tests à partir d'une description Lustre (cf. page 155, figure 11.3).

Améliorer l'efficacité de la validation par le test

Nous abordons dans ce paragraphe deux problèmes relatifs à la validation que nous avons rencontrés pendant la thèse. Il s'agit des problèmes de l'*oracle* et de la *terminaison du test*.

Ces problèmes sont le thème d'une littérature abondante. Nous nous appuyons sur quelques uns des travaux publiés pour proposer des directions de solution.

La construction d'oracles qui permettent de décider si un résultat est conforme à celui attendu est un problème général de la validation (et de la vérification aussi). Dans les études de cas que nous avons menées, nous avons construit des oracles à partir des propriétés attendues du système. Mais, nous avons noté que la détermination de ces propriétés est rarement complète et des erreurs peuvent passer inaperçues.

Il serait bon d'intégrer à Lutess d'autres types d'oracle. Par exemple, Whittaker [94] propose à l'utilisateur de définir une distribution attendues des sorties du système sous test, et de comparer cette dernière avec celle obtenue après un test. Une divergence entre les deux peut révéler une erreur dans le système. Nous l'avons d'ailleurs constaté dans l'expérience relative au convertisseur de clics (cf. page 100).

Ces oracles pourraient être simplement décrits avec des probabilités conditionnelles associées aux variables de sortie. Ils pourraient prendre une forme plus élaborée si l'utilisateur peut fournir la chaîne de Markov décrivant le fonctionnement du système sous test.

Décider de la terminaison d'un test pratiqué avec Lutess est une action pour laquelle nous n'offrons aucune aide à l'utilisateur. Ce dernier peut donc à raison s'interroger sur la qualité de son travail de validation, et sur la pertinence des données de test, si trop peu d'erreurs ont été révélées. Cette question provient d'abord du fait que pour les applications que nous avons considérées, il n'existe aucun cadre théorique analogue à celui du test de conformité pour les protocoles.

Pour le test de protocoles, la conformité d'un protocole avec sa spécification est déterminée sur un nombre fini de cas de test, produits à partir d'objectifs de tests, considérés comme représentatifs des seules situations à prendre en compte, compte tenu d'hypothèses de test simplificatrices. Les objectifs de tests à partir desquels les cas de test sont dérivés sont construits en tirant parti de l'expérience du testeur humain et de façon à assurer une certaine couverture des automates correspondant aux modèles des spécifications. Ils s'inspirent aussi parfois de modèles d'erreurs présentes dans les protocoles, caractérisés aussi sur les automates. La conformité est assimilée à un test exhaustif, et donc à une opération de vérification.

Dans le cadre de Lutess, plusieurs des points qui viennent d'être mentionnés constituent des problèmes ouverts, pour lesquels nous n'envisageons pas de solution. Il n'existe pas de caractérisation des erreurs typiques présentes dans les programmes écrits en Lustre; a fortiori, aucun modèle d'erreur n'a jamais été proposé. Les comportements des programmes Lustre n'ont pas été analysés en vue d'en définir des abstractions vis à vis des opérations de test; aussi aucune hypothèse simplificatrice n'est disponible.

Les axes de recherche qui nous semblent les plus prometteurs dans le cadre de Lutess, pour la terminaison du test, sont l'analyse de couverture et l'analyse de fiabilité.

L'analyse de couverture d'un programme consiste à décrire un critère minimum, basé sur la structure du programme, qui indique si le test effectué est trivialement insuffisant. Par exemple, pendant un test, on peut exiger que toutes les instructions d'un programme

soient activées. Beaucoup de critères de couverture ont été établis pour des programmes transformationnels (programmes C, ADA...), mais ces critères sont souvent peu adaptés pour des programmes synchrones (notamment des programmes Lustre) dont la structure est très pauvre.

Des expériences ont été menées pour l'établissement de critères de couverture sur l'automate produit par le compilateur Lustre [70]. Cette approche nécessite d'obtenir un automate non trivial (i.e. non réduit à un état). Au cours des études de cas FIW, nous n'avons pas été en mesure de produire ces automates, faute de temps et de place mémoire.

Actuellement, notre équipe mène une étude sur la possibilité d'établir un critère portant sur la couverture du codomaine des expressions conditionnelles. En effet, des études expérimentales indiquent que, même sur de longues séquences de test, il existe des expressions qui ne prennent pas toutes les valeurs de leur codomaine, ce qui laisse à penser qu'il s'agit d'un critère de couverture intéressant à observer.

La fiabilité consiste à estimer la confiance que l'on peut avoir dans un système à partir de l'observation de son exécution (ou de son test). Par exemple, il existe des analyses de fiabilité consistant à étudier le temps écoulé entre les occurrences d'erreurs, à partir de modèles de prédiction. Ce type d'approche nécessite des modèles de prédiction adaptés au langage de développement. Dans le cadre de Lutess, le premier travail consisterait à définir ces modèles, pour des applications typiques développées en Lustre. A partir d'expériences, plusieurs modèles pourraient être construits, car probablement ils dépendront des types d'applications.

Bibliographie

[1] S. Abramsky. Observation equivalence as a testing equivalence. *Theorical Computer Science*, 53:225–241, 1987.

[2] J.-R. Abrial and L. Mussat. Introducing Dynamic Constraints in B. In *Second International B Conference (B'98), LNCS 1393*, Montpellier, France, April 1998. Springer-Verlag.

[3] A. Aho, S. Gallagher, N. Griffeth, C. Schell, and D. Swayne. Scf3^{TM}sculptor with chisel: Requirements engineering for communications services. In *Feature Interactions in Telecommunications Systems V*, pages 45–63. IOS Press, 1998.

[4] S.B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27:509–516, 1978.

[5] C. André and H. Boufaïed. Vérification de comportement de machine d'exécution. In *2ème Congrès sur la Modélisation des systèmes réactifs*. Hermes, 1999.

[6] A. Arnold and I. Guessarian. *Mathematiques pour l'informatique*. Masson, 1993.

[7] S. Bensalem, P. Caspi, C. Dumas, and C. Parent-Vigouroux. A methodology for proving control programs with Lustre and PVS. In *Dependable Computing for Critical Applications, DCCA-7, San Jose*. IEEE Computer Society, January 1999.

[8] A. Benveniste and G. Berry. The Synchronous Approach to Reactive and Real-Time Systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[9] J. Blom, B. Jonsson, and L. Kempe. Using temporal logic for modular specification of telephone services. In L.G. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, pages 197–216. IOS Press, 1994.

[10] R. Blumenthal, N. Griffeth, J.-C. Gregoire, and T. Ohta. Feature interaction contest interaction description. *Comnet*, 1998. to be published.

[11] F. Boniol. Etude d'une sémantique de la réactivité : variations autour du modèle synchrone et application aux systèmes embarqués. Thèse, l'Ecole Nationale Supérieure de l'Aéronautique et de l'Espace, France, décembre 1997.

[12] L.G. Bouma and H. Velthuijsen, editors. *Feature Interactions in Telecommunications Systems*. IOS Press, 1994.

[13] F. Boussinot and R. De Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

[14] T.F. Bowen, F.S. Dworak, C.-H. Chow, N.D. Griffeth, Herman G.E., and Y.-L. Lin. The feature interaction problem in telecommunication systems. In *Proceedings of the seventh International Conference on Softwre Engineering for Telecommunication Switching Systems*, Bournemouth, United Kingdom, 1989.

[15] K. H. Braithwaite and J. M. Atlee. Towards automated detection of feature interactions. In L.G. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, pages 36–59. IOS Press, 1994.

[16] E. Brinksma. A theory for derivation of tests. In S. Aggrawal and K. Sabnani, editors, *Protocol Specification, Testing and Verification VIII*, 1988.

[17] E. Brinksma, J. Tretmans, and L. Verhaard. A framework for test selection. In *Protocol Specification, Testing and Verification XI*, 1991.

[18] R.E. Bryant. Graph-based algorithms for boolean functions manipulation. *IEEE Transactions on Computers*, pages 667–692, 1986.

[19] E.Jane Cameron and al. A feature interaction banchmark for in and beyond. In L.G. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, pages 1–23. IOS Press, 1994.

[20] 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.

[21] D. Cattrall, G. Howard, D. Jordan, and Buj S. An interaction-avoiding call processing model. In K.E. Cheng and T. Ohta, editors, *Feature Interactions in Telecommunications Systems III*, pages 173–184. IOS Press, 1995.

[22] K.E. Cheng and T. Ohta, editors. *Feature Interactions in Telecommunications Systems III*. IOS Press, 1995.

[23] P. Combes and S. Pickin. Formalization of a user view of network and services for feature interaction detection. In *Feature Interactions in Telecommunications Systems*, pages 120–135. IOS Press, 1994.

[24] R. de Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theorical Computer Science*, 34:83–133, 1984.

[25] L. du Bousquet. Feature interaction detection using testing and model-checking, experience report. In *World Congress on Formal Methods*, Toulouse, France, September 1999.

[26] 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, 1998.

[27] 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*, pages 262–275. IOS Press, 1998.

[28] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Feature interaction detection using synchronous approach and testing. *Computer Networks and ISDN Systems*, à paraître, 1999.

[29] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Lutess: a specification-driven testing environment for synchronous software. In *21st International Conference on Software Engineering*. ACM, May 1999.

[30] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Test et approche synchrone pour la détection d'interactions de services téléphoniques. technical report PFL, IMAG - LSR, Grenoble, France, 1999. à paraître dans la revue française Calculateurs Parallèles, Systèmes Répartis, Réseaux.

[31] L. du Bousquet and N. Zuanon. Feature interaction detection contest: Lutess testing tool. technical report PFL, IMAG - LSR, Grenoble, France, 1998.

[32] L. du Bousquet and N. Zuanon. An overview of lutess, a specification-based tool for testing synchronous software. In *14th IEEE International Conference on Automated Software Engineering*, volume à paraître. IEEE, Octobre 1999.

[33] J. Duran and S. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, 1984.

[34] M. Dyer. *The cleanroom approach to quality software development.* J. Wiley, 1992.

[35] ETSI. *Integrated Services Digital Network (ISDN); Call Forwarding No Reply, Service description*, December 1994. ETS 300 201.

[36] ETSI. *Integrated Services Digital Network (ISDN); Completion of Call to Busy Subscriber, Service description*, October 1995. ETS 300 357.

[37] ETSI. *Integrated Services Digital Network (ISDN); Explicit Call Transfert, Service description*, May 1995. ETS 300 367.

[38] A. S. Evans. Specifying and Verifying Concurrent Systems Using Z. In *Workshop on Formal Methods Europe (FME'94), LNCS 873*, Barcelona, Spain, October 1994. Springer-Verlag.

[39] M. Faci and L. Logrippo. Specifying features and analyzing their interactions in a lotos environment. In L.G. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, pages 136–151. IOS Press, 1994.

[40] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In *CAV'96, LNCS 1102*. Springer-Verlag, 1996.

[41] A. Gammelgaard and J. E. Kristensen. Interaction detection, a logical approach. In L.G. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, pages 178–196. IOS Press, 1994.

[42] P. Gibson. Towards a feature interaction algebra. In K. Kimbler and L.G. Bouma, editors, *Feature Interactions in Telecommunications Systems V*, pages 217–231. IOS Press, 1998.

[43] A-C. Glory. Vérification de propriétés de programmes flots de données synchrones. Thèse, Université Joseph Fourier, Grenoble, France, decembre 1989.

[44] J.-C. Godskesen. A formal framework for feature interaction with emphasis on testing. In K.E. Cheng and T. Ohta, editors, *Feature Interactions in Telecommunications Systems III*, pages 21–30. IOS Press, 1995.

[45] N. Griffeth, R. Blumenthal, J.-C. Gregoire, and T. Ohta. Feature interaction detection contest. In *Feature Interactions in Telecommunications Systems V*, pages 327–359. IOS Press, 1998.

[46] N. Griffeth and H. Velthuijsen. The negotiating agents approach to runtime feature interaction resolution. In L.G. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, pages 217–235. IOS Press, 1994.

[47] N. D. Griffeth and Y.-J. Lin. Extending telecommunication systems: The feature-interaction problem. In *IEEE Computer*, pages 14–18, August 1993.

[48] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. Programmation et Vérification des Systèmes Réactifs : le langage LUSTRE. *Technique et Science Informatique*, 10(2), 1991.

[49] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous Observers and the Verification of Reactive Systems. In *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93, Twente*. Workshops in Computing, Springer Verlag, 1993.

[50] N. Halbwachs, X. Nicollin, P. Raymond, and D. Weber. Test automatique de systèmes réactifs. In F. Cassez, C. Jard, O. Roux, and B. Rozoy, editors, *Modélisation et vérification des processus parallèles*, 1998.

[51] N. Halbwachs, D. Pilaud, F. Ouabdesselam, and A-C. Glory. Specifying, Programming and Verifying Real-Time Systems, using a synchronous declarative language. In *Workshop on automatic verification methods for finite state systems, LNCS 407*, Grenoble, France, 1989. Springer Verlag.

[52] R.J. Hall. Feature combination and interaction detection via foreground/background models. In K. Kimbler and L.G. Bouma, editors, *Feature Interactions in Telecommunications Systems V*, pages 232–246. IOS Press, 1998.

[53] D. Hamlet. Software Quality, Software Process and Software Testing. *Advances in Computers*, 1995.

[54] D. Hamlet and R. Taylor. Partition Analysis Does Not Inspire Confidence. *IEEE Transactions on Software Engineering*, pages 1402–1411, december 1990.

[55] D. Harel and A. Pnueli. On the Development of Reactive Systems. *Logics and Models of Concurrent Systems, NATO ASI Series*, F13:477–498, 1985.

[56] L. Jagadeesan, C. Puchol, and J. Von Olnhausen. A Formal Approach to Reactive System Software: A Telecommunications Application in Esterel. *Formal Methods in Systems Design*, 8(2), March 1996.

[57] L.J. Jagadeesan, A. Porter, C. Puchol, J.C. Ramming, and L. Votta. Specification-based Testing of Reactive Software: Tools and Experiments. In *19th International Conference on Software Engineering*, 1997.

[58] L.J. Jagadeesan, C. Puchol, and J.E. Von Olnhausen. Safety Property Verification of Esterel Programs and Applications to Telecommunications Software. In *7th Conference on Computer-Aided Verification*, 1995.

[59] B. Kelly, M. Crowther, J. King, R. Masson, and J. DeLapeyre. Service validation and testing. In K.E. Cheng and T. Ohta, editors, *Feature Interactions in Telecommunications Systems III*, pages 173–184. IOS Press, 1995.

[60] A. Khoumsi. Detection and resolution of interactions between telephone services. In *Feature Interactions in Telecommunications Systems IV*, pages 78–92. IOS Press, 1997.

[61] K. Kimbler and L.G. Bouma, editors. *Feature Interactions in Telecommunications Systems V*. IOS Press, 1998.

[62] K. Kimbler and Søbirk D. Use case driven analysis of feature interactions. In L.G. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, pages 166–177. IOS Press, 1994.

[63] D. Knuth. *The art of computer programming, chap. 3 : Random Numbers*. Addison Wesley, 1969.

[64] 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.

[65] G. Leduc. Conformance relation, associated equivalence and new canonical tester in LOTOS. In *Protocol Specification, Testing and Verification XI*, 1991.

[66] F.J. Lin and Y.-J. Lin. A building block approach to detecting and resolving feature interactions. In *Feature Interactions in Telecommunications Systems*, pages 86–119. IOS Press, 1994.

[67] Ross S. M. *A course in Simulation*. Macmillan, 1990.

[68] F. Maraninchi. The argos language: Graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, october 1991.

[69] B. Marre. Sélection de tests à partir de descriptions lustre. Séminaire, IMAG - LSR, Grenoble, France, juin 1999. http://www-lsr.imag.fr/Les.Seminaires/Resumes/index.html.

[70] C. Mazuet. Stratégies de Test pour des Programmes Synchrones - Application au Langage Lustre. Thèse, Institut National Polytechnique de Toulouse, Toulouse, France, decembre 1994.

[71] K.L. McMillan. An introduction to model-checking. In F. Cassez, C. Jard, O. Roux, and B. Rozoy, editors, *Modélisation et vérification des processus parallèles*, 1998.

[72] B. Mermet and D. Méry. Détection d'interaction de services : une approche avec b. In *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'97)*, Toulouse, France, 1997.

[73] G. Murakami and R. Sethi. Terminal call processing in Esterel. In *Proc. IFIP 92 World Computer Congress*, Madrid, Spain, 1992.

[74] J. Musa. Operational Profiles in Software-Reliability Engineering. *IEEE Software*, pages 14–32, march 1993.

[75] G. Myers. *The Art Of Software Testing*. Wiley-Interscience, 1979.

[76] R. Nelson. *Probability, stochastic processes and queuring theory, chap 8 : Markov processes*. Springer-Verlag, 1995.

[77] F. Ouabdesselam and I. Parissis. Testing Synchronous Critical Software. In *5th International Symposium on Software Reliability Engineering*, Monterey, USA, 1994.

[78] F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Using behavioral patterns for guiding the test of service specification. technical report PFL, IMAG - LSR, Grenoble, France, 1998. soumis à ASE'99.

[79] I. Parissis. Test de logiciels synchrones spécifiés en Lustre. Thèse, Université Joseph Fourier, Grenoble, France, septembre 1996.

[80] M. Phalippou. Relations d'implémentations et hypothèses de test sur des automates à entrées et sorties. Thèse, Université Bordeaux I, France, septembre 1994.

[81] 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, 1988. Springer Verlag.

[82] E. Pilaud and J-L. Bergerand. SAGA : A software Development Environment for Dependability Automatic Control. In *SAFECOMP'88*, Fulda, West Germany, 1988. IFAC.

[83] M. Plath and M. Ryan. Plug-and-play features. In K. Kimbler and L.G. Bouma, editors, *Feature Interactions in Telecommunications Systems V*, pages 150–164, 1998.

[84] A. Pnueli. Application of temporal logic to the specification and verification of reactive systems : a survey of current trends. *Current Trends in Concurrency, LNCS, Springer-Verlag*, 224:510–584, 1986.

[85] C. Ratel. Définition et réalisation d'un outil de vérification formelle de programmes Lustre: Le système Lesar. Thèse, Université Joseph Fourier, Grenoble, France, juin 1992.

[86] P. Raymond. Compilation efficace d'un langage déclaratif synchrone : le générateur de code LUSTRE-V3. Thèse, Institut National Polytechnique de Grenoble, Grenoble, France, novembre 1991.

[87] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium (RTSS'98)*. IEEE, 1998.

[88] C.E. Shannon. A symbolic analysis of relay and switching circuits. *Transactions AIEE*, 57:305–316, 1938.

[89] J. Tretmans. *A formal approach to conformance testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[90] K.J. Turner. An architectural description of intelligent network features and their interactions. *Computer Networks and ISDN Systems*, 30(15):1389–1420, 1998.

[91] G. Utas. A pattern language of feature interaction. In K. Kimbler and L.G. Bouma, editors, *Feature Interactions in Telecommunications Systems V*, pages 98–114. IOS Press, 1998.

[92] R. van der Linden. Using an architecture to help beat feature interaction. In L.G. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, pages 24–35. IOS Press, 1994.

[93] E. Weyuker, S. Weiss, and D. Hamlet. Comparison of Program Testing Stategies. In *Symposium on Testing, Analysis and Verification (TAV)*, Victoria, British Columbia, october 1991.

[94] J. Whittaker. *Markov chain techniques for software testing and reliability analysis*. PhD thesis, University of Tenessee, 1992.

[95] N. Zuanon. Modélisation et validation de services téléphoniques. Rapport de DEA, INPG - ENSIMAG, Grenoble, France, juin 1996.

Annexe A

Description des sémantiques citées

A.1 Définition formelle de Lustre

A.1.1 Sémantique basée sur des traces d'exécution finies

La valeur d'une expression Lustre se calcule à partir d'une trace d'exécution finie ou infinie[1]. Soit E une expression et $\Sigma = (\sigma_0, \sigma_1, ..., \sigma_n)$ une trace finie. On note $\Sigma \vdash E|v$, le fait que l'expression E est évaluée à v après l'exécution de Σ.

La valeur de toute expression Lustre sur une trace Σ est définie par les règles suivantes :

- Soit K une constante dont la valeur est k.
  $\Sigma \vdash K|k$
- Soit x une variable. Sa valeur sur Σ est définie par la "dernière" instanciation décrite par la trace.
  $\Sigma = (\sigma_0, ...\sigma_n) \vdash x|\sigma_n(x)$
- Soient $(E_1, .., E_m)$ des expressions quelconques du langage, et θ un opérateur m-aire arithmétique, booléen ou conditionnel standart. La valeur de l'expression $\theta(E_1, .., E_m)$ est définie par :
  $$\frac{\Sigma \vdash E_1|v_1, ..., \Sigma \vdash E_m|v_m}{\Sigma \vdash \theta(E_1, .., E_m)|\theta(v_1, ..., v_m)}$$
- Soient E_1 et E_2 deux expressions quelconques du langage. $E_1 \Leftrightarrow E_2$ a pour valeur :
  $$\frac{\Sigma = \sigma_0 \vdash E_1|v_1}{\Sigma = \sigma_0 \vdash E_1 \Leftrightarrow E_2|v_1} \qquad \frac{\Sigma = (\sigma_0 ... \sigma_j) \vdash E_2|v_2}{\Sigma = \sigma_0 \vdash E_1 \Leftrightarrow E_2|v_2}$$
- Soit E une expression quelconque et nil une valeur indéterminée; on note $\Sigma.\sigma$ la trace d'exécution obtenue par concatéation de la trace Σ et de la mémoire σ. La valeur de l'expression $\mathsf{pre}(E)$ est définie par :
  $$\Sigma = \sigma_0 \vdash \mathsf{pre}(E)|nil \qquad \frac{\Sigma \vdash E|v}{\Sigma.\sigma \vdash \mathsf{pre}(E)|v}$$

1. Dans sa thèse, C. Ratel détaille la compatibilité entre les traces finies et infinies. Nous n'exposerons ici que la partie de son travail sur les traces finies.

A.1.2 Sémantique basée sur deux mémoires

A partir de la sémantique sur les traces d'exécution, il est possible de construire une sémantique plus simple, ne reposant que sur les deux dernières instanciations des variables. Cette sémantique est appelée *sémantique opérationnelle* de Lustre.

La démarche consiste à imposer que l'opérateur pre ne soit appliqué qu'à des identificateurs (et non plus à des expressions complexes). Cette hypothèse n'est pas restrictive, puisque toute expression de Lustre peut être remplacer par une variable. Grâce à cette hypothèse, il est possible d'évaluer toute expression Lustre sur les deux dernières mémoires de la trace (l'avant dernière étant nécessaire et suffisante pour l'évaluation de toutes les expressions $\mathsf{pre}(x)$, x étant une variable).

La sémantique opérationnelle définit comment, à tout instant, une mémoire compatible avec un programme se déduit de la mémoire du programme à l'instant précédent. Le fait que pour un programme P, la mémoire σ' se déduit de la mémoire σ se note : $\sigma, \sigma' \vdash P$

A l'instant initial, la mémoire est indéfinie. Cette mémoire est dénotée par le symbol $\perp$.

La valeur de toute expression Lustre E sur une mémoire σ', sachant que la mémoire précédente est σ est définie par induction sur la structure de E. Le fait que E prend la valeur v est notée : $\sigma, \sigma' \vdash E|v$.

Les règles suivantes permettent de définir la valeur de E.

- Soit K une constante dont la valeur est k.
  $\sigma, \sigma' \vdash K|k$

- Soit x une variable.
  $\sigma, \sigma' \vdash x|\sigma'(x)$

- Soient $(E_1, .., E_m)$ des expressions quelconques du langage, et θ un opérateur m-aire arithmétique, booléen ou conditionnel standart. La valeur de l'expression $\theta(E_1, .., E_m)$ est définie par :
  $$\frac{\sigma, \sigma' \vdash E_1|v_1, ..., \sigma, \sigma' \vdash E_m|v_m}{\sigma, \sigma' \vdash \theta(E_1, .., E_m)|\theta(v_1, ..., v_m)}$$

- Soient E_1 et E_2 deux expressions quelconques du langage. $E_1 \Leftrightarrow E_2$ a pour valeur :
  $$\frac{\perp, \sigma' \vdash E_1|v_1}{\perp, \sigma' \vdash E_1 \Leftrightarrow E_2|v_1} \qquad \frac{\sigma, \sigma' \vdash E_2|v_2}{\sigma, \sigma' \vdash E_1 \Leftrightarrow E_2|v_2}$$

- Soit x un identificateur quelconque et nil une valeur indéterminée; la valeur de l'expression $\mathsf{pre}(x)$ est définie par :
  $\perp, \sigma' \vdash \mathsf{pre}(x)|nil \qquad \sigma, \sigma' \vdash \mathsf{pre}(x)|\sigma(x)$

Les règles définissant la compatibilité d'une mémoire σ' avec un programme P, sachant qu'à l'instant précédent la mémoire était σ ($\sigma, \sigma' \vdash P$) sont les suivantes :

- Compatibilité avec les équations :
  $$\frac{\sigma, \sigma' \vdash E|v,\ \sigma'(x) = v}{\sigma, \sigma' \vdash x = E}$$

– Compatibilité avec la composition :
$$\frac{\sigma, \sigma' \vdash P_1;\ \sigma, \sigma' \vdash P_2}{\sigma, \sigma' \vdash P_1; P_2}$$

– Compatibilité avec les assertions :
$$\frac{\sigma, \sigma' \vdash A|true}{\sigma, \sigma' \vdash \textsf{assert}\ A}$$

A.2 Définition formelle d'un noeud testeur

A.2.1 Sémantique opérationnelle pour la méthode 1

– Soit K une constante dont la valeur est k.
$\sigma, \sigma' \vdash K|k$

– Soit x une variable d'entrée ou locale.
$\sigma, \sigma' \vdash x|\sigma'(x)$

– Soit x une variable de sortie et soit out une valeur indéterminée. La valeur out a été introduite pour interdire l'utilisation de la valeur courante des variables de sorties pour le calcul dans le nœud testeur.
$\sigma, \sigma' \vdash x|out$

– Soit E une expression booléenne.
$$\frac{\sigma, \sigma' \vdash E \mid true}{\sigma, \sigma' \vdash \textsf{not}\ E \mid false} \qquad \frac{\sigma, \sigma' \vdash E \mid false}{\sigma, \sigma' \vdash \textsf{not}\ E \mid true}$$

– Soient E_1, E_2, E_3 trois expressions booléennes, et β une valeur égale à out ou nil.
$$\frac{\sigma, \sigma' \vdash E_1 \mid true}{\sigma, \sigma' \vdash \textsf{if}\ E_1\ \textsf{then}\ E_2\ \textsf{else}\ E_3 \mid E_2} \qquad \frac{\sigma, \sigma' \vdash E_1 \mid false}{\sigma, \sigma' \vdash \textsf{if}\ E_1\ \textsf{then}\ E_2\ \textsf{else}\ E_3 \mid E_3}$$
$$\frac{\sigma, \sigma' \vdash E_1 \mid \beta}{\sigma, \sigma' \vdash \textsf{if}\ E_1\ \textsf{then}\ E_2\ \textsf{else}\ E_3 \mid \beta}$$

– Soient E_1, E_2 deux expressions booléennes,
$\sigma, \sigma' \vdash E_1\ \textsf{or}\ E_2 \mid \textsf{if}\ E_1\ \textsf{then}\ true\ \textsf{else}\ E_2$
$\sigma, \sigma' \vdash E_1\ \textsf{and}\ E_2 \mid \textsf{if}\ E_1\ \textsf{then}\ E_2\ \textsf{else}\ false$

– Soient E_1 et E_2 deux expressions quelconques du langage. $E_1 \Leftrightarrow E_2$ a pour valeur :
$$\frac{\bot, \sigma' \vdash E_1 \mid v_1}{\bot, \sigma' \vdash E_1 \Leftrightarrow E_2 \mid v_1} \qquad \frac{\sigma, \sigma' \vdash E_2 \mid v_2}{\sigma, \sigma' \vdash E_1 \Leftrightarrow E_2 \mid v_2}$$

– Soit x un identificateur quelconque et nil une valeur indéterminée; la valeur de l'expression $\textsf{pre}(x)$ est définie par :
$\bot, \sigma' \vdash \textsf{pre}(x)|nil \qquad \sigma, \sigma' \vdash \textsf{pre}(x)|\sigma(x)$

- Compatibilité des équations définissant les variables locales :
$$\frac{\sigma, \sigma' \vdash E|v,\ \sigma'(x) = v}{\sigma, \sigma' \vdash x = E} \qquad \frac{\sigma, \sigma' \vdash P_1;\ \sigma, \sigma' \vdash P_2}{\sigma, \sigma' \vdash P_1; P_2}$$

- Compatibilité avec l'opérateur environment:
$$\frac{\sigma, \sigma' \vdash A_1|true, ..., \sigma, \sigma' \vdash A_n|true}{\sigma, \sigma' \vdash \mathsf{environment}(A_1, ..., A_n)}$$

A.2.2 Sémantique opérationnelle pour la méthode 2

Il s'agit de la même sémantique que précédemment, à laquelle il faut ajouter la règle de compatibilité avec l'opérateur safety :
$$\frac{\sigma, \sigma' \vdash P_1 \mid out, ..., \sigma, \sigma' \vdash P_n \mid out}{\sigma, \sigma' \vdash \mathsf{safety}(P_1, ..., P_n)}$$

A.2.3 Sémantique opérationnelle pour la méthode 3

Il s'agit de la même sémantique que précédemment, à laquelle il faut ajouter la règle de compatibilité avec l'opérateur proba.

- Soient $x_1...x_n$ des variables de sortie, $p_1...p_n$ des réels ($0 \leq p_i \leq 1$), $E_1...E_n$ des expressions booléennes. Soit γ une valeur égale à *nil*, *true* ou *false*.
  La compatibilité de l'opérateur proba est :
$$\frac{\sigma, \sigma' \vdash x_1 \mid out\ ...\ \sigma, \sigma' \vdash x_n \mid out \quad \sigma, \sigma' \vdash E_1 \mid \gamma\ ...\ \sigma, \sigma' \vdash E_n \mid \gamma}{\sigma, \sigma' \vdash \mathsf{proba}(\ \langle x_1, p_1, E_1\rangle ... \langle x_n, p_n, E_n\rangle\ )}$$

A.3 Opérateurs Lustre utilisés

On présente ici les opérateurs Lustre utilisés dans ce document. A chaque description, on associe un commentaire donnant la signification de l'opérateur. Dans [85], le lecteur trouvera la sémantique précise par rapport aux traces d'exécutions de certains des opérateurs temporels donnés ci-dessous.

```
- - - - - - - - - - - - - - Opérateurs pour les tableaux - - - - - - - - - -

- - LIN_OR : un entier N, un tableau de N booléens → un booléen
- - { LIN_OR(n,A) est vrai si au moins un élément de A est vrai }
node LIN_OR(const n: int; A: bool^n) returns (OR: bool)
let
    OR = with n=1 then A[0]
            else A[0] or LIN_OR(n-1, A[1..n-1]);
tel

- - LIN_AND : un entier N, un tableau de N booléens → un booléen
- - { LIN_AND(n,A) est vrai si tous les éléments de A sont vrais }
node LIN_AND(const n: int; A: bool^n) returns (AND: bool)
```

```
let
   AND =with n=1 then A[0]
          else A[0] and LIN_AND(n-1, A[1..n-1]);
tel

- - AuPlusUn : un entier N, un tableau de N booléens → un booléen
- - { AuPlusUn est vrai si au plus un élément de A est vrais }
node AuPlusUn(const n: int; A: bool^n) returns (APU: bool)
let
   APU = with n=1 then true
          else ((A[0] and not LIN_OR(n-1, A[1..n-1])) or
          (not A[0] and AuPlusUn(n-1, A[1..n-1])));
tel

- - ExactementUn: un entier N, un tableau de N booléens → un booléen
- - { ExactementUn est vrai si exactement un élément de A est vrais }
node ExactementUn(const n: int; A: bool^n) returns (EU: bool)
let
   EU = LIN_OR(n,A) and AuPlusUn(n, A) ;
tel

- - - - - - - - - - - - - - Opérateurs temporels - - - - - - - - - - - - - -

- - always_from_to : 3 booléens → un booléen
- - { always_from_to(A,B,C) est fausse si A a été fausse une fois entre la dernière
   occurence de B et la première de C suivant celle de B }
node always_from_to(A, B, C : bool) returns (always_A_from_B_to_C : bool);
let
   always_A_from_B_to_C = once_since(C,B) or always_since(A,B);
tel

- - once_from_to : 3 booléens → un booléen
- - { once_from_to(A,B,C) est fausse si A n'a jamais été vraie entre la dernière
   occurence de B et la première de C suivant celle de B }
node once_from_to(A, B, C : bool) returns (once_A_from_B_to_C : bool);
let
   once_A_from_B_to_C = C => once_since(A,B);
tel

- - once_since : 2 booléens → un booléen
- - { once_since(A,B) est vraie si A a été vraie depuis la dernière occurence de B }
node once_since(A, B : bool) returns (once_A_since_B : bool);
let
   once_A_since_B = if B then A else (true ⇔ (A or pre (once_A_since_B)));
tel

- - always_since : deux booléens → un booléen
```

```
- - { always_since(A, B : bool) est faux si A a été faux depuis la dernière occurence de B }
node always_since(A, B : bool) returns (always_A_since_B : bool);
let
    always_A_since_B = if never(B) then true
                       else if B then A
                       else (true ⇔ A and pre (always_A_since_B));
tel

- - never: un booléen → un booléen
- - { never(x) est vrai si x n'a jamais été vrai depuis l'instant initial }
node never(A:bool) returns (never_A : bool);
let
    never_A = not A ⇔ not A and pre never_A;
tel

- - jafter : un booléen → un booléen
- - { jafter(X) vrai si on l'instant courant suit immédiatement un instant ou X valait vrai
      (donc, faux initialement) }
node jafter(X : bool) returns (A : bool);
let
    A = false ⇔ pre X;
tel;

- - after : un booléen → un booléen
- - { vrai si X a déja été vrai dans le passé strict (donc, faux initialement) }
node after(X : bool) returns (A : bool);
let
    A = jafter(X or A);
tel;

- - once(X) : un booléen → un booléen
- - { once(X) est vrai si X est ou a été vrai (passé non strict) }
node once(X : bool) returns (O : bool);
let
    O = X or jafter(O);
tel;

- - edge : un booléen → un booléen
- - { edge(X) : front montant simple, inclue l'instant initial }
node edge(X : bool) returns (E : bool);
let
    E = X and not(jafter(X));
tel;

- - xedge : un booléen → un booléen
- - { xedge(X) : front montant strict, exclue l'instant initial }
node xedge(X : bool) returns (E : bool);
```

```
let
    E = X and (jafter(not X));
tel;

- - atlast : deux booléens → un booléen
- - { atlast(A,B) : prends la valeur de A chaque fois que B est vrai,
      sinon conserve sa valeur. Initialement faux. }
node atlast(A, B : bool) returns (Atl : bool);
let
Atl = if B then A else jafter(Atl);
tel;

- - switch : trois booléens → un booléen
- - { switch(past,on,off) : interrupteur. Automate à deux états: dans l'état "past"
      à l'origine des temps, activé par "on", et désactivé par "off". }
node switch(past, on, off : bool) returns (s : bool);
let
      s = if(not (past ⇔ pre s)) then (on) else (not off);
tel;
```