Test de spécifications de services de télécommunication

Nicolas Zuanon

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

Nicolas Zuanon

le 27 juin 2000

Test de spécifications de services de télécommunication

Composition du Jury :

P. Jacquet	**Président**
C. Jard	**Rapporteurs**
B. Legeard	
F. Klay	**Examinateurs**
F. Ouabdesselam	
J.-L. Richier	

2

Remerciements

Ce mémoire de thèse présente les travaux de recherche que j'ai effectués au sein du laboratoire Logiciels, Systèmes et Réseaux, de l'Institut IMAG. Je suis reconnaissant envers son directeur Paul Jacquet de m'y avoir accueilli. Je remercie également ce dernier d'avoir accepté de présider mon jury de thèse.

Mes travaux se sont déroulés dans le cadre d'une consultation thématique informelle initiée par le CNET de Lannion (nouvellement France Télécom Recherche & Développement), que je remercie ici de sa confiance.

Je remercie sincèrement les membres du jury qui ont accepté de juger ce travail et d'avoir bien voulu y apporter ainsi leur caution :

- Claude Jard, Directeur de recherche CNRS à l'IRISA (Rennes) et Gilles Bernot, Professeur à l'Université de Franche-Comté (Besançon), qui m'ont fait l'honneur d'être les rapporteurs de ma thèse,

- Farid Ouabdesselam, Professeur à l'Université Joseph Fourier, et Jean-Luc Richier, Chargé de recherche CNRS au LSR, mes directeurs de thèse, à la complémentarité exemplaire. Ils m'ont admirablement conseillé, guidé, soutenu et encouragé durant mes années de doctorat. Je rends ici hommage à leur toujours grande disponibilité malgré un emploi du temps extrêmement chargé,

- Françis Klay, responsable pour FT-R&D de la collaboration avec mon laboratoire d'accueil et qui a bien voulu être examinateur de mon travail.

Je tiens par ailleurs à remercier les membres du LSR, administratifs, chercheurs, doctorants et stagiaires qui ont, à divers titres, contribué au bon déroulement de cette thèse.
Je remercie en particulier :
- Ioannis à qui revient la paternité du premier prototype de LUTESS et l'essentiel de sa formalisation,
- Lydie avec qui j'ai longuement et utilement collaboré, nos directions de recherche étant relativement proches,
- Rémy et Jérôme pour leur aide précieuse lors du concours FIW'98, Patrice pour l'ergonomie nouvelle qu'il a su donner à LUTESS,
- Sophie pour m'avoir supporté trois ans durant comme collègue de bureau,
- sans oublier Liliane, Martine, Rachèle, Solange, Sonia, Bernard et François pour leur efficacité à leurs postes respectifs, et le mieux-être qu'il en découle pour le reste du laboratoire...

Enfin, ces remerciements ne seraient pas complets si j'oubliais Sophie pour son soutien et son amour de tous les instants et Lilou, qui a bien voulu attendre que "Papa" finisse de rédiger avant de se montrer au monde...

Table des matières

1 Introduction **9**

2 Objectifs **15**
2.1 Définitions préliminaires 15
2.2 Axes classiques de recherche 16
2.3 Présentation de l'approche choisie 17
2.4 L'approche synchrone 18
2.5 Justification d'une validation fondée sur le test 20
2.6 Présentation de la méthode de modélisation 20
2.7 Présentation de la méthode de validation 22
2.8 Définition formelle d'une interaction 23

I Contribution à un environnement de test efficace **27**

3 LUTESS **29**
3.1 Le langage LUSTRE 29
3.2 Principe de fonctionnement de LUTESS 30
3.3 Mise en œuvre du test par LUTESS 32
3.4 Présentation des méthodes de test 32
3.4.1 Test par simulation de l'environnement 33
3.4.2 Test de propriété de sûreté 33
3.4.3 Test guidé par profils opérationnels 34
3.5 Fondements théoriques 35
3.5.1 Simulateur d'environnement 35
3.5.2 Machine de génération guidée par propriétés 36
3.5.3 Machine de génération guidée par profil opérationnel 37
3.6 Détails des algorithmes et implémentation 38
3.7 Arrêt du test 42

4 Test efficace **45**
4.1 Introduction 45
4.1.1 La nécessité de guider dans le test 45
4.1.2 Description informelle de la notion de *schéma comportemental* 46
4.1.3 Guidage par schéma 46

4.2 Objectifs de la méthode . . . 47
4.3 Formalisation . . . 48
4.3.1 Définition formelle d'un schéma . . . 49
4.3.2 Définition formelle du guidage à base de schéma . . . 50
4.4 Aspects techniques . . . 51
4.5 Combinaison de schémas . . . 55
4.6 De schémas en profil opérationnel . . . 57
4.7 Caractérisation de la complétion d'un schéma . . . 58
4.7.1 Représentation graphique d'un schéma . . . 58
4.7.2 Caractérisation d'un schéma par un automate . . . 59
4.7.3 Caractérisation logique d'un schéma . . . 60

5 Évolutions 63
5.1 Schémas étendus . . . 63
5.1.1 Probabilités explicites . . . 63
5.1.2 Itération . . . 64
5.2 Optimisation de l'implantation . . . 65
5.2.1 Méthode de progression alternative . . . 65
5.2.2 Représentation optimisée . . . 66
5.3 Perspectives d'amélioration . . . 67
5.3.1 Schémas de remplacement . . . 67
5.3.2 Variation de l'influence . . . 68
5.3.3 Repérage d'un schéma sur une trace . . . 68
5.3.4 Combinaison avec d'autres méthodes . . . 68

II Application des méthodes de test à la validation de services de télécommunication 71

6 Modélisation 73
6.1 Contexte de modélisation . . . 73
6.2 Méthodologie de modélisation . . . 75
6.2.1 Choix d'un niveau d'abstraction . . . 76
6.2.2 Matérialisation du cœur réactif du modèle . . . 77
6.2.3 Organisation de la modélisation . . . 77
6.2.4 Choix d'un mode de communication . . . 77
6.2.5 Choix des mécanismes de composition et d'intégration . . . 79
6.2.6 Représentation d'un service . . . 80
6.2.7 Élaboration des propriétés de service . . . 81
6.3 Mise en œuvre sur une étude de cas . . . 82
6.3.1 Énoncé du concours FIW'2000 . . . 82
6.3.2 Application de la méthodologie . . . 85
6.4 Conclusion . . . 89

7 Principes de validation **91**
7.1 Description de l'environnement . . . 91
7.2 Élaboration des guides . . . 94
7.3 Définition de l'oracle . . . 95
7.4 Mise en œuvre du test . . . 96
7.5 Analyse . . . 98
7.5.1 L'oracle comme révélateur de situation . . . 98
7.5.2 Enrichissement de l'oracle par indicateurs . . . 99
7.6 Résultats et évaluation des études de cas . . . 101
7.6.1 Réalisation . . . 101
7.6.2 Résultats . . . 102
7.6.3 Influence du mode de composition de services . . . 102
7.6.4 Influence de l'ordre de composition . . . 103

III Travaux connexes et conclusions **105**

8 Travaux connexes **107**
8.1 Modélisation . . . 108
8.1.1 Utilisation de logiques temporelles linéaires standards . . . 108
8.1.2 Modélisation à base de processus communicants . . . 109
8.1.3 Piles d'automates . . . 111
8.1.4 Plug-and-play . . . 112
8.1.5 Langages dédiés . . . 112
8.2 Détection d'interactions et validation . . . 113
8.2.1 Analyse statique . . . 113
8.2.2 Model-checking . . . 116
8.2.3 Test statistique . . . 116
8.3 Autres travaux apparentés . . . 117
8.3.1 Test et simulation . . . 117
8.3.2 Test de conformité . . . 118
8.3.3 Test fonctionnel de systèmes réactifs . . . 120
8.3.4 Test guidé de programmes réactifs . . . 120

9 Test de propriétés non-invariantes **121**
9.1 Le problème de non-invariance . . . 121
9.1.1 Notion de vivacité . . . 121
9.1.2 La non-invariance pour les services de télécommunication . . . 122
9.1.3 Limites du test . . . 122
9.2 Proposition . . . 123
9.2.1 L'indice de confiance . . . 124
9.2.2 Intervention du guidage . . . 125
9.3 Expérimentation . . . 125
9.3.1 Énoncé . . . 126
9.3.2 Mode opératoire . . . 127

9.3.3 Résultats et analyses 127
9.4 Généralisation et conclusion 130

10 Conclusions et perspectives **133**
10.1 Conclusion 133
10.2 Perspectives 135
10.2.1 À propos de modélisation 135
10.2.2 À propos de la représentation de l'environnement 136
10.2.3 À propos de la méthodologie de validation 136
10.3 Perspectives 138

Annexes **146**

A Modélisation LUSTRE **147**

B Concours FIW'98 **153**

C Concours FIW'2000 **183**

D Propriétés de services du concours FIW'98 **203**
D.1 Définitions préliminaires 204
D.2 Propriétés du service de base 205
D.3 Propriétés des services 206

Chapitre 1

Introduction

Le domaine des télécommunications connaît à l'heure actuelle un essor formidable, dû en particulier sur le plan industriel à l'ouverture à la concurrence du marché de la téléphonie fixe, combinée à l'explosion du marché de la téléphonie mobile et à la vulgarisation de l'accès à l'Internet.
Les opérateurs et fournisseurs de services se font plus nombreux, les services s'orientent vers la mobilité, la communication multimedia et la conversation multi-acteurs. Cette diversification et cette multiplication de l'offre de services est soutenue par une évolution technologique importante, du fait de l'amélioration des réseaux de télécommunication. L'introduction du concept de "réseau intelligent" [33, 67] a permis en particulier d'aboutir à une architecture où la plupart des composants sont logiciels, facilitant ainsi l'introduction de nouveaux services. L'essor d'Internet, et son utilisation pour les télécommunications, tend à accentuer encore cette évolution.

Validation de services et interactions

La validation de ces services à divers stades de leur conception est indispensable pour assurer le bon fonctionnement du système de télécommunication qui les propose. Ce besoin de validation est d'autant plus important que la complexité des services va en augmentant.

Pour valider un service, il ne suffit pas de s'assurer que son fonctionnement propre est correct ; il est également nécessaire de vérifier si sa coexistence avec d'autres services risque ou non de conduire à une interaction. Une interaction peut apparaître lorsque plusieurs services, conçus indépendamment, sont proposés concurremment par un même système de télécommunication. Elle se manifeste par le fait que l'exécution de l'un des services, voire de plusieurs, se voit altérée. Considérons par exemple un usager ayant souscrit à un service de transfert d'appel et disposant également d'une messagerie vocale. Lorsqu'un appel lui parvient, celui-ci est censé être à la fois transféré et traité par le service de messagerie, ce qui n'est pas possible. Quel que soit le service qui traitera finalement l'appel, le comportement souhaité par l'autre service ne sera évidemment pas observé, ce qui constitue une altération de son exécution. Cette altération est identifiée dans la littérature par le terme d'*interaction*. Les modifications induites sur le comportement des services peuvent être voulues et sou-

haitées, notamment lorsqu'un service est conçu comme une extension d'un autre ; elles sont néanmoins le plus souvent inattendues, voire indésirables.

Parmi les interactions inattendues, si certaines, inoffensives, ne prêtent forcément à conséquence, d'autres, néfastes, peuvent conduire à un comportement anormal pouvant avoir des conséquences désastreuses : effondrement du réseau, pertes financières, insatisfaction ou mécontentement de la clientèle, ... Il est donc indispensable de pouvoir les détecter afin de pouvoir neutraliser leurs effets, dans le double objectif d'assurer la sûreté de fonctionnement du système et la qualité des services offerts.

Le problème de l'interaction de services a été soulevé il y a maintenant plus de dix ans [15]. Il connaît à l'heure actuelle un véritable engouement, tant dans les milieux universitaires qui trouvent là un champ d'expérimentations vaste et intéressant, le problème ayant de multiples facettes, que dans les milieux industriels qui ont pleinement conscience de son acuité.

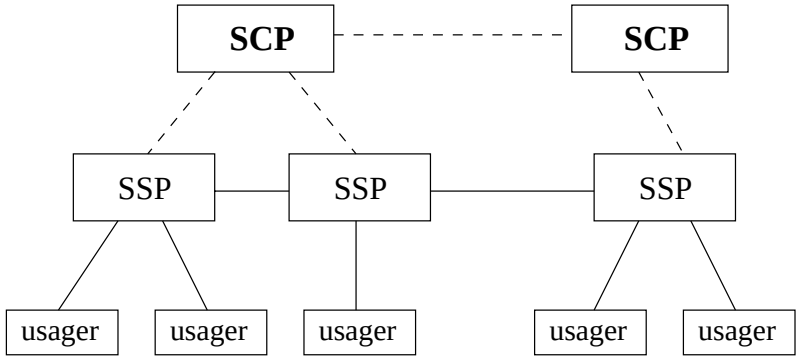
La nécessité d'automatiser la validation de services

Ainsi, la complexité d'une activité de validation de services vient notamment de l'existence du problème d'interaction entre ces services. Tant que l'offre de services reste de taille raisonnable, le problème de l'interaction peut être considéré comme marginal. En tout cas, il peut être analysé et résolu manuellement par un expert humain connaissant parfaitement l'ensemble de l'offre disponible. Ceci n'est envisageable que dans un contexte de marché faiblement développé, ce qui n'est plus le cas aujourd'hui, où l'offre de services augmente considérablement, notamment du fait du nombre croissant de fournisseurs.

L'augmentation du nombre de services, ainsi que la diversité de leur origine sont en train de rendre prohibitive toute technique d'analyse manuelle d'interaction au cas par cas. Des méthodes de détection et/ou de résolution automatiques deviennent donc nécessaires, pour des raisons de complexité et de coût.
C'est dans ce cadre que le CNET-France Telecom a entrepris en 1996 une collaboration avec l'équipe *Production Fiable de Logiciels* du laboratoire *Logiciels, Systèmes, Réseaux* dans l'objectif de caractériser un langage de spécification permettant l'expression de services au niveau logique, et de développer les outils de validation de ces spécifications.

L'évolution logicielle des systèmes de télécommunication

L'évolution des systèmes de télécommunication jusqu'à présent a abouti à une architecture désignée sous le terme de *Réseau Intelligent* [67], où la plupart des composants, notamment les services, sont logiciels. Le réseau intelligent est une solution appropriée pour favoriser le fonctionnement cohérent d'un ensemble intégré de services, en particulier grâce aux possibilités de contrôle logiciel des interactions entre les services de l'ensemble.



SSP point de commutation ———— réseau de commutation

SCP point de contrôle - - - - - - - réseau de signalisation

FIG. 1.1 – *Architecture du réseau intelligent*

Il s'agit de plus d'un effort de normalisation rendu nécessaire par la multiplication du nombre de fournisseurs de services. L'intérêt principal des réseaux intelligents est leur capacité à orchestrer l'exécution de services à partir de quelques points du réseau clairement identifiés. Ces points, appelés *points de contrôle*, sont reliés au réseau de commutation (composés des commutateurs, repérés sous le nom de *points de commutation*) grâce à un réseau dédié, dit réseau de signalisation. La standardisation d'un tel concept architectural permet la libre concurrence pour la conception des composants du réseau ainsi que pour celle des logiciels applicatifs (les services) destinés aux points de contrôle.

La figure 1.1 schématise les principales entités constituant le réseau intelligent, ainsi que leurs relations (cf. également [22]).

Les commutateurs (*Service Switching Point*, SSP) sont des relais permettant de mettre en communication deux usagers distants en établissant entre eux un canal de connexion logique. Chaque commutateur gère les usagers qui lui sont directement reliés, et il mémorise leur état. Les contrôleurs (*Service Control Point*, SCP) hébergent les services et récupèrent depuis les commutateurs les données dont les services ont besoin concernant l'évolution de l'état des usagers. En retour, les contrôleurs indiquent aux commutateurs les modifications de connexion ou d'état à effectuer. Chaque commutateur ne dépend que d'un seul contrôleur. Afin de pouvoir interroger un commutateur dont il n'a pas le contrôle, un contrôleur demandera ces renseignements au contrôleur dont dépend le commutateur concerné, en utilisant le réseau de signalisation.

Les caractéristiques de ces composants logiciels permettent de penser que toute approche issue du génie logiciel mérite d'être étudiée dans le contexte de la modélisation, de la conception, du développement et de la validation de services. La présente thèse montre que les techniques de modélisation et de validation conçues pour les systèmes réactifs sont en particulier bien adaptées pour aborder ces problèmes.

Proposition

Nous nous proposons dans cette thèse d'aborder le problème de la validation de services au niveau des spécifications en se basant sur la modélisation formelle dans un cadre synchrone et sur des techniques bien fondées de validation par le test.

La validation de spécifications est une activité d'une utilité certaine dans tout cycle de développement logiciel. En effet, elle intervient tôt dans ce processus et permet de déceler précocement tout défaut de conception qui risquerait autrement de se propager aux phases suivantes de développement. Le coût de correction de ces défauts est par suite bien moindre que s'ils avaient été détectés lors d'une phase ultérieure.

Les approches formelles et leurs méthodes associées acquièrent actuellement une certaine reconnaissance dans l'aide à la construction logicielle de qualité. Elles permettent en particulier de décrire des systèmes de manière rigoureuse et non-ambiguë et sont un support à l'automatisation du processus de validation. L'approche synchrone, que nous détaillerons plus loin (cf. section 2.4) a fait ses preuves dans des domaines partageant un certain nombre de caractéristiques avec celui des télécommunications, tel que celui des protocoles ou des systèmes critiques. La justification de nos choix, et notamment celui du test comme support de la validation, sera présentée au chapitre 2.

Ma contribution s'inscrit dans la continuité des travaux menés par l'équipe *Production Fiable de Logiciels* depuis 1994 autour du test de logiciels synchrones. I. Parissis [83] a posé les fondements théoriques de ce test et le principe de génération aléatoire de données sur lequel celui-ci se base. Il a également élaboré un premier prototype d'outil de test, LUTESS, incluant deux méthodes de génération.
L. du Bousquet [28] a ensuite poursuivi ce travail en apportant la preuve du bon fonctionnement du prototype par des tests d'hypothèses statistiques. Elle a aussi proposé une extension à l'une des techniques de génération, extension qu'elle a implanté dans LUTESS et validé expérimentalement.
Le travail présenté ici s'attache principalement à proposer une nouvelle méthode de génération, validée expérimentalement et fondée formellement. L'application de cette méthode au contexte particulier qu'est la validation de services a permis de montrer son efficacité et son utilité. Un important travail méthodologique a également été réalisé pour aborder le problème de la validation de services.

Structure du document

Ce document est constitué de deux grandes parties. La première est consacrée aux travaux que nous avons menés autour du test de programmes synchrones. Le chapitre 3 décrit l'environnement de test développé par mon équipe d'accueil, tandis que le chapitre 4 expose plus précisément ma contribution à cet environnement, sous la forme d'une méthode de test originale et efficace basée sur la notion de schéma comportemental. Le chapitre 5 termine

cette partie en discutant des évolutions possibles de la méthode.

La deuxième partie aborde le problème de la validation de service téléphoniques. Le chapitre 6 décrit une méthodologie de modélisation de services sous forme d'une spécification formelle. Le chapitre 7 s'intéresse à la manière de valider les spécifications ainsi obtenues, et détecter les interactions qui peuvent exister entre elles.

Une troisième partie dresse le bilan : le chapitre 8 compare nos travaux à d'autres approches s'y rapportant, le chapitre 9 propose une utilisation des schémas comportementaux au test de propriétés non-invariantes tandis que le chapitre 10 conclut sur notre travail et en ébauche les perspectives.

Chapitre 2

Objectifs

Ce chapitre définit les objectifs que nous avons fixés pour ce travail, et mentionne certains de nos choix les plus importants.

2.1 Définitions préliminaires

Qu'est-ce qu'un service ?
L'Union Internationale des Télécommunications (ITU) définit un service comme "une offre commerciale autonome constituée par un ou plusieurs éléments de services", chaque élément de service fournissant une fonctionnalité particulière pouvant être réutilisée en relation avec d'autres services ou compléments. Par exemple, l'acheminement d'un appel en fonction de sa zone géographique d'origine est un élément de service pour le service *numéro vert*, mais peut également être utilisé en tant que service autonome.

Pour être une offre commerciale autonome, un service n'est pas pour autant indépendant, et requiert un support sur lequel il pourra venir se greffer. Ce support est fourni par le service de base, qui correspond au traitement ordinaire d'appel. Ce service de base a pour fonction d'établir, maintenir et clôturer les connexions nécessaires à la mise en communication des abonnés qui en font la demande. Il gère également quelques aspects annexes comme la facturation.

Cycle d'exécution d'un service
Afin de préciser le vocabulaire qui sera utilisé dans la suite de ce document, nous présentons ici les différents termes pouvant se référer à la mise en œuvre d'un service, tels que définis par l'ETSI [37].

- `Réalisation`. Cette phase consiste à concevoir et spécifier le service, puis à l'implémenter.

- `Provisionnement` ou `déploiement`. Cette phase consiste à introduire le service dans le système, de manière à le rendre disponible auprès des utilisateurs.

- `Souscription` ou `abonnement`. Le fait pour un usager de demander à bénéficier du service, par exemple en s'inscrivant auprès du fournisseur.
- `Initialisation` ou `paramétrisation`. Le fait pour l'usager de préciser le cas échéant certaines informations variables nécessaires au fonctionnement du service (Cette étape peut avoir une influence sur les conditions de déclenchement du service ou plus précisément sur son exécution.).
- `Activation`. Cette phase consiste à indiquer au service qu'il doit être prêt à se déclencher.
- `Déclenchement`. Le service se déclenche lorsque certaines conditions requises pour son exécution sont remplies.
- `Exécution`. L'exécution du service consiste à modifier temporairement et localement le comportement du système de télécommunication.
- `Terminaison`. Le service se termine lorsque le traitement correspondant à son exécution est achevé.
- `Désactivation`. Opération inverse de l'activation.
- `Désabonnement`. Le fait pour l'usager de mettre fin à sa souscription et de renoncer à l'utilisation du service.
- `Retrait`. Le fait de rendre le service indisponible sur le réseau.

2.2 Axes classiques de recherche concernant la validation de services

Ainsi qu'on l'a mentionné en introduction, la validation d'un service ne peut se faire de manière isolée, du fait de la possible existence d'interactions entre services. En particulier, il est nécessaire de confronter le service à valider à l'ensemble des services avec qui il devra coexister.

La plupart des travaux de recherche concernant la validation de services portent sur l'application de techniques déjà éprouvées dans le domaine du génie logiciel. Les plus spécifiques se concentrent sur l'étude du problème d'interaction de services, sans forcément aborder la question plus générale de la validation de services. Ce sont ces axes de recherche que nous présentons ici, sans prétendre restreindre l'activité de validation à la seule résolution du problème d'interaction.

Il existe trois types d'approches concernant la gestion des interactions. La première vise à concevoir des architectures adaptées favorisant le déploiement et la coexistence de services, en résolvant notamment les problèmes liés à l'aspect distribué du système. C'est le cas notamment des travaux autour de TINA [66], mais d'autres travaux indépendants proposent

aussi des solutions architecturales matériels [101], mais aussi logicielles [99, 89]. Cette première catégorie a pour objectif l'*évitement* des interactions.

Un deuxième axe de recherche concerne la conception des services eux-même, avant leur déploiement sur le réseau. Les travaux correspondants s'appuient pour la plupart sur des techniques formelles de génie logiciel, plus précisément de vérification, validation et test [50, 9, 60, 43]. L'objectif de ces travaux est la détection des interactions.

Enfin, le troisième axe se consacre à l'étude et à la mise en place de mécanismes de résolution des interactions, une fois qu'elles se manifestent, après le déploiement des services sur le réseau [46, 97, 98, 23, 2].

Dans les deux dernières catégories, on peut distinguer les méthodes "off-line", qui s'appliquent lors de la phase de conception, et les méthodes "on-line" qui sont mises en œuvre une fois que le service est offert au public.

Bouma et Velthuijsen résument dans [13] sous la forme du tableau suivant les différents types d'approches en fonction de leur catégorie :

	approches *off-line*	approches *on-line*
Évitement	- plateformes de création de services - protocoles de communication dédiés - directives de programmation - ...	
Détection	- outils de vérification - outils de simulation - outils de validation - ...	- analyse de comportement - ...
Résolution	- correctifs de conception - correctifs d'implémentation - étude des besoins clients - ...	- gestionnaire d'interactions - mécanismes de négociation - restrictions de souscriptions

2.3 Présentation de l'approche choisie

Le travail que nous présentons ici a pour objectif principal de mener à la validation de services de télécommunication, le problème de la détection d'interaction étant traité comme un cas particulier de cette validation. Nous avons choisi de nous intéresser à la détection des interactions logiques, provenant d'une divergence de conception des services. Ceci nous permet de nous abstraire des aspects matériels de la réalisation et d'intervenir très tôt dans le cycle de développement du service, par exemple lors de sa spécification. Ce choix est conforté par le fait que, dans la pratique, la plupart des interactions néfastes relevées sont dues à des conflits d'intérêt de nature logique, sans lien avec la réalisation proprement dite des services.

Notre approche consiste donc à valider les spécifications des services et à détecter les

interactions *logiques* présentes entre elles.

Le choix d'agir au niveau des spécifications permet à la fois de raisonner sur des modèles abstraits et simplifiés, donc facilement manipulables et d'intervenir très tôt dans le cycle de développement du service, permettant ainsi de minimiser les coûts induits par une erreur : plus une erreur est révélée tardivement, plus l'effort nécessaire à sa correction sera important. Notre objectif principal est donc de constituer une aide à l'expertise, en permettant de restreindre l'intervention humaine à l'analyse de résultats de test.

Modéliser une spécification est une activité d'une utilité certaine, que ce modèle donne lieu à la génération automatique d'une implémentation ou pas. En effet, la validation d'un tel modèle permet d'assurer que la spécification sera non-ambiguë et cohérente, et élimine les erreurs de conception, tout en réduisant la complexité de l'activité d'implémentation [61, 50].

L'approche que nous proposons ici est par nature similaire à de nombreux travaux, consistant à réaliser une modélisation formelle du système de télécommunication et des services, puis à procéder à une validation desdits modèles. Ces travaux consistent en l'application de formalismes déjà utilisés avec succès dans d'autres domaines [25, 39, 75], ou en la mise au point de formalismes dédiés [50, 99, 16], ou enfin en la définition logique et formelle d'une interaction [9, 44]. On trouvera au chapitre 8 une discussion sur le positionnement de notre approche par rapport aux travaux analogues, ainsi que leur comparaison.

L'originalité de notre travail tient en deux points :

- La modélisation est fondée sur l'hypothèse de *synchronisme* [8]. Cette hypothèse permet de réaliser un certain nombre d'abstractions qui rendent le modèle plus facilement manipulable, tout en demeurant réaliste.

  Le modèle que nous proposons est par ailleurs exécutable, ce qui permet de l'animer et le rend plus intuitivement appréhendable.

- La validation s'opère par le biais d'un test automatisé et se base sur des méthodes de test orienté bien adaptées au problème. Outre le fait que l'utilisation du test est très peu répandue dans ce domaine, les méthodes que nous proposons sont originales et performantes.

2.4 L'approche synchrone

Un logiciel (ou système) réactif est un logiciel en interaction permanente avec son environnement et dont l'exécution est théoriquement infinie [51]. Son mode de fonctionnement est cyclique, le logiciel fournissant une réponse à chaque action de l'environnement.

De tels logiciels se distinguent des programmes informatiques classiques, dits *transformationnels*, qui acquièrent à l'initialisation leurs données et fournissent leur résultat à la

terminaison.

On trouve de nombreux exemples de systèmes réactifs dans les processus de contrôle-commande de procédés physiques et plus généralement dans nombre de systèmes dits critiques, c'est-à-dire dont le bon fonctionnement est indispensable pour raisons de sécurité.

Ces logiciels doivent obéir à de strictes contraintes temporelles ; en effet, à la différence des programmes dits *interactifs*, les programmes réactifs évoluent à la vitesse imposée par leur environnement, c'est-à-dire qu'ils doivent réagir plus rapidement que ce dernier. Un programme interactif au contraire peut avoir un délai de réponse important : c'est le cas d'applications de billeterie automatique par exemple.

La conception et la programmation de tels systèmes a commencé par faire appel à des langages traditionnels tels que C ou Ada, mais le degré de sûreté qu'exigent ces systèmes ont rapidement conduit à opter pour des langages plus spécialisés.

Des formalismes appropriés à la complexité et aux particularités de ces systèmes ont donc été développés. Leur hypothèse de base commune consiste à supposer que la réaction du système est instantanée, c'est-à-dire que le système a un temps de calcul égal à zéro (hypothèse dite de *synchronisme*) [8]. Cette façon de voir les choses revient à s'assurer que le système est capable de prendre en compte toute évolution significative de son environnement. Dans la pratique, cette hypothèse exige simplement du système qu'il réagisse plus vite que son environnement. Ces formalismes doivent également prendre en compte les aspects suivants :

- Les systèmes à considérer sont déterministes, i.e. leur réaction ne dépend que du comportement de leur environnement.
- Ces systèmes sont de plus non-bloquants, en cela que toute entrée est acceptable à tout moment.

La communauté "synchrone" a ainsi développé depuis une quinzaine d'années un certain nombre de langages adaptés aux caractéristiques de tels systèmes. Parmi eux, citons Esterel [14] dont le style de programmation est impératif, Lustre [21] et Signal [70], langages déclaratifs à flot de données. Nos travaux portent en particulier sur le langage Lustre, pour lequel la communauté universitaire grenobloise est très active.

Adéquation du synchrone à la modélisation de services téléphoniques

De nombreuses expériences ont montré que l'approche synchrone est adaptée à la modélisation de services de télécommunication [106], et plus généralement à celle, totale ou partielle, de systèmes de télécommunication [57, 77].
En effet, les systèmes de télécommunication présentent de nombreuses caractéristiques des systèmes réactifs : ils ont à réagir aux sollicitations d'un environnement dans des délais les plus brefs possibles. Certains de leurs composants peuvent être considérés comme synchrones [9, 6].

Le principal intérêt d'une approche synchrone repose sur la nature simplificatrice de l'hypothèse de synchronisme qui permet entre autres de faire abstraction des communications inter-entités. Le modèle est donc plus simple et plus compact, car moins de cas sont à considérer.

Par suite, cette manière de faire est intéressante dans la phase de validation du modèle, car elle permet d'éviter le problème d'explosion d'états fréquemment rencontré dans les méthodes classiques de validation de modèles asynchrones.

2.5 Justification d'une validation fondée sur le test

La détection d'interactions est typiquement une activité de recherche de défaut. Or, le test a justement cet objectif, à la différence des techniques de preuve ou "model-checking", dont le but est de prouver la correction du logiciel. Le test est donc bien adapté à la résolution d'un tel problème, bien qu'il offre une garantie moins absolue quant à la qualité du logiciel.

De plus, le test que nous proposons est dynamique, dans la mesure où les séquences de test sont générées "à la volée" (*on-the-fly testing*), en fonction des réactions du système sous test. Ceci permet de procéder au test quelle que soit la connaissance que l'on a du système sous test. Enfin, lorsque la production de données de test est effectuée statiquement (*batch testing*), celle-ci requiert des calculs complexes (puisqu'il faut prévoir les différentes réactions possibles du système) et se heurte au problème de l'explosion du nombre d'états [7]. Un choix de test "à la volée" permet d'éviter ces inconvénients.

2.6 Présentation de la méthode de modélisation

Chaque service S_i est composé d'un modèle exécutable décrivant son comportement, B_i, et d'une formule de logique temporelle traduisant la conjonction des propriétés que le service est censé satisfaire, P_i.

On se préoccupe essentiellement dans cette section de la manière dont peut être modélisé le comportement de chaque service, ainsi que du service de base qui va leur servir de support. La description logique sous forme de propriétés fera l'objet de la section suivante sur la validation.

Chaque service est conçu comme une extension au service de base. On va donc d'abord chercher à modéliser le système "nu", sans services supplémentaires, c'est-à-dire ne fournissant que le seul service de base. Modéliser un service supplémentaire consistera à intégrer ce dernier au modèle du système "nu". Une des propriétés attendues de ce modèle est qu'il permette une intégration aisée. Accessoirement, il devra être évolutif de sorte que le service de base ou n'importe quel autre de ses composants puisse être remplacé. L'architecture du

modèle devra à cette fin être la plus modulaire possible.

En plus de l'opération d'intégration, le modèle du système doit fournir une opération de composition qui permet de faire coexister plusieurs services et de mener à bien leur intégration.

Beaucoup de travaux existants proposent une intégration de service qui modifie une grande partie du modèle pré-existant [1]. Ce mode de conception pose le problème de non-monotonie [102] : le modèle lui-même est modifié pour pouvoir accueillir les nouveaux services. En conséquence, la complexité du modèle résultant d'une intégration est plus importante que la somme de celles du modèle initial et du service. En particulier, l'intégration d'un service dans un modèle, suivie de sa suppression, ne conduit pas forcément au modèle d'origine. Le choix d'une modélisation modulaire a pour but d'éviter ce problème.

Concernant l'intégration de plusieurs services au service de base, plusieurs possibilités sont envisageables. On peut décider :

- d'une intégration incrémentale : le premier service est intégré dans le modèle fournissant le service de base, le suivant le sera dans le modèle résultant de cette première intégration, etc.
- ou d'un processus en deux étapes consistant à :
  1. composer entre eux l'ensemble de services, pour en faire une sorte de super-service,
  2. puis intégrer ce dernier dans le modèle "nu".

Les deux points de vue ont leurs avantages propres :

- Le premier a un intérêt opérationnel : on peut considérer le modèle résultant d'une première intégration comme un tout, sans avoir à procéder à l'extraction du service intégré pour le composer avec le nouveau service.
- Le second a un avantage de lisibilité structurelle : on voit ce qui fait partie du modèle de base, et ce qui constitue l'offre de service supplémentaire. Ceci permet de conserver une architecture la plus structurée et modulaire possible et d'éviter de faire du modèle un amas informe et non-analysable ("Big Ball of Mud" [42]).

La structure modulaire que nous proposons de mettre en œuvre (cf. chapitre 6) permet d'adopter indifféremment les deux points de vue.

Par ailleurs, l'utilisation d'un langage de spécification exécutable tel que LUSTRE nous a semblé être une approche intéressante notamment de par le fait qu'une telle technique de description facilite la mise en œuvre d'une loi de composition [102].

1. Hall [50] par exemple, (cf. section 8.1.5) décrit chaque service en réécrivant entièrement le service de base. Devant le très grand nombre de conflits "parasites" engendrés, il est arrivé à la conclusion que chaque service devait être décrit comme une extension au service de base, de sorte à minimiser la redéfinition du service de base.

2.7 Présentation de la méthode de validation

De manière grossière, l'approche retenue pour la validation d'un système réactif incluant un certain nombre de services consiste à développer trois spécifications :

- M - une description déterministe, fonctionnelle et exécutable, modélisant le comportement du système. Cette description est construite à partir de la spécification du comportement de chaque service (B_i), ainsi que de celle du système fournissant le seul service de base. Ce comportement est noté $\mathcal{N}$.

- P - une spécification des propriétés *attendues* du système, c'est-à-dire ce que l'usager peut espérer concrètement de son utilisation. Cette spécification s'appuie sur la spécification des propriétés P_i des différents services fournis par le système.

- E - Une spécification de l'environnement du système, définissant des comportements possibles de celui-ci.

La validation consiste ensuite à s'assurer que M satisfait P, sous l'hypothèse que E soit satisfaite.

La spécification de l'environnement définit un ensemble de contraintes que l'on sait respectées par celui-ci. Il s'agit en fait de modéliser l'environnement de manière réaliste afin de rendre plus pertinents ses échanges avec le système. En réalité, cette spécification n'est pas forcément indispensable : il est en effet très facile de programmer un système réactif de sorte qu'il soit non-bloquant et qu'il ne tienne pas compte des environnements irréalistes ou aberrants. Néanmoins, en spécifiant au moins partiellement la manière dont évolue l'environnement, on restreint l'ensemble des comportements qui seront à étudier.

La validation est assurée par une méthode de test aléatoire contraint, qui génère de manière dynamique les comportements de l'environnement respectant les contraintes spécifiées. La description de l'environnement peut également inclure un certain nombre de comportements typiques de l'environnement, comportements qui seront générés de manière privilégiée. Ces comportements "privilégiés" sont basés sur la notion de schéma comportemental, qui sera introduite au chapitre 4.

Ce principe consiste à tester le système en le sollicitant de manière aléatoire mais réaliste dès les premières étapes de sa réalisation. Un tel test basé sur les usages réels du système et leur fréquence est appelé *test statistique*. Il a été appliqué à grande échelle dans le cadre de la méthode de développement logiciel nommée "cleanroom" [76, 32]. AT&T a développé au travers du concept de profil opérationnel une idée similaire [78]. Le test statistique est notamment utilisé pour mener une analyse de la fiabilité du système.
De manière plus générale, la méthode présentée ici se base sur le principe de simulation guidée, déjà mis en œuvre dans d'autres contextes [82]. Ce principe consiste à circonscrire le travail de validation à un sous-domaine de l'espace d'états du système à tester, en spécifiant des contraintes sur l'exécution de ce dernier.

Lorsque le modèle du système comporte plusieurs services, chacun fournit sa propre spécification sous forme de propriétés. Pour identifier plus facilement les problèmes qui pourraient être révélés, la validation se fait de manière incrémentale :

- Dans un premier temps, chaque service est validé séparément, comme s'il était seul disponible dans le modèle, en plus du service de base. Cette première étape permet de s'assurer que chaque service satisfait bien les propriétés qu'on en attend. C'est également l'occasion de corriger ou d'affiner éventuellement la modélisation ou l'expression des propriétés attendues.

- La deuxième phase consiste à valider le modèle incluant l'ensemble des services à confronter. La propriété à satisfaire par le système est la conjonction des propriétés attendues de chacun des services présents. Les problèmes révélés par cette phase sont des interactions de services.

2.8 Définition formelle d'une interaction

La notion d'interaction contient une certaine part de subjectivité, qui rend impossible sa caractérisation rigoureuse. En effet, chacun peut interpréter de manière sensiblement différente la description d'un service et l'appel à un expert demeure donc indispensable pour statuer objectivement sur la nature néfaste d'un cas d'interaction.

Nous proposons ici de préciser la notion d'interaction afin de fournir un cadre formel à sa détection et de faciliter ainsi le travail d'expertise requis. La définition que nous allons proposer ici est basée sur la notion de satisfaction de modèle et s'inspire des travaux de Combes [25]. Ce dernier propose la définition suivante : Soient $\mathcal{N}$ le modèle d'un système de télécommunication, $S_1 ... S_n$ n services. On représente par $\models$ la relation de satisfaction d'une propriété logique par un modèle et par $\oplus$ l'intégration de service dans un modèle. Une interaction existe lorsque :

$$\begin{cases} \forall i \in 1,..,n, \mathcal{N} \oplus B_i \models P_i \\ \mathcal{N} \oplus B_1 ... \oplus B_n \not\models \bigwedge_{i=1,..,n} P_i \end{cases}$$

Cette définition ne nous apparaît pas pleinement satisfaisante pour principalement deux raisons :

- La notion de composition de services n'apparaît pas, seule est présente celle d'intégration ; or nous avons vu en section 2.6 qu'il était souhaitable de pouvoir traiter séparément ces deux aspects.

- Les seules propriétés considérées sont celles associées à un service. En particulier, aucune propriété n'est fournie concernant le système de télécommunication proprement dit.

Nous allons donc procéder à certaines adaptations, de manière à rendre plus explicite la définition.

Composition et intégration
Suite à la distinction souhaitable que nous avons exprimée entre ces deux notions dans la section précédente, nous représenterons par $B_1 \oplus ... \oplus B_n$ le comportement du service composite formé par les services S_1, ..., S_n et $\frac{B_1 \oplus ... \oplus B_n}{\mathcal{N}}$ l'intégration de ce super-service dans le système nu.
On notera que l'opérateur $\oplus$ n'est pas commutatif, i.e. l'ordre de composition est significatif.

Propriétés du système et propriétés de services
Nous avons également fait le choix d'exprimer des propriétés non-associées à un service en particulier, mais associées au système "nu". Ces propriétés concernent plus particulièrement le service de base et la cohérence des données du modèle. Ces propriétés sont en général respectées par une majorité de services, mais peuvent être ponctuellement mises en défaut par certains services, sans que cela ne traduise un problème.

Ceci conduit à considérer deux niveaux d'interactions dans notre approche :

- Le premier niveau est observé lors de la mise au point d'un service isolé, lorsque l'on confronte ce dernier aux propriétés du système "nu". En effet, le service peut dans une certaine mesure modifier le comportement du service de base, et donc affecter les propriétés globales que l'on a pu émettre sur le système initial. Les interactions ainsi constatées sont pour la plupart prévues et attendues. Elles permettent à ce niveau d'avoir une connaissance précise de l'influence du service sur le comportement du système. On note $P_{\mathcal{N}}$ l'ensemble des propriétés offert par le système "nu". $P_{\mathcal{N}|S_i}$ représente l'ensemble des propriétés du système satisfaites lorsque le service S_i est intégré.

- Le deuxième niveau concerne la composition de plusieurs services. Les services peuvent avoir des objectifs diamétralement opposés. Les chances d'interactions néfastes sont beaucoup plus importantes, et seul l'expert peut trancher. À ce niveau, certaines propriétés du système peuvent également être mises en défaut, alors qu'elles ne l'étaient pas par chaque service pris indépendamment. On peut donc avoir $P_{\mathcal{N}|S_1} = P_{\mathcal{N}|S_2} = P_{\mathcal{N}}$, mais $P_{\mathcal{N}|S_1 \oplus S_2} \neq P_{\mathcal{N}}$

Les propriétés du système "nu" permettent également de faire une distinction entre les propriétés "critiques" (propriétés de service), dont la mise en défaut signale fort probablement un problème et les propriétés non-critiques (propriétés du système) dont on peut tolérer la mise en défaut par un service.

Définition de l'interaction On peut alors compléter la définition de l'interaction énoncée plus haut comme suit :

$$\begin{cases} \frac{B_i}{\mathcal{N}} \models P_i \wedge P_{\mathcal{N}|S_i}, 1 \leq i \leq n & \text{V}_1 \\ \frac{B_1 \oplus ... \oplus B_n}{\mathcal{N}} \not\models \bigwedge_{i=1..n}(P_i \wedge P_{\mathcal{N}|S_i}) & \text{V}_2 \end{cases}$$

V_1 correspond à la validation de chaque service de manière isolée, tandis que V_2 correspond à l'évaluation de la composition de services en rapport avec la conjonction des propriétés

de ces services. Cette définition formelle de l'interaction possède certaines limites ; en particulier, elle oblige à considérer les services comme des entités atomiques et ne permet donc pas de construire un service à partir d'autres services existants. Par ailleurs, elle ne s'intéresse qu'aux interactions d'ordre logique. Elle est cependant bien adaptée à la résolution du problème qui nous intéresse, i.e. la validation de spécifications de services.

Première partie

Contribution à un environnement de test efficace

Chapitre 3

L'environnement de test LUTESS

Ce chapitre présente l'environnement de test qui a servi de cadre à mes travaux concernant la conception et la réalisation de méthodes de test performantes et efficaces. Ceux-ci seront décrits au chapitre suivant.

LUTESS est un environnement de test pour la validation de programmes réactifs synchrones à entrées/sorties booléennes [80, 83, 30]. LUTESS permet de procéder à une activité de test fonctionnel (dit "boîte noire"), à partir de spécifications formelles éventuellement incomplètes et à l'aide de plusieurs techniques de test [30, 31, 80]. LUTESS est basé sur le langage synchrone LUSTRE [21, 48].

Étant donné un système réactif (logiciel, matériel ou mixte) à tester, le principe de base de LUTESS est de *simuler* son environnement et de contrôler les échanges entre ces deux entités. À partir d'une description de l'environnement, LUTESS permet la production automatique de données de test qui seront fournies au système sous test, l'objectif étant *in fine* d'analyser le comportement de ce dernier avant immersion dans son environnement réel. LUTESS propose à cette fin une génération automatisée de *verdicts*, ainsi qu'une assistance dans l'analyse des traces résultant du test.

LUTESS est dédié au test des systèmes synchrones dont les entrées et sorties sont booléennes. Cette condition peut sembler restrictive ; dans la pratique, elle ne s'est jamais révélée réellement problématique, notamment grâce au fait qu'il est souvent possible de réaliser des abstractions sur les données manipulées. On remarquera en particulier qu'à partir du moment où les grandeurs considérées peuvent être bornées, cette abstraction se ramène à une simple discrétisation.
Cette hypothèse permet de faire appel à des algorithmes de génération efficaces qui seraient inutilisables sur des données numériques.

3.1 Le langage LUSTRE

LUSTRE est un langage flot de données synchrone qui peut également être considéré comme

une logique temporelle du passé [85].

Un programme LUSTRE est essentiellement composé d'un ensemble d'équations définissant des flots. Un flot représente la séquence, théoriquement infinie, des valeurs prises par une variable. Un flot possède une horloge associée qui indique la fréquence à laquelle la variable peut changer de valeur. Tout flot prend sa n-ième valeur au n-ième cycle de son horloge. Un flot peut être décrit comme la succession de valeurs prises par la variable qu'il représente : $(f_0, f_1, ... f_n, ...)$ pour un flot F.

De manière générale, toute l'exécution d'un programme LUSTRE repose sur la notion d'horloge : le temps est discrétisé et évolue à la fréquence imposée par l'horloge du programme, dite *horloge de base*.

En plus des opérateurs arithmétiques classiques, LUSTRE dispose de deux opérateurs temporels pour manipuler les flots : *previous* (**pre**) et *followed-by* ($\Rrightarrow$).
À un instant n, l'opérateur **pre** permet d'accéder à la valeur du flot à l'instant $n \Leftrightarrow 1$. Ainsi, le flot **pre** F dénotera la séquence $(\perp, f_0, ... f_{n-1}, ...)$, où $\perp$ est une valeur indéfinie.
L'opérateur $\Rrightarrow$ permet quant à lui de fixer la valeur initiale d'un flot. Soient E et F deux flots, d'expressions $(e_0, e_1, ... e_n, ...)$ et $(f_0, f_1, ... f_n, ...)$, $E \Rrightarrow F$ dénotera le flot $(e_0, f_1, ... f_n, ...)$.
Les deux opérateurs **pre** et $\Rrightarrow$ sont fréquemment utilisés en association.

Un programme LUSTRE est réactif et synchrone : chaque donnée d'entrée induit une réaction immédiate. À chaque cycle de l'horloge de base, toutes les entrées sont lues et toutes les sorties sont fournies.
Un programme LUSTRE est par ailleurs non-bloquant : les entrées fournies à chaque instant sont acceptées, quelles qu'elles soient, et induisent une réaction.

3.2 Principe de fonctionnement de LUTESS

Il existe deux phases principales dans un processus de test : la dérivation des cas de test (c'est-à-dire les séquences d'entrée correspondant à une exécution du système à tester) et leur exécution [7]. Les deux peuvent être automatisées, que ce soit séparément ou de manière intégrée. Le choix de LUTESS est d'intégrer les deux phases en opérant une *simulation dynamique* de l'environnement du système à tester.
Ce choix est motivé par le fait qu'une automatisation séparée aboutit à la production de cas de test de taille parfois importante, puisqu'ils doivent tenir compte des différentes réponses possibles de l'objet à tester. Le stockage de ces donnés (au format TTCN, par exemple [54]) peut en conséquent s'avérer coûteux. De plus, le problème de l'explosion du nombre d'états, rencontré en vérification, peut également se poser.
Les intérêts des techniques de génération "au-vol" (c'est-à-dire lorsque les deux phases sont intégrées) ont été montrés dans plusieurs études [7, 40] et ont été mentionnés à la section 2.5.

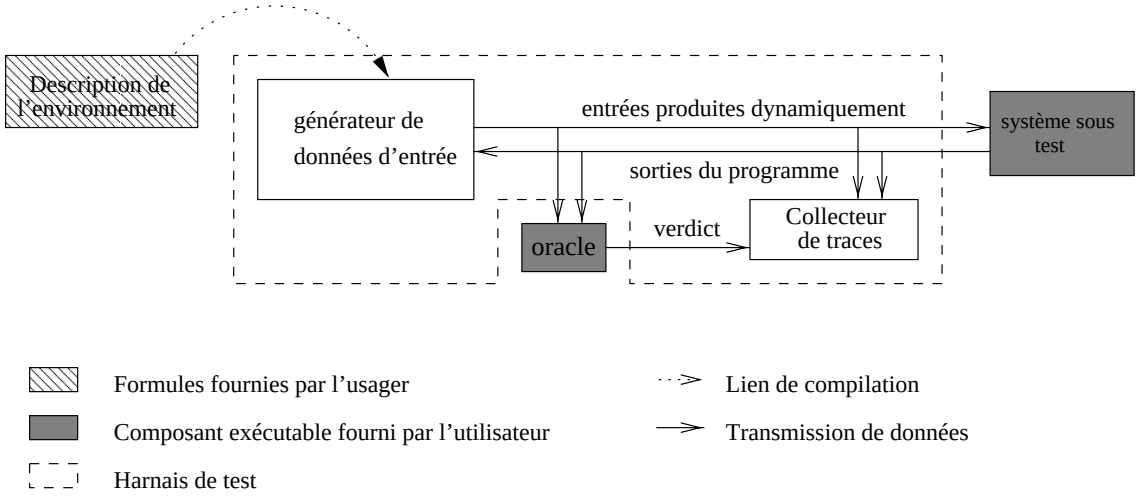


FIG. 3.1 – *Architecture de l'outil* LUTESS

LUTESS fonctionne sur un cycle simple de production/acquisition :

– production d'*une* donnée d'entrée transmise au système sous test,

– acquisition de la réaction correspondante du système.

Chaque donnée est produite en tenant compte de l'historique des échanges entre le système et son environnement. Il n'est donc pas nécessaire de savoir exactement de quelle manière va réagir le système à une entrée particulière. Ceci permet de mettre à profit toute connaissance parcellaire que l'on a du système.

Plus précisément, LUTESS requiert pour son fonctionnement les éléments suivants (cf. figure 3.1) :

– un système sous test, sous la forme d'un exécutable synchrone,

– une description de l'environnement sous forme de contraintes,

– un oracle, décrivant les propriétés que doit satisfaire le système.

LUTESS fournit également un collecteur de traces, permettant de rassembler et de formater les données à analyser. Ce collecteur est paramétré par l'utilisateur.

Le système sous test et l'oracle doivent tous deux être fournis sous la forme de programmes exécutables à entrées-sorties booléennes. Ils doivent également avoir un comportement synchrone (absence de blocage et réaction instantanée).

L'oracle décrit un invariant, exprimé par une formule de logique temporelle du passé : à chaque instant, le verdict qu'il fournit est calculé en fonction des échanges présents et passés entre le système et son environnement. On notera en particulier que la production de verdict est basée sur l'*observation* ; il n'est en conséquence pas possible d'exprimer de verdict

impliquant des instants futurs.

Les contraintes d'environnement sont décrites par des invariants de logique temporelle du passé. À chaque instant, ces invariants définissent un ensemble de données d'entrée qui satisfont les contraintes, en tenant compte des données d'entrée produites antérieurement et des réactions passées du système.

3.3 Mise en œuvre du test par LUTESS

La première tâche de LUTESS est de construire à partir des contraintes d'environnement un générateur de données d'entrée que l'on nommera *générateur aléatoire contraint* (autrement nommé *simulateur*). Plusieurs méthodes de génération étant proposées, le générateur peut également être complété par d'autres descriptions qui seront détaillées plus loin.

LUTESS construit ensuite un *harnais de test*, dont le rôle consiste à faire communiquer entre eux ses différents composants, selon un rythme cyclique. À chaque cycle, ou *pas de test*, le générateur va produire une donnée d'entrée qui sera transmise au système sous test. La réaction de ce dernier coorespond à l'émission d'une donnée de sortie qui sera récupérée par le générateur. Celui-ci pourra alors initier un nouveau cycle, en tenant compte de l'historique des échanges qu'il aura eus avec le système.

Ces échanges sont également observés par l'oracle qui sera alors en mesure d'évaluer à chaque pas de test la validité des propriétés qui le composent, et d'émettre un verdict en conséquence.

Toutes les données transitant dans le harnais sont récupérées par le module de collecte qui permet l'archivages des traces d'exécution, ainsi que leur manipulation : mise en forme, élaboration de statistiques, réévaluation d'une trace avec un nouvel oracle, ...

3.4 Présentation des méthodes de test

Nous nous intéressons dans cette section à la seule production de données de test par LUTESS, en laissant de côté la construction du harnais.

Les techniques proposées par LUTESS antérieurement à ma contribution et mises en œuvre dans le générateur de données sont décrites et illustrées ici sur l'exemple d'une spécification de système téléphonique simple. Ces techniques sont formalisées dans la section suivante.

L'expression des contraintes d'environnement, ainsi que les descriptions complémentaires requises par chaque méthode, sont toutes basées sur le langage LUSTRE (cf. section 3.1).

3.4.1 Test par simulation de l'environnement

Cette méthode produit les données de façon équiprobable par rapport au domaine d'entrée de l'application, défini par les contraintes de l'environnement [80].
Par exemple, l'environnement d'un système téléphonique est constitué de l'ensemble des postes physiques qui peuvent se connecter au réseau. Une contrainte simple est que chaque combiné ne peut pas être décroché deux fois de suite sans être raccroché entre-temps. À chaque instant, LUTESS considère toutes les entrées acceptables, c'est-à-dire celles satisfaisant les contraintes d'environnement compte tenu de l'évolution passée de ce dernier. L'outil procède ensuite sur cet ensemble à un choix aléatoire équiprobable d'une donnée d'entrée, qui sera fourni au système sous test.

Observations
Cette méthode donne des résultats intéressants pour des systèmes relativement simples, c'est-à-dire dont le nombre de comportements différents est peu important, ou lorsque l'hypothèse d'équiprobabilité entre les entrées est valide. Cependant, dès que le système à tester devient plus complexe, la part des comportements réalistes de l'environnement peut devenir minime par rapport à l'ensemble de ses comportements possibles. Une distribution uniforme des probabilités conduit à observer trop souvent des situations peu réalistes, au détriment d'autres comportements plus vraisemblables. Par exemple, chaque usager d'un système téléphonique peut, sous cette hypothèse, composer son propre numéro aussi souvent que n'importe quel autre numéro possible, alors que dans la réalité, ce comportement est assez rare.

3.4.2 Test de propriété de sûreté

Cette technique permet d'insister sur la détection de défauts qui peuvent être caractérisés par le non-respect de certaines propriétés. Ce sont par exemple les défauts pouvant conduire à une défaillance grave du logiciel. Elle s'appuie pour cela sur la définition des propriétés de sûreté du logiciel, et vise à produire les données d'entrée les plus susceptibles de mettre une de ces propriétés en défaut. Cette technique impose de compléter la description de l'environnement par une spécification de l'ensemble des propriétés de sûreté que le testeur estime essentielles.
La procédure de sélection assure que les contraintes d'environnement sont respectées ; l'existence à un instant donné d'une entrée satisfaisant les contraintes d'environnement et susceptible de mettre une propriété en défaut n'est donc pas garantie. Lorsqu'une telle entrée n'existe pas, la sélection se fait en tenant compte des seules contraintes d'environnement, de manière analogue à la méthode précédente.

Si l'on considère une propriété (i *or* o), avec i une entrée et o une sortie du logiciel, il est clair que si i prend la valeur *vrai*, la propriété est trivialement satisfaite. En conséquent, choisir i valant *faux* permet de tester la propriété de manière plus pertinente. Ce choix n'est évidemment possible que s'il vérifie les contraintes d'environnement.

Observations
Une propriété qui pourrait être considérée pour notre exemple est celle qui garantit qu'"un poste X que l'on vient de raccrocher $(On\ X)$ retourne instantanément dans son état de repos $(Idle\ X)$" :

$$On\ X \Rightarrow Idle\ X$$

Cependant, utiliser cette propriété comme guide revient à favoriser l'apparition d'événements de type "raccrocher". En conséquence, de nombreux comportements plus intéressants ne sont pas générés ni testés.

De plus, la méthode ne recherche que des mises en défaut immédiates et n'est pas capable de déterminer par exemple que la propriété $(o\ or\ pre\ i)$[1] sera testée de manière adéquate au cycle suivant si i vaut *faux* à l'instant présent.

3.4.3 Test guidé par profils opérationnels

Le principe des profils opérationnels [78] a été conçu pour décrire les usages réels d'une application. Un profil est alors utilisé afin de guider le test en sollicitant le système sous test de manière plus conforme à la réalité. La technique présentée ici offre à l'utilisateur la possibilité de décrire un tel profil en spécifiant une répartition statistique des données d'entrée du système sous test.

La spécification du profil consiste à associer des probabilités conditionnelles à des variables d'entrée du système. Les conditions sont des prédicats (expressions booléennes exprimées en Lustre) pouvant porter sur l'histoire du programme et ses entrées. Ces associations, désignées par abus de langage par le terme de *probabilités conditionnelles*[2] sont définies comme un complément à la description de l'environnement.

À la différence des travaux classiques du domaine [103, 104], la spécification du profil pour LUTESS ne décrit pas nécessairement de manière exhaustive les usages du système. L'utilisateur peut ainsi tirer profit de toute information, même partielle, concernant le comportement de l'environnnement.

Un algorithme permet de traduire automatiquement un profil en un ensemble de probabilités conditionnelles [29].

Observations
Cette technique permet de remédier au faible réalisme de certaines situations. Il est par exemple possible de rendre la composition d'un numéro plus vraisemblable en affectant au numéro composé une probabilité dépendant de l'identité de l'usager qui compose. Ainsi, on peut spécifier que "si l'usager X compose, il effectuera dans 1 cas sur 100 seulement son

1. On rappelle que *pre* i représente la valeur de i à l'instant précédent.
2. Un terme plus approprié aurait été *affectations conditionnelles probabilisées.*

propre numéro".

Cette dernière méthode est bien adaptée au test de systèmes dont les différentes variables d'entrée du système n'ont que peu de dépendances temporelles les unes par rapport aux autres. En revanche, dès lors que les comportements à décrire sont complexes et portent sur une durée importante, elle s'avère beaucoup plus difficile à mettre en œuvre. Cette limite sera plus abondamment discutée au chapitre 4.

3.5 Fondements théoriques

Nous décrivons ici le cadre formel des méthodes de test présentées plus haut. Les générateurs aléatoires contraints correspondant à chacune sont formalisés par des machines génératrices, basées sur des machines réactives d'entrée-sortie [17]. Cette formalisation s'inspire de celle fournie par I. Parissis dans sa thèse, mais est établie ici dans un cadre simplifié.

Dans la suite, et pour tout ensemble de variables booléennes X, V_X dénotera l'ensemble des valeurs possibles du vecteur[3] formé par toutes les variables de X. $x \in V_X$ correspondra à une affectation de toutes les variables de X.

3.5.1 Simulateur d'environnement

Définition 3.1 *Un simulateur d'environnement (ou machine génératrice) est défini comme* $M_{env} = (Q, q_{init}, O, I, t, env, out_{env})$ *où :*

- O *(resp. I) est l'ensemble des variables de sortie (resp. d'entrée) du système sous test (UUT,* Unit Under Test*),*
- Q *est un ensemble fini d'états,*
- $q_{init} \in Q$ *est l'état initial,*
- $env \subseteq Q \times V_I$ *représente les contraintes d'environnement,*
- $t : Q \times V_O \times V_I \rightarrow Q$ *est la fonction de transition (totale),*
- out_{env} *est une méthode qui, étant donné $q \in Q \backslash \{q | \nexists i, (q,i) \in env\}$, sélectionne un élément de manière non-déterministe parmi $S_{env}(q) = \{i | (q,i) \in env\}$, l'ensemble de toutes les entrées valides du système sous test.*

Dans chaque état, une entrée est produite et fournie au système sous test. Celui-ci génère une sortie, ce qui permet d'effectuer une transition. Ce comportement peut être représenté

3. On emploiera dans la suite indifféremment les termes *donnée d'entrée* et *vecteur d'entrée*.

comme ci-dessous, en termes des entrées (i_k) et sorties (o_k) successives du système sous test :

$$\left|\begin{array}{l} Pour\ k = 0, \ldots \\ \quad i_k \leftarrow out_{env}(q_k) \\ \quad read(o_k) \\ \quad q_{k+1} \leftarrow t(q_k, o_k, i_k) \end{array}\right. \qquad \text{(B)}$$

Dans le cadre de la méthode de test décrite en section 3.4.1, la méthode de sélection est basée sur l'hypothèse d'une distribution équiprobable des entrées. En d'autres termes, pour un état q donné, chaque élément de $S_{env}(q)$ a la même chance d'être choisi.

Réactivité du simulateur

Un simulateur d'environnement peut ne pas être réactif ; en d'autres termes, on ne peut pas affirmer que son évolution ne le mènera pas dans un état q_b de blocage, où $S_{env}(q_b) = \emptyset$.

Pour garantir l'absence de blocage d'une telle machine, il est nécessaire de déterminer si pour tout état accessible depuis l'état initial, il est possible de générer une nouvelle entrée pour le système. Ceci se ramène au calcul d'un plus petit point fixe [49].

Soit $post_{env} : Q \rightarrow 2^Q$ la fonction de calcul des états successeurs d'un état par t :

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

On calcule l'ensemble des états accessibles $\widetilde{post}(q_{init})$ comme l'image de q_{init} par la fermeture transitive de *post*. *env* est génératrice si

$$\forall q \in \widetilde{post}(q_{init}), \exists (i, o, q'), q' = t(q, o, i) \wedge q' \in \widetilde{post}(q_{init}).$$

Ce calcul n'est pas toujours réalisable en pratique [90]. Dans LUTESS, nous préférons construire *a priori* la machine génératrice sans effectuer cette vérification. En revanche, LUTESS sait repérer le blocage du simulateur et alerte l'utilisateur en conséquence. Puisqu'un tel problème est révélateur d'une mauvaise spécification de l'environnement, l'utilisateur a alors la charge de reconcevoir cette dernière pour éviter que le problème ne se présente à nouveau.

3.5.2 Machine de génération guidée par propriétés

On formalise dans ce paragraphe le générateur aléatoire contraint guidé par des propriétés,

tel que défini en section 3.4.2, après avoir au préalable formalisé la notion de *donnée de test pertinente* vis-à-vis d'une propriété.

Définition 3.2 *Une donnée de test (donnée d'entrée du système sous test) est dite* pertinente *vis-à-vis d'une propriété si elle rend possible la mise en défaut immédiate de la propriété considérée.*
Soit $P \subseteq Q \times V_O \times V_I$ un prédicat représentant une propriété, $q \in Q$ un état de l'environnement.

$$\mathsf{pertinente}_P(i,q) \Longleftrightarrow \exists o \in V_O,\ P(q,o,i) = faux$$

Définition 3.3 *Une machine de génération guidée par propriétés est une machine génératrice $M_{prop} = (Q, q_{init}, O, I, t, env, out_{prop})$ où :*

- *$prop \subseteq Q \times V_O \times V_I$ est une propriété (dans la pratique, prop sera l'intersection des propriétés que l'on attend du système),*
- *out_{prop} est une méthode de calcul de données de test. Pour un état q donné, la méthode choisit tout d'abord un ensemble parmi $S_{env \cap pertinente_{prop}}(q) = \{i \in V_I \mid (q,i) \in env \cap pertinente_{prop}\}$ et $S_{env}(q) = \{e \in V_I \mid (q,e) \in env\}$. $S_{env \cap pertinente_{prop}}(q)$ est choisi avec une forte probabilité s'il n'est pas vide. Un élément est ensuite sélectionné dans l'ensemble choisi, selon le processus propre au simulateur d'environnement. $S_{env}(q) = \{e \in V_I \mid (q,e) \in env\}$.*

À partir du moment où, dans un état q, il existe une donnée de test pertinente, toutes les données non-pertinentes sont ignorées. La génération se fait suivant une distribution équiprobable parmi les données de test pertinentes.
On peut constater, ainsi qu'annoncé en section 3.4.2, que la recherche de données de test pertinentes est instantanée et ne considère que l'état courant.

3.5.3 Machine de génération guidée par profil opérationnel

En préliminaire à cette définition, il est utile de préciser ce que recouvre dans ce contexte précis le terme de profil opérationnel. Un profil opérationnel est défini ici par une liste de probabilités conditionnelles CPL$= (cp_0, cp_1, ..., cp_k)$. Chaque cp_j est un triplet (i_j, p_j, f_j) où i_j est une variable d'entrée ($i_i \in I$), v_j est une valeur de probabilité ($p_i \in [0..1]$) et f_j est un prédicat décrivant une condition ($f_i \subseteq Q \times V_O \times V_I$). p_j est la probabilité que la variable i_j prenne la valeur *vrai* lorsque la condition f_j est vérifiée.

Définition 3.4 *Une machine de génération guidée par profil opérationnel est une machine génératrice $M_{oper} = (Q, q_{init}, O, I, t, env, out_{CPL})$ où :*

- CPL$= (cp_0, cp_1, ..., cp_k)$ *est une liste de probabilités conditionnelles,*

– *out_{CPL} est la méthode de calcul de sortie. La sélection d'une valeur de sortie est telle que l'affectation de chacune des variables de sortie dépend des probabilités conditionnelles associées à cette variable, lorsqu'elles existent. Lorsque cette dernière condition n'est pas vérifiée, la variable est affectée en accord avec l'algorithme équiprobable classique.*

On peut noter que si la liste de probabilités conditionnelles est vide, une telle machine est équivalente à une machine génératrice, c'est-à-dire sans guidage.

3.6 Détails des algorithmes et implémentation

L'environnement du système peut être décrit par trois ensembles de variables booléennes :

– $\mathcal{E}$ (pour *Entrées*) est l'ensemble des variables d'entrée du système (donc de *sortie* de l'environnement),

– $\mathcal{S}$ (pour *Sorties*) est l'ensemble des variables de sortie du système (donc d'*entrée* de l'environnement),

– $\mathcal{L}$ (pour *Locales*) est l'ensemble des variables définissant l'état de l'environnement. Ces variables sont internes à l'environnement, d'où leur dénomination de variables *locales*. Elles sont exprimées en fonction des valeurs *passées* des variables de $\mathcal{E}$ et $\mathcal{S}$.

L'instanciation des machines génératrices présentées plus haut se fait donc en identifiant $I = \mathcal{E}$ et $O = \mathcal{S}$. L'espace d'états Q correspond à l'ensemble des différentes combinaisons possibles des variables de $\mathcal{L}$: $Q = V_{\mathcal{L}}$.

Comme on l'a vu, la construction du générateur s'appuie essentiellement sur l'énoncé des contraintes décrivant l'environnement. L'ensemble de ces contraintes est compilé de manière à produire un automate. Les états de l'automate sont codés par l'ensemble de variables booléennes Q, tandis que les transitions sont représentées par des fonctions booléennes.

Ces fonctions sont implémentées sous la forme d'un unique graphe de décision binaire (BDD, *Binary Decision Diagram*) [4, 19]. Les BDDs sont une optimisation de la représentation de fonction booléenne sous forme d'arbre de Shannon [94]. À partir d'un tel arbre, un BDD est construit par élimination des nœuds redondants et le partage des sous-arbres isomorphes. Ce mode de représentation efficace permet une manipulation aisée et une grande facilité de calcul. Les structures de BDDs ont été largement mises à contribution, notamment dans le cadre de travaux sur la vérification de programmes réactifs [49, 91].

On notera dans la suite Δ_f le BDD représentant la fonction f.

Sur le BDD Δ_{env} que nous souhaitons construire, chaque nœud désigne une variable d'état ou d'entrée[4], et chaque arc sortant est étiqueté par une valeur que peut prendre la variable du nœud. Par convention, la branche gauche correspond à la valeur *faux*, la branche

4. Rappelons qu'une contrainte d'environnement ne peut désigner que les valeurs *passées* d'une variable de sortie du système, et ce, par le biais des variables d'état de l'environnement.

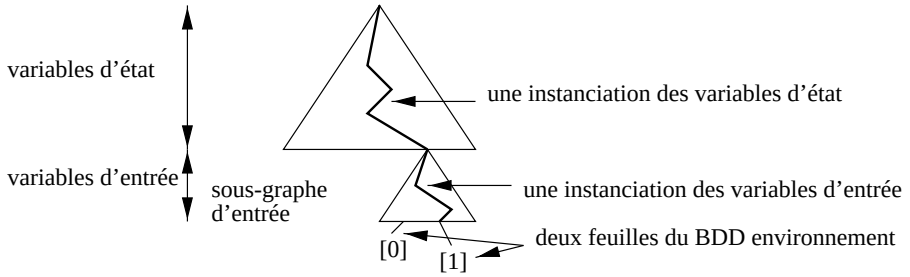


FIG. 3.2 – *Description générale de la structure du BDD d'environnement*

droite à *vrai*. Les nœuds terminaux (ou *feuilles*) sont valués par les constantes booléennes *vrai* ou *faux*. Un chemin depuis la racine jusqu'à une feuille de l'arbre correspond donc à une valuation de l'ensemble des variables de la fonction représentée. La valuation est valide si la feuille désignée porte la valeur *vrai*.

Les variables portées par le BDD appartiennent à $\mathcal{L} \cup \mathcal{E}$. Les variables sont ordonnées ainsi : en premier lieu (à la racine de l'arbre) figurent les variables d'état, puis viennent les variables d'entrée du système sous test. La relation d'ordre est celle de l'ordre d'expansion : pour deux variables x et y appartenant à $\mathcal{L} \cup \mathcal{E}$, $x < y$ signifie que x est plus proche de la racine que y. La seule contrainte imposée à cet ordre est que, $\forall y \in \mathcal{E}, \forall x \in \mathcal{L}, x < y$.

Considérant la valeur courante des variables décrivant l'état de l'environnement, cet ordonnancement permet de parcourir facilement le BDD pour atteindre le sous-arbre décrivant les transitions tirables depuis cet état. Ce sous-arbre sera appelé dans la suite *sous-arbre d'entrée* (cf. figure 3.2).

L'utilisation d'un BDD permet d'aboutir à une représentation optimisée de l'arbre de Shannon correspondant. Cependant, les raisonnements que nous menons se font par référence à l'arbre "expansé" (i.e., l'arbre de Shannon sur lequel toutes les variables apparaissent sur tous les chemins, et dont les sous-arbres isomorphes ne sont pas partagés). Lorsque l'on parcourt un BDD, on reconstruit donc "à la volée" l'arbre expansé correspondant.

Le reste de cette section se consacre à la description du choix d'une transition pour un état donné. Cette dernière fournit une valuation des variables d'entrée et conduit donc à la génération d'une donnée d'entrée.

Les divers algorithmes de génération sont tous basés sur un étiquetage des nœuds du sous-arbre d'entrée. Cet étiquetage est effectué une fois pour toutes, lors de la construction du BDD. Chaque nœud correspondant à une variable d'entrée $e \in \mathcal{E}$ est étiqueté par un couple d'entiers (v_0, v_1). v_0 (resp. v_1) indique le nombre de transitions *tirables* si la variable est affectée de la valeur *faux* (resp. *vrai*).

Considérons par exemple la fonction booléenne $f(e0, e1, e2) = e0 \vee (\neg e1 \wedge e2)$. Sa table de vérité peut s'écrire :

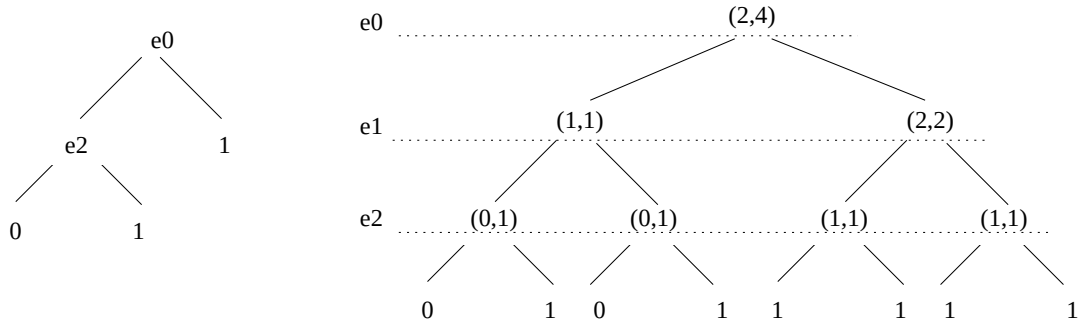


FIG. 3.3 – *Un exemple de BDD et l'étiquetage de sa version expansée*

		e1.e2			
		00	01	11	10
e0	0	0	1	0	0
	1	1	1	1	1

La figure 3.3 montre en sa partie gauche la représentation sous forme de BDD correspondant à la fonction f ci-dessus, tandis qu'à droite est indiqué l'étiquetage correct à réaliser. On peut constater sur cet exemple que les six vecteurs d'entrée valides apparaissent bien.

Génération aléatoire
Pour assurer l'équiprobabilité de la génération, l'algorithme d'affectation des variables d'entrée est le suivant :

1. localiser dans le BDD décrivant l'environnement, le sous-arbre d'entrée correspondant à l'état courant de celui-ci ;

2. parcourir ce sous-arbre de la façon suivante :
   - affecter la variable e identifiée par le nœud à la racine du sous-arbre en fonction des probabilités suivantes :
     $p(e = vrai) = \frac{v_1}{v_0+v_1}$ et $p(e = faux) = \frac{v_0}{v_0+v_1}$,
   - suivre la branche correspondant au choix effectué (i.e., considérer le fils droit si e a été affectée à *vrai*, le gauche dans le cas contraire.),
   - réitérer le processus jusqu'à accéder à une feuille de l'arbre.

Le vecteur d'entrée ainsi formé satisfait bien les contraintes d'environnement. Pour qu'une affectation des variables d'entrée satisfasse les contraintes d'environnement, il faut et il suffit que le chemin ainsi décrit aboutisse à une feuille valuée *vrai*. Pour peu qu'il existe dans le sous-arbre d'entrée au moins **un** chemin menant à une valuation *vrai*, le processus de sélection aboutira au choix d'un tel chemin. En effet, à chaque nœud, une branche qui ne mène qu'à des feuilles valuées *faux* ne peut être choisie (sa probabilité d'être sélectionnée est nulle). Si aucun chemin n'existe, la situation sera automatiquement détectée au début du processus car le sous-arbre d'entrée sera isomorphe à la feuille *faux*. Ceci correspond à un sous-arbre

expansé où $p(e = vrai) = p(e = faux) = 0$.

Le choix du vecteur d'entrée respecte bien l'équiprobabilité parmi l'ensemble des vecteurs possibles. Celle-ci est assurée par le fait que chacun de ces vecteurs est pris en compte et contribue à part égale dans le décompte affiché par l'étiquetage de la racine du sous-arbre d'entrée. Pour chaque nœud, l'étiquetage rend compte du nombre de vecteurs d'entrée valides dans les sous-arbres gauche et droit. En effectuant un choix tenant compte de cette répartition, on préserve l'équiprobabilité.
Remarque : on aurait pu, avec le même résultat, décider d'effectuer un seul tirage pour désigner le vecteur à sélectionner, puis parcourir le sous-arbre avec cette seule donnée.

Génération basée sur des propriétés
Cette méthode impose la construction d'un second BDD, supplémentaire à celui décrivant l'état de l'environnement. Ce BDD représente la fonction booléenne $env \wedge prop$, où env est la fonction représentant les contraintes d'environnement, et $prop$ l'énoncé de la propriété servant de guide. Ce BDD est étiqueté de la même manière que le BDD d'environnement dans sa partie inférieure correspondant aux entrées.

L'algorithme d'affectation des variables d'entrée est modifié comme suit :

1. Localiser dans Δ_{env} et $\Delta_{env \wedge prop}$ les sous-arbres d'entrée δ_{env} et $\delta_{env \wedge prop}$ correspondant à l'état courant de l'environnement.

2. Sélectionner ensuite un sous-arbre parmi les deux ci-dessus en se basant sur la constatation suivante :
   Si $\delta_{env \wedge prop}$ n'est pas isomorphe à Δ_{faux}, il existe au moins une donnée d'entrée pertinente pour l'état courant de l'environnement. $\Delta_{env \wedge prop}$ est alors sélectionné avec une forte probabilité et δ_{env} avec une probabilité beaucoup plus faible. Dans le cas contraire, on choisit par défaut δ_{env}.

3. Appliquer ensuite le point 2 de l'algorithme développé page 40 en considérant le sous-arbre d'entrée sélectionné.

Génération basée sur des profils opérationnels
On se base pour réaliser cette méthode sur le BDD d'environnement Δ_{env} et la liste de probabilités conditionnelles définissant le profil :CPL= $(cp_0, cp_1, ..., cp_k)$, chaque cp_i étant un triplet (e_i, p_i, f_i).

Le sous-arbre d'entrée de Δ_{env} est parcouru de la même manière que décrit page 40 pour l'algorithme standard, à la différence près que l'affectation de la variable e contenue dans chaque nœud du chemin dépend des probabilités ci-dessous :

Soient $(e_{i_1}, p_{i_1}, f_{i_1})..(e_{i_r}, p_{i_r}, f_{i_r})$ les r triplets de CPL tels que $(e_{i_j}) = e$.

$$\begin{cases} p(e = \texttt{true}) = & \texttt{if } f_{i_1} \texttt{ then } p_{i_1} \\ & \texttt{else if } f_{i_2} \texttt{ then } p_{i_2} \\ & \ldots \\ & \texttt{else if } f_{i_r} \texttt{ then } p_{i_r} \\ & \texttt{else } \frac{v_1}{v_0+v_1} \\ p(e = \texttt{false}) = & (1 \Leftrightarrow p(e = \texttt{true})) \end{cases}$$

(v_0, v_1) est l'étiquetage du nœud considéré.

Notes sur l'implémentation
En pratique, la sélection aléatoire à la base des algorithmes décrits plus haut est fondée sur un mécanisme de génération de nombres pseudo-aléatoires. Cette implémentation a été démontrée équitable dans [28]. L'intérêt d'un tel choix d'implantation est de permettre de répéter plusieurs fois une même expérimentation. En effet, l'initialisation du générateur pseudo-aléatoire requiert de fournir un germe. Pour reproduire une expérience, il suffit donc de fournir le même germe.

3.7 Arrêt du test

Le critère d'arrêt du test est totalement arbitraire. Il est donné par l'usager sous la forme d'un entier désignant la longueur (en nombre de cycles) de la séquence de test.
Une autre solution aurait pu consister à décider de l'arrêt du test dès que l'oracle révèle une erreur. Cette possibilité n'a pas été retenue pour les raisons suivantes :

- Formellement, la procédure de production de verdict est déconnectée de la phase de génération de données de test. C'est ce qui permet de réévaluer une même séquence avec plusieurs oracles.

- Un oracle n'a pas forcément valeur de verdict définitif, puisque le testeur humain est censé procéder à l'analyse des résultats du test. L'oracle peut d'ailleurs même avoir été conçu pour être un révélateur de situation (cf. section 7.5.1), c'est-à-dire être destiné à informer le testeur de l'apparition de certaines situations jugées dignes d'intérêt, mais pas forcément synonymes d'erreur. Il est dans ces conditions peu judicieux de se baser sur les informations que l'oracle apporte pour décider de l'arrêt du test.

Dans le cas précis de l'utilisation de notre approche pour la détection d'interactions entre services, il est apparu intéressant de continuer le test même après qu'une erreur ait été décelée pour les raisons suivantes :

- Comme on le verra au chapitre 6, les services sont composés de telle sorte que le comportement du modèle demeure cohérent. Il n'est donc pas absurde de poursuivre le test après une erreur.

– Le modèle a un comportement cyclique, et se retrouve fréquemment dans un état équivalent à son état initial. On peut alors considérer qu'une séquence de test est constituée de plusieurs séquences quasi-indépendantes mises bout à bout.

Chapitre 4

Une méthode efficace de test de systèmes à environnement complexe

4.1 Introduction

4.1.1 La nécessité de guider dans le test

Un profil opérationnel [78] est une description statistique des différents modes d'utilisation d'un système, logiciel ou matériel. L'utilisation de profils à des fins de guidage du test permet d'optimiser le processus de validation en garantissant qu'à tout moment, les fonctions les plus utilisées auront été testées avec le plus d'attention.

L'utilité de tels profils est d'autant plus importante que l'environnement du système sous test a des comportements complexes, pouvant s'étendre sur une longue période de temps[1]. Dans ce cas, en effet, le nombre de comportements possibles peut être très grand par rapport au nombre de comportements "plausibles" ou "réalistes", i.e. qui correspondent à un comportement observable dans la réalité avec une fréquence non-négligeable. Il est clair que dans ces conditions, un test purement aléatoire ne sera pas en mesure de produire avec une chance raisonnable une séquence d'entrée correspondant à un comportement plausible.

Pour illustrer ce propos, considérons un exemple emprunté au domaine des services téléphoniques : le service de transfert d'appel explicite (ECT). Celui-ci permet à son souscripteur de placer son correspondant en attente, d'appeler un autre usager et de connecter ensemble ces deux intervenants.
Son exécution nécessite d'établir tout d'abord une première communication, de demander ensuite l'exécution de la mise en attente, puis de composer le numéro d'un autre usager avant de pouvoir enfin composer le code permettant l'exécution du service de transfert proprement dit. Un tel comportement est relativement long et complexe et sera difficilement observable dans le cas d'une génération aléatoire.

1. On pourra formaliser la notion de *comportement* comme une sous-séquence d'événements d'entrée participant à la réalisation d'un objectif commun (activation d'un service, par exemple).

On peut aussi mentionner le service de facturation *Charge Call*, qui permet à son souscripteur de facturer un appel sur un compte d'un poste autre que celui à l'origine de l'appel. Sa mise en œuvre requiert de la part de son souscripteur de respecter un protocole complexe : il lui faut en effet commencer par composer le numéro du correspondant souhaité, préfixé par "0", puis composer le numéro de l'abonné à facturer, et enfin le numéro personnel d'identification correspondant à cet abonné.

Nous proposons dans ce chapitre une méthode de test adaptée à ce genre de situations. Cette méthode repose comme celles décrites au chapitre 3 sur une simulation de l'environnement du système. Cependant, avec cette méthode, la simulation est *dirigée* par un ou des guides décrits par des schémas comportementaux. Le guidage se base notamment sur la notion de *progression temporelle*, intervenant de manière explicite dans les exemples ci-dessus : un certain nombre d'instants sont clairement identifiés comme des étapes à franchir.

4.1.2 Description informelle de la notion de *schéma comportemental*

Un schéma est la description d'un comportement "typique" de l'environnement que l'on va chercher à mettre en avant par rapport à l'ensemble infini des comportements possibles. Un schéma ne représente pas forcément un comportement unique, mais une *classe* de comportements : tous les comportements vérifiant un certain nombre de conditions données seront concernés. Ainsi, étant donné un schéma, tout comportement contenant au moins une occurrence de ce schéma est membre de cette classe.

Un schéma est décrit comme une succession de conditions portant alternativement sur des instants et des intervalles. Les conditions d'instant caractérisent principalement les entrées que l'on souhaite voir engendrées, tandis que les conditions d'intervalle décrivent les invariants à maintenir vrais entre deux conditions d'instant consécutives.

Ainsi, sur l'exemple du service ECT mentionné plus haut, un schéma pertinent consisterait à indiquer que, à partir d'une situation où il est en communication, le souscripteur active la mise en attente de son correspondant (première condition d'instant), puis compose le numéro d'un autre abonné (seconde condition d'instant), sans raccrocher entre temps (première condition d'intervalle), puis active la mise en œuvre du service de transfert (troisième condition d'instant) avant que l'un des usagers impliqués n'ait raccroché (deuxième condition d'intervalle).

4.1.3 Guidage par schéma

Le guidage par un schéma consiste à augmenter la probabilité d'occurrence de ce schéma, de manière à favoriser la classe de comportements qu'il représente.

La méthode de guidage basée sur ce concept va conduire à favoriser successivement chaque condition portant sur des instants, tout en minimisant les chances de mise en défaut des

conditions d'intervalle. Toute séquence satisfaisant les conditions imposées par un schéma aura une chance accrue d'être observée, par rapport à ce qui serait obtenu par une génération purement aléatoire.

Le guidage proposé par cette méthode est "conciliant", en cela qu'il ne cherche qu'à *favoriser* une classe de comportements, et non à *forcer* son apparition. En particulier, aucune donnée d'entrée n'est jamais imposée, et seules les probabilités d'occurrence sont affectées. De même, la mise en défaut des conditions d'intervalle n'est pas interdite, mais seulement rendue moins probable. De la sorte, tous les comportements valides restent possibles, mais ceux recherchés apparaîtront avec une plus grande fréquence. L'utilisation de cette méthode au sein d'une procédure de génération aléatoire *contrainte* permet de garantir que les contraintes imposées sont préservées et qu'elles ne subissent aucun renforcement.

4.2 Objectifs de la méthode

L'utilisation de la notion de profil opérationnel (cf. chapitre 3), comme celle des schémas de comportement correspond à une certaine philosophie de test : l'objectif est de tester en priorité les comportements les plus fréquents, de manière à éliminer le plus rapidement possible les erreurs qui risqueraient de se manifester le plus souvent. L'idée sous-jacente consiste donc à traiter les erreurs par ordre d'importance en fréquence, sans tenir compte de la gravité de leurs conséquences [2].

L'objectif de la méthode de guidage par schémas est de permettre un test large : en n'imposant jamais de valeur d'entrée et en se limitant à influencer la génération, elle autorise le parcours de l'espace des comportements possibles *autour* des comportements sélectionnés (i.e., ceux représentés par un schéma). Ainsi, non seulement on peut observer les comportements désirés suffisamment souvent, mais on peut également constater et évaluer les effets des variations élémentaires par rapport à ces comportements.

Considérons par exemple un schéma décrivant un usage d'un service. Cet usage correspond à une intention de l'utilisateur du service. On peut tout à fait imaginer que l'utilisateur change d'avis, et modifie son comportement en cours de route. Le fait que le schéma sélectionné ne s'impose pas à tous les comportements engendrés autorise cette possibilité. Dans l'exemple du service ECT, le fait d'utiliser le schéma annoncé en fin de section 4.1.2 peut permettre d'observer un comportement où le souscripteur mettra son correspondant en attente, composera le numéro d'un autre usager, mais changera d'avis et ne demandera pas la mise en œuvre du transfert. Un tel comportement sera partiellement influencé, jusqu'à l'instant où il aura dévié du schéma. En conséquence, ses chances d'apparition seront accrues.

On a donc une sorte de test de *proximité* : plus un comportement sera "proche" d'un schéma, plus il sera fréquent. Un comportement sera d'autant plus proche d'un schéma qu'il

2. L'oracle peut néanmoins faire le tri et être exprimé de manière à ne considérer que les erreurs les plus critiques.

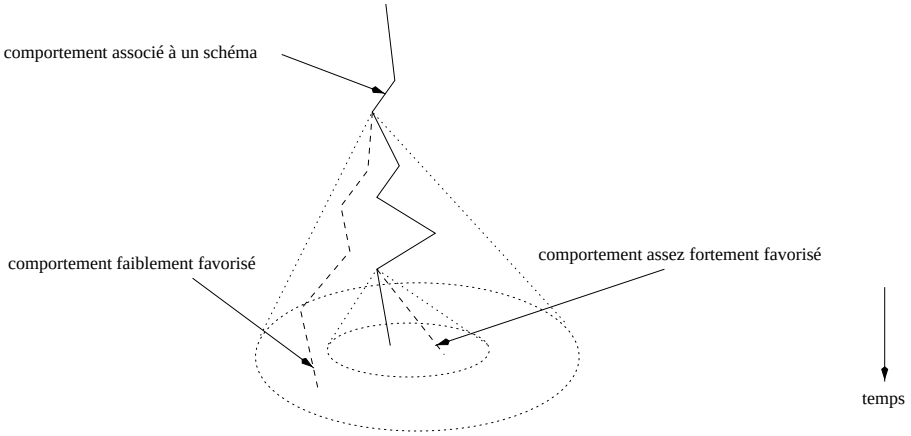


FIG. 4.1 – *Influence du guidage sur divers comportements*

en diverge tardivement. La divergence est observée lorsque une condition d'intervalle n'est plus satisfaite.
Il faut noter à ce point de la discussion que le fait pour un comportement de diverger d'un schéma ne signifie pas que le schéma ne finira pas par être satisfait : en effet, une divergence correspond à une réinitialisation du guidage (cf. section 4.3), ce dernier recommençant ensuite à influencer la génération.

Le concept de test de proximité peut s'illustrer sur l'espace des états de l'environnement, en faisant abstraction de la possibilité de réinitialisation. Sur la figure 4.1, on a représenté plusieurs comportements sous la forme de lignes brisées reliant certains états de l'environnement.
Le comportement symbolisé par un trait plein fait partie de la classe représentée par le schéma sélectionné, tandis que les deux autres, tracés par des tirets sont des variations par rapport à ce comportement de référence. Le plus petit cercle concentrique représente l'ensemble des états accessibles en déviant du schéma le plus tard possible. Les comportements qui y mènent sont donc relativement bien favorisés. Le cercle de plus grand diamètre représente pour sa part l'ensemble des états accessibles après des comportements ayant rapidement divergé du schéma. Ces derniers ont une probabilité plus faible d'être générés.

Cette discussion montre l'intérêt d'une génération simplement "guidée" par rapport à une génération "forcée", qui imposerait une séquence d'entrée totalement figée. Elle offre le double avantage de ne pas requérir une connaissance trop fine du système à tester et de pouvoir mener le test sans faire d'hypothèses trop restrictives sur le comportement de l'environnement.

4.3 Formalisation

Un schéma est une séquence de conditions ordonnées, à laquelle est attachée une notion explicite de *progression*. Dans la succession de conditions d'instant et d'intervalle constituant un schéma, on peut associer les conditions par *paire*. Une paire sera constituée des deux

conditions, l'une d'intervalle et l'autre d'instant, qui correspondront à un même niveau de progression.
À chaque instant n'est considérée qu'une paire de conditions, et c'est en relation avec ces deux conditions que la génération d'entrées sera influencée. Si la condition d'instant est satisfaite, il y a *progression*, et à partir du cycle suivant sera considérée la paire de conditions suivantes. Dans le cas contraire, si de plus la condition d'intervalle est mise en défaut, il y a *régression* : le guidage est réinitialisé et ce sera la première paire de conditions du schéma qui sera ensuite prise en compte.

4.3.1 Définition formelle d'un schéma

Rappelons (cf. section 3.5) que, pour tout ensemble de variables booléennes X, V_X est l'ensemble des valeurs possibles du vecteur formé par toutes les variables de X et $x \in V_X$ est une affectation de toutes les variables de X.

Un schéma étant censé décrire un usage du système, nous commençons ici par définir le vocabulaire qui permet cette description. Rappelons que l'environnement du système peut être décrit par trois ensembles de variables booléennes : d'entrée ($\mathcal{E}$), de sortie ($\mathcal{S}$) et d'état ($\mathcal{L}$) (cf. section 3.5).

Définition 4.1 *On définit l'ensemble des expressions* Π_V *sur un ensemble de variables* V *par induction grâce aux règles suivantes :*

- $false \in \Pi_V$, $true \in \Pi_V$
- $\forall p \in V, p \in \Pi_V$
- $\forall p, p' \in \Pi_V, \neg p \in \Pi_V, p \vee p' \in \Pi_V, p \wedge p' \in \Pi_V$

Définition 4.2 *On construit ensuite un* prédicat simple *comme la conjonction d'une expression* $\epsilon_{\mathcal{L}} \in \Pi_{\mathcal{L}}$ *(désignée comme la composante d'état du prédicat), considérée à l'instant précédent et d'une expression* $\epsilon_{\mathcal{E}} \in \Pi_{\mathcal{E}}$ *(désignée comme la composante d'entrée du prédicat) :*

$$SP = pre\ \epsilon_{\mathcal{L}} \wedge \epsilon_{\mathcal{E}}$$

Définition 4.3 *Une condition d'instant ou d'intervalle (CI) est construite à partir de conjonctions* ($\wedge$)*, de disjonctions* ($\vee$) *et de négations* ($\neg$) *de prédicats simples.*

Cette définition, et essentiellement l'utilisation de l'opérateur de disjonction, permet d'exprimer le fait que différentes situations peuvent convenir à un instant donné pour satisfaire une condition d'instant ou d'intervalle[3]. La condition d'instant ci-dessous, portant sur l'exemple d'un modèle de système téléphonique introduit au chapitre précédent, est satisfaite si :

- la sortie à l'état précédent satisfait le prédicat $RingingTone_A$ et si l'entrée générée au cycle présent est Off_A (ce qui correspond à la situation où l'usager A décroche pour répondre à un appel),

3. Cette définition n'autorise pas une condition à faire directement référence à une variable appartenant à $\mathcal{S}$. Néanmoins, les valeurs passées de ces variables participent à la définition de l'état de l'environnement, et on pourra donc les consulter dans $\mathcal{L}$.

– ou si le prédicat $RingBackTone_A$ était vrai à l'instant précédent et Off_{party_A} est l'entrée générée au cycle présent (ce qui correspond à la situation où le correspondant de A répond à l'appel de ce dernier).

$$(\textbf{pre}\ RingingTone_A \wedge Off_A) \vee (\textbf{pre} RingBackTone_A \wedge Off_{party_A})$$

Définition 4.4 *Un schéma comportemental*[4] *est composé d'une séquence ordonnée de paires de conditions* $(inter_i, cond_i)_{i=1..n}$ *où* n *est le rang du schéma.* $inter_i$ *représente la* i^e*me condition d'intervalle,* $cond_i$ *la* i^e*me condition d'instant.*

La première condition d'instant est dite condition *initiale* du schéma, et la première condition d'intervalle est toujours égale à *true*. On peut donc faire abstraction de celle-ci, et donner la règle syntaxique suivante :

$$BP ::= CI \mid CI\ [CI]\ BP$$

Les prédicats entre crochets décrivent les conditions d'intervalle, les autres prédicats sont les conditions d'instant.

4.3.2 Définition formelle du guidage à base de schéma

Notion de trace

Une mémoire est une affectation des variables d'entrée, de sortie et d'état d'un programme quelconque. Une trace d'exécution est une séquence non-vide, finie ou infinie, de mémoires. Elle caractérise le comportement du programme. Dans notre cas, on écrira $\Sigma = \mathcal{S} \cup \mathcal{E} \cup \mathcal{L}$. Une trace s'écrira alors $\sigma = (\sigma_1, .., \sigma_m)$ avec $\sigma_i \in V_\Sigma (i = 1..m)$. On pourra éventuellement faire référence à σ_i en utilisant la notation (σ, i), par analogie aux définitions introduites par Pnueli [87].

Complétion d'un schéma

Un schéma de rang n sera dit *complété* par une trace finie $\sigma = (\sigma_1, .., \sigma_m)$ aux conditions suivantes :

– $\exists (i_1, ..., i_n), (1 = i_1 < ... < i_n = m) \cdot \forall k \in 1..n, (\sigma, i_k) \models cond_k$

– $\forall k \in 1..(n \Leftrightarrow 1), \forall j, i_k < j < i_{k+1} \cdot (\sigma, i_k) \models inter_{k+1} \wedge \neg cond_{k+1}$

En d'autres termes, un schéma est complété à l'instant où sa dernière condition d'instant et satisfaite, et si toutes les conditions d'instant et d'intervalle précédentes ont été satisfaites dans l'ordre voulu.

Définition 4.5 *Le nombre de complétions d'un schéma de rang* n *dans une trace finie* $\sigma = (\sigma_1, .., \sigma_m)$ *est le nombre entier* κ *tel qu'il existe* κ *sous-traces distinctes dans* σ *qui complètent le schéma.*

4. parfois abrégé par *BP* pour *behavioral pattern*

Soit une trace finie $\sigma = (\sigma_1, .., \sigma_m)$ et deux sous-traces $\sigma' = (\sigma_i, .., \sigma_j), \sigma'' = (\sigma_k, .., \sigma_l) \subset \sigma$. σ' et σ'' sont dites *distinctes* si $j < k$ ou $l < i$.

L'objectif du guidage est de produire une trace dans laquelle le nombre de complétion du schéma sera significatif ou, à tout le moins, accru en regard de ce qu'une génération non-guidée aurait permis d'obtenir.

La formalisation d'un simulateur d'environnement guidé par schéma s'appuie sur les définitions introduites en section 3.5.

Définition 4.6 *Un simulateur d'environnement guidé par schéma est une machine génératrice* $G_{pat} = (Q, q_{init}, \mathcal{S}, \mathcal{E}, t, env, out_{BP})$ *avec :*

- $BP = cond_1[inter_2]cond_2...cond_{n-1}[inter_n]cond_n$ *est un schéma comportemental.*
- progress *est une variable entière prenant ses valeurs dans* $V_{progress} = \{0..n\}$. progress *est l'indice de progression du schéma BP et indique à tout instant quel préfixe du schéma a été complété.*
- $S_{\mathcal{P}}, S_{\mathcal{R}}, S_{\mathcal{N}} : Q \times V_{progress} \rightarrow V_{\mathcal{E}}^*$ *sont des sous ensembles de* $\mathcal{E}$ *tels que*[5], $\forall q \in Q, \forall j \in V_{progress}$*:*
  - $S_{\mathcal{P}}(q,j) = \{i \in V_{\mathcal{E}} \mid (q,i) \in cond_{j+1} \cap env\}$
  - $S_{\mathcal{R}}(q,j) = \{i \in V_{\mathcal{E}} \mid (q,i) \in \neg inter_{j+1} \cap \neg cond_{j+1} \cap env\}$
  - $S_{\mathcal{N}}(q,j) = \{i \in V_{\mathcal{E}} \mid (q,i) \in inter_{j+1} \cap \neg cond_{j+1} \cap env\}$

  Note : pour des raisons d'homogénéïté d'écriture, $cond_{n+1}$ *et* $inter_{n+1}$ *sont supposés exister et valoir respectivement* true *et* false.
- out_{BP} *est la méthode de sélection de sortie de la machine. La sélection d'une valeur de sortie est effectuée de telle manière que, étant donné l'état courant de l'environnement* q *et la valeur courante de l'indice de progression* j*, les entrées valides appartenant à* $S_{\mathcal{P}}(q,j)$ *soient favorisées par rapport à celles appartenant à* $S_{\mathcal{N}}(q,j)$*, elles-mêmes favorisées par rapport à celles appartenant à* $S_{\mathcal{R}}(q,j)$*.*

4.4 Aspects techniques

Description de la méthode de génération de données

Dans la pratique, la méthode consiste à parcourir le schéma comportemental au moyen d'un indice de progression, et à privilégier toute entrée satisfaisant la condition d'instant pointée par l'indice, tout en diminuant la probabilité de faire apparaître une entrée pouvant mettre en défaut la condition d'intervalle correspondante. Pour cela, **pour chaque état de l'environnement**, l'ensemble des entrées possibles (i.e. compatibles avec les contraintes

5. Un prédicat simple est ici vu comme une relation entre ses arguments.

liées à l'environnement) est partitionné en trois catégories suivant la valeur de l'indice de progression :

- $\mathcal{P}$ (pour vecteurs de *Progression*) regroupe les entrées satisfaisant la condition d'instant pointée ;
- $\mathcal{R}$ (pour vecteurs de *Régression*) est composée des entrées mettant en défaut la condition d'intervalle correspondante;
- $\mathcal{N}$ (pour vecteurs *Neutres*) inclut les entrées n'appartenant à aucune des deux autres catégories.

Il est important de noter que cette répartition se fait en tenant compte de l'état de l'environnement : considérant par exemple le prédicat simple décrivant la condition d'instant pointée par l'indice de progression, il faut que sa composante d'état (cf. définition 4.2) soit satisfaite pour que l'entrée susceptible de satisfaire la composante d'entrée soit placée dans $\mathcal{P}$.

Dans chaque état, une probabilité est associée à chaque catégorie en fonction de sa cardinalité et d'un coefficient de pondération.

La génération d'une entrée se fait en deux étapes :

1. Sélection d'une catégorie en accord avec les probabilités qui leur sont associées ;
2. Choix aléatoire équiprobable d'un vecteur d'entrée dans la catégorie sélectionnée.

Les coefficients de pondération sont fournis par l'utilisateur pour chacune des valeurs de l'indice de progression d'un schéma : $w^z_{\mathcal{P}}$, $w^z_{\mathcal{R}}$ et $w^z_{\mathcal{N}}$ sont les poids respectifs de $\mathcal{P}$, $\mathcal{R}$ et $\mathcal{N}$, pour une valeur z de l'indice de progression.

Chacune des catégories c parmi $\mathcal{C}=\{S_{\mathcal{P}}, S_{\mathcal{R}}, S_{\mathcal{N}}\}$ a une probabilité p^z_c d'être sélectionnée, compte tenu de la valeur z de l'indice de progression :

$$p^z_c = \frac{w^z_c * card(c)}{\sum_{j\in\mathcal{C}} w^z_j * card(j)}$$

La probabilité pour un vecteur d'entrée i de la catégorie c d'être choisi est donc par conséquent $p^z_{i,c}$, :

$$p^z_{i,c} = \frac{1}{card(c)} * p^z_c = \frac{w^z_c}{\sum_{j\in\mathcal{C}} w^z_j * card(j)}$$

Dans la version actuelle de LUTESS, les pondérations ne sont pas variables en fonction de la valeur de l'indice de progression. L'utilisateur ne fournit qu'un coefficient par catégorie. La possibilité de les spécifier de manière variable présente cependant un certain intérêt : on peut par exemple décider de guider plus fortement une partie du schéma que l'on sait être difficile à obtenir, et plus lâchement le reste, plus facilement observable.

Dans la pratique, les pondérations sont fixées de manière à favoriser les entrées qui permettent de faire progresser le schéma par rapport aux entrées qui n'ont pas d'influence sur

cette progression, elles-mêmes prioritaires par rapport à celles dont les conséquences feraient régresser le schéma.

$$\forall z \in V_{progress}, w^z_{\mathcal{P}} > w^z_{\mathcal{N}} > w^z_{\mathcal{R}}$$

Implantation de la méthode de guidage

La méthode de guidage proposée réutilise la technologie BDD déjà mise en oeuvre dans les techniques de génération aléatoire présentées au chapitre 3. Chaque condition d'instant ou d'intervalle présente dans le schéma est représentée par un BDD. Ces derniers sont combinés entre eux et avec le BDD représentant l'environnement afin d'identifier les ensembles $S_{\mathcal{P}}$, $S_{\mathcal{R}}$ et $S_{\mathcal{N}}$.

On a ainsi, pour un schéma de rang n $(inter_i, cond_i)_{i=1..n}$, la construction de $3.n$ BDDs: $\Delta_{env \cap cond_i}$, $\Delta_{env \cap \neg cond_i \cap \neg inter_i}$ et $\Delta_{env \cap \neg cond_i \cap inter_i}$ représentant respectivement les BDDs décrivant les possibilités de progression, de régression ou de stagnation d'un schéma par rapport à la valeur courante i de son indice de progression.

Chaque génération implique par conséquent le parcours (partiel) des trois BDDs correspondant à la valeur courante de l'indice de progression. Le parcours consiste à retrouver le sous-arbre correspondant à l'état courant de l'environnement, où la cardinalité de chacun des ensembles peut être retrouvée grâce à l'étiquetage effectué lors de la construction des BDDs, de la même manière que pour les autres méthodes proposées par LUTESS. On dispose alors des informations suffisantes pour calculer les cardinalités des ensembles $S_{\mathcal{P}}$, $S_{\mathcal{R}}$ et $S_{\mathcal{N}}$ et sélectionner une catégorie.

L'utilisation d'une technologie à base de BDDs a permis une implantation efficace de la méthode. En effet, le calcul des probabilités s'effectue statiquement une fois pour toutes au début du test. Le processus de sélection d'une entrée demeure quant à lui sensiblement identique à celui développé dans le chapitre précédent pour la génération aléatoire simple.

Mise en œuvre de la méthode de guidage

Lors de la génération proprement dite, l'indice de progression est mis à jour après chaque nouveau choix d'entrée selon les règles suivantes : si l'entrée proposée appartient à $\mathcal{P}$, l'indice de progression est incrémenté, si l'entrée choisie fait partie de $\mathcal{N}$, l'indice demeure inchangé, si l'entrée fait partie de $\mathcal{R}$, l'indice est réinitialisé.

Lorsque l'indice atteint la fin du schéma, la procédure de guidage élémentaire est achevée. Selon l'objectif du test, l'algorithme principal considèrera le guidage comme terminé, ou réitèrera la procédure.

Définition 4.7 *On nommera "procédure de guidage élémentaire"* une *complétion d'un schéma. Celle-ci débute en fixant la valeur de* progress *à 0 et se termine lorsque* progress *atteint la valeur n ou que le guidage est réinitialisé (mise en défaut d'une condition d'intervalle).*

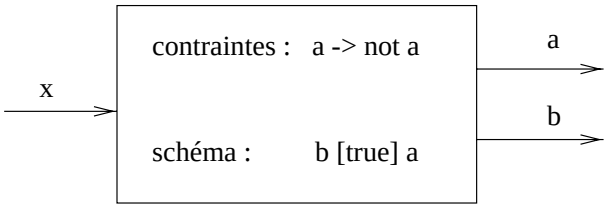


FIG. 4.2 – *Exemple d'environnement incluant un schéma ne pouvant être complété*

L'algorithme implémenté dans LUTESS consiste à effectuer autant de procédures de guidage élémentaire que possible : lorsqu'une procédure se termine avec succès, *progress* est réinitialisé (remis à zéro) et une autre procédure de guidage est initiée. Le critère d'arrêt du test n'est pas affecté : le test se termine lorsque la longueur de séquence fixée arbitrairement par le testeur est atteinte.

Satisfaisabilité de la complétion

Il faut noter que rien ne garantit dans la définition 4.6 et dans les algorithmes présentés ci-dessus qu'une procédure de guidage élémentaire puisse aboutir à la complétion du schéma. En effet, l'environnement peut atteindre un état à partir duquel il n'est plus possible ni de progresser ni de régresser. Une telle situation est liée à une mauvaise description de schéma. Un testeur aura donc intérêt à définir ses schémas avec précaution.

Considérons l'exemple naïf suivant. Soit un environnement disposant d'une entrée x et fournissant deux sorties a et b au système sous test. Supposons que cet environnement n'ait à respecter qu'une seule contrainte ($a \Leftrightarrow$ not a), caractérisant le fait que la variable booléenne a doive être vraie à l'instant initial, et seulement à l'instant initial (cf. figure 4.2). Un schéma de la forme $b[true]a$ ne pourra alors jamais être complété, car dès que l'environnement aura quitté son état initial, la seconde condition d'instant ne pourra plus être satisfaite.

Le problème de la terminaison de la procédure de guidage élémentaire (cf. définition 4.7) est analogue à un problème d'accessibilité d'état sur l'automate décrivant le schéma (cf. section 4.7.2). Il s'agit de prouver que l'état accepteur du schéma (correspondant à un indice de progression égal à n) peut être atteint quelle que soit la séquence d'entrées générée en accord avec les contraintes d'environnement.

Définition 4.8 *Un état q d'un simulateur d'environnement guidé par schéma est dit* neutre *pour une valeur p de l'indice de progression si et seulement si $cond_p$ et $\neg inter_p$ ne peuvent être satisfaits.*

$$q \text{ est neutre pour } p \text{ si } S_{\mathcal{P}}(q,p) = \emptyset \text{ et } S_{\mathcal{R}}(q,p) = \emptyset.$$

Il peut exister un état $q \in Q$ et une valeur $p \in V_{progress}$ tels que tout état accessible depuis q soit neutre pour p. Dans ce cas, la procédure de guidage ne terminera pas.

Définition 4.9 *Soit un simulateur d'environnement guidé par schéma $G_{pat} = (Q, q_{init}, \mathcal{S}, \mathcal{E}, t_{env}, out_{BP})$ Une valeur p de l'indice de progression est dite* insurmontable *lorsque*

$$\forall q \in Q, S_{\mathcal{P}}(q,p) = \emptyset$$

S'il existe une valeur insurmontable de l'indice de progression, le schéma ne pourra être complété.

De tels problèmes sont révélateurs d'une mauvaise conception de schéma (par exemple, non-prise en compte de la globalité des contraintes d'environnement). De manière pratique, une solution consiste à avertir le testeur d'une non-progression perdurant, charge restant à ce dernier de corriger en conséquence le schéma.

La caractéristique "test en boîte noire" de notre méthode implique l'absence de toute possibilité d'anticiper ou prévoir les réactions du système sous test. Cela ne nous permet pas de vérifier l'absence de valeurs insurmontables, ainsi que l'inaccessibilité d'états neutres ; il n'est donc pas possible d'effectuer la preuve de la satisfaisabilité de la complétion.

4.5 Combinaison de schémas

L'expressivité de la notion de schémas permet déjà de combiner deux schémas, sous la forme d'un seul, à condition de définir de nouvelles variables intermédiaires. Cette opération sera dénotée sous le terme de *combinaison*.

Soient par exemple A[B]C et D[B]E, deux schémas dont nous n'expliciterons pas les conditions. Si on veut les combiner dans un seul et même schéma, on peut écrire :

$$A \ \vee \ D \ [\ B\ ] \ \texttt{if}\ \mathit{cond}\ \texttt{then}\ C\ \texttt{else}\ E$$

avec *cond*=`between`*(A,D)*[6]. `between` est un opérateur classique de logique temporelle. On trouvera sa définition en annexe D.

Néanmoins, ainsi qu'on le verra en fin de cette section, cette solution est lourde à mettre en place et augmente notablement la complexité du schéma résultant.
Nous avons donc défini explicitement deux opérateurs permettant de combiner les schémas.

Opérateur de priorité

Cet opérateur est binaire, commutatif et associatif. Il est dénoté "||". Un schéma composite $BP_1||BP_2$ est strictement équivalent à la combinaison de schémas proposée ci-dessus. Les deux schémas composants sont exclusifs l'un par rapport à l'autre. Le premier à se déclencher (i.e., satisfaisant sa condition d'instant initiale, cf. section 4.4) est considéré *prioritaire* sur l'autre jusqu'à ce qu'il soit complété ou réinitialisé. Pendant ce temps, le second schéma est inactif. Par analogie, cet opérateur peut être considéré comme un contrôleur d'accès à une section critique.

6. La construction "if..then...else...", absente de la définition d'un prédicat simple (cf. définition 4.2) est un raccourci syntaxique pour $(A \wedge B) \vee (\neg A \wedge C)$

Opérateur de mise en concurrence

Cet opérateur est binaire, commutatif et associatif. Il est dénoté ";". Il a été introduit pour éviter qu'un schéma ayant commencé à progresser n'empêche les autres de pouvoir progresser également.

La sélection d'un vecteur d'entrée est alors menée en considérant à un même instant les paires ($inter_i$,$cond_i$) correspondant à la valeur courante de l'indice de progression de chaque schéma. Une fois le vecteur choisi, chaque schéma est invité à évoluer en conséquence.

Cet opérateur permet de décrire, pour une même séquence de test, différents schémas, correspondant à différents usages du système. Le fait de les mettre en concurrence permet d'obtenir une description du comportement de l'environnement beaucoup plus riche.
Son usage est particulièrement pertinent lorsque le système fournit plusieurs services indépendants. Cette situation a été notamment rencontrée dans les études de cas que nous avons réalisées (cf. partie II).

Coût de la réalisation

Techniquement, l'introduction de ces opérateurs n'a pas imposé d'importantes modifications de l'implémentation existante :

- L'opération de priorité correspond à une affectation temporaire à "0" des coefficients de pondération des schémas qui ne sont pas prioritaires.

- La mise en concurrence a demandé certains aménagements dans l'algorithme de sélection, mais relativement minimes et en nombre réduit.
  En pratique, il a fallu rajouter une étape préalable dans la sélection d'une entrée : avant de choisir une catégorie, il est nécessaire de choisir un schéma parmi tous ceux décrits. Ce choix est effectué en tenant compte des cardinalités de chacune des catégories de chacun des schémas, et de leurs pondérations. Une fois le vecteur d'entrée choisi, une ultime étape consiste à répercuter ce choix sur tous les schémas, afin de les faire éventuellement évoluer en conséquence.
  Cette solution a été préférée à celle, plus lourde, qui aurait consisté à considérer le produit des catégories de chacun des schémas pour partitionner l'ensemble des vecteurs d'entrée valides. Ceci se serait avéré rapidement prohibitif en termes de puissance de calcul requise, puisque le nombre de partitions aurait été égal à 3^n pour n schémas.

Évaluation

Ainsi qu'on l'a mentionné plus haut, l'opération de priorité est strictement équivalente à la combinaison de schémas. Il n'est en revanche pas possible d'exprimer l'opération de mise en concurrence avec la seule expressivité de la notion de schéma unique.

Le tableau 4.1 offre quelques éléments de comparaison entre l'opération de priorité (||) et l'opération de combinaison. Les temps de calcul, en secondes, proviennent de deux expéri-

	Exp 1		Exp 2	
	Construction	Génération	Construction	Génération
Témoin	18.08	35.67	101.16	33.89
Priorité	20.82	34.62	105.38	36.83
Combinaison	42.00	36.09	137.95	38.61
Schéma double	20.86	34.97	117.97	21.02

TAB. 4.1 – *Temps de calcul comparatifs (en secondes) pour concurrence et combinaison de schémas*

mentations portant chacune sur deux schémas de rang 3. La ligne "Témoin" donne les temps de calcul pour l'environnement seul. La ligne "Schéma double" permet de comparer les résultats avec un schéma de rang 6 (construit par concaténation des deux schémas). La première expérimentation (Exp 1) concernait un environnement simple, tandis que la seconde (Exp 2) était relative à un environnement plus complexe.

L'utilisation de l'opérateur de priorité aboutit à une description du profil sous forme de plusieurs BDDs $b_1, ..., b_n$, un par schéma, tandis que la combinaison implicite conduit à la construction d'un unique BDD, B. Ce dernier comporte plus de variables que chacun des b_i, puisqu'un certain nombre de variables intermédiaires supplémentaires doivent être définies. La taille d'un BDD étant au pire exponentielle en fonction du nombre de variables à considérer, la différence de coût entre les deux manières d'opérer une combinaison s'explique aisément.

4.6 De la notion de schéma comportemental à celle de profil opérationnel

Ainsi qu'on l'a mentionné en introduction à ce chapitre, un profil opérationnel permet de décrire de manière quantitative les différents usages d'un système. Sa prise en compte dans une activité de validation permet de maximiser la confiance que l'on peut avoir dans le bon fonctionnement du système en regard du temps consacré à l'activité. De la sorte, si des impératifs (délais de livraison, dépassement des coûts, ...) imposent de réduire la durée de cette dernière, voire de l'interrompre brutalement, la fiabilité du système sera aussi bonne que possible.

De manière plus précise, un profil opérationnel définit des ensembles de vecteurs d'entrée et associe à chacun de ces ensembles une probabilité, autrement dit, une fréquence d'occurrence. La finesse de description du profil et la définition subséquente des ensembles dépend de l'usage qui doit en être fait.

Fondamentalement, la notion de schéma est connexe à celle de profil opérationnel, puisque l'une et l'autre s'attachent à la description des usages du système ou du logiciel à tester.

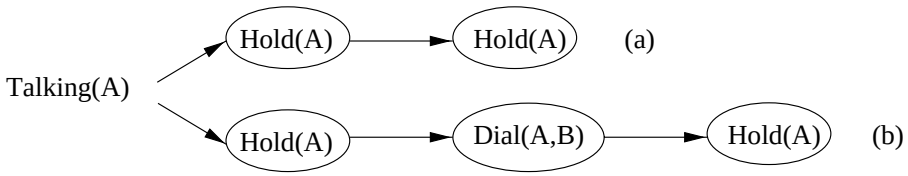


FIG. 4.3 – *Deux comportements concurrents*

L'intérêt du concept de schéma est de fournir un moyen simple de décrire un usage complexe. Une description à base de schémas se veut moins complète et moins précise qu'un profil, car on n'attribue généralement pas explicitement une probabilité à un usage, on se borne simplement à augmenter ses chances d'occurrence en favorisant la satisfaction successive des conditions qui le compose (On a vu cependant en section 5.1.1 qu'il était possible si souhaité d'associer une probabilité explicite, non pas à un schéma, mais à certaines de ses conditions).

Notons par ailleurs que rien n'oblige le testeur à utiliser les schémas dans le seul but de décrire les usages du système. On peut par exemple tout à fait guider le système vers une situation supposée critique.

La connexion entre schémas et profil opérationnel est renforcée par la possibilité de mettre plusieurs schémas en concurrence (cf. §4.5), de manière à favoriser lors d'une même génération plusieurs classes de comportements. On peut ainsi avoir un schéma pour décrire chaque type particulier de sollicitation (ou d'usage) du système, l'ensemble des schémas décrivant alors le profil opérationnel de l'environnement.

On peut considérer qu'un schéma représente un usage, ou une classe d'usages, du système sous test. Permettre de guider une génération de données de test avec plusieurs schémas offre le moyen de décrire partiellement un profil opérationnel.

Par exemple, dans le domaine des services téléphoniques, un service de mise en attente typique permet deux comportements, représentés sur la figure 4.3. La situation initiale est caractérisée par le fait que le souscripteur est en communication (*Talking(A)*). Après avoir mis son correspondant en attente (*Hold(A)*), le souscripteur peut soit initier une nouvelle communication (*Dial(A,B)*) et revenir ultérieurement à son premier correspondant (b), soit choisir de retourner tout de suite à la première communication (a).

4.7 Caractérisation de la complétion d'un schéma

4.7.1 Représentation graphique d'un schéma

Afin de permettre de distinguer plus aisément les conditions d'instant de celles d'intervalle, il est possible de représenter un schéma sous forme graphique, de manière analogue à la description logique graphique *GIL* développée par Dillon et al. [26].
Sur le schéma de la figure 4.4, A, B et C sont des conditions d'instant, tandis que I et J sont les conditions d'intervalle.

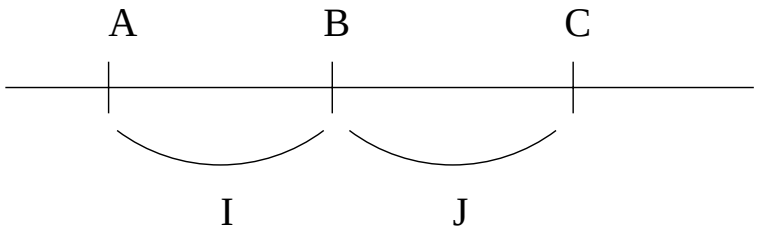


FIG. 4.4 – *Représentation graphique du schéma A[I]B[J]C*

4.7.2 Caractérisation d'un schéma par un automate

On a introduit plus haut (cf. section 4.3.2) la notion de complétion d'un schéma de rang n sur une trace finie $\sigma = (\sigma_1, .., \sigma_m)$. Cette satisfaction est obtenue si la trace σ correspond à l'expression régulière suivante :

$$cond_1.(inter_2 \wedge \neg\ cond_2)^*.cond_2 \ldots . cond_{n-1}(inter_n \wedge \neg\ cond_n)^*.cond_n$$

(Rappelons que $inter_1$=*Vrai*.)

Une expression régulière pouvant être reconnue par un automate, on peut également construire un automate permettant de caractériser la progression d'un schéma.
Chaque condition d'intervalle ou d'instant est représentée par une transition. Chaque état caractérise une étape dans la progression. Pour plus de clarté, ces états sont étiquetés par la valeur de l'indice de progression à considérer.
L'état initial, noté I, correspond à une progression nulle ; le schéma n'a pas encore influencé la génération de données. L'état initial n'a qu'une transition sortante, correspondant à la satisfaction de la condition *initiale* du schéma (cf. section 4.4).
L'état final, ou état accepteur, noté F, permet de reconnaître la complétion du schéma. Lui aussi n'a qu'une transition sortante, notée *Tau*. Il s'agit d'une transition silencieuse, tirée de manière inconditionnelle et qui permet de retourner à l'état initial.
Les autres états caractérisent les étapes intermédiaires de progression sur le schéma. Chacun possède deux transitions sortantes. L'une correspond à la satisfaction de la condition d'instant associée à la valeur de l'indice de progression portée par l'état, et conduit à l'état successeur dans la progression. L'autre correspond à la mise en défaut de la condition d'intervalle associée à la valeur de l'indice de progression portée par l'état, et conduit à un état dont l'étiquette désigne une valeur plus faible de l'indice de progression.

L'automate représenté sur la figure 4.5 caractérise ainsi le schéma

$$A[I]xx : B[J]C[K, xx]D$$

(Remarquons que ce schéma contient une régression partielle : la mise en défaut de la condition d'intervalle K conduit à une régression à l'indice 1, repéré par l'étiquette xx :. En cas de mise en défaut de K, la complétion du schéma ne demande de répéter que le sous-schéma B[J]C[K]D. La notion de régression partielle est une évolution de la notion de schéma et sera présentée au chapitre 5.)

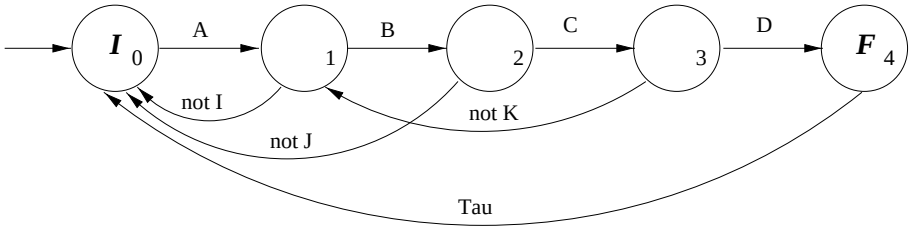


FIG. 4.5 – *Caractérisation d'un schéma par un automate*

4.7.3 Caractérisation logique d'un schéma

Il est également possible de caractériser la progression au sein d'un schéma par des propriétés de logique temporelle. Nous pouvons nous appuyer pour cela sur les opérateurs temporels `between` et `always_since`, définis en annexe D. Chaque instant significatif d'un schéma de rang n est caractérisé par rapport au précédent comme une conjonction :

$$P_i = cond_i \texttt{ and between}(P_{i-1}, \texttt{pre } P_i) \texttt{ and always\_since}(inter_i, P_{i-1}) \text{ (pour } i \in 2..n)$$

Le premier terme conjoint sert à caractériser la progression, le second à se situer par rapport à l'instant précédent, et le troisième à garantir la non-régression depuis ce dernier. Pour l'instant initial du schéma, il est nécessaire de spécifier sa condition, et le fait que le guidage recommence une fois complété :

$$P_1 = cond_1 \texttt{ and (not after}(P_1) \texttt{ or between}(P_n, \texttt{pre } P_1))$$

Le deuxième terme, *after*, sert pour la première tentative de complétion du schéma, le troisième, *between*, pour les suivantes.

La complétion du schéma est alors caractérisée par la propriété P_n. Cette représentation est néanmoins limitée en cela qu'elle ne permet pas d'exprimer de régression partielle (cf. section 5.1.2), à moins de passer par une multitude d'expressions intermédiaires complexes.

Exemple 4.1
En reprenant l'exemple ci-dessus (cf. figure 4.5), il faudra, pour décrire cette possibilité de régression, définir RP_2 pour caractériser la satisfaction de la deuxième condition d'instant lorsqu'il y a eu régression partielle :

$$RP_2 = B \texttt{ and between}(P_3, RP_2) \texttt{ and once\_since(not } K, P_3)$$

Il sera ensuite nécessaire de redéfinir P_3 pour prendre en compte cette éventualité :

$$P'_3 = P_3 \texttt{ or } (C \texttt{ and between}(RP_2, \texttt{pre } P'_3) \texttt{ and always\_since}(J, RP_2))$$

Dans LUTESS, un utilitaire effectue automatiquement cette caractérisation à partir de la spécification LUSTRE du schéma et de l'interface de l'environnement. Cet utilitaire produit un programme LUSTRE dont les entrées sont les entrées et sorties de l'environnement, et qui fournit en sortie un unique booléen valant *vrai* aux instants où le schéma est complété.

Ce programme peut ensuite être réutilisé, par exemple pour construire une propriété d'oracle.

On pourra remarquer qu'une même représentation (la forme graphique décrite plus haut) permet de dériver à la fois à la fois la spécification LUSTRE implémentant le schéma et la spécification LUSTRE implémentant la propriété qui permettra de valider son observation.

Chapitre 5

Propositions d'évolution du guidage par schémas

5.1 Schémas étendus

Afin d'améliorer l'expressivité d'un schéma, et permettre au testeur de contrôler plus efficacement le guidage, nous proposons deux extensions à la notion de schéma : l'affectation de probabilités explicites, et l'itération.

5.1.1 Probabilités explicites

Le guidage basé sur une pondération des catégories d'entrée (cf. section 4.4) peut être critiqué sur le point suivant. L'influence obtenue par ce biais demeure relative, puisqu'elle dépend de la cardinalité de chaque catégorie. Si les cardinalités diffèrent fortement, il se peut que cela ne permette pas au schéma de progresser raisonnablement.
Si on considère par exemple une situation volontairement extrême où $\mathrm{card}(S_{\mathcal{P}}) = 1$, $\mathrm{card}(S_{\mathcal{R}}) = 10000$, $\mathrm{card}(S_{\mathcal{N}}) = 100$, avec des poids $w_{\mathcal{P}} = 100$, $w_{\mathcal{R}} = 1$ et $w_{\mathcal{N}} = 10$. Cette répartition permet d'aboutir aux probabilités suivantes :

$$p_{S_{\mathcal{P}}} = \frac{1 * 100}{1 * 100 + 10000 * 1 + 100 * 10} = \frac{1}{111}$$
$$p_{S_{\mathcal{R}}} = \frac{10000 * 1}{1 * 100 + 10000 * 1 + 100 * 10} = \frac{100}{111}$$
$$p_{S_{\mathcal{N}}} = \frac{100 * 10}{1 * 100 + 10000 * 1 + 100 * 10} = \frac{10}{111}$$

Il semble évident que cette distribution ne permet pas d'assurer la complétion du schéma, la probabilité de progression étant beaucoup trop faible devant celle de régression.

Pour remédier à ce défaut, l'affectation de probabilités explicites consiste à associer directement à une valeur de l'indice de progression les probabilités de progression, stagnation et régression que l'on souhaite voir considérées. Une application intéressante de ce principe consiste à ne probabiliser que la mise en œuvre du schéma : on ne fait qu'indiquer les chances qu'a le schéma de commencer à progresser, sans avoir à préciser ses chances de progression ultérieures.

5.1.2 Itération

Lorsqu'une condition d'intervalle donnée est mise en défaut, plutôt que de reconsidérer la première étape du schéma, on peut souhaiter repartir d'un point intermédiaire.
Par exemple, on peut vouloir exprimer dans un même schéma deux étapes différentes du cycle d'exécution d'un service (cf. section 2.1), comme la paramétrisation et le déclenchement. La première n'a pas à être réitérée à chaque fois que le déclenchement échoue.

L'itération est le moyen de spécifier une régression partielle : on associe à une condition d'intervalle la nouvelle valeur de l'indice de progression qu'il faudra considérer si la condition en question est mise en défaut.

Syntaxe

L'introduction des deux notions ci-dessus ont requis de modifier légèrement la syntaxe d'un schéma comportemental. Celle-ci est désormais décrite sous la forme ci-dessous :

$$BP ::= CI \quad | \quad CI\ \{l :\}\ \{Pr\}\ [CI\ \{, l\}]\ BP$$

- Les accolades désignent le caractère optionnel de certaines informations.

- CI représente une condition d'instant ou d'intervalle. Les conditions d'intervalle sont entre crochets [].

- l est une étiquette qui permet d'identifier une paire (condition d'intervalle, condition d'instant) et la valeur de l'indice de progression qui s'y rapporte. La notation $[CI, l]$ représente une régression partielle à l'indice désigné par l'étiquette l, lorsque la condition d'intervalle CI est mise en défaut.

- Pr est un triplet de valeurs réelles entre 0 et 1, dont la somme vaut 1. Les valeurs représentent, dans l'ordre, les probabilités de progression, de stagnation et de régression associées à une valeur de l'indice de progression.

Dans la version actuelle de l'outil, seule l'extension concernant l'itération est implémentée. La possibilité d'utiliser des probabilités explicites ne l'a pas encore été par manque de temps.

5.2 Optimisation de l'implantation

5.2.1 Méthode de progression alternative

Nous décrivons ici une proposition destinée à réduire le coût de mise en œuvre du guidage. Cette optimisation porte sur la méthode de progression et plus particulièrement sur la manière dont une catégorie d'entrée est sélectionnée.

Cette proposition se base sur la constatation que l'utilisation du guidage est pertinente surtout lorsque l'on souhaite observer un comportement n'ayant qu'une fréquence faible dans une génération uniquement aléatoire (équiprobable).

Dans cette situation, nous pouvons émettre l'hypothèse que, dans tout le schéma, les cardinaux des catégories de progression et de régression sont petits devant celui de la catégorie de stagnation. On peut exprimer cette hypothèse en considérant, pour un BDD donné, le nombre de chemins menant à une feuille *vrai* (noté *Card*) sur l'arbre expansé correspondant :

$$Card(\Delta_{env \cap cond_i}) + Card(\Delta_{env \cap \neg cond_i \cap \neg inter_i}) \ll Card(\Delta_{env \cap \neg cond_i \cap inter_i})$$

On peut alors procéder à l'approximation suivante :

$$Card(\Delta_{env \cap \neg cond_i \cap inter_i}) \approx Card(\Delta_{env})$$

Cela permet de considérer, à chaque instant et pour une valeur i de l'indice de progression, les 3 BDDs $\Delta_{env \cap cond_i}$, $\Delta_{env \cap \neg cond_i \cap \neg inter_i}$ et Δ_{env}. Ainsi, pour un schéma de rang n, plutôt que de stocker $3.n$ BDDs, on se limite à n'en considérer que $2.n + 1$.

Cette décision impose de vérifier *a posteriori*, lorsque le choix s'est porté sur Δ_{env}, si une progression ou une régression s'est produite. Deux solutions se présentent alors :

1. refuser l'entrée et répéter la génération,
2. accepter l'entrée et faire évoluer en conséquence la progression du schéma.

Ces deux solutions offrent chacune des avantages mais présentent des inconvénients :

- Pour la première, les pondérations fournies par le testeur sont strictement conservées. En revanche, le temps de calcul n'est plus constant.
- Dans le deuxième cas, le temps de calcul demeure constant, car toute entrée générée est acceptée. Par contre, il existe potentiellement un biais des pondérations.

Si la supposition initiale s'avère fondée[1], le biais potentiel dû à la deuxième méthode est négligeable, d'autant plus que les pondérations sont généralement données à titre indicatif, sans grande précision. Dans ce cas, la deuxième méthode est à privilégier car plus simple

1. i.e., si le guidage a pour objectif de rendre plus fréquents des comportements rares avec une génération purement aléatoire.

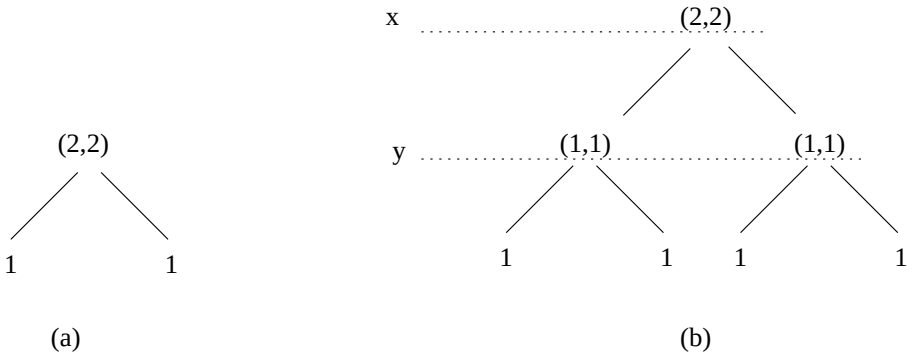


FIG. 5.1 – *Un exemple simple de BDD non "multi-étiquetable"*

à mettre en œuvre. Si en revanche les pondérations sont à respecter impérativement, ou si l'hypothèse n'est que moyennement valide (i.e., l'approximation est litigieuse), on utilisera de préférence la première méthode.

Le gain obtenu par non-construction de $\Delta_{env \cap \neg cond_i \cap \neg inter_i}$ est important, et ce, d'autant plus que le rang du schéma est élevé.
Sur un exemple relativement simple de schéma de rang 3, on a pu constater que la méthode initiale requérait 37,46 secondes pour construire le générateur, contre 15,40 secondes pour la méthode optimisée. Pour deux schémas concurrents de rang 3, ces chiffres étaient de 137,02 secondes contre 103,01.

5.2.2 Représentation optimisée

Une représentation optimisée de schéma consisterait à ne manipuler qu'un seul BDD, Δ_{env}, muni d'un étiquetage multiple. Ce dernier devrait permettre de déterminer, pour une valeur d'indice de progression donnée, l'appartenance de chaque vecteur d'entrée valide à une des catégories (progression, régression, stagnation).

Cette proposition se heurte au fait qu'un BDD est une représentation réduite d'un arbre de Shannon, et que les variables inutiles en sont abstraites. Or, les conditions de progression et de régression peuvent porter sur des variables absentes de certains chemins du BDD d'environnement. Pour prendre un exemple extrêmement simple, considérons la seule contrainte d'environnement *env =vrai* , et un schéma *BP = vrai [not y] x*, x et y étant les deux seules variables d'entrée du système.

La méthode d'étiquetage simple permet d'obtenir un étiquetage de la racine $(v_0, v_1) = (2, 2)$. La figure 5.1 présente le BDD ainsi étiqueté (5.1(a)), de même que sa version "expansée", sous forme d'arbre de Shannon (5.1(b)) Il n'est clairement pas possible sur le BDD (a) d'étiqueter le nœud en fonction des conditions x et *not y* du schéma.

Les conditions du schéma *précisent* les contraintes d'environnement. Pour évaluer correctement l'appartenance de chaque vecteur d'entrée valide à une classe (progression, régression

ou stagnation), il est nécessaire de se baser sur une représentation expansée (i.e., dans laquelle, chaque variable apparaîtrait, même si elle n'influence pas la fonction décrivant les contraintes d'environnement).

La seule solution est de remplacer la structure de données de type BDD par une autre, où toutes les variables seraient explicites. Cette structure de données serait en elle-même plus lourde et plus encombrante qu'un BDD.

Une telle proposition requiert donc au préalable d'étudier les conséquences de ces changements, en termes de gain en temps et en place.

5.3 Quelques perspectives d'amélioration de la méthode de guidage par schémas

5.3.1 Schémas de remplacement

À l'heure actuelle, le processus de test s'achève en fournissant un rapport indiquant le nombre de complétions de chacun des schémas ou, à défaut, la valeur maximale atteinte par leur indice de progression. Nous nous proposons ici de tirer profit de cette information.

Nous avons évoqué en section 4.4 qu'il était possible qu'un schéma ne puisse être complété pour des raisons liées à une mauvaise spécification : existence d'un état neutre ou d'une valeur de progression insurmontable. On pourrait tout à fait envisager de proposer des schémas de remplacement qui, lorsqu'un schéma stagne trop longtemps ou boucle sans jamais réussir à satisfaire une des conditions d'instant, prennent sa relève pour tenter de surmonter autrement le problème.
Le schéma de remplacement pourrait être par exemple plus tolérant, c'est-à-dire être composé de conditions moins contraignantes. Ou alors, au contraire, celui-ci pourrait être une description plus fine d'un comportement pas assez détaillé dans le schéma d'origine.

Ainsi, dans le schéma proposé pour le service ECT (cf. figure 7.1, page 95), la condition initiale contient le prédicat "Talk(A,B)", qui indique le fait que l'usager est en communication. Le guidage n'est pas en mesure d'influencer la satisfaction de ce prédicat, qui dépend uniquement des sorties du système. Si ce prédicat demeure trop longtemps non-satisfait, on pourra trouver utile de le remplacer par une séquence d'événements qui peut mener à sa satisfaction, par exemple : Off(A) [not On(A)] Dial(A,B) and pre Idle(B) [not On(A)] Off(B).

D'une manière générale, lorsqu'une condition fait référence à une sortie du système, ou à l'état de l'environnement, aucun guidage ne peut être réalisé pour que cette condition soit satisfaite. Il peut être alors utile, lorsque l'évolution du système ne conduit pas à sa satisfaction, de remplacer le schéma par un autre plus dirigé et plus complet.
Une telle fonctionnalité serait aisément implantable dans LUTESS, à condition que le testeur soit en mesure de fournir d'éventuels schémas de remplacement.

5.3.2 Variation de l'influence

Au cours d'une même séquence de test, on peut vouloir faire varier l'influence d'un schéma au cours du temps. Cela peut permettre de traduire certaines variations d'usages au cours de la journée. Par exemple, un particulier travaillant à domicile n'utilisera pas forcément son téléphone de la même façon durant ses heures de travail et de repos. Un ascenseur dans un lieu professionnel sera sollicité différemment à l'arrivée des employés (tout le monde monte), et à leur départ (tout le monde descend).

On peut également faire dépendre cette variation du nombre de complétions d'un schéma : avoir une influence plutôt forte tant que le schéma n'a jamais été complété, puis plus modérée à mesure que son nombre de complétion augmente.
On peut envisager d'avoir une inter-dépendance entre schémas : si un schéma tarde à être complété, on pourra vouloir réduire l'influence des autres.

Les possibilités de variation d'influence sont modulables à l'infini, et il ne s'agit pas de vouloir tout intégrer dans LUTESS. En revanche, offrir la possibilité à l'usager de fixer lui-même les critères de variation conférerait à l'outil une grande souplesse. Dans ce but, une solution consiste à définir un langage de description d'influence, qui permette de définir librement les critères. Ce langage serait idéalement basé sur LUSTRE.
Cette extension peut demander un travail supplémentaire relativement important, et il serait donc pertinent de déterminer au préalable son utilité réelle.

5.3.3 Repérage d'un schéma sur une trace

Lorsqu'un schéma n'a pas pu être complété, il n'est pas toujours évident d'analyser les causes de cet échec. Permettre à l'utilisateur de matérialiser le schéma sur la trace, en repérant quand chaque condition d'instant a été satisfaite, peut être d'une grande aide dans cette analyse. Ce repérage peut être réalisé par l'oracle, en utilisant la caractérisation logique d'un schéma et de chacune de ses étapes (cf. section 4.7.3) : chaque étape du schéma est caractérisée en fonction de la satisfaction passée de la condition d'instant de l'étape précédente et de la satisfaction présente de la condition d'instant de l'étape courante. On peut facilement traduire cette caractérisation par une propriété qui sera insérée ensuite dans l'oracle. L'analyse des traces permettrait par exemple de savoir si une valeur de progression n'a jamais pu être surmonté. Une solution plus simple consisterait à réaliser cette analyse automatiquement et à inclure son résultat dans le rapport produit à la fin du test.
Cette analyse est d'ailleurs très facilement automatisable, dans la mesure où il est immédiat de repérer dans l'algorithme de génération aléatoire guidée les instants où une étape du schéma est satisfaite.

5.3.4 Combinaison avec d'autres méthodes

La méthode de test guidé par schémas pourrait être utilement combinée avec la méthode de guidage basée sur des probabilités conditionnelles, de manière à tirer parti des avantages

des deux approches.

La nouvelle procédure de génération pourrait par exemple consister à :

1. sélectionner un sous-ensemble de vecteurs d'entrée parmi les 3 catégories de progression, régression et stagnation, en accord avec la procédure de test guidé par schéma ;
2. puis procéder au choix du vecteur d'entrée en prenant en compte les éventuelles probabilités conditionnelles associées à chacune des variables d'entrée.

Comme on l'a dit précédemment, les deux méthodes de guidage conviennent pour décrire des aspects différents. La possibilité de les utiliser conjointement permet d'enrichir la représentation de l'environnement.

Deuxième partie

Application des méthodes de test à la validation de services de télécommunication

Chapitre 6

Principes de modélisation

Cette partie s'intéresse au problème de la validation de services téléphoniques et à l'adéquation des méthodes de test que nous avons développées pour la résolution de ce problème. Le présent chapitre décrit la méthodologie que nous avons développée pour la spécification formelle de services de télécommunication, tandis que le chapitre suivant décrit la manière dont l'activité de validation a été menée. Ce travail s'appuie sur les trois études de cas que nous avons réalisées. La première a consisté à analyser les documents normatifs spécifiant de manière formalisée mais en langue naturelle un certain nombre de services [34, 35, 36]. Les deux autres se sont déroulées dans le cadre des deux éditions du concours de détection d'interactions organisées en marge de la conférence internationale dédiée au problème (Feature Interaction Workshop) [63] et [1]. Les énoncés de ces deux dernières études sont disponibles en annexes B et C.

6.1 Contexte de modélisation

Nous présentons dans cette section le contexte dans lequel nous allons chercher à modéliser un système de télécommunication et les services que celui-ci peut fournir. Afin d'assurer la crédibilité et la bonne lisibilité de notre travail de modélisation, nous avons jugé utile de demeurer proches des modèles communément reconnus et utilisés dans le domaine. Dans ce but, nous avons choisi de baser notre travail sur les vues et modèles génériques référencés par l'ITU, l'organisme international de normalisation pour les télécommunications.

Les recommandations de l'ITU concernant les réseaux intelligents en donnent un modèle conceptuel, formé de quatre plans [55]. Chaque plan présente une vue du système à un niveau d'abstraction différent. La hiérarchie de plans se décompose ainsi, du plus abstrait au plus concret :

- Le plan des services, qui décrit les services sous le point de vue de l'usager, en faisant abstraction du système sous-jacent.
- Le plan fonctionnel global, qui décrit les fonctions élémentaires permettant de construire chaque service et indique comment elles se combinent. De telles fonctions sont par

exemple la facturation, la diffusion d'un message d'attente, ... Ces "briques" peuvent être composées soit par une mise en séquence, soit par un choix conditionnel, autorisant différents comportements du service en fonction de la réaction de l'usager.
Parmi ces briques, le Basic Call Process (BCP) décrit la procédure d'appel "classique", correspondant au traitement ordinaire d'un appel. Ce traitement correspond au "service de base".
On ignore sur ce plan la question de la répartition des fonctions sur le réseau.

– Le plan fonctionnel distribué, qui décrit la structure fonctionnelle du réseau et prend notamment en compte la localisation des composants. Par exemple, on indiquera à ce niveau que le système de facturation est centralisé et est localisé à tel endroit.

– Le plan physique, qui décrit l'architecture physique du réseau.

Pour en rester à une description logique des services, il suffit de considérer les deux premiers niveaux qui permettent d'obtenir en même temps une définition logique, en termes de propriétés, et une description fonctionnelle, présentant de manière opérationnelle les fonctionnalités offertes.

Au niveau du plan fonctionnel global, le service offert aux usagers se compose d'un BCP et de services supplémentaires. Le BCP et les services supplémentaires peuvent être vus comme des processus concurrents, chacun étant en mesure de réclamer le contrôle d'un appel. Par défaut, le contrôle est donné au BCP. Le BCP peut donner à tout instant le contrôle de l'appel au service qui aura préalablement indiqué son intention de réagir à la situation actuelle. Les situations auxquelles un service peut réagir correspondent à un état observable du BCP et sont identifiées par la notion de point d'initiation (POI). Le retour du contrôle au BCP se fait par l'intermédiaire d'un état pareillement identifié, que l'on nomme point de retour (POR).

Un service supplémentaire peut donc être vu comme une alternative à une partie du service de base (décrit par le BCP). Un service pouvant avoir éventuellement plusieurs comportements différents selon son contexte d'exécution, ses points d'initiation et de retour peuvent être multiples. Le schéma de la figure 6.1 décrit le service global offert aux usagers en termes des entités décrites dans le plan fonctionnel global.

Il est important de noter que la description faite ici concerne le traitement global d'un appel. En pratique, le service de base est divisé en deux parties, une décrivant le comportement de l'usager *appelant*, l'autre décrivant celui de l'usager *appelé*. Ces deux modèles sont désignés dans la norme par les termes anglais de *Originating Basic Call Model* (O-BCM) et *Terminating Basic Call Model* (T-BCM). Ceci conduit à la constation qu'un service n'a qu'une influence locale, dans la mesure où il n'affecte que l'un de ces deux modèles. Ce principe de localité des services et le fait que le contrôle d'un appel soit distribué font partie des éléments essentiels qui ont guidé notre approche.

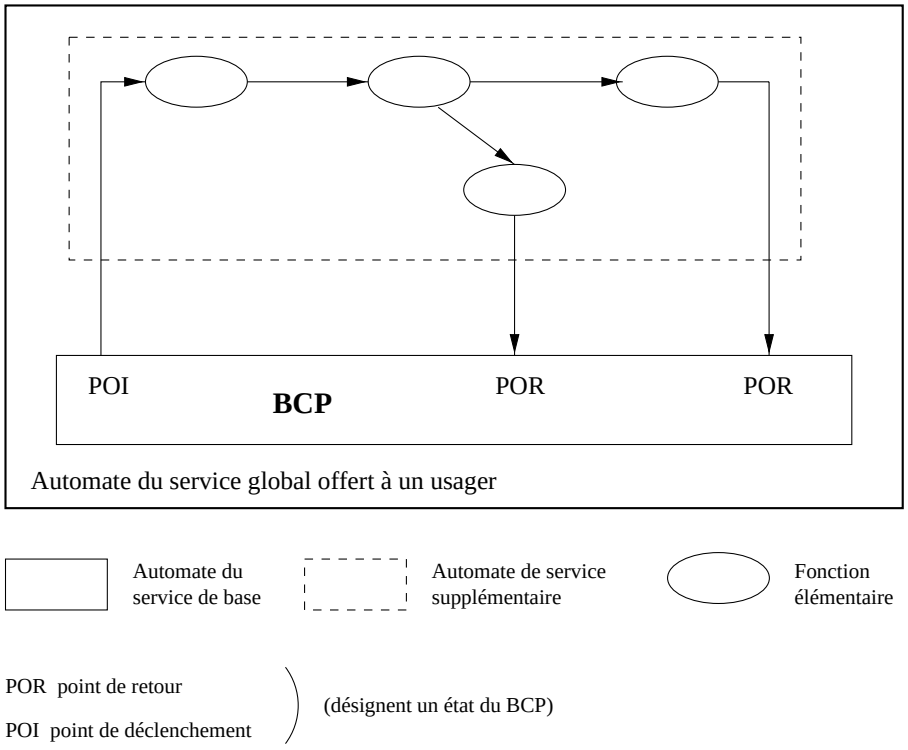


FIG. 6.1 – *Description d'un service dans le plan fonctionnel global*

6.2 Proposition de méthodologie de modélisation

Au travers des diverses études de cas que nous avons réalisées, nous avons élaboré une certaine méthodologie dont nous décrivons ici les lignes directrices.

La modélisation s'effectue à partir de documents qui peuvent plus ou moins précis et détaillés. Nous supposons disposer d'une spécification décrivant chaque service ainsi que le système leur servant de support commun. La modélisation que nous proposons d'effectuer n'a pour seul objectif que de servir la validation logique des spécifications par rapport aux attentes des usagers. La conception du modèle doit donc être réalisée en tenant compte de ce but.

Les étapes de modélisation que nous avons recensées sont les suivantes :

1. Choix d'un niveau d'abstraction.
2. Matérialisation d'un cœur de modèle réactif.
3. Organisation des composants du modèle.
4. Choix d'un mode de communication.
5. Choix des mécanismes de composition et d'intégration.

6. Représentation et traduction des services.

7. Élaboration des propriétés de services.

6.2.1 Choix d'un niveau d'abstraction

La spécification originelle décrit le système et les services à un certain niveau de détail et opère certaines abstractions. Le modèle que nous devons en fournir doit a priori demeurer au même niveau d'abstraction. Néanmoins, pour le type de validation fonctionnelle qui nous intéresse, certains aspects peuvent être omis ou simplifiés.

Abstraction de composants du système

Certains composants et acteurs d'un système de télécommunication n'ont a priori aucune influence directe sur le comportement des services. Il n'est donc pas nécessaire de les faire apparaître dans le modèle. On peut en particulier faire abstraction des éventuels composants relatifs à l'administration du réseau, et des aspects concernant la répartition des composants dans le système. Les seuls composants à représenter sont ceux dont la fonctionnalité est perceptible pour l'usager final : ce sont la fourniture de services et -accessoirement- la facturation.

Hypothèses simplificatrices sur les composants

De la même manière que certains composants décrits par la spécification sont inutiles pour notre propos, d'autres composants peuvent être définis à un niveau de détail trop fin. Il est dans ce cas possible de réaliser certaines simplifications sur ces composants. On pourra par exemple sur un réseau de communication faire une hypothèse de bon fonctionnement ou encore abstraire partiellement ou totalement les données transmises.

Simplifications de certaines notions

Une autre source de simplification concerne, non plus des composants du système tel qu'il est spécifié, mais certaines notions. Ainsi, on pourra réaliser une discrétisation du temps, si la notion de délai apparaît dans la spécification de certains services ; on pourra l'abstraire dans le cas contraire. On peut de la même manière faire abstraction des données vocales échangées par les usagers finaux du système.

Par ailleurs, la description des services peut se faire avec une précision variable : on peut distinguer chacune de ses fonctionnalités élémentaires (cf. le plan fonctionnel global du modèle conceptuel de l'ITU) et la manière dont elles sont combinées, ou simplement considérer chaque service comme une unité atomique. La notion de fonctionnalité élémentaire ne semblant pas primordiale pour notre propos, nous proposons de l'abstraire, si elle est présente dans les spécifications.

6.2.2 Matérialisation du cœur réactif du modèle

Pour pouvoir décrire de façon réactive le modèle d'un système de télécommunication, il est nécessaire de représenter explicitement l'état des postes téléphoniques associés à chaque usager. Cette représentation se fait sous la forme de terminaux virtuels (ou *postes logiques*), décrivant chacun l'état d'un poste. Si cette notion est absente de la spécification, il est nécessaire de la matérialiser. Tous les terminaux virtuels sont identiques et incluent chacun l'ensemble des services disponibles sur le réseau. Un terminal virtuel est composé d'un ensemble d'automates d'entrées-sorties, représentant chacun un service (y compris le service de base). On discutera plus loin de la manière dont ces automates sont connectés à l'intérieur d'un même terminal virtuel, et comment les terminaux virtuels communiquent entre eux.

6.2.3 Organisation de la modélisation

Une fois identifiés les composants utiles à la validation, il s'agit de les organiser et de les connecter de manière à respecter les principes de l'approche synchrone, qui est celle que l'on souhaite mettre en œuvre.

Cela consiste à repérer la part du système que l'on va considérer comme réactive et la part du système qui va constituer l'environnement de cette entité réactive.

Toujours dans l'optique de favoriser la validation fonctionnelle des services, le système réactif incluera les composants prenant part à la mise en œuvre des services, tandis que l'environnement sera constitué des usagers finaux du système (les abonnés au téléphone), ainsi qu'éventuellement d'autres entités périphériques non-essentielles au fonctionnement des services, mais utiles à la détection de défauts de conception. Un journal de facturation aura par exemple sa place dans l'environnement. De manière générale, l'interface entre le système réactif et son environnement sera principalement constituée des combinés téléphoniques permettant aux abonnés d'accéder aux services du réseau. L'organisation du modèle suivra donc le schéma représenté sur la figure 6.2, sur laquelle on fait abstraction des éventuelles entités périphériques incluses dans l'environnement.

Le choix d'une telle interface offre l'intérêt de fournir une vue du comportement du système intuitive et compréhensible.

6.2.4 Choix d'un mode de communication

L'hypothèse de réactivité que nous souhaitons pouvoir appliquer à notre modèle nous impose de garantir un temps de réaction inférieur à celui de l'environnement.

Le mode de communication privilégié par l'approche synchrone est la diffusion instantanée : tous les destinataires d'un message le reçoivent simultanément et instantanément. Selon le niveau de détail utilisé pour la description des services dans la spécification, il sera ou non possible de mettre en œuvre ce principe. Tout dépend de la portée des messages échangés entre le système réactif et son environnement. Si une action de l'environnement est propagée

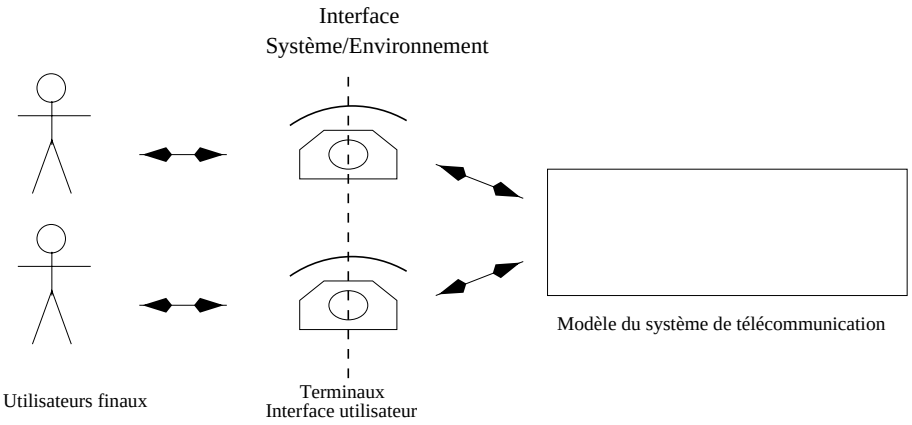


FIG. 6.2 – *Choix d'une interface*

directement à tous les terminaux virtuels concernés, il sera possible de mettre en place une communication totalement synchrone. Si au contraire une action de l'environnement peut conduire à la génération d'autres messages au sein du système réactif, l'hypothèse de synchronisme se heurte au problème de *causalité*. Un terminal virtuel pourrait en effet recevoir un message en provenance du poste utilisateur qui lui correspond, interroger en conséquence un autre terminal virtuel et recevoir la réponse de celui-ci, tout ceci dans le même temps. En d'autres termes, le terminal en question recevrait *à la fois* deux messages, dont l'un est une conséquence de l'autre !

La seule manière de résoudre ce conflit est d'affecter un temps non-nul à la transmission de message. Pour ne pas trop compliquer le modèle, cette durée est constante, et égale à un cycle de l'horloge de base du système[1]. Ainsi, un message émis sur le réseau au cycle t sera délivré à son destinataire au cycle $t+1$. Ceci correspond à un affaiblissement de l'hypothèse synchrone, proche des travaux de F. Boniol autour de la notion de synchronisme faible [12, 11].

Ce choix implique d'imposer sur l'environnement certaines hypothèses : en effet, pour garantir la réactivité du modèle, il faut empêcher l'environnement de générer de nouvelles actions avant la fin de sa réaction.
Cette contrainte peut être établie soit statiquement (en donnant un délai minimum à respecter entre deux actions de l'environnement), soit dynamiquement (en interdisant à l'environnement de produire une nouvelle action tant que le système ne lui a pas explicitement indiqué la fin de sa réaction).
Cette hypothèse est raisonnable dans la mesure où il existe un facteur d'échelle important entre la vitesse de réaction du système, que l'on peut estimer de l'ordre de la milliseconde, et celle de l'environnement, composé dans notre cas exclusivement d'êtres humains, proche de la seconde.

1. On rappelle que l'un des choix initiaux de notre travail était d'opter pour une modélisation synchrone en LUSTRE (cf. sections 2.4 et 3.1), dont le modèle d'exécution est bâti sur la notion d'horloge.

6.2.5 Choix des mécanismes de composition et d'intégration

Afin de permettre l'intégration aisée de nouveaux services et de faciliter leur validation, le modèle doit être modulaire et extensible.

Pour ce faire, il nous faut fournir au concepteur un cadre de développement approprié, c'est-à-dire un modèle de base sur lequel le nouveau service pourra s'appuyer (on rappelle que l'on parle ici de services *supplémentaires*), un mécanisme d'intégration, ainsi qu'un mécanisme de composition permettant la coexistence de plusieurs services.

Localité des services

Nous avons présenté page 74 le principe de localité des services sous-jacent à la description de l'ITU, en insistant sur le fait qu'un appel est représenté de manière distribuée par deux modèles (voire plus) représentant chacun le comportement d'un des usagers impliqués dans l'appel.

Ce principe est traduit en pratique par le fait que chaque service est représenté dans chaque terminal virtuel par une instance de service. On peut donc considérer que le contrôle d'appel est géré de manière locale à chaque terminal, seules les instances de service associées à un terminal étant en mesure d'affecter le comportement de ce terminal.

Les opérations de composition et d'intégration présentées ci-dessous sont donc de la même façon locales à chaque terminal.

Mode d'intégration des services

Un service définit une extension du service de base pour chaque terminal virtuel, sous la forme de transitions à rajouter ou à modifier sur l'automate du service de base.

Au niveau de l'implémentation LUSTRE, pour continuer à distinguer le service de base et le service supplémentaire, chacun est considéré comme une entité à part qui propose une évolution de l'état du terminal. La procédure de sélection consiste à évaluer tout d'abord la proposition du service supplémentaire : si ce dernier émet une proposition valide, on fait évoluer l'automate en conséquence, s'il s'abstient, c'est l'évolution proposée par le service de base qui est prise en compte. Un mécanisme doit également permettre d'indiquer au service la suite donnée à sa proposition.

Mode de composition des services

Lorsque plusieurs services souhaitent étendre le service de base, il est nécessaire de les composer entre eux, avant de vouloir les intégrer au service de base. Cette composition vise à obtenir un super-service, qui s'intégrera de la même manière qu'un service isolé.

Deux solutions sont envisageables :

- Établir une priorité entre les services, de la même manière que l'on considère le super-service prioritaire sur le service de base (Composition par *superposition*).

– Considérer les propositions d'évolution de chaque service sur un même plan (Composition par *juxtaposition*). Dans ce cas, la proposition du super-service peut être incohérente, car constituée des propositions de plusieurs services[2].
Cette dernière solution ne remet pas en cause le mode d'intégration, selon lequel les services supplémentaires sont prioritaires sur le service de base.

La première proposition permet de demeurer cohérent avec une description à base d'automates, et c'est en conséquence celle que nous avons choisie. La seconde proposition a un intérêt pour la validation elle-même, et nous discuterons de ce point et des conséquences de notre choix en section 7.6.3.

6.2.6 Représentation d'un service

Le service de base
Le service de base décrit l'évolution de l'état du terminal virtuel associé à un usager lorsqu'aucun service supplémentaire n'est disponible. Le service de base est représenté par un automate d'entrées-sorties. Étant donné qu'il est censé servir de support à chacun des services supplémentaires, il est nécessaire que les transitions et états présents dans cet automate soient suffisamment explicites pour que les services supplémentaires sachent où se greffer. Pour que les services puissent prendre appui sur le service de base, il est en effet indispensable que soient clairement identifiés sur le service de base les états pouvant servir de points d'initiation ou de retour (POI et POR, cf. section 6.1).

Services supplémentaires
Un service est présent dans chaque terminal virtuel sous la forme d'un ou plusieurs automates d'entrées-sorties décrivant les extensions et modifications qu'il propose d'appliquer au service de base. Chaque automate correspond à un comportement que peut induire le service. Ces comportements décrivent les différents rôles qu'un usager peut avoir dans l'exécution d'un service. De manière générale, on peut distinguer le comportement du souscripteur du service, et celui des autres usagers qui "subissent" le service. Chacun de ces comportements pourra être considéré comme une extension différente au service de base.

Lorsque les différents rôles ne sont pas explicites dans la spécification d'un service, une étape préliminaire à la traduction impose de les faire apparaître.

Pour conserver au modèle sa nature modulaire, chaque comportement fait l'objet d'une modélisation indépendante, sous la forme d'un automate. Ces comportements sont ensuite composés de la manière décrite en section 6.2.5, comme si chacun désignait un service différent.

2. Elle ne l'est pas forcément, car les services peuvent faire la même proposition, ou faire des propositions concernant des aspects indépendants du système.

L'implémentation du modèle sous forme exécutable peut être réalisée en Lustre [21], qui est un langage bien adapté à la description d'automates.

Ainsi qu'on l'a indiqué ailleurs (cf. section 6.2.5), chaque service fait à chaque instant une proposition d'évolution concernant l'état du terminal virtuel. En fonction des règles de composition, cette proposition n'est pas forcément retenue, et il est nécessaire que le service puisse prendre en compte ce refus. Pour cela, il faut que soient distinguées sur l'automate décrivant un comportement la fonction de sortie, matérialisant la proposition d'évolution du service, et la fonction de transition de l'automate. De la sorte, la transition peut être conditionnée par le fait que la proposition d'évolution du service ait été choisie ou non. Si cette dernière a été ignorée, la transition n'est pas effectuée.

Répartition d'un service Le principe de localité des services (cf. page 74) fait que l'exécution d'un service peut concerner plusieurs terminaux virtuels. Selon le mode de communication choisi (cf. section 6.2.4), l'échange d'information entre terminaux prend une forme différente.
Si on a opté pour une communication purement synchrone (diffusion instantanée), les terminaux doivent évoluer en parallèle, de manière synchronisée. Il est pour cela nécessaire de rendre disponible de manière globale l'état de l'ensemble des terminaux virtuels, de sorte que chaque terminal virtuel puisse le consulter de manière instantanée pour évoluer en conséquence.
Si le mode de communication choisi est au contraire un affaiblissement de la diffusion synchrone, les différents terminaux virtuels et leurs services communiquent entre eux par échange de messages.

Dans le premier cas de figure, les sorties des automates représentant chaque service ne sont destinées qu'à faire évoluer l'environnement ; dans le second, il existe également des sorties destinées aux autres terminaux virtuels.

6.2.7 Élaboration des propriétés de service

Ainsi qu'on l'a indiqué en section 2.7, un service est composé d'une description fonctionnelle et d'un ensemble de propriétés invariantes qui expriment les attentes des usagers quant à son fonctionnement. Ces propriétés sont déduites généralement de la description informelle des services en langue naturelle.

De notre expérience, les propriétés qui sont à exprimer peuvent avoir deux formes typiques :

- Invariante : "telle situation n'est jamais possible". C'est par exemple le cas d'une propriété du service de transfert d'appel inconditionnel, qui exprime que le poste de tout souscripteur au service ne sonne jamais pour l'avertir de l'arrivée d'un appel (puisqu'aucun appel n'est censé lui parvenir).

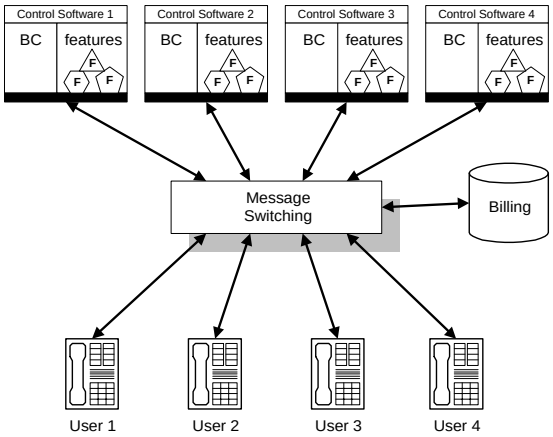


FIG. 6.3 – *Spécification du réseau de télécommunication*

– Comportementale : "tel comportement aboutit à telle situation". Ce peut être le cas d'une autre propriété du service de transfert cité ci-dessus. Un usager qui compose le numéro d'un souscripteur du service sera mis en communication avec la personne désignée par le souscripteur comme cible du transfert.

Les problèmes de correction et de complétude des propriétés seront abordés au chapitre 7.

6.3 Mise en œuvre sur une étude de cas

Nous illustrons dans cette section la mise en œuvre de l'ébauche de méthodologie décrite en section 6.2 sur l'étude de cas la plus récente, qui s'est déroulée dans le cadre du concours FIW'2000. Lorsque les choix réalisés ont été très différents dans l'autre étude de cas conséquente que nous avons réalisée (FIW'98), nous indiquons les deux choix et les raisons qui nous y ont conduits. Les énoncés de ces études de cas sont fournis en annexe B et C.

6.3.1 Énoncé du concours FIW'2000

L'étude de cas menée dans le cadre du deuxième concours de détection d'interactions associé à la conférence FIW'2000 était constituée de 12 services décrits de manière informelle par quelques phrases en anglais, et formellement sous forme de diagrammes états-transitions. Ces descriptions peuvent être vues comme des spécifications dont on souhaite donner une modélisation exécutable.

Description du système à modéliser (voir figure 6.3)
La spécification du système comporte la description d'un commutateur (*Message Switching*), d'un ensemble de postes utilisateurs (les combinés sur lesquels peuvent agir les usagers), d'un ensemble de postes logiques (*Control Software*) représentant chacun un poste utilisateur, et

d'un système de facturation (*Billing*).

Les différentes entités communiquent par le biais de messages, distribués et transmis par le commutateur. Tous les messages circulant dans le système transitent par le commutateur. Le système de facturation est quant à lui un simple journal qui stocke diverses informations que lui transmet le reste du système.

Description des messages échangés

Seuls deux types de composants peuvent générer des messages : les postes utilisateurs et les postes logiques.

Les postes utilisateurs produisent des messages correspondant aux actions des usagers sur leurs combinés : décrocher, composer, raccrocher, ... Ces messages sont transmis aux postes logiques correspondants (Chaque poste utilisateur est en correspondance avec *un* poste logique).

Les postes logiques peuvent produire des messages vers trois destinations différentes :

- Vers le poste utilisateur auquel chacun est associé, de manière à le faire évoluer. Ces messages sont essentiellement destinés à produire différentes sonneries ou tonalités sur le poste (e.g. sonnerie d'appel, signal occupé, annonce de message personnalisé, ...).

- Vers un autre poste logique. Les messages transmis par ce biais servent à indiquer l'évolution du poste logique émetteur à un poste distant. C'est indirectement un moyen pour un poste d'en interroger un autre. Ainsi, un message *Alert* X Y permet au poste logique X de notifier le poste Y d'une tentative d'appel ; le poste Y répondra en émettant à destination de X un message indiquant s'il est en mesure d'accepter ou non cette demande.

- Vers le système de facturation. Ces messages servent à connaître l'instant de début et de fin d'une communication, ainsi que l'identité de l'usager facturé.

Définition des services

Les services supplémentaires et le service de base sont définis par des diagrammes états-transitions en fonction des messages échangés. La figure 6.4 représente ainsi la définition du service de base (BC sur la figure 6.3).

Les services sont conçus comme des extensions du service de base : chaque service s'intègre au service de base en introduisant une ou plusieurs nouvelles transitions, ou en remplaçant certaines transitions existantes.

La figure 6.5 présente la spécification du service TeenLine qui restreint les possibilités d'appel depuis le poste de son souscripteur : selon l'heure de la journée, l'utilisateur doit ou non composer un code secret pour pouvoir passer un appel. La transition $BC1 \rightarrow TL1$ remplace donc sur chacun des postes souscripteurs la transition $BC1 \rightarrow BC2$ du service de base. Les autres transitions présentées par TeenLine sont des transitions supplémentaires aux transitions du services de base.

L'énoncé du concours ne précise pas la manière dont les services se composent.

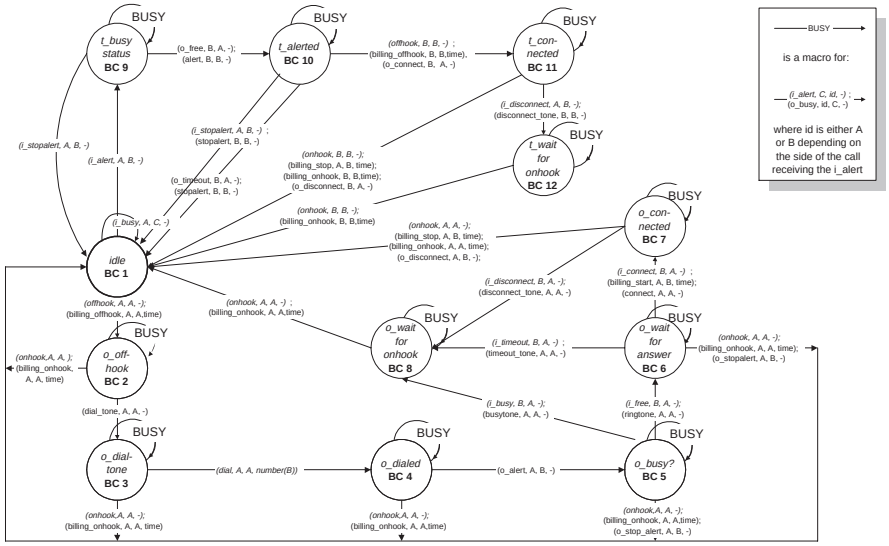


FIG. 6.4 – *Spécification de l'appel de base*

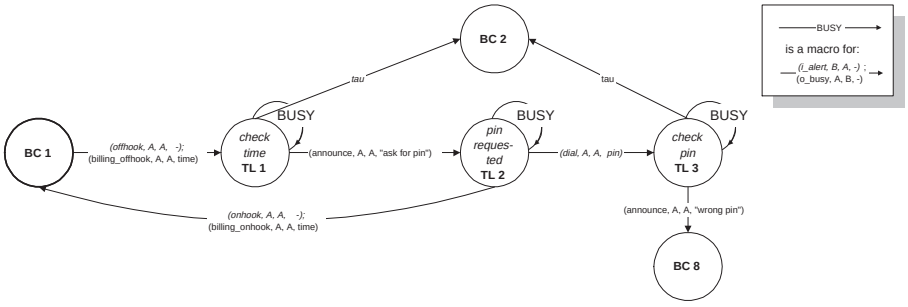


FIG. 6.5 – *Spécification du service TeenLine*

Hypothèses
Un certain nombre d'hypothèses étaient proposées pour cette modélisation, notamment :

- Les seuls messages sont ceux définis dans l'énoncé : en d'autres termes, l'ensemble de messages ne peut pas évoluer.

- Le réseau de commutation n'est jamais saturé.

- La souscription de services n'est pas prise en compte. Elle est considérée s'effectuer à l'initialisation une fois pour toutes.

- Les transitions entre états sont instantanées.

- Les transitions "internes", i.e. se déclenchant sur un message en provenance d'un poste logique, sont déclenchées sans délai. En d'autres termes, elles sont prioritaires sur toute transition résultant d'une action en provenance de l'environnement.

6.3.2 Application des principes méthodologiques de modélisation

Choix d'un niveau d'abstraction
Cette étape n'a pas nécessité beaucoup de travail, les spécifications étant décrites de manière suffisamment explicite et détaillée.

- La spécification du système (cf. figure 6.3) ne fait apparaître qu'un nombre réduit de composants, tous utiles pour la validation fonctionnelle des services. En revanche, pour l'étude de cas FIW'98, la spécification incluait explicitement des composants spécifiques à la notion de réseau intelligent (SCP, cf. chapitre introductif page 10)

- Le réseau de signalisation, permettant aux différents composants du système de communiquer n'est pas considéré explicitement ; il est implicitement supposé que ce réseau est sûr, fonctionnant sans panne ni erreur, et ne subissant ni perte de message ni congestion.

- La notion de fonctionnalité élémentaire est absente, il n'a donc pas été nécessaire de l'abstraire.

- La notion de terminal virtuel existe, et est représentée par le composant intitulé *Control Software* (cf. figure 6.3).

Organisation du modèle
Le système réactif à considérer est composé du module *Message Switching* et de l'ensemble des terminaux virtuels (*Control Software*), tandis que son environnement inclut l'ensemble des usagers, ainsi que le système de facturation. Ceci est une stricte application de la directive énoncée en section 6.2.3.

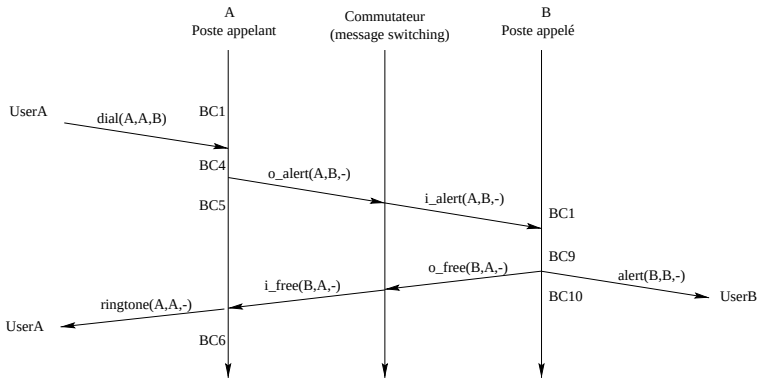


FIG. 6.6 – *Échange de messages correspondant à une demande de connexion*

Mode de communication
Lors de la description des types de messages pouvant être échangés dans le modèle (page 83), on a mentionné le fait que certains messages provenant de l'environnement pouvaient avoir des répercussions sur plusieurs entités, et nécessiter la création de plusieurs autres messages. C'est essentiellement le cas dans le cadre du service de base lors d'une demande de connexion (cf. figures 6.4 et 6.6) : l'usager appelant compose un numéro (transition $BC3 \rightarrow BC4$), provoquant la production d'un message d'alerte (*o_alert*) qui est transmis à l'usager appelé. Dans l'hypothèse où celui-ci n'est pas occupé, son poste prend la transition $BC1 \rightarrow BC9$, et émet en retour un message *o_free* à destination de l'appelant. Ce dernier peut alors évoluer en conséquence ($BC5 \rightarrow BC6$).

La figure 6.6 décrit les échanges de messages entre 3 entités du système lors d'une demande de connexion. Les lignes verticales indiquent les évolutions d'état de chacun des postes. Les flèches transversales indiquent les transmissions de messages.

Il a donc été nécessaire dans ce contexte d'imposer un délai non-nul pour la transmission des messages, de manière à éviter le problème de causalité.

Dans l'expérience menée dans le cadre du premier concours de détection d'interactions (FIW'98), l'évolution de l'ensemble des postes était décrite comme parfaitement simultanée. Un même événement d'entrée devait donc agir éventuellement de manière instantanée sur plusieurs postes. Dans un tel contexte, chaque poste doit connaître à tout instant l'état de tout autre terminal, afin de pouvoir évoluer correctement.

Par exemple, lorsqu'un usager compose un numéro, l'événement est transmis simultanément aux terminaux associés à l'appelé et à l'appelant. Ce dernier doit "supposer" en fonction de l'état de son correspondant quelle suite devra être donnée à sa demande : mise en connexion ou échec.
Cette manière de voir les choses a abouti à une modélisation plus complexe mais au synchronisme plus "pur", puisque la réaction du modèle est instantanée.

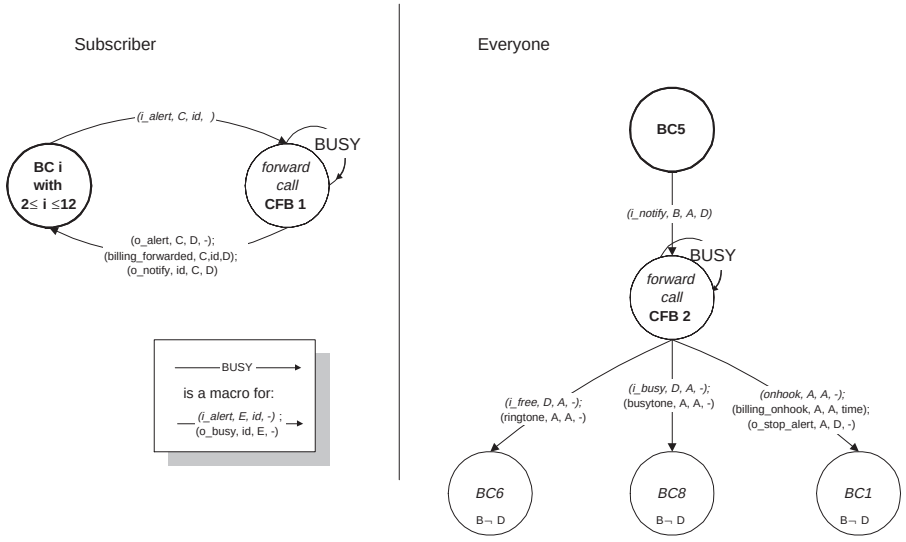


FIG. 6.7 – *Spécification du service* Call Forwarding on Busy

Composition et intégration
La distinction que nous avons proposée entre ces deux opérations pour obtenir une définition rigoureuse de l'interaction existe également dans les spécifications du système. En effet, on peut noter sur la figure 6.3 que le *Control Software* sépare explicitement le service de base (BC, à gauche) des autres services (F).

Traduction
Les spécifications dont nous disposions pour l'étude de cas considérée se présentaient en partie sous la forme de diagrammes états-transitions. La transposition sous forme d'automates a pu se faire de manière immédiate. De plus, le fait que la description des services distinguait déjà explicitement les différents comportements (rôles) a permis de rendre cette traduction systématique.

Par exemple, la spécification du service TeenLine (cf. figure 6.5) ne fait apparaître qu'un seul rôle, car elle n'affecte que le comportement de l'abonné. En revanche, le service de transfert d'appel sur signal occupé (*Call Forwarding on Busy*, CFB) affecte le comportement de deux usagers, l'appelant (6.7.a) et la cible du transfert (6.7.b).

Pour le concours FIW'98, les services étaient décrits d'un point de vue global, sans distinguer leur impact sur chacun des différents intervenants. La traduction devait donc comporter les étapes suivantes :

1. Identifier 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, un troisième rôle concerne la *cible du transfert*.

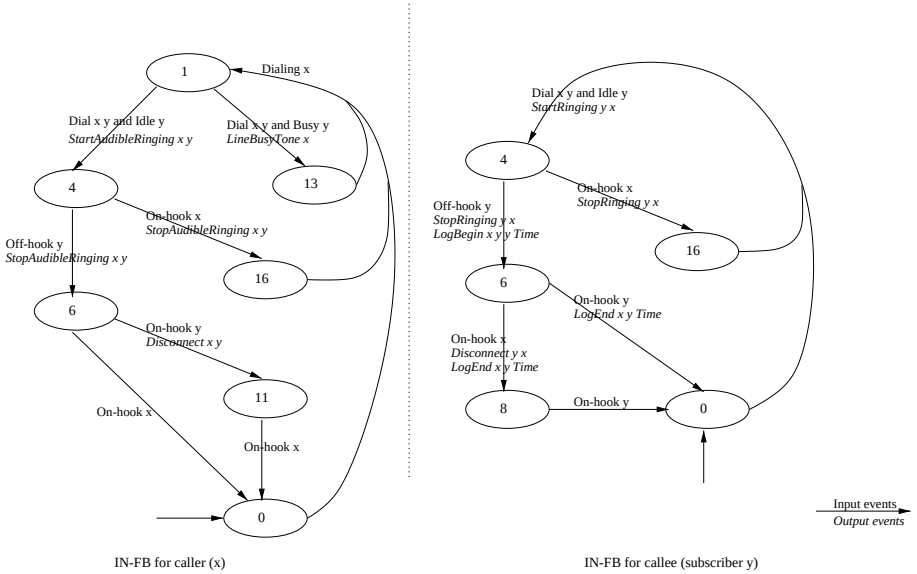


FIG. 6.8 – *Découpage en rôles du service* FreePhone Billing

2. La description du service est ensuite projetée sur chacun de ces rôles, de manière à déterminer quels messages concernent quels usagers. On obtient ainsi un automate par mode de fonctionnement. La figure 6.8 décrit par exemple les deux automates issus de l'analyse du service *FreePhone Billing*, dont la description est donnée en annexe B.

3. Tous ces automates sont composés dans un même module qui constituera la modélisation du service[3]. Chacun de ces automates évolue de manière indépendante. Il est donc tout à fait possible pour un même usager d'occuper des rôles différents pour plusieurs instances d'un même service. En revanche, il ne lui est pas possible d'avoir plusieurs rôles identiques pour un même service. Par exemple, un usager transférant un appel ne pourra en transférer un autre tant que celui en cours ne sera pas terminé.

Les difficultés soulevées par cette dernière expérience sont issues du fait que la description des services était totalement synchrone et n'autorisait pas de messages intermédiaires. On notera par exemple que pour un service de transfert d'appel, il est nécessaire d'impliquer les trois usagers dont le comportement est modifié par le service (appelant, appelé et cible du transfert), alors qu'une modélisation utilisant des messages intermédiaires permet de ne limiter l'influence directe du service sur uniquement deux usagers, l'appelant et la cible du transfert. Ainsi, la limitation citée en exemple dans la troisième étape ci-dessus n'existe pas dans le modèle issu du second concours.

Expression des propriétés

Dans les deux concours, les spécifications fournies étaient de deux natures : l'une fonctionnelle, sous forme de diagrammes, l'autre plus informelle décrivant en langue naturelle les

3. On peut également choisir 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. Le choix fait ici permet néanmoins de mieux identifier l'influence de chaque service.

objectifs du service. C'est essentiellement de cette dernière description qu'ont été extraites les attentes de l'utilisateur, et les propriétés qui traduisent celles-ci. La spécification des services sous forme de diagrammes a parfois été mise à profit pour préciser l'expression des propriétés lorsque la description informelle présentait des imprécisions.

6.4 Conclusion sur l'activité de modélisation

L'activité de modélisation formelle est un préalable indispensable à la validation de services. Elle permet d'obtenir une spécification rigoureuse et non-ambiguë, sur laquelle il est possible de raisonner mathématiquement.
Au travers des études de cas que nous avons menées, nous avons pu dégager un certain nombre de principes qui mènent dans les meilleures conditions possibles à un modèle fiable et rigoureux.

La méthodologie présentée ici n'est encore qu'une ébauche. Des travaux sont actuellement menés au sein de l'équipe de manière à compléter les étapes encore imprécises, telles que la définition des attentes de l'usager et la systématisation de la traduction de la spécification d'un service dans un formalisme adapté.

Nous avons également pu évaluer l'intérêt d'opter pour une modélisation basée sur des principes de synchronisme, même si ceux-ci peuvent requérir d'être légèrement affaiblis.
L'utilisation du langage LUSTRE a été un support pour cette étude ; nous ne prétendons pas qu'il s'agit là du langage le plus adapté à la modélisation de services, même si son caractère formel est en la matière d'un intérêt indéniable. Tous les langages synchrones ayant un pouvoir d'expressivité équivalent, son utilisation a permis en tout cas de montrer les avantages de cette approche de manière générale.

Modularité
Les principes d'intégration et de composition des services décrits en section 6.2.5 permettent de disposer d'une modélisation très modulaire. L'introduction d'un nouveau service se fait simplement, par une opération d'intégration du modèle du service dans le modèle du système. Le système initial n'a pas lui-même à subir de modifications. Tous les services ont la même interface et sont interchangeables. Le modèle est par ailleurs extensible, le nombre de services étant paramétrable.

Intérêt d'un formalisme exécutable
Le choix d'un formalisme exécutable a prouvé être très profitable, pour certaines raisons déjà énoncées plus haut (cf. section 2.6). Le recours à un tel type de modélisation a été par ailleurs indispensable pour décrire certaines fonctionnalités impliquant des structures de boucles, tel que le service de retour d'appel (cf. annexe B). Il semblerait que ce point ait posé de nombreux problèmes à un certain nombre de concurrents lors du concours FIW'98.

Chapitre 7

Principes de validation

Le principe de notre méthode de validation a été introduit en section 2.7. Nous décrivons dans ce chapitre la méthodologie de sa mise en œuvre et l'illustrons principalement sur l'étude de cas relative au premier concours de détection d'interactions (FIW'98). En effet, faute de temps, nous n'avons pu aborder que très partiellement cet aspect lors de l'étude de cas relative au concours FIW'2000. Cette méthodologie comporte les étapes suivantes :

- construction de l'environnement,
- construction des guides de test,
- construction de l'oracle,
- mise en œuvre du test et analyse des résultats.

7.1 Description de l'environnement

L'environnement du système sous test est décrit sous la forme de contraintes qui spécifient ses comportements valides. Ces contraintes, définies comme des invariants de logique temporelle, peuvent avoir des natures temporelles différentes : elles peuvent être instantanées, et énoncer une propriété immédiate que doit vérifier chaque entrée, ou porter sur une plus longue durée, définissant ainsi une relation entre chaque entrée et l'historique des interactions avec le système.

Ces invariants définissent principalement deux types de contraintes : d'une part des contraintes physiques, liées à la manière dont peut se comporter l'environnement, et d'autre part des contraintes d'intégrité, assurant la validité des messages échangés ou décrivant l'état initial de l'environnement.
Les invariants sont également utilisés pour rendre compte de certaines abstractions que l'on souhaite faire sur le comportement de l'environnement. Ils sont enfin le moyen de fixer les paramètres du test, ainsi qu'on le décrira en fin de section.

Dans le cadre des études que nous avons menées, l'environnement est principalement composé des usagers du système téléphonique. Les contraintes doivent donc autoriser l'ensemble des comportements que chaque usager peut réaliser, à l'exclusion de tout autre.

Contraintes physiques
Les contraintes physiques traduisent les impératifs auxquels doit se plier la simulation de l'environnement pour que son comportement corresponde en permanence à un comportement possible dans la réalité.

Une contrainte physique typique, commune aux trois études de cas réalisées, est celle qui exprime l'impossibilité, pour un même usager, de décrocher deux fois de suite son combiné sans raccrocher entre-temps, et inversement. Cette propriété de l'environnement s'exprime en LUSTRE grâce à un opérateur de logique temporelle programmé par l'utilisateur et dont la signification est intuitive : *once ... from ... to*[1]. Dans un formalisme pseudo-LUSTRE, elle s'écrit, avec i prenant ses valeurs dans l'ensemble des postes :

$$once\ \mathrm{On}_i\ from\ \mathrm{Off}_i\ to\ \mathrm{Off}_i\ and\ once\ \mathrm{Off}_i\ from\ \mathrm{On}_i\ to\ \mathrm{On}_i$$

De plus, tous les postes étant initialement raccrochés, il n'est pas possible pour un usager de raccrocher avant d'avoir décroché. Ceci s'exprime aisément grâce à l'opérateur *after* (défini en annexe D).

$$\mathrm{On}_i \Rightarrow after(\mathrm{Off}_i)$$

Cohérence des données
Ces contraintes correspondent à des nécessités imposées par la modélisation que nous réalisons en LUSTRE.

Un événement d'entrée du système est en général composé de trois variables : *type*, *orig* et *param*. *type* décrit la nature de l'événement (typiquement décrocher, composer, ...), *orig* désigne l'usager à l'origine de l'événement et *param* est un paramètre optionnel (comme par exemple le numéro composé).

Un exemple de contrainte d'intégrité concerne la modélisation des messages de numérotation. Un tel message doit obligatoirement contenir le numéro composé. Ceci s'exprime de la manière suivante :

$$type = Dial \Rightarrow Valide(param)$$

Valide est une fonction retournant *vrai* lorsque son argument représente un numéro.

Par ailleurs, le fait que LUTESS ne sache traiter que des données booléennes conduit à représenter des variables numériques entières sous forme de tableaux de booléens. Si on choisit un codage *1 parmi n*, il devient nécessaire de spécifier qu'il n'est pas possible d'avoir plusieurs cellules d'un même tableau affectées de la valeur booléenne *vrai*.

1. *once A from B to C* est vrai si et seulement si A a été vrai au moins une fois entre la dernière occurrence de B et celle de C lui faisant suite (cf. annexe D).

Ainsi, les trois variables décrivant un message sont représentées chacune par un tableau de booléens. Pour des raisons de lisibilité, *type* et *orig* sont codés en 1 parmi n, ce qui se traduit par la contrainte suivante[2] :

$$\#(type)\ and\ \#(orig)$$

Les invariants d'intégrité décrits ici sont typiquement instantanés, à la différence de ceux présentés auparavant.

Simplifications de comportements

Les contraintes peuvent également traduire des hypothèses simplificatrices que l'on estime raisonnables concernant le comportement de l'environnement. Par exemple, on peut supposer que le fait de composer alors que le combiné est raccroché n'a jamais aucune conséquence, et que l'on peut donc réduire l'ensemble des comportements valides de l'environnement en supprimant cette possibilité :

$$always\ \text{not Dial}_i\ from\ \text{On}_i\ to\ \text{Off}_i$$

Paramètres de test

Il est nécessaire de préciser pour chaque service quel usager y a souscrit, et éventuellement quels sont les paramètres associés à chaque souscription. Ceci permet de déterminer une *configuration* de test.

Dans notre contribution au concours FIW'98, nous avons choisi de rendre les listes de souscription statiques, i.e. ne changeant pas durant tout le test. Ces listes sont définies et initialisées manuellement. Ceci se traduit par une contrainte par usager imposant qu'à tout instant, les souscriptions réalisées par cet usager sont invariantes. Par exemple, la contrainte ci-dessous indique que l'usager A n'a souscrit qu'au second des trois services que fournit le système :

$$Liste\_abonnements_A = [false]|[true]|[false]$$

Dans le second concours, nous avons choisi de ne plus considérer de manière explicite ces listes d'abonnement aux différents services, ainsi que les paramètres liés à chaque souscription. Nous avons représenté ces données par des événements aléatoires discrets, générés par l'environnement en supprimant simplement les contraintes imposées sur les souscriptions.

La première possibilité (listes fixées et statiques) est intéressante si on peut intuitivement supposer quelles configurations sont susceptibles de poser problème, ou si le nombre de configurations possibles est faible (c'est le cas par exemple lorsque les services ne sont pas paramétrables). En revanche, lorsqu'il existe un grand nombre de configurations possibles, il est préférable d'opter pour la seconde possibilité, en choisissant des listes de souscription variables et aléatoires.

2. # est un opérateur LUSTRE retournant *vrai* si au plus un de ses opérandes vaut *vrai*.

7.2 Élaboration des guides

Afin d'assurer une bonne efficacité au test, la mise au point de schémas dans le but de guider le test peut s'avérer utile. L'utilisation d'un tel guidage vise à atteindre deux objectifs complémentaires : (1) fournir le moyen de produire des séquences de test réalistes, proches de ce qui pourrait être observé dans la réalité et (2) permettre de conduire le système sous test dans des états dont l'analyse est jugée pertinente.

Les schémas sont élaborés à partir de l'analyse des spécifications de chaque service, pris indépendamment. L'objectif est d'extraire de ces spécifications la description de chacun des comportements génériques les plus significatifs concernant le service. Ces comportements doivent notamment inclure les différentes procédures relatives à l'activation et au déclenchement du service[3].

On aura soin de décrire chaque schéma de la manière la plus générique possible, de sorte qu'il permette de couvrir la totalité des comportements qu'il est censé représenter. On évitera en particulier de donner des conditions trop restrictives, que ce soit pour caractériser un instant ou un intervalle.

Dans certains cas, néanmoins, un schéma trop général ne sera pas efficace. Il faut en effet se rappeler que l'influence du guidage n'agit que sur la production d'entrées ; il n'est pas possible de favoriser directement le passage du système dans un état donné. Sur l'exemple décrit par la figure 4.3 (section 4.6), on a par exemple une condition initiale portant sur le fait que l'usager est dans l'état *Talking(A)*. Si cette condition n'est que rarement satisfaite, le schéma aura rarement la possibilité d'être complété. On peut alors choisir de spécifier le schéma, en le rendant moins général. Dans notre exemple, cela peut se faire en précisant une manière pour le système d'aboutir dans un état où la condition *Talking(A)* est satisfaite. En l'occurrence, on peut par exemple remplacer la condition initiale par le schéma :

$$Off(A)\ [not\ On(A)]\ Dial(A,C)\ and\ Idle(C)\ [not\ On(A)]\ Off(C)$$

Le service de transfert d'appel explicite (ECT) a été décrit au §4.1.1. Rappelons qu'il permet à son souscripteur de transférer le correspondant avec qui il est en communication vers un autre poste de son choix.
Pour solliciter ce service de manière efficace, il était donc nécessaire de conduire un usager en communication à produire la séquence d'actions suivantes : mettre en attente son correspondant (*Hold*), composer le numéro vers lequel transférer l'appel (*Dial*) et enfin procéder au transfert (*InvokeECT*). La figure 7.1 propose un schéma simple permettant de réaliser ce guidage. A représente le souscripteur au service, B son correspondant initial, *Talk(A,B)* le fait qu'une communication soit établie entre eux, et C le numéro du correspondant vers qui A veut transférer B.
Pour affiner ce guidage, il serait possible de contraindre plus fortement les intervalles, en imposant des conditions sur le comportement des usagers B et C.

3. Ces deux phases du cycle d'exécution d'un service ont été introduites en section 2.1.

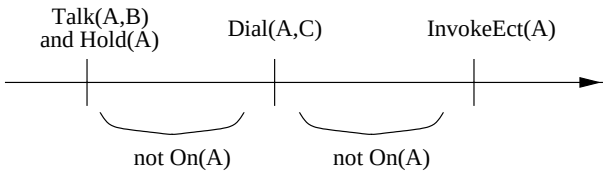


FIG. 7.1 – *Schéma de guidage pour le service ECT*

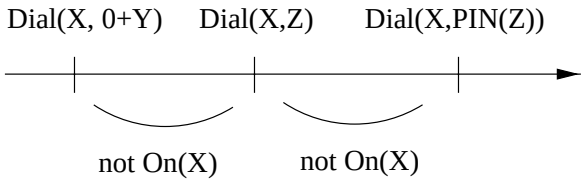


FIG. 7.2 – *Schéma de comportement pour le service Charge Call*

Dans les faits, ce guidage a permis d'accroître considérablement la fréquence de sollicitation du service. Sur 8.000 pas de test, la technique purement aléatoire n'avait permis de n'observer que 2 exécutions du service, ce qui était insuffisant pour mener à une quelconque conclusion quant à la validité du service. Selon les pondérations choisies, le guidage par schéma permettait d'augmenter sensiblement ce nombre : pour une pondération très raisonnable (poids peu déséquilibrés), on a obtenu par exemple 30 occurrences. En influençant plus fortement la génération, ce nombre a dépassé 500.

Dans l'étude de cas relative au concours FIW'98, les schémas ont également été fort utiles pour de nombreux services relativement complexes. Considérons par exemple le service de facturation *Charge Call*, décrit également au §4.1.1. L'invocation du service requiert de composer tout d'abord le numéro du correspondant souhaité, préfixé par 0, puis composer le numéro du poste que l'on désire facturer, et enfin entrer le code d'identification valide (PIN) pour ce poste. Les comportements intéressants à tester seront ceux où cette séquence d'actions est menée à son terme (cf. figure 7.2), car ils permettent d'invoquer le service. Les résultats de l'utilisation de cette méthode de guidage pour cette étude de cas seront présentés en section 7.6.2.

7.3 Définition de l'oracle

L'oracle se compose des propriétés que l'on souhaite voir satisfaites par le système de télécommunication. En particulier, il s'agit d'exprimer les attentes des usagers concernant chacun des services fournis par le système. Ces propriétés sont issues de l'analyse de la description de chaque service. L'oracle est ensuite constitué de la mise en conjonction des propriétés attendues des services, ainsi que des propriétés du système (cf. section 2.8). En cas de violation de l'oracle, LUTESS permet d'assister le testeur dans son analyse des traces afin de déterminer la propriété mise en défaut.

On a mentionné en fin du chapitre précédent (section 6.2.7) les deux formes que pouvait

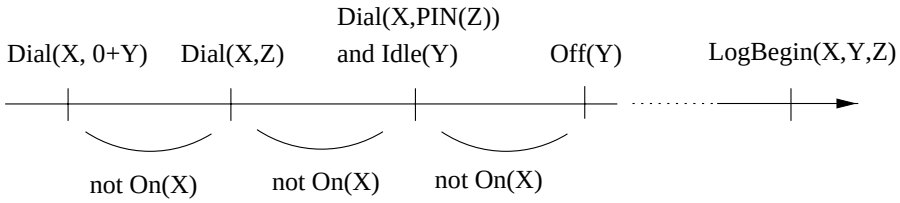


FIG. 7.3 – *Propriété du service Charge Call*

prendre une propriété de service, soit invariante, soit comportementale.

Les propriétés invariantes s'expriment sous la forme "Telle situation n'apparaît jamais". Par exemple, un service de transfert d'appel inconditionnel doit forcément empêcher le poste concerné de sonner. Cette relation peut s'écrire ($CFsub(i)$ étant le prédicat indiquant que i est abonné au service de transfert) :

$$CFsub(i) \Rightarrow notRing_i$$

Les propriétés comportementales s'expriment quant à elles sur le schéma suivant : "tel enchaînement de conditions mène de manière immédiate à telle conclusion". Un formalisme graphique permet de les représenter simplement, en adjoignant à celui déjà présenté pour le guidage la notion d'implication. Celle-ci est représentée par une ligne pointillée qui sépare la séquence de prémisses à gauche de la conclusion à droite, celle-ci devant apparaître à l'instant même où la prémisse est satisfaite.

En reprenant l'exemple du service *Charge Call* présenté plus haut, on peut vouloir tester la propriété suivante : "lorsqu'un usager invoque le service Charge Call (cf. figure 7.2) et que son interlocuteur accepte l'appel, ce dernier est facturé sur le compte voulu". Sur la figure 7.3, le prédicat LogBegin(X,Y,Z) signifie que la facturation de la communication de X vers Y est portée sur le compte de Z.

La correction et la complétude des propriétés est un aspect délicat. En effet, celles-ci doivent traduire les attentes de l'usager, lesquelles attentes sont forcément subjectives. Il faut donc au préalable s'attacher à décrire de manière non-ambigue et rigoureuse ces attentes avant de pouvoir en extraire les propriétés. Concernant la complétude de cette traduction *attentes* $\Rightarrow$ *propriétés*, nous ne disposons pas des moyens de nous en assurer. Néanmoins, il est utile de vérifier que chacun des aspects abordés par le service fait au moins l'objet d'une propriété. Par exemple, un service de transfert d'appel affecte à la fois la mise en connexion des usagers, ainsi que la facturation des appels ; il faudra donc que le service soit validé par rapport à ces deux aspects. On discutera plus loin (section 7.5) du problème de la correction des propriétés.

7.4 Mise en œuvre du test

Dans le cadre du premier concours de détection d'interactions, l'énoncé demandait à confronter les services uniquement par paires. Ceci repose sur une hypothèse largement par-

tagée (cf [25], par exemple) selon laquelle la plupart des interactions se produisent déjà entre deux services. Nous discuterons néanmoins dans les perspectives (cf. section 10.2.3) de l'extension de la méthode de validation aux cas d'ensembles plus larges de services.

Nous avons par ailleurs décidé arbitrairement de fixer le nombre d'usagers du système à 4, en faisant l'hypothèse que la plupart des interactions se produisent en impliquant un petit nombre d'usagers.

Ainsi qu'on l'a déjà mentionné, nous avons dans cette même étude de cas choisi que les *configurations de test* soient précisées par le testeur. Il ne s'agit pas de tester toutes les configurations possibles. En effet, tous les postes étant identiques et pouvant accéder aux mêmes services, de nombreuses configurations sont équivalentes. Afin de déterminer un ensemble minimum de configurations à tester, il est nécessaire de procéder à l'analyse manuelle de la description informelle de chaque service. Cette analyse doit permettre de déterminer pour chaque configuration les listes de souscription et les paramètres éventuellement associés à certaines souscriptions.

Minimiser la complexité de l'environnement en limitant le nombre d'usagers modélisés et réduire le nombre de configurations à tester permet un gain de temps non négligeable dans la phase de test proprement dite. En particulier, un environnement moins complexe, modélisant moins d'usagers, sera plus rapide à construire.

La validation d'un service seul comporte deux phases. La première consiste à valider une seule instance du service (i.e. tester une configuration où un seul usager est souscripteur). Une seconde consiste à confronter le service à lui-même en rajoutant un deuxième abonné. Lorsqu'il s'agit de confronter plusieurs services, le nombre d'abonnés à chacun varie entre 1 et 2, sous l'hypothèse déjà mentionnée qu'une interaction implique généralement peu d'usagers.

Les paramètres de souscription requièrent une analyse plus poussée. Par exemple, pour des services de transfert comme FreephoneRouting, les paramètres sont la donnée d'une plage horaire et d'un numéro de téléphone. Il est pertinent de choisir en priorité des valeurs de ces paramètres telles que les plages puissent se chevaucher et que les numéros de téléphone puissent désigner des usagers disposant d'autres services.

L'autre possibilité que nous avons étudiée, bien que nous n'ayons pas eu matériellement le temps de la mettre en œuvre, consistait à faire abstraction des aspects de souscription pour les remplacer par des variables aléatoires (cf. paragraphe "paramètres de test", section 7.1). Pour être valide, cette solution impose néanmoins que certaines précautions soient prises. Il faut en particulier veiller à la cohérence des données au cours d'un même appel : les paramètres liés à une instance de service ne doivent pas changer. Cela peut demander éventuellement de modéliser les services avec attention de sorte que chacun ne fasse appel qu'une seule fois aux variables aléatoires entre le moment où il se déclenche et le moment où il se termine.

7.5 Analyse

La méthode de validation que nous proposons ici accorde une importance considérable à la manière dont les propriétés sont formulées puisque c'est sur ces dernières, et uniquement sur elles, que se base l'activité de détection d'interaction.

L'activité d'analyse des attentes des usagers, puis de spécification des propriétés traduisant ces attentes, est donc cruciale pour l'efficacité de la méthode. Cette spécification est particulièrement malaisée à concevoir de manière parfaitement objective : en effet, le testeur qui conçoit une propriété a toujours plus ou moins inconsciemment à l'esprit le type d'interactions que celle-ci va permettre de révéler. C'est ce que fait par exemple remarquer H. Velthuijsen dans [102].

Par ailleurs, il est difficile de s'assurer de la correction des propriétés exprimées. Le fait de réaliser la traduction *attentes* $\Rightarrow$ *propriétés* avec précaution est utile, mais pas forcément suffisant. Cette section propose deux méthodes de mise au point d'indicateurs permettant d'aider à l'analyse du test.

7.5.1 L'oracle comme révélateur de situation

Certaines attentes de l'usager peuvent être trop vagues ou trop complexes pour s'exprimer simplement sous la forme d'une propriété de sûreté. Dans ce cas, plutôt que de risquer de définir une propriété erronée ou imprécise, on peut préférer remplacer l'oracle *de propriété* par un *révélateur de situation.*

Un révélateur de situation est un indicateur qui n'a pas valeur de verdict, mais qui permet au testeur humain de concentrer son analyse autour de certaines situations particulièrement dignes d'intérêt.

Considérons par exemple la propriété "$A \Rightarrow B$", où A caractérise un comportement de l'environnement et B la réponse du système. Supposons que B soit relativement complexe, car basé sur des calculs arithmétiques de date et de durée. Plutôt que de risquer de considérer une expression incorrecte de B, on ne considérera que "A" dans l'oracle, sous forme de révélateur, et on laissera au testeur humain la tâche de mener les calculs nécessaires à l'évaluation de B.

Le problème peut aussi se poser dans le cas où la propriété que l'on souhaite exprimer n'est pas invariante, c'est-à-dire que la réponse B du système à un comportement A n'est pas forcément attendue instantanément, et puisse être espérée après un délai indéterminé a priori. On présentera au chapitre 9 une autre solution à ce genre de problème.

Les propriétés propres au système (cf. section 2.8) peuvent être considérées comme des révélateurs, puisque leur mise en défaut ne repère pas forcément une erreur. Par exemple, il est supposé que tout poste logique désigné comme "correspondant" par un autre poste

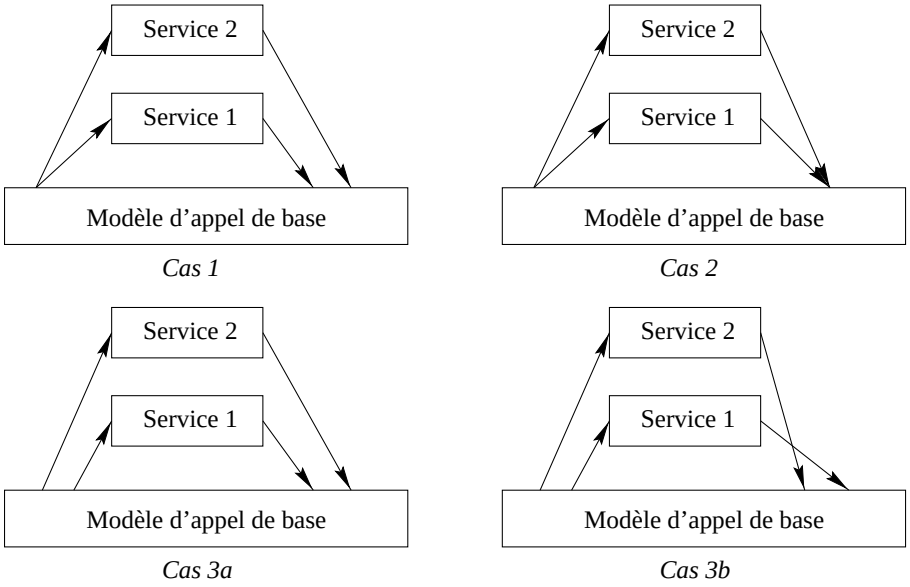


FIG. 7.4 – *Différentes situations de conflits de contrôle*

logique ait lui-même un correspondant. Il est intéressant de fournir un révélateur permettant d'indiquer quand cette convention n'est plus respectée. C'est ensuite au testeur humain de juger, si l'outrepassement de cette règle mène à une interaction ou non. Le révélateur se borne à localiser et à désigner les différents points d'apparition de la situation dans les traces.

7.5.2 Enrichissement de l'oracle par indicateurs

Les révélateurs de situation décrits plus haut requièrent du testeur qu'il fournisse une certaine quantité d'information, puisqu'il lui faut au préalable repérer les situations à révéler. Nous proposons ici d'instrumenter le modèle afin que soient remontées vers l'oracle certaines informations susceptibles d'identifier une interaction selon des critères stables, indépendants des services impliqués. Cette technique est contraignante, car elle suppose de modifier le système. Elle est en revanche performante, car elle permet de rendre le processus de détection partiellement indépendant de l'activité de spécification des propriétés de service. Ceci est particulièrement intéressant si l'on considère que cette dernière activité pose un certain nombre de problèmes quant à la correction et à la complétude desdites spécifications.

Nous avons identifié deux indicateurs pertinents, que nous décrivons après avoir introduit les notions de domaine d'action d'un service et de conflit de contrôle.

Domaine d'action d'un service

Ce paragraphe introduit la notion de domaine d'action et de conflit de contrôle. On identifie sous le nom de *domaine d'action* d'un service l'union des parties du modèle d'appel de base inhibées par la présence du service. En d'autres termes, le domaine d'action décrit les phases d'un appel contrôlées par le service.

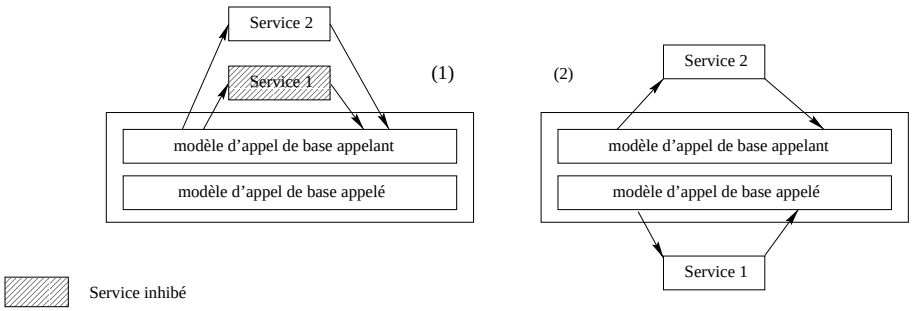


FIG. 7.5 – *Concurrence de services sur un même appel*

Il existe un conflit de contrôle sur un appel dès lors que deux services ont des domaines d'action non disjoints. On peut identifier sur le schéma 7.4 les cas de conflit suivants :

- Les services se déclenchent sur un même point d'initiation et leurs points de retour diffèrent (cas (1)).
- Les services se déclenchent sur un même point d'initiation, leurs points de retour sont identiques, mais les mises à jour requises par chacun sont incompatibles (cas (2)).
- Un service se déclenche avant l'autre, mais son point de retour se situe chronologiquement après le point d'initiation de ce dernier (cas (3a) et (3b)).

Conflit de réaction

Lorsque deux services réagissent en simultané à une même entrée (cas (1) ou (2) de la figure 7.4), le module de composition choisit une des réactions, en fonction des priorités relatives des services, et ignore l'autre, qui est alors *inhibée*. Cette inhibition est révélatrice d'un conflit, qu'il est utile de signaler à l'usager[4].

Pour informer l'oracle des inhibitions de ce type, il est nécessaire de rajouter dans le modèle un observateur sur chaque poste qui surveille à chaque instant si l'opération de composition a donné lieu à un conflit ou non.
L'implémentation de ces observateurs a dans la pratique consisté à modifier l'opérateur de composition, de sorte que celui-ci fournisse une sortie supplémentaire correspondant à l'information attendue[5].

Remarque : cet indicateur suppose que le mode de composition choisi est une composition par *superposition* (cf. section 6.2.5). Si la composition est par *juxtaposition*, un tel indicateur est inutile (cf. section 7.6.3).

Conflit de contrôle

On a présenté page 79 le principe de localité, selon lequel le contrôle d'appel est effectué

4. Il faut toutefois noter que ce conflit ne traduit pas obligatoirement une interaction, car ses conséquences ne seront pas forcément observables ou jugées néfastes par l'usager.

5. La composition est implémentée en LUSTRE par le module nommé "Sélecteur/réinitialiseur" (cf. figure A.2 en annexe).

localement sur chaque poste. Dans le cas où deux services souhaitent intervenir sur un même poste, à des instants différents (cf. cas (3a) et (3b) de la figure 7.4), le service intervenant le plus tardivement est inhibé (cf. cas (1) sur la figure 7.5), car les deux services sont ouvertement en conflit pour le contrôle de l'appel. En revanche, lorsque deux services souhaitent prendre sur deux postes différents le contrôle du traitement d'un même appel (cas (2) sur la figure 7.5), ils n'ont pas connaissance l'un de l'autre et le fait que le contrôle de l'appel soit conflictuel n'est pas forcément visible. Il est donc utile de rendre observable cette coexistence, qui a de grandes chances d'être vectrice d'interactions.

Nous n'avons pas à l'heure actuelle effectué l'instrumentation du modèle qui permettrait de remonter vers l'oracle les informations nécessaires à la réalisation de cet indicateur. Ces informations doivent faire apparaître :

- l'indication des instants de début et fin d'appel, de manière à savoir à tout moment quels appels sont en cours ;
- l'indication du déclenchement et de la terminaison d'un service, et du poste sur lequel il se déclenche.

Tous ces renseignements peuvent être extraits sans difficulté du modèle. L'oracle doit pour sa part procéder à l'analyse des informations ainsi fournies, afin de fournir à l'usager un état des situations de contrôle conflictuel.

7.6 Résultats et évaluation des études de cas

7.6.1 Réalisation

L'expérimentation que nous avons menée autour de la validation proprement dite a porté principalement sur l'étude de cas relative au concours FIW'98. Il s'agissait de considérer 78 paires de services et de détecter toutes les interactions néfastes pouvant apparaître entre chaque couple.
Chaque paire de services a été testée en variant les configurations d'abonnement et de paramétrisation, selon les principes décrits plus haut. Sur le plan quantitatif, 42 situations de tests ont requis l'utilisation de schémas, 26 celle de probabilités conditionnelles, et dans 10 cas, aucun guide n'a été nécessaire. Pour ces derniers cas, les services impliqués étaient simples, et invoqués suffisamment fréquemment pour que des tests plus dirigés s'avèrent inutiles.

Lorsqu'une paire de services imposait le recours aux schémas, le guidage était réalisé en mettant en concurrence l'ensemble des schémas de chaque service, grâce au mécanisme présenté en section 4.5.

La validation de l'étude de cas (hors modélisation du système) a requis un effort de 90 hommes*jours, se divisant grossièrement en 1 homme*jour pour déterminer les propriétés de chaque service et produire l'oracle correspondant (soit 12 hommes*jours) et 1 homme*jour

pour tester chaque paire de services (soit 78 hommes*jours).

7.6.2 Résultats

Lutess a été déclaré "meilleur outil", vainqueur de la compétition. Sur l'ensemble des 78 situations de test, le jury a recensé 98 interactions, considérées "valides". Notre outil a quant à lui permis d'identifier 82 cas d'interaction, dont 72 valides. Certaines interactions valides sont donc absentes de nos résultats, tandis que plusieurs autres ont été introduites. D'autres enfin se sont manifestées sous une forme différente de celle attendue.

En ce qui concerne les 26 interactions valides non détectées par Lutess, la différence de résultats s'explique comme suit :

- Dans 10 cas, l'interaction était détectable par Lutess, et seul le manque de temps nous a empêché de l'identifier.

- 11 de ces interactions ne sont en réalité que d'autres manifestations de conflits déjà révélés par Lutess. Nous avons estimé inutile de recenser tous les symptômes identifiant un même problème.

- Enfin, 5 interactions n'étaient pas détectables, compte tenu des propriétés que nous avions choisi de considérer dans l'oracle. Ceci provient d'une interprétation différente de l'attente de l'usager. Par exemple, nous n'avons pas jugé essentiel de vérifier la bonne terminaison de chaque appel, alors que le jury a considéré cette notion comme primordiale.

Pour ce qui est des 10 interactions non-recensées par le jury, elles sont toutes dues à une divergence dans l'analyse des attentes de l'usager. En d'autres termes, nous avons imposé sur les services certaines propriétés, considérées comme non-justifiées par le jury. Nous avons en particulier défini une propriété trop forte, concernant 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 a supposé 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.

7.6.3 Influence du mode de composition de services

Nous avons discuté en section 6.2.5 des différentes manières de procéder à la composition de services. Une de nos suppositions est qu'une interaction logique est inhérente à un ensemble de services (tout du moins, étant donné le niveau de détail auquel chacun d'eux est représenté), et qu'elle est indépendante de la manière dont les services sont composés.

Afin de valider cette hypothèse, nous avons procédé à une expérimentation sur la plupart des paires de services testées dans le cadre du concours FIW'98, en utilisant successivement les deux modes de composition. Dans l'immense majorité des cas d'interaction, les deux méthodes de composition ont permis de révéler le problème, sous des manifestations parfois différentes.

La méthode de composition par juxtaposition possède l'intérêt de faciliter la détection des conflits de réaction (i.e., lorsque deux services réagissent sur un même événement, cf. section 7.5.2), qui se manifestent par une réponse incohérente.

La méthode de composition par superposition requiert pour ce même résultat de faire appel aux indicateurs de conflits de réaction. Néanmoins, cette deuxième méthode, qui fournit une réponse toujours cohérente, peut apparaître plus satisfaisante pour deux raisons : sur le plan formel, le fonctionnement de chaque poste conserve une description sous forme d'automate. En pratique, le comportement du système préserve ainsi sa cohérence, et les traces résultant d'un test sont plus facilement analysables.

7.6.4 Influence de l'ordre de composition

Ainsi qu'on a pu le voir au chapitre précédent, l'opération de composition n'est pas commutative. En toute logique, pour vérifier l'absence d'interaction entre un ensemble de services, il faudrait tester chaque composition possible.
Néanmoins, selon l'hypothèse énoncée au paragraphe précédent, une interaction est inhérente à un ensemble de services. Dans ces conditions, l'ordre de composition n'a pas d'importance et l'interaction se manifestera dans tous les cas, sous des formes différentes.

Troisième partie

Travaux connexes et conclusions

Chapitre 8

Travaux connexes

Comme on l'a suggéré en introduction, la création et le déploiement de services téléphoniques constituent une activité relevant du génie logiciel. Ainsi, on retrouve dans ce domaine l'application de nombreuses méthodes du génie logiciel, comme les méthodes orientées objet [64], les méthodes à base de patrons [100], les systèmes à agents [20], ...

Pour cette même raison, de nombreuses méthodes formelles ont été étudiées pour être appliquées à ce domaine. Cette exploration a été d'autant plus précoce qu'historiquement certaines de ces méthodes (telles SDL [25] ou LOTOS [38]) sont depuis longtemps utilisées avec succès dans le domaine proche des protocoles de communication.

La manière dont nous avons modélisé les services est similaire à de nombreuses propositions à base de processus communicants [71, 25, 39]. La différence essentielle réside dans le fait que l'emploi de techniques synchrones a permis de réduire significativement la complexité des modèles.

Dans le domaine de la détection d'interactions, les travaux les plus aboutis s'appuient en général sur des modèles et des techniques outillées de vérification-validation, sans lesquels la complexité du problème est impossible à maîtriser. Ces travaux assimilent la recherche d'interactions à un problème de vérification. La réussite de la détection et le degré d'automatisation dépendent alors du formalisme employé et de la méthode de vérification mise en œuvre.
Parmi les différents formalismes ayant fait leurs preuves dans d'autres domaines et appliqués à la recherche d'interaction, on peut citer les modèles à base de machines d'états finis [10, 9, 74] ou de processus parallèles et communicants [71, 25, 39]. On trouve également certains formalismes dédiés [50, 99].
De rares approches ont conduit à la formalisation de la notion d'interaction [9, 44] ; néanmoins, la définition de l'interaction qui en est issue est simplifiée et ne couvre qu'un spectre restreint de cas d'interaction. En conséquence, de nombreuses interactions ne peuvent être découvertes.

Notre approche relative au problème de détection d'interactions peut se réduire à l'éva-

luation d'une propriété de logique temporelle sur un modèle.
Cette approche est classique mais s'appuie habituellement sur des techniques de vérification du type model-checking [9, 25, 39]. Quelques rares travaux abordent le problème du test. Ainsi, Combes [2] propose une validation basée sur la simulation exhaustive d'un modèle exécutable, ce qui présente certaines similitudes avec le test. Godskesen [45] mentionne le test de conformité comme étant adapté à la détection d'interaction au niveau de l'implémentation du service.

Les deux premières sections de ce chapitre décrivent des approches concurrentes à la nôtre concernant la modélisation de services téléphoniques (section 8.1) et la détection d'interactions entre services (section 8.2). Une dernière section situe notre approche par rapport à d'autres travaux basés sur le test.

8.1 Approches de modélisation de services

8.1.1 Utilisation de logiques temporelles linéaires standards

Les travaux de Jonsson et al. [10, 9, 79] mettent en avant l'intérêt des formalismes à base de logique temporelle pour spécifier un système de télécommunication ; ceux-ci présentent notamment l'avantage d'être flexibles et modulaires et donc de faciliter la composition de services.

Le formalisme choisi est la logique temporelle définie par Lamport, TLA [69]. Dans ce contexte, les auteurs se proposent de spécifier chaque service de manière indépendante sous la forme d'un système réactif. Le système est décrit par un ensemble de variables d'état et réagit aux événements extérieurs en changeant d'état.
Ce style de spécification à base de changements d'états pourrait également se baser sur un langage de spécification algébrique tel que Z [95].

La spécification d'un service se résume à l'énoncé :

$$\mathit{Initial} \ \wedge \ \Box \ \mathit{React} \ \wedge \ \Box \ \mathit{Frame}$$

où :

- *Initial* représente l'état initial.

- *React* est l'ensemble des réactions (spécifications de changement d'état). Chaque réaction est de la forme $\mathit{event} \Rightarrow (\mathit{allowed} \wedge (\mathit{condition} \Rightarrow \mathit{effects}))$.
  effects est le changement d'état (modification de variables) à effectuer lorsque la condition *condition* est remplie et que l'événement *event* est fourni en entrée du système. *allowed* est une garde qui indique dans quelles conditions l'événement est pris en compte.

- *Frame* indique quels événements affectent quelles variables. Ceci permet d'assurer que les variables non-affectées par un événement conservent une valeur constante.

Idéalement, le souhait d'un concepteur de services est de pouvoir définir chaque service de manière indépendante, puis procéder à la composition de la manière la plus simple possible. L'intérêt d'un style de spécification comme celui présenté ici est que la composition de services se fait simplement grâce à l'opérateur de conjonction logique $\wedge$.

En termes de modélisation de services, ces travaux prouvent l'utilité d'un style de spécification à base de logique temporelle pour décrire de manière structurée un système modulaire. Néanmoins, le fait qu'il n'existe pas en logique temporelle de mécanisme de communication évolué interdit de spécifier le modèle à un niveau plus concret, en faisant apparaître certains éléments de son architecture, ou encore d'envisager de dériver automatiquement une implémentation. La modélisation ne peut être réalisée qu'avec un haut niveau d'abstraction.

8.1.2 Modélisation à base de processus communicants

SDL

Combes et al. proposent dans [25] une modélisation de services téléphoniques basée sur le langage SDL. La spécification modélise une vue externe du système observé par un usager et consiste en un ensemble de processus parallèles communicants.

Un appel est décrit par une instance d'un processus décrivant l'appel de base, tandis qu'un service est représenté par un processus concurrent s'exécutant en parallèle. Un service est considéré comme une *alternative* à une portion de l'appel de base. La modélisation choisie suit en cela les recommandations de l'ITU à propos du plan fonctionnel global (cf. section 6.1).

Les relations entre le processus du service de base et les processus des services supplémentaires correspondent à un transfert du contrôle de l'appel. Les services supplémentaires sont capables d'indiquer au processus du service de base la ou les situations dans lesquelles ils souhaitent prendre le contrôle de l'appel. Initialement, c'est le service de base qui le possède. Le processus du service de base fait évoluer l'appel jusqu'à atteindre un état où une des situations indiqués par les services se produit. Le service de base passe alors le contrôle de l'appel au processus du service concerné, et attend que celui-ci lui rende le contrôle.

Cette approche de modélisation est analogue à la nôtre, malgré quelques différences essentielles, propres au choix d'un formalisme asynchrone. Ce choix est justifié par les auteurs par la volonté de rester proche des standards de modélisation dans les télécommunications.

Lotos

Le travail de Logrippo et al. présenté en [38] consiste en l'application d'une méthode de modélisation particulière au domaine des télécommunications. Les auteurs proposent une spécification dont le niveau d'expression est un intermédiaire à mi-chemin entre les exigences des services et leurs spécifications.

La modélisation proposée est faite en LOTOS et consiste à n'exprimer que les contraintes associées au système sans jamais faire explicitement référence à un quelconque état interne. On rend ainsi opaque le fonctionnement dudit système. Les contraintes sont uniquement exprimées en fonction des états externes (i.e, observables de l'extérieur) des postes. La spécification distingue trois types de contraintes, que l'on peut considérer comme autant de niveaux d'abstraction :

- les contraintes locales, permettant d'assurer un séquencement correct des observations sur un poste donné ;
- les contraintes de connexion, garantissant le synchronisme entre les séquences d'observations des postes en communication. Ces contraintes peuvent également être considérées comme des liens de cause à effet entre les observations effectuées sur les différents postes ;
- les contraintes globales, assurant la cohérence globale du système, par supervision de toutes les observations effectuées.

Les observations effectuées sur un poste regroupent aussi bien les actions de l'usager (comme le fait de composer, de décrocher, ...) que les signaux provenant du réseau (signal de sonnerie, signal occupé, ...), sans distinction d'origine. Le système global est composé d'un service de base et de services supplémentaires. Chaque service possède ses propres contraintes. Les contraintes locales et de connexion d'un service supplémentaires sont considérées comme une alternative aux contraintes du service de base.
Dans le cadre du service de base, les contraintes globales se limitent à garantir qu'un poste n'apparaît que dans une seule connexion à la fois, en gérant des ensembles représentant les postes libres et occupés. Pour un service supplémentaire, elles incluront également des contraintes sur le contrôle des appels : par exemple, pour un service devant faire coexister plusieurs connexions sur un même poste, les contraintes globales associées devront permettre la gestion de cette coexistence, par la définition de variables appropriées.

L'introduction d'un nouveau service se fait en deux étapes :

1. Il faut définir tout d'abord ses contraintes locales et de connexion, et les composer -au sens de la composition parallèle LOTOS- avec les contraintes du système préexistant. Un poste pourra alors avoir plusieurs comportements possibles, chacun correspondant aux contraintes dues à un service particulier.
2. Il faut ensuite modifier les contraintes globales du système afin de prendre en compte les contraintes spécifiques au service à ajouter.

L'intérêt de cette technique incrémentale est de circonscrire les modifications à apporter à la spécification préexistante lors de l'ajout d'un service, celles-ci étant limitées aux seules contraintes globales.
Un autre point intéressant dans cette approche est la distance prise par rapport à l'origine des observations. On ne soucie pas en effet de distinguer leur nature ni leur cause. Cette abstraction est spécifique au mode de communication de LOTOS et n'est malheureusement

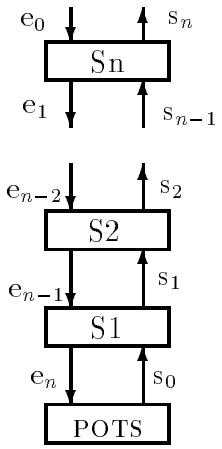


FIG. 8.1 – *Structure de composition des services*

pas exploitable dans notre approche.

8.1.3 Piles d'automates

Braithwaite et Atlee proposent dans [16] une modélisation d'un système de télécommunication par des piles d'automates communicants. Les services sont conceptualisés par des filtres agissant sur les entrées d'un poste.

Chaque usager est représenté par une ou plusieurs piles, correspondant chacune à la vision qu'il a d'une connexion dans laquelle il est impliqué. Les piles peuvent être créées et modifiées dynamiquement, suivant les activations/désactivations de services.
Chaque pile comporte au moins une vue de l'automate du service de base, à laquelle viennent s'ajouter les automates des services activés concernant l'usager. La machine de plus bas niveau représente le service de base, tandis que les machines de niveaux supérieurs sont associées aux services supplémentaires (cf. fig. 8.1).

Les automates évoluent en réagissant à des événements, soit provenant d'autres automates de la pile, soit provenant du reste du système (i.e. de l'usager ou d'autres piles).

Les échanges entre une pile et le reste du système se font par le sommet de pile ; les événements entrant peuvent être consultés, interceptés ou modifiés par chacun des automates traversés avant d'aboutir éventuellement au service de base.

L'empilement définit une relation de priorité entre les automates, celui situé en sommet de pile étant le plus prioritaire. De plus, chaque transition souhaitée par un automate doit recevoir l'aval de tous les automates de plus forte priorité: une notification de changement d'état est soumise en sommet de pile, et redescend jusqu'à l'automate émetteur. Si aucun des automates notifiés n'a jugé bon d'intercepter le message, alors seulement l'automate émetteur reçoit l'autorisation d'effectuer sa transition.

Un service peut être réparti sur plusieurs piles, lorsqu'il concerne plusieurs postes simultanément. L'empilement et le dépilement sont donc des processus dynamiques. De plus, les parties d'un même service communiquent via un canal dédié.

Notre modélisation des services s'inspire du concept de filtrage présenté ici, mais est cependant plus simple. Les filtres sont exclusifs, en ce sens que dès que l'un d'eux a émis une proposition, les autres sont inhibés. Les différents composants d'un service ne communiquent pas entre eux par un canal dédié.

8.1.4 Plug-and-play

Les travaux décrits dans [86] proposent une approche générale visant à construire les services de la manière la plus indépendante possible et à faciliter ensuite leur composition.

L'objectif principal de ces travaux est d'étendre un langage de spécification donné avec un opérateur de construction de services et de fournir une méthode simple pour intégrer ces services dans un système pré-existant. Chaque service est considéré comme une extension et/ou une modification du système pré-existant. L'utilisation d'un opérateur dédié à la description des services permet de contrôler l'influence de ces services.

La syntaxe imposée par l'opérateur de construction demande de préciser :

1. les éléments (variables, modules) que doit au minimum contenir le système de base,
2. les éléments (variables, modules) qui seront rajoutés au système par l'intégration du service,
3. les changements apportés par le service dans la définition des variables du système.

Bien que l'approche ne soit pas liée à un quelconque formalisme, les auteurs ont choisi de l'instancier avec l'outil de model-checking SMV [24], qui offre à la fois un langage de modélisation et une méthode de vérification. L'opérateur de description est en fait une extension du langage, qui est ensuite traduite grâce à l'outil d'intégration prévu à cet effet, *SMV Feature Integrator*.

L'intérêt de cette approche est de proposer un mécanisme de composition très simple et adaptable à de nombreux formalismes. Dans une certaine mesure, l'approche que nous proposons qui consiste à identifier pour chaque service les modifications que celui-ci souhaite imposer au service de base peut être assimilée à une adaptation de ces principes.

8.1.5 Langages dédiés

Les travaux présentés par Hall [50] proposent un formalisme dédié sous la forme d'un langage procédural, P-EBF (*procedural event-based formalism*), pour la modélisation de spécifications de services.

Un modèle est composé d'une collection de variables décrivant son état et d'un ensemble de gestionnaires d'événements (*event-handlers*) qui définissent chacun une réaction du modèle face aux événements qui lui sont fournis en entrée.

La composition de services se fait en opérant l'union des variables d'état et la concaténation des gestionnaires se rapportant à un même événement.

8.2 Approches de détection d'interactions ou de validation de services

8.2.1 Analyse statique

Logique temporelle

Dans [9] (cf. section 8.1.1), les spécifications s'attachent uniquement à la description des changements d'état. En particulier, la notion de propriété n'est pas définie.
La détection s'opère en deux phases : analyse statique des spécifications, pour repérer les réactions incohérentes (caractérisées par des conflits de gardes sur un même événement d'entrée, ou par des effets contradictoires en réaction à un même événement), puis calcul de l'accessibilité des états autorisant ces réactions par une méthode de calcul en arrière [59].
Dans le cadre de l'application de cette approche au concours FIW'98, les effets contradictoires se manifestaient par une sortie incohérente, comme par exemple la production de deux tonalités ou annonces simultanées, ou deux facturations concurrentes.

Dans [79], les interactions citées plus haut sont dénotées sous le terme d'interactions de contrôle. Pour les besoins du concours FIW'98, les auteurs ont dû rajouter un second type d'interactions, désignées par le terme d'interactions d'intentions. Ces interactions sont révélées par une divergence entre les attentes exprimées vis-à-vis d'un service et le comportement observé de ce dernier. Cependant, les attentes considérées étaient la spécification même de chaque service. En conséquence, toute composition de services conduisait à une interaction.

En termes de détection d'interactions, l'approche est limitée par le fait que le modèle n'est pas exécutable. La seule analyse possible est par conséquent statique. Il n'est pas évident de déterminer ensuite si une situation d'interaction repérée par analyse est réalisable ou pas, i.e. si elle correspond à un état accessible ou non. La logique temporelle permet de disposer d'un moyen de spécifier des propriétés que les services doivent respecter, mais il manque le moyen de constater si celles-ci sont respectées par les spécifications. Ceci limite la portée de la notion d'interaction, telle qu'elle peut être explicitée dans ce modèle. En revanche, la formalisation de cette notion entre ici dans un cadre parfaitement maîtrisé.

Langages dédiés

La méthode de validation présentée par Hall (cf. section 8.1.5) est outillée et permet entre autres la détection de conflits symboliques et la vérification de propriétés de correction. Ces

dernières doivent être fournies par l'usager. Une interaction est caractérisée par une des trois situations suivantes :

- incohérence d'un gestionnaire d'événement (type I),
- mise en défaut d'une des propriétés de correction d'un service (type II),
- mise en défaut d'une propriété formulée sur la combinaison de services considérée (type III).

Les interactions de type I correspondent chez nous aux indicateurs (introduits en section 7.5.2). Les interactions de type II sont celles déterminées par l'oracle constitué des propriétés de services. Concernant les interactions de type III, elles sont chez nous révélées par les propriétés associées au système (cf. section 2.8).

La limite de cette approche est qu'il n'existe pas de processus d'intégration de services au service de base. Chacun des services redéfinit le comportement de ce dernier en totalité. En conséquence, un grand nombre d'interactions "parasites" sont introduites. En effet, chaque service a alors un domaine d'action (cf. section 7.5.2) couvrant la totalité du service de base, et l'interaction est inévitable.

Considérons par exemple les services *TeenLine* et *Terminating Call Screening* (TL et TCS, issus du concours FIW'98 et décrits en annexe D). Le service TL affecte le comportement du service de base à l'instant où l'usager décroche, en demandant à ce dernier de fournir un code d'identification ; le service TCS pour sa part agit sur le service de base au moment où l'usager compose. Les domaines d'action de chaque service sont donc en théorie disjoints et les services ne devraient pas interagir. Cependant, si on considère que chaque service redéfinit totalement le service de base, chaque service empiétera sur le domaine de l'autre : TCS voudra contrôler l'appel à l'instant où l'usager décroche, tandis que TL cherchera aussi à contrôler l'appel lorsque l'usager compose.

Pour éviter ce problème et réduire le nombre d'interactions parasites, l'auteur propose de gérer deux modèles en même temps, un modèle d'avant-plan (F) et un modèle d'arrière-plan (B). Pour chaque entrée, les réponses des deux modèles sont considérées. Si les réponses sont incompatibles, la partie conflictuelle de la réponse de B est ignorée, et le nouvel état est alors calculé en procédant à l'union de la proposition de F et de la partie non-ignorée de la proposition de B.

Le modèle d'arrière-plan est identique pour chaque service, il s'agit en fait d'un modèle du seul service de base. Le modèle d'avant-plan consiste alors à exprimer uniquement les *différences* introduites par le service supplémentaire dans le comportement du service de base. La composition de services s'effectue en conséquence uniquement sur les modèles d'avant-plan.

En termes de détection d'interactions, cette approche présente également le défaut de requérir du testeur qu'il analyse lui-même les spécifications pour en extraire les scénarios

permettant la vérification des propriétés (détection d'interactions de type II). Notre approche requiert également une analyse des spécifications, mais dans le seul but de définir un profil opérationnel, et non dans celui d'élaborer manuellement les cas de test utiles à la conduite de l'opération de validation.
De plus, lorsqu'un gestionnaire incohérent est détecté (interactions de type I), il faut procéder à l'analyse de son accessibilité, ce qui est pour l'instant déterminé manuellement. Concernant les interactions de type III, leur détection sollicite également le testeur de manière importante, puisqu'il lui incombe de définir le comportement attendu de la combinaison de services et de caractériser ce comportement par une propriété.

Lotos
La modélisation proposée par Logrippo (cf. section 8.1.2) permet de tester la conformité des traces d'observations d'une implémentation vis-a-vis des contraintes exprimées, et qui constituent la spécification. Néanmoins, les auteurs proposent une analyse comportementale des spécifications afin de déceler d'éventuelles interactions entre deux services. Pour ce faire, on fixe des buts que l'on sait atteints isolément par chacun des deux services, et on teste l'accessibilité simultanée des deux buts sur la spécification globale. L'apparition d'un blocage dans l'exécution démontrera la présence d'un conflit dans les buts. L'absence de blocage, en revanche, ne permet pas de conclure à l'absence d'interaction. Il est donc nécessaire d'exprimer des buts aussi variés que possible pour obtenir les meilleurs résultats.

L'originalité de ce travail est de ne baser la recherche d'interactions que sur la seule donnée des spécifications : en particulier, on ne distingue pas explicitement les attentes de l'usager de la spécification des services.

Détection "goal-oriented"
Le groupe LOTOS de l'université d'Ottawa a récemment proposé une méthode de détection d'interactions dite "goal-oriented" [60]. Cette méthode consiste en trois étapes :

1. établissement de propriétés censées être garanties par le système,

2. description de *buts* à atteindre, correspondant à la violation de ces propriétés,

3. analyse statique de l'ensemble des comportements possibles pour atteindre ces buts, grâce à un outil de simulation favorisant la satisfaction de buts.

Les buts ainsi décrits ont une certaine similitude avec le concept de schéma, en cela qu'ils désignent également une suite d'actions à observer, même si leur expressivité est moindre (pas de conditions d'intervalles, ...). La différence essentielle provient de l'utilisation qui en est faite : au lieu d'une génération dynamique, on fait ici appel à une analyse statique. Le modèle évalué doit donc être complètement analysable, ce qui interdit entre autres le choix d'un langage de modélisation autre que LOTOS.

8.2.2 Model-checking

Le modèle développé par Combes en SDL (cf. section 8.1.2) peut être évalué par model-checking des propriétés de logique temporelle linéaire (grâce à l'outil GEODE).

En termes de détection d'interactions, l'approche se heurte au problème de l'explosion combinatoire. De plus, le fait de manipuler deux formalismes différents rend plus délicate la procédure de traduction des propriétés pour les rendre utilisables par l'outil de vérification. En particulier, il est plus difficile de traduire les notions présentes dans le modèle pour les intégrer aux propriétés logiques.

Dans les travaux de Ryan et al. (cf. section 8.1.4), le système obtenu après intégration d'un service est ensuite confronté à un ensemble de propriétés déduites de la description informelle du service tel que vu par le client.

L'utilisation de l'outil SMV permet de réaliser cette confrontation. Pour des raisons d'explosion du nombre d'états, la modélisation est obligatoirement opérée à un haut niveau d'abstraction. Dans la perspective de détecter des interactions logiques, ceci n'est pas forcément un inconvénient, car elles seront plus visibles dans un tel modèle. Dans la pratique, la modélisation d'un système de télécommunication n'a pu se faire qu'au prix de nombreuses abstractions, parfois non-désirées. C'est pourquoi les auteurs envisagent d'instancier leur approche avec une autre méthode plus efficace, telle que celle basée sur l'outil SPIN [52]. Les approches de validation à base de test leur paraissent également prometteuses.

8.2.3 Test statistique

Dans [62], Kimbler défend l'idée d'un test statistique pour permettre de réduire le coût de l'opération de détection d'interactions. Ceci constitue une suite logique à ses travaux précédents, qui consistaient à modéliser des cas d'utilisation des services fournis par le système [64]. Les cas d'utilisation, dont l'usage est répandu dans l'ingéniérie orientée objet [56], décrivent de manière informelle les diverses manières dont le système peut être sollicité. De ces cas d'utilisation sont déduits des scénarios, dont l'analyse manuelle peut permettre de découvrir des situations d'interaction potentielle.

L'auteur remarque cependant qu'une telle approche favorise la détection des interactions correspondant aux usages les plus fréquents, mais ne prend pas du tout en compte l'importance des conséquences de chaque interaction. En particulier, cela ne permet pas d'évaluer le risque inhérent au déploiement d'un service donné. L'auteur propose donc comme future direction de recherche, de combiner cette dimension statistique avec une autre plus subjective qui pourrait prendre en compte la criticité des différents services (selon par exemple que le service est à très forte valeur ajoutée ou non, ou qu'il fasse appel à des ressources limitées...). Cette analyse rejoint le travail de Brinksma [18] qui, pour des raisons d'efficacité, propose de réduire la taille des jeux de test pour la conformité en complétant la spécification formelle par des informations précisant par exemple quelles fonctionnalités du système seront probablement complexes à mettre en œuvre, quelles sont les erreurs fréquemment rencontrées

dans des systèmes analogues, quelles erreurs potentielles risquent d'avoir des conséquences critiques, ... L'idée consiste à définir la *puissance* d'un jeu de test à partir de données sur la gravité des erreurs qu'il est susceptible de révéler.

Dans LUTESS, cette dimension critique peut être prise en compte dans la définition des comportements à favoriser. Rien n'oblige en effet à décrire de manière objective le comportement de l'environnement, et il est tout à fait possible d'exagérer certains aspects jugés plus importants.

Un dernier point intéressant de ces travaux est précisé dans [65] et consiste à séparer la *modélisation* des usages du système et le *profil* de ces derniers. Le modèle des usages permet de décrire qualitativement le comportement de l'environnement, tandis que le profil des usages quantifie ce comportement en termes de fréquence d'occurrence. Cette distinction peut s'appliquer au guidage par schéma proposé dans LUTESS : le schéma modélise les usages et peut être considéré indépendamment de la manière dont il sera utilisé pour le guidage.

8.3 Autres travaux apparentés

8.3.1 Test et simulation

L'utilisation de la simulation à des fins de validation d'un logiciel a fait l'objet de nombreux travaux, notamment dans le domaine des protocoles [58, 47, 82].

Groz [47] en particulier propose d'assimiler la simulation à une méthode de vérification et justifie cette proposition par les arguments suivants. Si la simulation présente le défaut d'être "borgne" (elle sait indiquer la présence d'une erreur, mais ne peut pas garantir l'absence d'erreur dans un système correct), il en est de même de toute preuve automatique : en effet, une telle approche requiert une modélisation préalable du système, et cette dernière n'est pas forcément exempte d'erreur. La confiance conférée par cette vérification n'est donc pas non plus absolue : le fait que le modèle soit prouvé correct ne garantit pas automatiquement que le système soit lui-même correct [1]. Il est donc parfaitement valable de considérer la simulation comme conférant un degré de confiance analogue à celui de la preuve. De plus, la simulation s'avère en général moins coûteuse à mettre en œuvre que les méthodes à base de preuve. Elle offre ainsi l'intérêt de permettre la manipulation de modèles plus complexes, voire du système réel.

La simulation purement aléatoire peut s'avérer mal adaptée dans certains cas, par exemple lorsque les situations potentiellement sources d'erreur ne peuvent se produire qu'à la suite d'enchaînements bien précis qu'une simulation aléatoire aura du mal à faire surgir. Pageot [82] propose par exemple pour la validation de protocoles une méthode de simulation guidée basée sur une procédure de séparation et évaluation.

Dans cette méthode, l'opérateur doit définir par une stratégie les zones qu'il souhaite voir atteintes ou évitées par la simulation. Cette stratégie décrit les états souhaités et leur attribue

1. En revanche, dans le cas de la preuve comme dans celui de la simulation, il est facile de déterminer, après qu'une erreur ait été révélée sur le modèle, s'il s'agit d'une erreur de modélisation ou d'une erreur également présente dans le système.

une signification : comportement attendu, interdit, intéressant, inconnu. Cette stratégie est ensuite combinée au protocole afin de produire un arbre de décision qui sera parcouru en tenant compte des informations qui le décorent. La méthode de parcours est un compromis entre aléatoire et exhaustivité, offrant ou non la possibilité de mémoriser des états ou de réaliser des retours dans la simulation. Une fonction d'évaluation doit permettre d'aider au choix des branches de l'arbre à explorer à un instant donné, mais cet aspect de la méthode n'a été, de l'aveu de l'auteur, que peu développé dans ce travail.

L'objectif des techniques de simulation guidée est de sélectionner un sous-domaine pertinent de l'espace d'états du système sous test afin d'y réaliser un travail de validation plus efficace et plus rapide. Néanmoins, le parcours du sous-domaine proposé ici fait largement appel à des techniques de parcours exhaustif, ce qui peut s'avérer rapidement coûteux. La technique atteint en particulier ses limites lorsque le nombre de choix possibles à un instant donné est trop important.

Ainsi qu'on vient de le voir, la simulation consiste à parcourir partiellement l'espace d'états d'un système fermé, de manière plus ou moins aléatoire. Notre conception de ce terme est légèrement différente, puisque nous ne simulons pas directement le système considéré. Nous réalisons un simulateur d'environnement qui va fournir au système les sollicitations qui vont permettre de parcourir son espace d'états de manière pertinente. Nous considérons donc le système comme une boîte noire dont nous n'observons que les réactions externes. En conséquence, la méthode de parcours et d'évaluation présentée ci-dessus n'est pas applicable telle quelle. Néanmoins, ces résultats seraient facilement adaptables et utilement mis en œuvre si nous rendions l'état du système observable et manipulable.

8.3.2 Test de conformité

Le test de conformité est une activité de validation d'implémentation très utilisée dans le développement de systèmes réactifs et de protocoles [53]. Elle consiste à assurer qu'une implémentation du système (logiciel et/ou matériel) est *conforme* aux exigences formulées par la spécification du système. La notion de conformité s'énonce généralement selon une relation formelle, dite d'implantation, entre le modèle de la spécification et celui de l'implémentation [84, 96].

Un *cas de test* correspond à une séquence particulière d'interactions entre le système et son environnement, menant explicitement à un verdict. Ce verdict indique si la séquence correspond à un comportement correct du système (succès) ou non (échec). Informellement, la conformité de l'implémentation par rapport à la spécification est assurée s'il n'existe pas de cas de test pouvant aboutir à un verdict d'échec.
Dans l'absolu, il n'est pas possible de tester exhaustivement l'ensemble des comportements. Des hypothèses fortes d'uniformité et de régularité peuvent dans certains cas permettre de généraliser les résultats obtenus sur un nombre réduit de cas de test et autorisent à considérer le test de conformité comme une activité de vérification, mais ces hypothèses ne sont que rarement vérifiables. Dans la pratique, il faut donc élaborer un ensemble significatif de cas de test de manière à garantir une couverture raisonnable et suffisante des comportements possibles du système.

Pour mener à bien la sélection de cet ensemble, une approche consiste à mettre en évidence des *objectifs de test* (*test purpose*). Les objectifs de test expriment chacun une condition de conformité particulière. Ils définissent en d'autres termes certaines propriétés sur le comportement externe du système. Ces objectifs de test permettent de dériver des cas de test caractéristiques des interactions entre le système et son environnement. L'utilisation des objectifs de test, sous forme d'automates ou de *message sequence chart* (MSC, [93]) est une pratique courante dans le domaine du test de protocoles.

Les objectifs de test peuvent être représentés par un automate dont les transitions sont étiquetées par des interactions entre le système et son environnement. Une séquence d'interactions entre le système et son environnement sera reconnue correcte si elle inclut une sous-séquence permettant de conduire l'automate dans un état accepteur. La notion d'objectif de test présente ainsi certaines similitudes avec celle de schéma comportemental, en ce sens que toutes deux définissent un comportement de l'environnement.

Dans [41], une approche est proposée pour la génération automatique de cas de test fondée sur des techniques de vérification "à-la-volée". Les cas de test sont dérivés automatiquement et sont basés sur le parcours *dynamique* du produit synchrone de deux automates, celui représentant l'objectif de test concerné et celui décrivant la spécification[2].

Ce travail a en commun avec les idées développées dans LUTESS le fait de proposer un test fonctionnel, à base de spécifications formelles et de proposer une analyse des résultats basée sur une notion d'oracle. La génération des cas de test partage avec la méthode de génération guidée par schéma la notion de test orienté : les fonctionnalités les plus importantes du système sous test sont les plus sollicités. Elle en diffère par le fait que les cas de test produits sont des scénarios qui imposent la génération de certaines données, au contraire des schémas qui ne font qu'influencer cette génération.

Cette approche diffère également de la nôtre de par le fait qu'elle nécessite une spécification complète du système sous test, tandis que la description d'un schéma peut se faire sans connaissance précise du comportement du système ; on peut donc tirer parti de toute spécification, aussi incomplète soit-elle. A priori, les deux approches concernent des phases différentes du cycle de développement : la nôtre intervient plus en amont, à un niveau où il n'est pas envisageable d'avoir une description complète. Les auteurs reconnaissent d'ailleurs que l'exigence d'une spécification formelle complète constitue un frein à l'utilisation de ces méthodes dans un contexte industriel, les spécifications disponibles étant fréquemment informelles.

2. En d'autres termes, cet automate-produit n'est jamais représenté de manière explicite.

8.3.3 Test fonctionnel de systèmes réactifs

L'outil et les méthodes de test fonctionnel présentés par Jagadeesan et al. [57] ont certaines ressemblances avec Lutess, notamment par le fait qu'ils sont également dédiés au test de programmes réactifs. Le langage sur lequel est basée l'approche est Esterel [14], un langage synchrone proche de LUSTRE.
Un générateur de données est construit à partir d'une description de l'environnement et de propriétés de sûreté exprimées dans une logique temporelle.

Les techniques employées sont plus limitées que celles utilisées dans LUTESS ; la sélection des données n'est en particulier orientée que vers la violation de propriétés de sûreté et l'aspect statistique de la distribution des données n'est pas considéré.

Par ailleurs, contrairement à LUTESS, le harnais de test construit par cet outil constitue une seule entité, non-modulaire, dont la compilation s'avère rapidement prohibitive (de l'avis même des concepteurs de l'outil). En conséquence, toute modification, même minime, des caractéristiques du test (changement d'oracle, mise à jour du programme, raffinement de la description de l'environnement, ...) est extrêmement coûteuse.

8.3.4 Test guidé de programmes réactifs

Dans [92], Halbwachs et al. décrivent un outil, Lurette, au fonctionnement analogue à celui de Lutess, permettant la génération dynamique de données de test à partir de contraintes décrivant l'environnement. Cet outil a l'avantage sur LUTESS de pouvoir prendre en compte des contraintes numériques linéaires. En revanche, il ne dispose que d'une seule méthode de génération, basée sur une hypothèse de distribution équiprobable des données.

Lurette permet de faire du test "épais", en tirant parti du fait que le logiciel sous test s'exécute dans le même espace-mémoire que l'outil. Ceci permet de stocker temporairement l'état du logiciel sous test, et il est alors possible de "défaire" une transition, pour revenir à l'état précédent du logiciel.
Cette possibilité est mise à profit dans Lurette en tirant successivement plusieurs données (en particulier, les valeurs aux bornes pour les variables numériques) pour un même état, de les soumettre au logiciel pour constater sa réaction à chacune, avant de choisir l'entrée qui fera réellement évoluer le logiciel.

Pour la mise en œuvre des schémas dans LUTESS, il serait possible de tirer parti de cette capacité en répertoriant les différents états atteints par le logiciel lorsqu'un schéma est complété. Ceci permettrait de mesurer la richesse de situations obtenue par un schéma, par exemple par comparaison à ce que pourrait donner un guidage non-permissif, basé sur un scénario.

Chapitre 9

Vers une utilisation de la méthode des schémas pour le test de propriétés non-invariantes

Nous avons montré dans les chapitres précédents l'intérêt que représentait LUTESS et la méthode de test basée sur les schémas pour le test de services de télécommunication. Une des limites de l'approche est qu'il n'est possible d'exprimer et de valider uniquement des propriétés invariantes, exprimables dans une logique temporelle du passé.

Ce chapitre décrit un travail exploratoire ayant pour objectif d'élargir le champ d'utilisation du guidage par schémas, en évaluant en particulier dans quelle mesure il peut aider à la validation de propriétés non-invariantes.
On verra en particulier que le recours à des propriétés non-invariantes peut être intéressant pour la validation de certains services de télécommunication, notamment ceux dont il n'est pas possible de déterminer *a priori* le temps de réaction.

9.1 Le problème de non-invariance

9.1.1 Notion de vivacité

À la différence d'une propriété de sûreté qui garantit que "quelque chose de mauvais ne se produit jamais", une propriété de vivacité annonce que "quelque chose de bien finit par arriver". La notion de vivacité a été introduite par L. Lamport [68] et formellement définie par Alpern et Schneider [5].
La particularité d'une propriété de vivacité est qu'elle considère qu'aucune trace d'exécution n'est irrémédiablement mauvaise. Si le "quelque chose de bien" attendu n'est pas observable sur la trace, il est possible que cela se produise dans le futur.

La notion de vivacité est dérivée de conditions d'équité, imposées sur le programme à tester. En général, on impose que toutes les actions exécutables infiniment souvent soient exécutées infiniment souvent. L'équité faible (ou justice) [88] garantit qu'une action exé-

cutable continûment n'est pas ignorée infiniment longtemps. L'équité forte garantit qu'une action exécutable infiniment souvent (sans l'être forcément de manière continue) sera exécutée infiniment souvent.

9.1.2 La non-invariance pour les services de télécommunication

Bien que la plupart des propriétés que l'on souhaite pouvoir exprimer sur des services soit de nature invariante, il peut être intéressant de permettre l'expression de propriétés non-invariantes, notamment pour faciliter la validation de services dont on ne peut déterminer *a priori* le délai de réponse.

Ainsi, une propriété commune à de nombreux services pourrait s'écrire élégamment sous la forme "l'activation du service doit mener *inévitablement* à son aboutissement". Une formulation de ce genre est particulièrement adaptée aux services de retour d'appel (ReturnCall, extrait du concours FIW'98, cf. annexe B) et de complétion d'appel (Call Completion to Busy Subscriber [35]), pour lesquels on ne peut prévoir le temps nécessaire à leur aboutissement.

Dans ces deux exemples, le service n'est pas immédiatement délivré mais est retardé le temps que certaines conditions soient satisfaites (e.g., retour d'un poste à l'état de repos). Le problème est qu'il n'est pas possible *a priori* de déterminer dans quel délai ces conditions pourront être observées (notamment de par le fait que les intervenants sont nombreux : n'importe quel usager peut appeler un poste et l'empêcher momentanément de retourner dans l'état de repos).

Une propriété exprimant un caractère inéluctable de ce type est dite de *fatalité* ou encore de *progression* [69]. Elles peuvent être décrites sur le modèle "*requête* $\leadsto$ *réponse*". Ce dernier se lit "*requête* `leads to` *réponse*" et s'interprète de la manière suivante :

"Toute occurrence de p est inévitablement suivie d'une occurrence de q dans le futur". Ceci correspond à la formule de logique temporelle "$\Box(p \Rightarrow \Diamond q)$" (qui se lit intuitivement "Toujours(p implique UnJour(q))") [72].

9.1.3 Limites du test

Les propriétés ci-dessus, comme toutes celle de vivacité, ne s'évaluent que sur des traces d'exécution *infinies* [81], puisque toute trace finie est supposée pouvoir être complétée de manière à mener à la satisfaction de la propriété.

Le test peut alors sembler inadapté, car limité à une évaluation sur des traces d'exécution *finies*. En d'autres termes, notre cadre de travail nous permet de manipuler des formules de logique temporelle du *passé*, alors qu'une propriété de vivacité fait référence au *futur*.

Il est dans certains cas possible de "retourner" une formule de vivacité (i.e., l'exprimer dans une logique du passé) lorsque sa conclusion (la réponse) peut être bornée, soit par une condition qui lui est postérieure (retour dans un état initial, par exemple) soit grâce à un délai qui fixe dans l'absolu une limite temporelle au temps de réponse.

C'est ce que propose Mermet dans sa thèse [73], en définissant un *variant* qui décroît lorsque certaines opérations, définies comme *progressistes*, sont exécutées. La propriété doit être satisfaite avant que le variant ne parvienne à 0. Une hypothèse d'équité forte est imposée sur les opérations progressistes, de manière à ce qu'au moins une d'entre elles soit exécutable infiniment souvent.

Nous ne nous intéresserons pas ici à ce type particulier de propriétés dans la mesure où celles-ci peuvent être *in fine* assimilées à de la sûreté.

L'évaluation d'une propriété de vivacité sur une trace finie ne peut être réalisée avec certitude. Deux cas peuvent se présenter :

- soit la conclusion attendue s'est produite et permet une évaluation de la propriété à *vrai*,
- soit au contraire on est toujours dans l'expectative à la fin de la trace, et rien ne permet de savoir si une trace plus longue aurait pu aboutir ou non à une satisfaction de la propriété.

Le problème se pose avec encore plus d'acuité dans le cas particulier des propriétés de fatalité, qui possèdent un caractère permanent. En effet, le fait d'observer un nombre *fini* de fois la réponse à une requête ne garantit pas la validité absolue de la propriété : rien ne dit qu'une requête ultérieure ne restera pas sans réponse.

9.2 Proposition

Puisqu'il n'est pas possible de procéder à une évaluation de la propriété sur une trace finie, nous nous proposons de procéder par analyse comparée. Considérant un certain nombre d'informations collectées par ailleurs, nous pouvons envisager de déterminer un verdict provisoire, auquel serait associé un *indice de confiance*.
Un verdict provisoire permet d'indiquer si, au vu de la trace considérée, la propriété semble vraie ou fausse. L'indice de confiance précise pour sa part la probabilité que le verdict soit valide.

Les informations utiles à la détermination de ce verdict et de son indice peuvent être des données statistiques issues d'autres tests similaires ou des prévisions de comportement réalisées à partir des spécifications. Elles peuvent également être déduites de la trace considérée : données sur les occurrences des requêtes, temps séparant la fin de la trace de la première requête non-satisfaite, ...

Il faut insister sur le fait que les verdicts que nous manipulons ne sont que provisoires. Ils se basent sur une trace considérée et ne peuvent permettre une conclusion définitive. En effet, si l'on considère une trace faisant état de plusieurs requêtes, toutes ayant reçu une réponse, rien ne permet d'affirmer que dans ce qui aurait été le *futur* de cette trace, il n'y

aura pas une requête qui demeurera sans réponse. Inversement, le fait d'observer sur une trace finie une requête sans réponse ne permet pas de conclure que cette dernière n'arrivera jamais.

9.2.1 L'indice de confiance

Nous commencerons dans cette section par décrire la forme des données pertinentes que nous pouvons envisager d'utiliser pour calculer l'indice de confiance, puis nous discuterons de la manière d'obtenir ces données.

Cas d'une propriété simple

Nous ne considérons tout d'abord que la seule propriété "requete $\Rightarrow \Box$ reponse". Supposons l'existence d'une loi de distribution dont la densité de probabilité indique à chaque instant les chances d'observer "reponse". Par souci de simplification, nous considérerons cette distribution continue, même si le temps est pour nous discrétisé. Nous supposerons également que l'origine des temps est l'instant où est émise la requête.

La densité est considérée nulle sur $]\Leftrightarrow\infty, 0[$, car pour être prise en compte, la réponse doit succéder à la requête, et non la précéder. L'intégration de cette densité sur $[0, +\infty[$ vaut 1, ce qui permet de rester cohérent avec la philosophie de la notion de vivacité qui affirme que rien n'est jamais perdu.
La fonction de répartition $F(x) = \int_0^x f(t)dt$ décrit l'évolution des chances qu'a la propriété d'être valide à l'instant x. En d'autres termes, les chances pour une propriété d'être valide alors que sa conclusion n'a pas été observée à un instant x sont égales à $1 \Leftrightarrow F(x) = \int_t^{+\infty} f(t)dt$.

S'il est possible de construire une telle courbe, l'indice de confiance pourra alors être lu directement comme le résultat de cette intégration.

Extension aux propriétés de fatalité

Nous faisons l'hypothèse que la situation n'est guère différente dans le cas d'une propriété de fatalité. Tant qu'une requête n'a pas reçu de réponse, la confiance continue de décroître : le fait d'observer une nouvelle requête n'influence pas le phénomène. Nous validerons cette hypothèse dans l'expérience décrite en section 9.3.

Construction de la courbe

La construction de la courbe ci-dessus peut reposer sur des estimations (ou des espoirs) basées sur le contenu du cahier des charges. On pourra par exemple estimer le temps moyen acceptable par l'usager pour qu'un certain service réponde à une requête précise. Ce temps, que l'on pourrait repérer sous le terme de Mean Time To Service (MTTS), par analogie avec le Mean Time To Failure (MTTF) fréquemment utilisé en fiabilité, correspondrait au maximum de cette distribution. D'autres données concernant l'écart-type, ... pourraient alors

suffire à brosser le profil de la courbe. De telles données sont obtenues à partir des mêmes informations permettant de construire le profil opérationnel de l'environnement : on sait par exemple que le système de télécommunication sera accédé par tant de personnes à la fois, qui effectueront en moyenne tant de requêtes à telle fréquence...

On peut également, lors du test lui-même, utiliser les données au fur et à mesure de leur apparition pour corriger la courbe de distribution. On sera en particulier intéressé par les cas où la propriété trouve sa conclusion. On affinera ainsi la distribution estimée, en lui ajoutant des informations issues de l'expérimentation. Cette opération revient à augmenter dynamiquement la taille de l'échantillon utilisé pour construire la courbe.

Le plus simple est évidemment de partir d'une courbe déjà construite dans des conditions analogues. Ainsi, pour le cas des services téléphoniques, on peut supposer avoir validé la propriété pour le service isolé. De l'expérience ayant permis cette validation, on aura tiré des statistiques indiquant la manière dont le service répond. On peut alors utiliser ces données lors de la composition de ce service avec d'autres, puisqu'idéalement le comportement du service ne sera pas altéré par l'existence des autres services existants.

C'est ce que nous nous proposons de réaliser dans l'expérience décrite en 9.3.

9.2.2 Intervention du guidage

Dans le cas particulier d'une propriété de fatalité, il convient de distinguer les sous-cas suivants :

1. Aucune requête n'a été émise. La trace satisfait trivialement la propriété, mais ne donne aucune indication sur la validité de cette dernière dans l'absolu.

2. Un certain nombre de requêtes ont été formulées. Le service correspondant à la propriété a donc été sollicité, et l'analyse de la trace pourra alors donner des résultats pertinents.

Pour donner une plus grande pertinence à une trace, il peut donc s'avérer utile de guider la génération de données de test de manière à observer un nombre significatif de requêtes.

Le même commentaire s'applique également à toute propriété de vivacité, dès lors qu'elle peut s'écrire comme une implication de la forme "f(entrees) $\Rightarrow$ g(sorties)" où f et g sont des fonctions logiques pouvant impliquer des opérateurs temporels.

L'expérience qui suit permettra de dégager un certain nombre de conclusions quant à la "bonne" manière de guider.

9.3 Expérimentation

Nous relatons dans cette section un cas d'école, adapté à partir des cas d'étude réalisés par ailleurs. Cette expérience vise à évaluer si une propriété de service de type "fatalité" peut

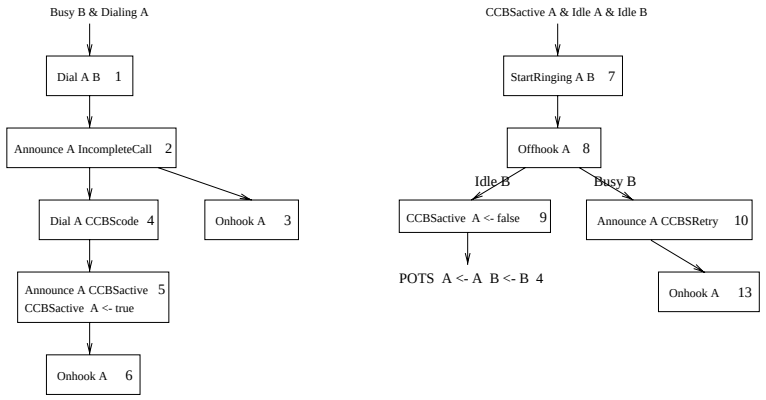


FIG. 9.1 – *Diagrammes de Chisel pour le service CCBS*

être satisfaite lorsque le service est seul disponible, mais ne plus l'être lorsque ce dernier est composé avec d'autres. Sous ces conditions, on montrera que l'on peut conclure avec une relative confiance sur la mise en défaut de la propriété, quand bien même celle-ci n'est pas explicite.

9.3.1 Énoncé

Considérons deux services de retour d'appel sur signal occupé, l'un souscrit par l'appelant (Call completion to Busy Subscriber, CCBS), l'autre par l'appelé (Return Call on Busy, RCB). Le service CCBS provient de l'étude de cas proposée par le CNET. Le service RCB est une adaptation du service *Return Call* présenté dans le concours FIW'98. Il en diffère principalement par le fait qu'il s'active automatiquement lorsque son souscripteur reçoit un appel en étant déjà occupé, tandis que le service initial s'activait à la demande du souscripteur, après non-réponse de celui-ci à un appel.

Les deux services ont des conditions d'activation similaires : l'appelé doit être occupé lorsque l'appelant effectue sa demande de connexion. La seule différence tient dans le fait que RCB est activé automatiquement, sans intervention de l'usager souscripteur, tandis que CCBS requiert de son souscripteur qu'il compose un code d'activation spécifique lorsque la situation décrite ci-dessus survient.

Après activation, les services sont mis en veille jusqu'à apparition de la condition suivante "Idle(appelant) & Idle(appelé)" (condition dite de *déclenchement*). Chacun fait alors évoluer son souscripteur vers l'état "Ringing" et attend que celui-ci décroche pour appeler le correspondant désigné, à condition que celui-ci soit toujours dans l'état Idle.

Les figures 9.1 et 9.2 décrivent les deux services sous forme de diagrammes de Chisel [3] [1].

La propriété à tester concerne le service CCBS, et garantit que chaque requête de service est inévitablement suivie d'une satisfaction. Elle est basée sur l'observation de la condition

1. Chisel est le formalisme de description de services sous forme d'automates utilisé dans le cadre du concours FIW'98 (cf. annexe B).

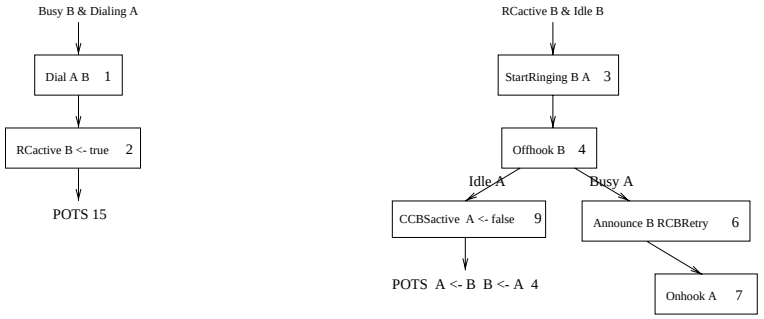


FIG. 9.2 – *Diagrammes de Chisel pour le service RCB*

événementielle "Off A and StartAudibleRinging A". Cette condition est discriminante car dans le cadre du service de base, le fait de décrocher mène soit à "DialTone A", soit à "Talking A".

$$\mathcal{P} : \mathit{Announce\ A\ IncompleteCall} \rightsquigarrow (\mathit{Off\ A\ and\ StartAudibleRinging\ A})$$

9.3.2 Mode opératoire

L'expérimentation comportait 3 étapes :

1. Collecter des données concernant la satisfaction de $\mathcal{P}$ dans un système où CCBS était le seul service supplémentaire. Les données collectées ont été :
   - le nombre de fois où la propriété a été satisfaite,
   - le nombre de requêtes émises,
   - le temps séparant chaque requête de sa conclusion.
2. Réitérer le même processus pour un système où CCBS cohabite avec RCB. Le même ensemble de données a été collecté.
3. Analyser et comparer les deux jeux de données et tenter d'en tirer des conclusions.

Afin d'évaluer l'apport de l'utilisation du guidage par schéma dans le test de propriétés non-invariantes, nous avons procédé à la génération de plusieurs jeux de test, en variant la forme du guidage, ainsi que sa force.

9.3.3 Résultats et analyses

Il est important de noter que l'étude décrite ici a un caractère totalement exploratoire. Nous n'avons pas cherché à analyser les données en détail, mais plus à nous faire une idée de la validité de notre approche. Ainsi l'analyse a été pour l'essentiel manuelle, soit directement à partir des données brutes collectées, soit après visualisation par le logiciel de tracé *Gnuplot*.

L'analyse devait permettre de répondre aux questions suivantes :

- Quel est l'intérêt du guidage, et comment guider pertinemment ?
- Enfin, cette façon de faire permet-elle de révéler des problèmes ?

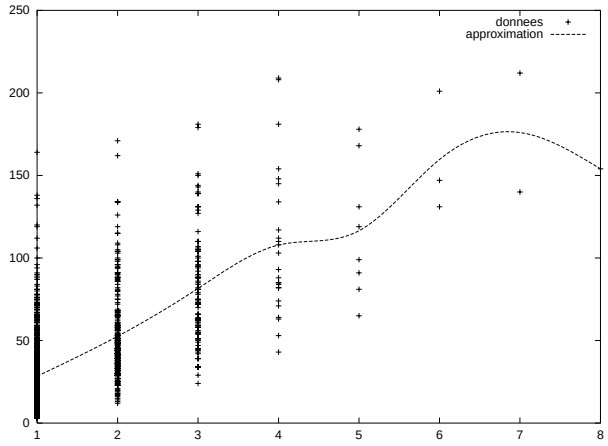


FIG. 9.3 – *Relation* Nombre de requêtes simultanées / temps de réponse *pour CCBS seul*

	nb requêtes		
	totales	simultanées	nb réponses
Sans guidage	814	4	670
Guidage faible	1105	5	790
Guidage moyen	1515	9	817
Guidage fort	1786	15	797

TAB. 9.1 – *Influence du poids du guidage*

Influence du guidage

La première étape de notre analyse a consisté à vérifier l'hypothèse selon laquelle le nombre de requêtes en suspens n'influence pas l'apparition de la réponse. En effet, on soupçonne qu'il est inutile de favoriser l'émission d'une requête lorsqu'il en existe déjà une en attente (i.e, encore sans réponse).

La figure 9.3 (en abscisse, nombre de requêtes simultanées, en ordonnée, temps de réponse) permet de constater que l'approximation peut être considérée comme une droite, si l'on exclut les valeurs trop peu significatives. Si le nombre de requêtes avait une influence sur le temps de réponse, l'approximation devrait montrer un fléchissement important, ce qui n'est pas le cas.

Pour confirmer ce résultat, le tableau 9.1 compare, pour différents niveaux de guidage, les résultats obtenus en termes de nombre de requêtes, nombre de requêtes simultanées et nombre de réponses du service. On peut au vu de ces chiffres constater que le fait d'émettre plusieurs requêtes simultanément n'a pas une influence directe sur le temps de réponse.

La courbe de la figure 9.4 montre bien le retard éventuellement introduit par le guidage lorsque celui-ci est réalisé de manière irréfléchie : multiplier le nombre des requêtes n'apporte rien en termes de réduction du temps de réponse, et au contraire, peut retarder le moment

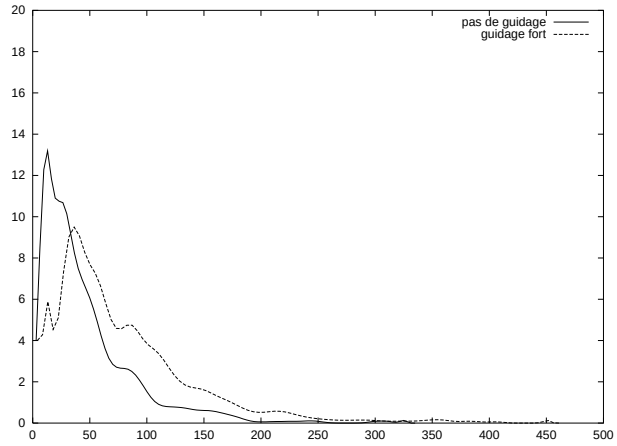


FIG. 9.4 – *Répartition des réponses en fonction du temps les séparant des requêtes*

	Nombre de requêtes	Nombre de réponses	Temps moyen	Requêtes simultanées 1	2	3+
Sans guidage	88	77	40.7	68	8	1
Guidage de référence	171	89	63.3	45	25	19
Guidage modifié	187	147	45	118	23	6

TAB. 9.2 – *Résultats des tests pour divers guides*

où les conditions nécessaires à la production de la réponse seront réunies.

Augmenter la pertinence du guidage

Pour pallier le défaut énoncé plus haut, une solution peut consister à guider plus intelligemment, de manière à n'obtenir qu'une seule requête à la fois. Pour cela, il suffit de compléter la première condition du schéma avec une formulation du genre : between(Réponse,Requête).

Le tableau 9.2 offre une comparaison des résultats obtenus pour 10 000 pas de tests sans guide, avec le guide d'origine décrit précédemment, et avec la modification ci-dessus, conçue pour obtenir des requêtes "exclusives".

La remarque principale que l'on peut faire au vu de ces résultats concerne l'augmentation notable du nombre de réponses du service, qui double presque avec le guidage modifié, tandis que le temps de réponse n'est pas significativement augmenté. On peut également observer la corrélation apparente entre l'accroissement du temps de réponse et le nombre de requêtes simultanées en comparant le guidage de référence au guidage modifié.

Si on compare le nombre total de réponses au nombre total de requêtes, on a une idée du pourcentage de requêtes "utiles", i.e. celles qui activent effectivement le service. Ce taux varie de 52% à 87%. Cela confirme bien l'idée selon laquelle il est nécessaire de guider intelligemment pour obtenir de meilleurs résultats. On notera qu'avec un taux de 79%, le guide

modifié fournit des données presque aussi utiles que sans guidage. En revanche, le volume de données qu'il permet d'obtenir est significativement plus important. C'est donc un bon compromis entre l'absence de guidage qui fournit essentiellement des données utiles, mais peu, et le guidage de référence dont les données sont plus nombreuses mais moins utiles en moyenne.

Il est indispensable de noter que, dans cet exemple précis, l'influence du guidage est limité par le fait que la production de requêtes est facilement obtenue par une génération aléatoire. Il est évident qu'une expérience comparable sur une propriété reposant sur un modèle de requête plus complexe aurait montré de manière plus flagrante l'intérêt du guidage.

Comparaison entre service isolé et services composés

Ce paragraphe présente les résultats obtenus en sollicitant le service CCBS dans deux configurations : une première où il était le seul service supplémentaire disponible, et une deuxième où il était composé avec le service RCB.

La courbe 9.4 est issue de l'analyse d'une génération sans guidage pour le seul service CCBS. Elle a été obtenue en classant chaque réponse du service en fonction du temps la séparant de la requête initiale (celle ayant activé cette instance du service). Elle permet de faire apparaître un MTTS d'approximativement 15 cycles, et montre bien comme on l'espérait une décroissance relativement rapide en fonction du temps. On peut reconnaître dans cette distribution une allure gaussienne, décalée par rapport à l'origine.

La configuration issue de la composition des deux services n'a quant à elle pas permis d'obtenir la moindre satisfaction de la propriété CCBS, que ce soit avec ou sans guidage, sur 100.000 pas de test. Avec guidage, on a observé 996 requêtes, dont la première au bout de 98 pas ; sans guidage, il a fallu attendre 256 pas pour voir la première des 408 requêtes.

Au vu de l'indice de confiance calculé pour le service CCBS seul, ceci permet d'affirmer avec une relativement bonne confiance que la propriété n'est pas satisfaite par la configuration. On peut donc raisonnablement conclure que la composition des deux services génère une interaction empêchant au service CCBS de s'exécuter correctement (ce que l'analyse manuelle de la spécification des services permet effectivement de confirmer).

9.4 Généralisation et conclusion

Il est acquis qu'une propriété de vivacité ne peut être évaluée de manière absolue et certaine sur une trace finie. Nous avons néanmoins fourni dans ce chapitre quelques éléments objectifs pouvant permettre au testeur de conclure avec une confiance relative sur ce point.

Nous avons montré sur un cas d'étude comment déterminer la confiance en la validité d'une propriété de fatalité dans le cadre du test de services téléphoniques.

La procédure consistant à mesurer statistiquement les délais de réponse associés à divers services pris indépendamment, puis à comparer ces délais avec ceux observés lorsque les services sont composés, peut permettre effectivement de découvrir des interactions, se manifestant par l'absence de réponse d'un service particulier à une requête. L'utilisation du guidage par schéma a montré son utilité dans l'objectif de rendre pertinentes les séquences de test.

Les résultats issus de ce travail peuvent également être mis à profit pour ce qui concerne l'arrêt du test. Un des problèmes que rencontre en effet notre approche est l'absence de mécanisme permettant de savoir quand considérer le test comme étant suffisamment significatif. Le calcul d'un indice de confiance, tel que nous le proposons ici peut être un moyen de déterminer un critère d'arrêt pertinent vis-à-vis d'une propriété de vivacité.

Si l'on suppose valide l'hypothèse selon laquelle la probabilité d'occurrence d'une réponse à une requête suit une distribution gaussienne au cours du temps, un critère d'arrêt pourrait consister à considérer qu'il est inutile de continuer le test lorsque l'indice de confiance est inférieur à une valeur donnée (0,01%, par exemple).

Chapitre 10

Conclusions et perspectives

10.1 Conclusion

Le travail exposé dans ce document s'inscrit dans le domaine de la validation de systèmes réactifs synchrones. Nous nous intéressons plus particulièrement aux systèmes dont le comportement est complexe et peut s'étendre sur de longs intervalles de temps. Nous nous sommes plus précisément attaché à étudier la validation de spécifications de services de télécommunication.

La méthode de validation que nous utilisons se base sur trois éléments : le système sous test $\mathcal{S}$, une description de l'environnement $\mathcal{E}$ dans lequel doit s'exécuter $\mathcal{S}$, et une description des propriétés $\mathcal{P}$ que le système doit satisfaire. Elle consiste principalement à simuler l'environnement de manière à fournir des données de test au système sous test. Un oracle observe les réactions du système et évalue si elles satisfont les propriétés annoncées $\mathcal{P}$.
La production de données est dynamique : chaque donnée est générée en fonction des réponses antérieures du système sous test. Cette production est aléatoire mais respecte un certain nombre de contraintes imposées par $\mathcal{E}$.
Notre contribution a consisté à proposer, implémenter et valider un moyen de guider la simulation de l'environnement, de manière à la rendre non seulement plausible, mais réaliste. Nous avons à cette fin élaboré une méthode de test basée sur la notion de schémas comportementaux. Chaque schéma décrit une classe de comportement typique de l'environnement ; la méthode de génération prend en compte ces schémas de manière à privilégier l'apparition des comportements qu'ils décrivent.

Nous avons mis en application cette méthode pour valider des spécifications de services de télécommunication sur trois exemples de taille conséquente. Ces expérimentations nous ont permis de constater l'efficacité de la méthode et nous ont permis d'ébaucher une méthodologie de validation de spécifications.

Adéquation de la validation par le test
Ainsi qu'on l'a mentionné plus tôt (cf. section 2.5), le test est par nature plus adapté à une

activité de recherche de défaut, ce qui est exactement le cas lorsque l'on cherche à détecter des interactions.
De plus, la possibilité d'observer les traces résultant du test permet de s'assurer *de visu* que le comportement du système et de son environnement est sensé. Par ailleurs, une expérience complémentaire menée sur la même étude de cas avec des outils de vérification a confirmé que, dans ce domaine, le test permet de valider des programmes de taille bien plus importante [27].

Cette adéquation entre la nature du problème abordé et l'activité de test est probablement un des atouts essentiels de notre méthode.

Pertinence du guidage par schéma
L'utilisation des schémas à fin de guidage s'est montré extrêmement bénéfique dans le cadre du premier concours de détection d'interactions (cf. chapitre 7). De manière plus générale, les services fournis par un système de télécommunication peuvent être relativement complexes à solliciter. Dès lors, il n'est pas possible de procéder à un test purement aléatoire si l'on souhaite des résultats pertinents.
Le guidage par schéma s'est révélé particulièrement adapté, permettant de décrire de manière simple et efficace les comportements les plus fréquents et pertinents des usagers. Nous comparons dans la suite cette approche aux méthodes pré-existantes de LUTESS, et nous discutons de sa généralité.

Intérêt des schémas par rapport aux probabilités conditionnelles
La différence essentielle entre les deux méthodes provient du fait que la méthode à base de probabilités conditionnelles s'attache à caractériser des *variables*, tandis qu'une description constituée de schémas s'intéresse aux *vecteurs* d'entrée.
De plus, la manipulation de vecteurs est facilitée par le fait qu'un schéma rend explicite l'aspect temporel de la description. L'utilisation de schémas se rapproche plus d'un contexte de test à base de scénarios décrivant des cas de test (cf. section 8.3.2).

Selon les cas, il sera plus adapté de choisir l'une ou l'autre méthode. Les schémas seront donc préférés lorsque l'on souhaite décrire explicitement un comportement qui se déroule dans le temps, tandis que les probabilités conditionnelles seront vraisemblablement plus appropriées pour décrire les probabilités d'occurrence d'un ensemble de variables n'ayant qu'une faible relation entre elles.

Concernant plus particulièrement la validation de services, un autre intérêt des schémas est leur compositionnalité. Lorsque l'on souhaite confronter deux services, on obtient très facilement une description de leur environnement en mettant en concurrence les schémas associés à chaque service. La même opération serait beaucoup moins aisée à réaliser si les profils d'utilisation des services sont décrits par des probabilités conditionnelles.

Intérêt des schémas par rapport au guidage par propriété
L'utilisation d'un guidage par schémas peut se rapprocher par plusieurs aspects du guidage

à base de propriétés. On peut notamment avec cette dernière méthode obtenir une description sensiblement équivalente à un schéma unique, en utilisant la relation entre schéma et propriété définie en section 4.7.3. Chaque instant significatif (i.e., chaque instant soumis à condition) d'un schéma de rang n étant caractérisé par une propriété P_i, on peut construire une propriété de guidage en procédant à la mise en conjonction de la négation des propriétés caractéristiques du schéma :

$$P = \bigwedge_{i=1..n} \neg P_i$$

Cependant, le guidage par schéma est plus efficace, en termes de coût de calcul, ce qui s'explique par le fait que les BDDs utilisés sont moins complexes. Les schémas sont par ailleurs plus expressifs et donnent lieu à des résultats de test plus riches et plus variés, grâce à l'aspect probabiliste qu'ils contiennent (cf. section 4.2).

Généralité de l'approche de test guidé par schéma
Nous avons cherché à montrer l'intérêt des schémas pour le test d'un type d'applications. Nous nous sommes posés la question de la généralité de l'approche, et à cette fin, nous avons également étudié un autre exemple, le modèle d'un contrôleur de cabine d'ascenseur. Nous avons éprouvé certaines difficultés à appliquer la méthode à cet exemple, et nous avons notamment été obligés de renforcer les contraintes d'environnement.

En effet, dans cet exemple, les usagers n'étaient pas identifiés en tant que tels : ils n'existaient que par le biais des capteurs (boutons d'appel...) du système. Il n'était donc pas évident d'isoler un comportement particulier d'un usager, et surtout, il s'est avéré très difficile de le favoriser par rapport aux autres. Nous avons donc été amené à concrétiser l'existence des usagers en les modélisant explicitement.

Cette étude succincte a montré que les schémas ne sont pas forcément appropriés à n'importe quel contexte. Leur fonction première est de décrire les comportements-types de l'environnement ; leur utilisation n'est valable que lorsque ce dernier est constitué d'entités parfaitement discernables.

10.2 Perspectives concernant la validation de services

10.2.1 À propos de modélisation

Dans la méthodologie de modélisation décrite ici, nous considérons un service comme une unité élémentaire. Nous faisons donc abstraction des différentes fonctions dont il est composé, ce qui permet de ne pas contraindre inutilement sa description.

Une autre possibilité aurait consisté à considérer un niveau de détail plus fin, et à rendre visibles les fonctions élémentaires (éventuellement paramétrables) qui constituent chaque service. L'intérêt d'une telle approche est la suivante. Les "briques" de construction de services étant répertoriées, on peut, facilement et avec un faible risque d'erreur, exprimer sur

chacune une propriété caractéristique. La propriété du service constitué de plusieurs briques sera alors automatiquement construite à partir des propriétés de ces dernières.

Par ailleurs, répertorier les briques peut avoir un autre intérêt. On peut analyser celles-ci pour déterminer le ou les aspects d'un appel qu'elles affectent, puis regrouper en fonction de ce critère les briques en diverses classes. Lorsque deux services différents utilisent une brique d'une même classe, on pourra soupçonner un risque d'interaction entre les deux. En revanche, lorsque deux services n'ont pas de briques de classe commune, le risque d'interaction pourra être considéré comme faible, voire nul. Cette proposition demande bien évidemment à être étudiée plus avant afin de pouvoir être validée.

10.2.2 À propos de la représentation de l'environnement

Un des obstacles à l'utilisation intensive de l'outil LUTESS pour la simulation d'environnements très complexes reste le temps de construction des BDDs. L'utilisation de schémas complexes et concurrents accentue encore ce problème.

Afin de démultiplier la complexité, il serait intéressant de pouvoir disposer de plusieurs générateurs, chacun gérant le comportement d'un composant de l'environnement (dans le cas des téléphones, un générateur par usager, ou par type d'usager, par exemple). Cette voie semble prometteuse, surtout lorsque les composants sont relativement indépendants.

Par exemple, l'environnement de système de télécommunication "décortiqué" au chapitre 6 contient d'une part les usagers, d'autre part un système de facturation, qui sont deux entités sans aucun lien.
De plus, on pourrait tout à fait considérer que chaque usager agit indépendamment des autres, en fonction des seules informations qui lui sont transmises (notamment par le biais des sonneries). S'il doit y avoir coopération[1], il faudra étudier comment synchroniser et faire communiquer les différents générateurs, et assurer la cohérence des comportements produits.

10.2.3 À propos de la méthodologie de validation

Prise en compte des interactions complexes, apparaissant entre plus de deux services

La grande majorité des interactions se manifeste déjà lorsque les services sont confrontés par paires. Il n'est pas possible d'exclure cependant toute possibilité d'interaction ne se manifestant qu'en présence concurrente d'un plus grand nombre de services.

Un exemple d'interaction complexe implique par exemple les trois services usuels "Liste rouge", "Facture détaillée" et "Rappel du dernier appelant" (en considérant que ce dernier service ne permet pas de connaître l'identité de l'appelant, mais autorise juste à retourner son appel). Un appel provenant d'un usager A sur liste rouge pourra être retourné par un

1. C'est le cas pour les téléphones à cause de la contrainte simplificatrice suivante : un seul événement à chaque instant est produit par l'environnement.

usager B ayant souscrit à la fois au service de retour d'appel, et au service de facturation détaillée. Le problème qui se pose alors est de savoir si cet appel doit faire apparaître sur la facture de B l'appel vers A, alors que celui-ci a souhaité ne pas rendre public son numéro.

Les trois services sont cohérents deux à deux et n'entrent en conflit que lorsque tous trois sont exécutés. Les exemples d'interactions complexes sont encore très peu nombreux, mais rien ne permet de penser qu'il n'existe pas d'interactions impliquant plus de trois services. Personne n'a encore suggéré que le problème d'interaction soit NP-complet, mais il est en tout cas de complexité au moins égale à $\mathcal{O}(n^3)$, n étant le nombre de services.

Le modèle que nous proposons est adapté à l'intégration d'un nombre quelconque de services. Nous ne discuterons donc pas de ce point dans cette section, mais nous nous consacrerons aux stratégies qui peuvent prévaloir pour valider un ensemble de services concurrents.

La méthode la plus immédiate consiste à chercher à valider la composition de l'ensemble des services, et à analyser ensuite les résultats de manière à déterminer quels sous-ensembles de services sont responsables de chacune des éventuelles interactions détectées. Malheureusement, cette approche montre rapidement ses limites, en raison notamment de la profondeur d'analyse qu'elle requiert.

Pour automatiser partiellement cette analyse, on peut mettre en œuvre la technique dite du "delta-debugging" afin de déterminer, pour une interaction donnée, quel est le sous-ensemble minimal de services responsable du problème [105]. Cette technique, empruntée au domaine de la mise au point de programme, semble en effet bien adaptée à cette fin. Basée sur une recherche dichotomique, elle consiste à partitionner un ensemble de services $\mathcal{C}$ montrant une interaction en deux sous-ensembles c_1 et c_2 et à procéder au test de chacun. Si l'interaction se manifeste dans un des deux sous-ensembles, la recherche devra se concentrer sur celui-ci ; sinon, l'interaction est forcément issue de la combinaison de certains services inclus de c_1 et d'autres contenus dans c_2. Dans ce cas, il s'agit de partitionner à nouveau l'un des deux sous-ensembles, par exemple c_1, et de tester les deux sous-ensembles produits par cette dernière partition, composés chacun avec c_2. Par approximations successives, on peut ainsi parvenir à déterminer l'ensemble minimal responsable d'une interaction particulière.
La bonne mise en œuvre de cette technique est garantie par le fait que notre opération de composition est associative : rien n'empêche, pour confronter un ensemble de services {a,b,c,d}, de composer d'une part a et b, de l'autre c et d, de mener à bien la validation de chacune des compositions puis d'effectuer ensuite la composition de celles-ci et sa validation.

Généralisation des cas d'interaction

Lorsqu'une séquence de test aboutit à la révélation d'un problème que l'expert aura analysé comme étant réellement une interaction, il est également nécessaire de procéder à une seconde analyse. Celle-ci doit viser à généraliser la description du cas d'interaction : à partir du scénario concret observé, on veut pouvoir déduire une description symbolique permettant de regrouper tous les conflits équivalents. Cela passe par une abstraction de

l'identité des usagers concernés, et par une minimisation de la séquence, de manière à ne conserver que les événements pertinents. Ce travail, pour l'instant complètement manuel, mériterait d'être partiellement automatisé.

10.3 Perspectives de l'environnement de test LUTESS

Les méthodes de test fonctionnel développées dans le cadre de l'environnement de test LUTESS couvrent à l'heure actuelle un spectre intéressant de types de systèmes réactifs. Cependant, nous ne fournissons à l'heure actuelle que peu d'information quant à l'efficacité du test en termes de couverture des comportements du système. En particulier, nous ne savons pas déterminer de critère d'arrêt du test raisonnable.

Si l'on demeure dans le strict cadre de la validation fonctionnelle, en considérant le système sous test comme une boîte noire, il ne sera de toute manière pas possible de mener une telle analyse. En revanche, il peut être pertinent de considérer la description de l'environnement et son profil opérationnel, donné sous forme de schémas comportementaux et d'analyser le rapport entre les comportements réellement produits et cette description. En réutilisant et précisant la notion de proximité introduite en section 4.2, on pourra par exemple décider qu'il est nécessaire de couvrir tous les comportements relativement proches d'un schéma pour garantir une couverture convenable et assurer une certaine fiabilité au système. Cette idée pourrait être le point de départ de futurs travaux.

Par ailleurs, LUTESS a acquis désormais une certaine maturité qu'il pourrait être intéressant d'exploiter de manière pratique en transférant cette technologies vers l'industrie. Une première ébauche de transfert a déjà été réalisée dans le courant de l'été 1999, en intégrant l'outil à un environnement de maquettage et de simulation de services utilisé en interne par France Télécom R&D.

Bibliographie

[1] *Feature Interactions in Telecommunications Systems VI.* – IOS Press, mai 2000.

[2] Aggoun I. et Combes P. – Observers in the SCE and the SEE to detect and resolve services interactions. *In : Feature Interactions in Telecommunications Systems IV.* pp. 198–212. – IOS Press, 1997.

[3] Aho A., Gallagher S., Griffeth N., Schell C. et Swayne D. – SCF3^{TM}Sculptor with Chisel: requirements engineering for communications services. *In : Feature Interactions in Telecommunications Systems V.* pp. 45–63. – IOS Press, 1998.

[4] Akers S.B. – Binary Decision Diagrams. *IEEE Transactions on Computers*, vol. C-27, 1978, pp. 509–516.

[5] Alpern B. et Schneider F. B. – Defining liveness. *Information Processing Letters*, vol. 21, octobre 1985, pp. 181–185.

[6] Au P.K. et Atlee J.M. – Evaluation of a state-based model of feature interactions. *In : Feature Interactions in Telecommunications Systems IV.* pp. 153–167. – IOS Press, 1997.

[7] Belinfante A., Feenstra J., de Vries R. G., Tretmans J., Goga N., Feijs L., Mauw S. et Heerink L. – Formal test automation: a simple experiment. *In : 12th International Workshop on Testing of Communicating Systems.* pp. 179–196. – Kluwer Academic Publishers, 1999.

[8] Benveniste A. et Berry G. – The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, vol. 79, n° 9, 1991, pp. 1270–1282.

[9] Blom J., Bol R. et Kempe L. – Automatic detection of feature interactions in temporal logic. *In : Feature Interactions in Telecommunications Systems III.* pp. 1–20. – IOS Press, 1995.

[10] Blom J., Jonsson B. et Kempe L. – Using temporal logic for modular specification of telephone services. *In : Feature Interactions in Telecommunications Systems.* pp. 197–216. – IOS Press, 1994.

[11] Boniol F. – *Étude d'une sémantique de la réactivité : variations autour du modèle synchrone et application aux systèmes embarqués.* – Thèse, École Nationale Supérieure de l'Aéronautique et de l'Espace, décembre 1997.

[12] Boniol F. et Adelantado M. – Programming distributed reactive systems: a strong and weak synchronous coupling. *In : 7th International Workshop on Distributed Algorithms*. pp. 294–308. – LNCS, septembre 1993.

[13] Bouma L.G. et Velthuijsen H. – Introduction of the proceedings. *In : Feature Interactions in Telecommunications Systems V*. pp. vii–xiv. – IOS Press, 1998.

[14] Boussinot F. et De Simone R. – The Esterel language. *Proceedings of the IEEE*, vol. 79, n° 9, 1991, pp. 1293–1304.

[15] Bowen T.F., Dworak F.S., Chow C.-H., Griffeth N.D., Herman G.E. et Lin Y.-L. – The feature interaction problem in telecommunication systems. *In : 7th International Conference on Softwre Engineering for Telecommunication Switching Systems.* – Bournemouth, United Kingdom, 1989.

[16] Braithwaite K. H. et Atlee J. M. – Towards automated detection of feature interactions. *In : Feature Interactions in Telecommunications Systems*. pp. 36–59. – IOS Press, 1994.

[17] Brinksma E. – A theory for derivation of tests. *In : Protocol Specification, Testing and Verification VIII*, éd. par Aggrawal S. et Sabnani K. – North-Holland, 1988.

[18] Brinksma E., Tretmans J. et Verhaard L. – A framework for test selection. *In : Protocol Specification, Testing and Verification XI*, éd. par Jonsson B. – North-Holland, 1991.

[19] Bryant R.E. – Graph-based algorithms for boolean functions manipulation. *IEEE Transactions on Computers*, 1986, pp. 667–692.

[20] Buhr R.J.A., Amyot D., Elammari M., Quesnel D, Gray T. et Mankovski S. – Feature-interaction visualization and resolution in an agent environment. *In : Feature Interactions in Telecommunications Systems V*. pp. 135–149. – IOS Press, 1998.

[21] Caspi P., Halbwachs N., Pilaud D. et Plaice J. – LUSTRE, a declarative language for programming synchronous systems. *In : 14th Symposium on Principles of Programming Languages (POPL 87), Munich*. pp. 178–188. – ACM, 1987.

[22] CCITT. – *Intelligent Network Recommendation.* – Recommandation n° Q.1200, 1992.

[23] Chen Y.-L., Lafortune S. et Lin F. – Resolving feature interactions using modular supervisory control with priorities. *In : Feature Interactions in Telecommunications Systems IV*. pp. 108–122. – IOS Press, 1997.

[24] Clarke E., Grumberg O. et Long D. – Verification tools for finite-state concurrent systems. *In : A Decade of Concurrency, Lecture Notes in Computer Science (803).* pp. 124–175. – Springer Verlag, 1993.

[25] Combes P. et Pickin S. – Formalization of a user view of network and services for feature interaction detection. *In : Feature Interactions in Telecommunications Systems.* pp. 120–135. – IOS Press, 1994.

[26] Dillon L., Kutty G., Melliar-Smith P., Moser L. et Ramakrishna Y. – A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology*, vol. 3, n° 2, avril 1994, pp. 131–165.

[27] du Bousquet L. – Feature interaction detection using testing and model-checking, experience report. *In : World Congress on Formal Methods.* – Toulouse, France, septembre 1999.

[28] du Bousquet L. – *Test fonctionnel statistique de systèmes spécifiés en Lustre.* – Thèse, Grenoble, France, Université Joseph Fourier, septembre 1999.

[29] du Bousquet L., Ouabdesselam F. et Richier J.-L. – Expressing and implementing operational profiles for reactive software validation. *In : 9th International Symposium on Software Reliability Engineering.* – Paderborn, Germany, 1998.

[30] du Bousquet L., Ouabdesselam F., Richier J.-L. et Zuanon N. – Lutess: a specification-driven testing environment for synchronous software. *In : 21st International Conference on Software Engineering.* – ACM, mai 1999.

[31] du Bousquet L., Ouabdesselam F., Richier J.-L. et Zuanon N. – Feature interaction detection using synchronous approach and testing. *Computer Networks and ISDN Systems*, vol. 32, n° 4, avril 2000, pp. 419–431.

[32] Dyer M. – *The cleanroom approach to quality software development.* – John Wiley & Sons, 1992.

[33] ETSI. – *Intelligent network (IN); Intelligent networks framework.* – Standard n° DTR/NA-060101, juin 1991.

[34] ETSI. – *Integrated Services Digital Network (ISDN); Call Forwarding No Reply, Service description.* – Standard n° ETS 300 201, décembre 1994.

[35] ETSI. – *Integrated Services Digital Network (ISDN); Completion of Call to Busy Subscriber, Service description.* – Standard n° ETS 300 357, octobre 1995.

[36] ETSI. – *Integrated Services Digital Network (ISDN); Explicit Call Transfer, Service description.* – Standard n° ETS 300 367, mai 1995.

[37] ETSI. – *Service life cycle reference model for services supported by an IN.* – Standard n° DTR/NA-060109, novembre 1995.

[38] Faci M. et Logrippo L. – Specifying features and analyzing their interactions in a LOTOS environment. *In : Feature Interactions in Telecommunications Systems.* pp. 136–151. – IOS Press, 1994.

[39] Faci M., Logrippo L. et Stepien B. – Structural models for specifying telephone systems. *Computer Networks and ISDN Systems*, vol. 29, n° 4, 1997, pp. 501–528.

[40] Fernandez J.-C., Jard C., Jéron T. et Viho C. – Using on-the-fly verification techniques for the generation of test suites. *In : 8th Conference on Computer-Aided Verification.* – Springer-Verlag, 1996.

[41] Fernandez J.-C., Jard C., Jéron T. et Viho C. – An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, vol. 29, 1997, pp. 123–146.

[42] Foote B. et Yoder J. – Big ball of mud. *In : Proceedings of the 4th conference on Patterns Languages of Programs.* – Université de Washington, 1997.

[43] Frappier M., Mili A. et Desharnais J. – Detecting feature interactions on relational specifications. *In : Feature Interactions in Telecommunications Systems IV.* pp. 123–137. – IOS Press, 1997.

[44] Gammelgaard A. et Kristensen J. E. – Interaction detection, a logical approach. *In : Feature Interactions in Telecommunications Systems.* pp. 178–196. – IOS Press, 1994.

[45] Godskesen J.-C. – A formal framework for feature interaction with emphasis on testing. *In : Feature Interactions in Telecommunications Systems III.* pp. 21–30. – IOS Press, 1995.

[46] Griffeth N. et Velthuijsen H. – The negotiating agents approach to runtime interaction resolution. *In : Feature Interactions in Telecommunications Systems*, éd. par Bouma L. G. et Velthuijsen H. pp. 217–235. – IOS Press, 1994.

[47] Groz R. – *Vérification de propriétés logiques des protocoles et systèmes répartis par observation de simulations.* – Thèse, Université de Rennes I, 1989.

[48] Halbwachs N., Caspi P., Raymond P. et Pilaud D. – The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, vol. 79, n° 9, 1991, pp. 1305–1320.

[49] Halbwachs N., Lagnier F. et Raymond P. – Synchronous observers and the verification of reactive systems. *In : Third International Conference on Algebraic Methodology and Software Technology.* – Springer Verlag, 1993.

[50] Hall R.J. – Feature combination and interaction detection via foreground/background models. *In : Feature Interactions in Telecommunications Systems V.* pp. 232–246. – IOS Press, 1998.

[51] Harel D. et Pnueli A. – On the development of reactive systems. *Logics and Models of Concurrent Systems, NATO ASI Series*, vol. F13, 1985, pp. 477–498.

[52] Holzmann G. J. – *Protocol Design and Validation.* – Prentice Hall, 1991.

[53] ISO. – *Conformance Testing Methodology and Framework - Part 1: General concept.* – Standard international n° ISO/IEC 9646-1, 1991.

[54] ISO. – *Conformance Testing Methodology and Framework - Part 3: Tree and Tabular Combined Notation.* – Standard international n° ISO/IEC 9646-3, 1991.

[55] ITU-T. – *Principles of intelligent network architecture.* – Recommandation n° Q.1201, 1993.

[56] Jacobson I., Christerson M., Jonsson P. et Övergaard G. – *Object-oriented software engineering, a use case driven approach.* – Addison-Wesley, 1994.

[57] Jagadeesan L.J., Porter A., Puchol C., Ramming J.C. et Votta L. – Specification-based testing of reactive software: tools and experiments. *In : 19th International Conference on Software Engineering.* pp. 525–535. – ACM, 1997.

[58] Jard C. – *Protocoles et services : test des spécifications.* – Thèse, Université de Rennes I, 1984.

[59] Jonsson B. et Kempe L. – Verifying safety properties of a class of infinite-state distributed algorithms. *In : 7th International Conference on Computer Aided Verification.* pp. 42–53. – Springer-Verlag, 1995.

[60] Kamoun J. et Logrippo L. – Goal-oriented feature interaction detection in the intelligent network model. *In : Feature Interactions in Telecommunications Systems V.* pp. 172–186. – IOS Press, 1998.

[61] Kelly B., Crowther M., Kling J., Masson R. et DeLapeyre J. – Service validation and testing. *In : Feature Interactions in Telecommunications Systems III.* pp. 173–184. – IOS Press, 1995.

[62] Kimbler K. – Towards a more efficient feature interaction analysis - a statistical approach. *In : Feature Interactions in Telecommunications Systems III.* pp. 201–211. – IOS Press, 1995.

[63] Kimbler K. et Bouma L.G. (édité par). – *Feature Interactions in Telecommunications Systems V.* – IOS Press, septembre 1998.

[64] Kimbler K. et Søbirk D. – Use case driven analysis of feature interactions. *In : Feature Interactions in Telecommunications Systems*, éd. par Bouma L. G. et Velthuijsen H. pp. 167–177. – IOS Press, 1994.

[65] Kimbler K. et Wohlin C. – A statistical approach to feature interaction. *In : Telecommunications Information Networking Architecture (TINA95)*, pp. 219–230. – 1995.

[66] Kolberg M. et Magill H. – Service and feature interactions in TINA. *In : Feature Interactions in Telecommunications Systems V.* pp. 78–84. – IOS Press, 1998.

[67] Kung R. et Maitre X. – L'architecture du réseau intelligent. *L'écho des recherches*, vol. 157, 1994, pp. 5–14.

[68] Lamport L. – Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, vol. 3, n° 2, mars 1977, pp. 125–143.

[69] Lamport L. – The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, vol. 16, n° 2, mai 1994, pp. 872–923.

[70] Le Guernic P., Gautier T., Le Borgne M. et Le Maire C. – Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, vol. 79, n° 9, 1991, pp. 1321–1336.

[71] Lin F.J. et Lin Y.-J. – A building block approach to detecting and resolving feature interactions. *In : Feature Interactions in Telecommunications Systems.* pp. 86–119. – IOS Press, 1994.

[72] Manna Z. et Pnueli A. – *The temporal logic of reactive and concurrent systems: specification.* – Springer-Verlag, 1991.

[73] Mermet B. – *Qualité de service dans une logique temporelle compositionnelle.* – Thèse, Nancy, France, Université Henri Poincaré, décembre 1997.

[74] Mermet B. et Méry D. – Détection d'interaction de services : une approche avec B. *In : Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'97).* – Toulouse, France, 1997.

[75] Mermet B. et Méry D. – Incremental specification of telecommunication services. *In : First IEEE International Conference on Formal Engineering Methods (ICFEM).* – 1997.

[76] Mills H. D., Dyer M. et Linger R. C. – Cleanroom software engineering. *IEEE Software*, septembre 1987.

[77] Murakami G. et Sethi R. – Terminal call processing in Esterel. *In : World Computer Congress.* – Madrid, Spain, 1992.

[78] Musa J. – Operational profiles in software-reliability engineering. *IEEE Software*, mars 1993, pp. 14–32.

[79] Naesser G., Nyström J. et Jonsson B. – *Contribution to feature interaction detection contest.* – Rapport interne, Uppsala University, juillet 1998. accessible à l'URL http://www-db.research.bell-labs.com/user/nancyg/Contest/uppsala1.ps.gz.

[80] Ouabdesselam F. et Parissis I. – Testing synchronous critical software. *In : 5th International Symposium on Software Reliability Engineering.* – Monterey, USA, 1994.

[81] Owicki S. et Lamport L. – Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, vol. 4, n° 3, juillet 1982, pp. 455–495.

[82] Pageot J.-M. – *Guider la simulation pour valider les protocoles.* – Thèse, Université de Rennes I, 1989.

[83] Parissis I. – *Test de logiciels synchrones spécifiés en Lustre.* – Thèse, Grenoble, France, Université Joseph Fourier, septembre 1996.

[84] Phalippou M. – *Relations d'implémentations et hypothèses de test sur des automates à entrées et sorties.* – Thèse, France, Université de Bordeaux I, septembre 1994.

[85] Pilaud D. et Halbwachs N. – 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.* – Springer Verlag, 1988.

[86] Plath M. et Ryan M. – Plug-and-play features. *In : Feature Interactions in Telecommunications Systems V.* pp. 150–164. – IOS Press, 1998.

[87] Pnueli A. – Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. *Lecture Notes in Computer Science, Current Trends in Concurrency*, vol. 224, 1986, pp. 510–584.

[88] Pnueli Amir, Shankar Natarajan et Singerman Eli. – Fair synchronous transition systems and their liveness proofs. *In : 5th International School and Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT '98).* pp. 198–209. – Lyngby, Denmark, septembre 1998.

[89] Prehofer C. – An object-oriented approach to feature interaction. *In : Feature Interactions in Telecommunications Systems IV.* pp. 313–325. – IOS Press, 1997.

[90] Ramadge P. et Wonham W. – Supervisory control of a class of discrete event processes. *SIAM J. CONTROL AND OPTIMIZATION*, vol. 25, n° 1, janvier 1987, pp. 206–230.

[91] Ratel C. – *Définition et réalisation d'un outil de vérification formelle de programmes Lustre : Le système Lesar.* – Thèse, Grenoble, France, Université Joseph Fourier, juin 1992.

[92] Raymond P., Weber D., Nicollin X. et Halbwachs N. – Automatic testing of reactive systems. *In : 19th IEEE Real-Time Systems Symposium.* – IEEE, 1998.

[93] Rudolph E., Graubmann P. et Grabowski J. – Tutorial on message sequence charts. *Computer Networks and ISDN Systems*, vol. 28, 1996, pp. 1629–1641.

[94] Shannon C.E. – A symbolic analysis of relay and switching circuits. *Transactions AIEE*, vol. 57, 1938, pp. 305–316.

[95] Spivey J. M. – *The Z Notation.* – Prentice-Hall, 1989.

[96] Tretmans J. – *A formal approach to conformance testing.* – Thèse, Enschede, The Netherlands, University of Twente, 1992.

[97] Tsang S. et Magill E. H. – Detecting feature interactions in the intelligent network. *In : Feature Interactions in Telecommunications Systems*, éd. par Bouma L. G. et Velthuijsen H. pp. 236–248. – IOS Press, 1994.

[98] Tsang S. et Magill E.H. – Behaviour based run-time feature interaction detection and resolution. *In : Feature Interactions in Telecommunications Systems IV*. pp. 254–270. – IOS Press, 1997.

[99] Turner K.J. – An architectural description of intelligent network features and their interactions. *Computer Networks and ISDN Systems*, vol. 30, n° 15, 1998, pp. 1389–1420.

[100] Utas G. – A pattern language of feature interaction. *In : Feature Interactions in Telecommunications Systems V*. pp. 98–114. – IOS Press, 1998.

[101] van der Linden R. – Using an architecture to help beat feature interaction. *In : Feature Interactions in Telecommunications Systems*. pp. 24–35. – IOS Press, 1994.

[102] Velthuijsen H. – Issues of non-monotonicity in feature-interaction detection. *In : Feature Interactions in Telecommunications Systems III*. pp. 31–42. – IOS Press, 1995.

[103] Whittaker J. – *Markov chain techniques for software testing and reliability analysis*. – Thèse, University of Tenessee, 1992.

[104] Woit D. – Specifying operational profiles for modules. *In : International Symposium on Software Testing and Analysis*, pp. 2–10. – Cambridge, Massachusetts, USA, juin 1993.

[105] Zeller A. – Yesterday, my program worked. Today, it does not. Why ? *In : Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pp. 253–267. – Toulouse, France, septembre 1999.

[106] Zuanon N. – *Modélisation et validation de services téléphoniques*. – Rapport de DEA, Grenoble, France, INPG - ENSIMAG, juin 1996.

Annexe A

Modélisation LUSTRE

Nous donnons dans ce chapitre quelques éléments de réalisation du modèle exécutable dans le cadre du second concours de détection d'interactions (FIW'2000) et illustrant la méthodologie décrite au chapitre 6.

La figure A.1 décrit la structure générale de la spécification LUSTRE choisie pour modéliser le système. L'environnement est composé de l'ensemble des usagers pouvant utiliser le système, ainsi que du module de facturation. Ce dernier n'a qu'un rôle passif, puisqu'il ne fait que recevoir des informations du système, sans rien émettre en retour.

La représentation est ici légèrement simplifiée: on a fait notamment abstraction des expirations de timers. Celles-ci sont représentées par des variables supplémentaires que nous avons préféré ne pas faire apparaître pour rendre plus lisible le code LUSTRE.
Par ailleurs, on notera que les messages provenant de l'environnement sont directement transmis aux postes logiques concernés, pour des raisons analogues de simplicité de lecture.

```
-- NBFEATURES et NBUSERS sont des constantes entieres definissant le nombre
-- de services et le nombre d'usagers.
-- Le type "message" est decrit par un tableau de booleens et permet de
-- representer n'importe quel message defini dans l'enonce du concours

type messages = message^NBUSERS;

node System
(
        UserInitiatedMessages : messages;
        SubscriptionLists: bool^NBFEATURES^NBUSERS;
        Condition : bool;
)
returns
(
        ControlSoftwareInitiatedMessages : messages;
```

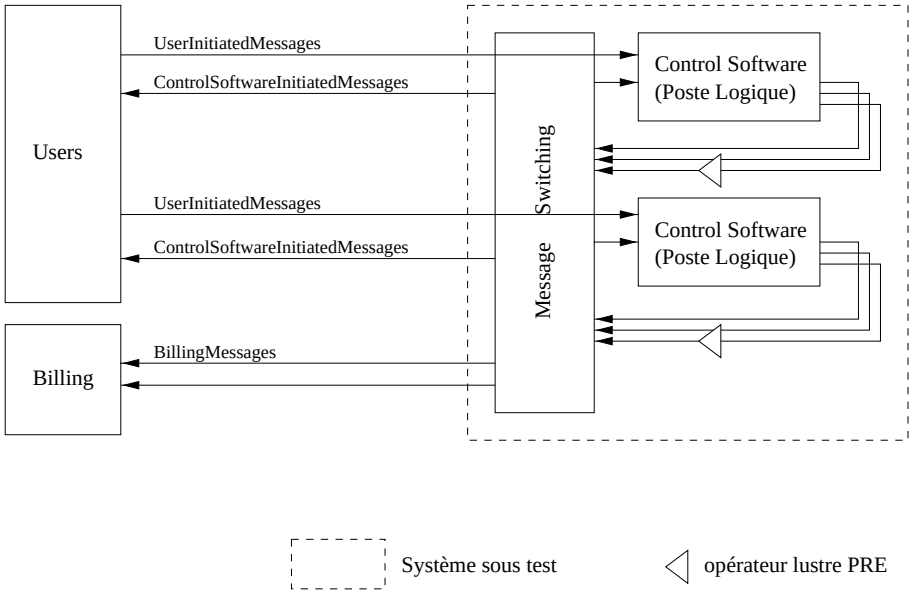


FIG. A.1 – *Structure générale du programme* LUSTRE

```
        BillingMessages : messages;
);
var
-- Un poste logique peut emettre trois types de sorties en simultane :

-- * une sortie pour l'usager qu'il represente
   ControlSoftwareUsersOutputs : messages;

-- * une sortie pour le logiciel de facturation
   ControlSoftwareBillingOutputs : messages;

-- * une sortie pour un autre poste logique
   ControlSoftwareOutputs : messages;

-- Un poste logique dispose de deux entrees :
-- * l'une externe, traduisant les actions des usagers : UserInitiatedMessages,
-- * l'autre interne, vehiculant les messages echanges entre postes :
   SwitchedMessagesFromControlSoftware : messages;

let

   (ControlSoftwareInitiatedMessages,
   BillingMessages,
   SwitchedMessagesFromControlSoftware)=
        MessageSwitching(ControlSoftwareUsersOutputs,
                         ControlSoftwareBillingOutputs,
```

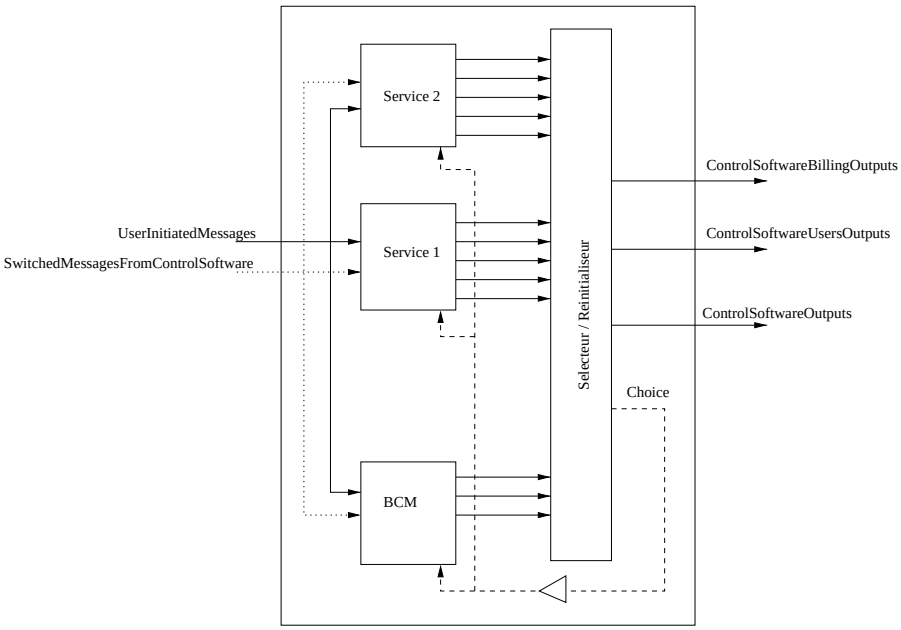


FIG. A.2 – *Détail d'un poste logique* (Control Software)

```
                    empty_messages -> pre ControlSoftwareOutputs);


   (ControlSoftwareOutputs,
    ControlSoftwareUsersOutputs,
    ControlSoftwareBillingOutputs)=
        ControlSoftware(UserInitiatedMessages,
                        SwitchedMessagesFromControlSoftware,
                        SubscriptionLists,
                        Condition^NBUSERS);

tel;
```

La figure A.2 représente la manière dont est structuré un poste logique. *Choice* est une variable entière qui indique quel service a été choisi. La réaction fournie en sortie est celle proposée par ce dernier. Les autres réponses sont ignorées, et les autres services sont réinitialisés en conséquence.

```
node ControlSoftware
(
              UserInitiatedMessage : message;
        SwitchedMessageFromControlSoftware : message;
```

```
        SubscriptionList : bool^NBFEATURES;
        Condition : bool;
)
returns
(
        ControlSoftwareOutput : message;
        ControlSoftwareUserOutput : message;
        ControlSoftwareBillingOutput : message;
)

var
        Choice : bool;
        ReturnState : int^NBFEATURES;
        FeatureOutput : message^NBFEATURES;
        FeatureUserOutput : message^NBFEATURES;
        FeatureBillingOutput : message^NBFEATURES;
        BCMOutput : message;
        BCMUserOutput : message;
        BCMBillingOutput : message;

let

-- l'equation ci-dessous permet de selectionner la reaction d'un service a
-- l'entree courante.
        (ControlSoftwareOutput,
         ControlSoftwareUserOutput,
         ControlSoftwareBillingOutput,
         Choice)=
                Selecteur(BCMOutput,
                          BCMUserOutput,
                          BCMBillingOutput,
                          FeatureOutput,
                          FeatureUserOutput,
                          FeatureBillingOutput,
                          ReturnState);

-- Appel des services
-- Les 4 derniers arguments d'appel du noeud "Selecteur"
-- sont des tableaux parametres par le nombre de services.
-- La reaction d'un service sera decrite par les elements de meme indice des
-- tableaux FeatureOutput, FeatureUserOutput, FeatureBillingOutput et
-- ReturnState.
-- Pour rajouter un service, il suffit d'incrementer la constante NBFEATURES et
-- d'inclure ci-dessous la definition de la reaction du nouveau service.
```

```
        (FeatureOutput[0],
         FeatureUserOutput[0],
         FeatureBillingOutput[0],
         ReturnState[0])=
                 Service1(UserInitiatedMessage,
                         SwitchedMessageFromControlSoftware,
                         SubscriptionList[0],
                         Condition,
                         pre Choice);

        (FeatureOutput[1],
         FeatureUserOutput[1],
         FeatureBillingOutput[1],
         ReturnState[1])=
                 Service2(UserInitiatedMessage,
                         SwitchedMessageFromControlSoftware,
                         SubscriptionList[1],
                         Condition,
                         pre Choice);

tel;
```

Dans le fragment de description ci-dessus, *ReturnState* indique, pour chaque service, l'état de retour : cet état est celui dans lequel sera placé le modèle du service de base lorsque le contrôle de l'appel lui sera rendu.

Annexe B

Énoncé du premier concours de détection d'interaction (FIW'98)

Feature Interaction Detection Contest

5th International Workshop on Feature Interactions

Instructions

Contest committee: Nancy Griffeth (chair), Ralph Blumenthal , Jean-Charles Gregoire, and Tadashi Ohta

The goal of this contest is to compare different feature interaction detection tools according to a single benchmark collection of features. We have tried to describe the features both precisely and simply. This has not been easy, but we hope that the result will be adequate to the purpose.

The original contest announcement is attached in Appendix A. The feature definitions, in the form of Chisel sequence diagrams, are in Appendix B.

In order to define the features, we model a network as a collection of black boxes (Section 1), communicating with each other via defined interfaces (Section 2). We define the POTS service and the features as sequences of events taking place on these interfaces (Section 3).

The choice of features has been dictated by the need for them to interact. They must have some commonality. For this first contest, we are using versions of familiar POTS features, to ensure that there are plenty of interactions. To provide a bit of a challenge for the tools, we introduce features in several different categories. There are billing features as well as call control features, and some features are switch-based whereas others are implemented on an IN platform.

Don't assume that because these are familiar features, the interactions will be the usual ones. The features are defined on a simplified network model, their definitions have been changed slightly, and – most important – we have sincerely tried to define each feature without thinking about the other features. Each feature was defined as if it is the only feature that will be used and as if it will be active on only one call at a time. As a result, the interactions may be quite different from the expected ones! In particular, beware of interactions that are normally so embedded in the features that they appear to be part of them.

Our models of POTS, IN, and billing are quite simple. We hope that more detailed models of these will be available for future contests. However, for this first contest, it seemed important to keep things simple as is reasonable (and no simpler). In future contests, we would also like to see themes involving ISDN, wireless, PCS, and even the Web.

1. The Network

The network consists of:

- End-user equipment (telephones)
- A switch (fast enough to handle all telephones at once)
- A Service Control Point (SCP) that processes IN features like credit card calling, 800/900 numbers, IN Call Forwarding, and so on
- An Operations System (OS) that does billing

There is also a global clock whose value is represented in the variable Time.

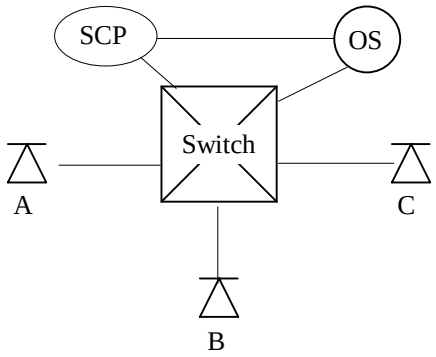


Figure 1. Diagram of the network

2. Events on the Network Interfaces

The network interfaces are the interface between the user and the switch (on which the telephone is used for signaling); the interface between the switch and the SCP (on which IN messages are used); and the interface to the billing system (for tracking the beginning and end of each call).

2.1 User/Switch

The telephone provides the events on the interface between the user and the switch. The tables below summarize the events between the user and the switch. The format of an event is:

<event> <parameter>:<type> … <parameter>:<type>

where <event> is an event name, <parameter> is a symbol, and <type> indicates the type of the parameter. We use types Address (for a telephone number), Cadence (for a special ring or tone), and Message (for a string).

User to Switch
Off-hook X:Address
On-hook X:Address
Dial X:Address Y:Address
Flash X:Address

Switch to User
DialTone X:Address [C:Cadence]
Start AudibleRinging X:Address Y:Address
Start Ringing X:Address Y: Address [C:Cadence]
Start CallWaitingTone X: Address Y: Address [C:Cadence]
Stop AudibleRinging X:Address Y:Address
Stop Ringing X:Address Y:Address
Stop CallWaitingTone X:Address Y:Address
LineBusyTone X: Address [C:Cadence]
Announce X:Address M:Message
Disconnect X:Address Y:Address
Display X:Address M:Message

The first parameter of an event is the address of the telephone on whose interface the event occurs. A number of events have more parameters:

- *Dial A B* means that the subscriber at address A dials the address B.
- *Flash A* is equivalent to an On-hook A followed by an Off-hook A, unless a feature uses it otherwise. We assume that end-users have a Flash button.
- *DialTone A n* means that dialtone occurs at address A. The second parameter, if present, specifies a special cadence for the dialtone. We provide stop and start events for ringing and call-waiting tone, but for the given features, no interesting events occur between starting and stopping DialTone or LineBusyTone, so we left them as single events.

- *Start Ringing A B n* and *Start CallWaitingTone A B n* mean that alerting starts at address A for a call originated at address B. If the third parameter is present, it specifies the cadence of the alerting. It not, the cadence is the usual cadence.
- *Start AudibleRinging A B* means that the ringback tone is provided at address A while waiting for the user at address B to answer the call.
- *Stop Ringing A B, Stop CallWaitingTone A B,* and *Stop AudibleRinging A B* mean to stop the ringing or tone occurring at address A in relation to a call to or from B.
- *LineBusyTone A n* means that the telephone to which A is attempting a connection is busy. The second parameter, if present, specifies the cadence of the tone.
- *Announce A M* means that announcement M is played to address A.
- *Disconnect A B* informs A that B has disconnected a connection with A. (We use a restricted definition for Disconnect. It is a signal from the switch to a user, signaling the user that a connected party has gone On-hook. The On-hook event is the signal from the user to the switch that the user is disconnecting.)
- *Display A M* uses a display screen on telephone at address A (or a Caller ID box at the same address) to display the message M concerning the current call. Typically, the message is the number of the calling party, but it could be another string such as "Anonymous."

We assume that the various cadences and messages have well-defined implementations, but we will just use descriptive strings to distinguish them.

2.2 Switch/SCP

The Bellcore AIN document GR-1298-CORE has been a reference for this interface, but we use a much simplified version of the message parameters. A trigger message includes only the trigger name, the subscriber address, the calling party address, the called party address, and the time. The response messages include only the response type, the subscriber name, the calling party address, and other necessary parameters for that response type.

Switch to SCP
Trigger N:TriggerName S:Address A:Address B:Address T:Time Resource S:Address A:Address M:Message

SCP to Switch
Response R:ResponseType S:Address A:Address … (see below)

The type TriggerName is an enumeration of the names of IN triggers. Some valid TriggerName's are ORIGINATION_ATTEMPT, INFO_COLLECTED, INFO_ANALYZED, and NETWORK_BUSY. In the trigger message, the first address parameter is that of the subscriber, the second of the calling party, the third of the called party, and the last parameter is the current time.

The type ResponseType is an enumeration of the SCP responses to trigger messages. Some valid ResponseType's are ANALYZE_ROUTE, CONTINUE, FORWARD_CALL, and SEND_TO_RESOURCE.

The *Resource S A M* message responds to the SEND_TO_RESOURCE message. The string M is a string of collected digits.

Additional parameters (after the subscriber and calling party addresses) for the ResponseTypes are given in the following table.

Response Type	Additional Parameters
ANALYZE_ROUTE	B:Address C:Address
FORWARD_CALL	B:Address C:Address
CONTINUE	B:Address
SEND_TO_RESOURCE	M:Message
DISCONNECT	

ANALYZE_ROUTE S A B C means to route a call from A to B with C as the paying party (normally, C will be A). S is the subscriber on whose behalf the trigger was activated, usually A or B.

FORWARD CALL S A B C means to forward a call to C, originated by A, with terminating address B.

CONTINUE S A B means to continue processing the call as if no trigger had occurred.

SEND_TO_RESOURCE S A M means to play the message M at address A and collect the response (if any).

DISCONNECT S A means to terminate processing of calls from A until after A has gone on-hook.

2.3 Interface to the billing system

We assume the existence of a billing system recognizing messages from the switch or the SCP and that billing is based entirely on subscriber addresses and calls placed. These messages provide the time, the calling party, called party, and (LogBegin only) the paying party.

to OS
LogBegin X:Address Y:Address P:Address T:Time
LogEnd X:Address Y:Address T:Time

2.4 Simplifying Assumptions

1. The above messages are all there are.
2. User to Switch “messages” get an instantaneous response from the switch, i.e., DialTone starts immediately after Off-hook and stops immediately after Dial. This means that two user messages won’t be sent one immediately after the other; there will always be a switch response between.
3. On-hook is instantaneously followed by Disconnect. There is no extended disconnect timing.
4. There are no network busy conditions.
5. There is no provisioning or de-provisioning of features. Also, there is no feature activation or de-activation. A subscriber either has a feature or doesn’t.
6. The end-user equipment has a button for Flash.

3. Services and Features

We define POTS and all features of POTS as sets of sequences of events on interfaces between network elements (telephones, switch, SCP, and OS). The definitions are expressed in the form of Chisel sequence diagrams. A Chisel sequence diagram is just a directed graph, whose nodes are events on the various interfaces. The diagram defines a set of event sequences for a single call, one for each path through the graph. Event sequences involving multiple calls can be interleaved to define global system activity.

3.1 Interpreting the Chisel sequence diagram for POTS

For illustration, a basic two-party POTS diagram is given at the beginning of the Appendix. It includes both telephones in a two-party call, and also some messages for the billing system.

A node (one of the ovals) in a sequence diagram contains a number, which uniquely identifies the node within the feature, and one or more events and variable assignments. The nodes are connected by directed edges (arrows in the diagrams). Multiple events in a node are separated by vertical bars (|||). A node containing multiple such events is equivalent to the sequence diagram representing any possible sequence of those same events (i.e., A ||| B means {AB, BA}; A ||| B ||| C means {ABC, ACB, BAC, BCA, CAB, CBA}; and so forth).

Variables are used in conditions on the edges, to define when an edge can be followed in constructing an event sequence from the diagram and to restrict possible interleavings of event sequences. A variable defines one or more sequences of events, in the sense that Busy B defines the set of event sequences having one of the following properties:

- An event sequence containing an Off-hook B not followed by On-hook B.
- An event sequence containing Ringing B not followed by Disconnect B A.

Variables may be conceptually part of a feature but we don't have any expectation that they will be implemented (or not).

A condition next to an edge means that to continue an event sequence by following that edge, the condition must be true at the end of the event sequence. C syntax is used in the conditions (~ for not, && for and, || for or).

We use two techniques to define the value of a variable after an event. First, at the beginning of a feature definition, we provide some rules about the values of the variables (to minimize the complexity of the diagram). Second, especially for variables introduced for the individual features, we include an assignment statement with an event to say that the variable takes on a new value after the event. The format of this is `<event> / <var> <- <value>.`

Consider the POTS diagram at the beginning of the Appendix. In this POTS diagram, we describe the values of several variables – Busy A, Ringing A B, and AudibleRinging A B – in rules at the top of the page. Because they are already defined in the rules, we don't really need to set the values of any variables in this diagram, but for illustration we set the values for Busy B in nodes 4, 9, 10, and 14.

The POTS diagram represents only two telephones and a single call. To use a sequence to determine all possible event sequences representing a single call, substitute all pairs of telephone addresses for A and B (and any other symbols). Let's use O(a) to designate the set of all such sequences with originating telephone a. Multiple calls can be originated at a, so let U(a) be the set of all sequences derived by concatenating any number of sequences of O(a) (U(a) = $O(a)^*$, the Kleene closure of O(A)). Then, to determine all possible sequences of events in the network, we can interleave the sets U(A), over all possible call originators A, in all ways allowed by the conditions on the edges.

3.2 Interpreting sequence diagrams for features

Each feature provides some modification to POTS, thereby redefining the subscribed service for that subscriber. We don't model service provisioning or activation, so if a subscriber has a service, that service is always active.

The POTS diagram represents a single call, originated by party A. The feature diagrams represent modifications to this POTS diagram – subdiagrams that can be "pasted" into the POTS diagram at given points. To specify where a feature can be pasted in, we need to designate the node in the POTS diagram and the relationship of the symbols A and B in the POTS diagram to the symbols used in the feature diagram. The designation of a node in the POTS diagram looks like: POTS A<-X B<-Y n, meaning use

node n from the POTS sequence diagram with X substituted for A and Y substituted for B. If either X or Y is "any," then any telephone address is possible there.

A root feature node (i.e., a node with no parents) will be a child of a POTS node. The designation of the POTS node goes above an arrow leading into the node. If the feature node replaces one of the children of this POTS node, the designation of that child goes inside the feature node. Otherwise, the feature node is an additional child of the POTS node.

A leaf feature node (i.e., a node with no children) will be a parent of one or more POTS nodes. The designation of each POTS node goes below an arrow leading out of the leaf feature node.

For illustration, see Figure 1 in the Appendix, which is the sequence diagram for Call Forwarding/Busy Line. Each sequence in the CFBL diagram starts with A dialing (i.e., it will follow a sequence in which the last event was DialTone A) and ends with A and C idle (i.e., both A and C are on-hook and neither is ringing – the next event, if taken from the POTS sequence diagram, would be an off-hook or a ringing event).

4. Variables

The following variables are used in the sequence diagrams.

Busy A: true between an Off-hook A event and the next On-hook A event or between a Start Ringing A event and the next Stop Ringing A event.

Other variables are:

Ringing A B: true between a Start Ringing A event immediately following a Dial B A event and the next Stop Ringing A event

AudibleRinging A B: true between a Start AudibleRinging A event immediately following a Dial A B event and the next Stop AudibleRinging A event.

Note that Busy is true for a telephone if any of the other variables are. We also use *Idle A* to mean that Busy A is not true.

Additional variables may be introduced by a features. Most often, these "variables" define feature parameters (e.g., addresses of screened telephones or PIN's for charging calls), and are fixed for each subscriber (but different for different subscribers). For variables that change during a call, the convention is to describe how the variable behaves with the unfeatured POTS service, in terms of sequences of events (as above), and then to include changes in the value of the feature in the sequence diagram. For the above variables, to keep the diagrams as simple as possible, we will not usually show each change of value. However, if a feature affects the value of one of them (as Call Waiting and Three-Way Calling do the value of Busy), the changes will be shown in the sequence diagram.

5. The Phase 1 Features

5.1 Call Forwarding Busy Line

With the **Call Forwarding Busy Line** feature, all calls to the subscribing line are redirected to a predetermined number when the line is busy. The subscriber pays any charges for the forwarded call from his station to the new destination. The subscriber's originating service is not affected.

Sequence Diagram: Figure 1, Appendix

5.2 Calling Number Delivery

Calling Number Delivery (CND) is a feature that allows the called telephone to receive a calling party's Directory Number (DN) and the date and time. In the on-hook state, in a real network, the delivery of this information occurs during the long silent interval between the first and second power ringing cycles.

For the purposes of the contest, we assume the capability of delivering the number, and deliver it whenever an idle called party receives the Ringing event.

Sequence Diagram: Figure 2, Appendix

5.3 IN Freephone Billing

The **IN Freephone** feature allows the subscriber to pay for incoming calls. Call routing is normally part of this feature, but we define that in the next feature.

Sequence Diagram: Figure 3, Appendix

5.4 IN Freephone Routing

The **IN Freephone** feature allows the subscriber to redirect a call to various telephones, potentially using the all or part of calling number and/or the time of day.

Sequence Diagram: Figure 4, Appendix.

5.5 IN Teen Line

Teen Line restricts outgoing calls based on the time of day (i.e., hours when homework should be the primary activity). This can be overridden on a per-call basis by anyone with the proper identity code. We describe this as an IN feature.

Sequence Diagram: Figure 5, Appendix

5.6 Terminating Call Screening

Terminating call screening (TCS) restricts incoming calls. Calls from lines that appear on a screening list are redirected to a vague but polite message.

Sequence Diagram: Figure 6, Appendix

5.7 Three-way Calling

Three-way calling allows the connection of three parties in a single conversation.

Sequence Diagram: Figure 7, Appendix.

5.8 IN Call Forwarding

The **Call forwarding (CF)** feature permits the subscriber to have incoming calls redirected to another number, no matter what the called party line status is. The user's originating service is unaffected, even for charging. We describe this as an IN service.

Sequence Diagram: Figure 8, Appendix.

5.9 Call Waiting

The **Call waiting (CW)** feature allows the subscriber to be notified that another party is trying to reach his number while her line is busy, and to accept the new call by placing the original call on hold. Subsequently, the subscriber can switch back and forth between the calls.

Sequence Diagram: Figure 9, Appendix

5.10 Charge Call

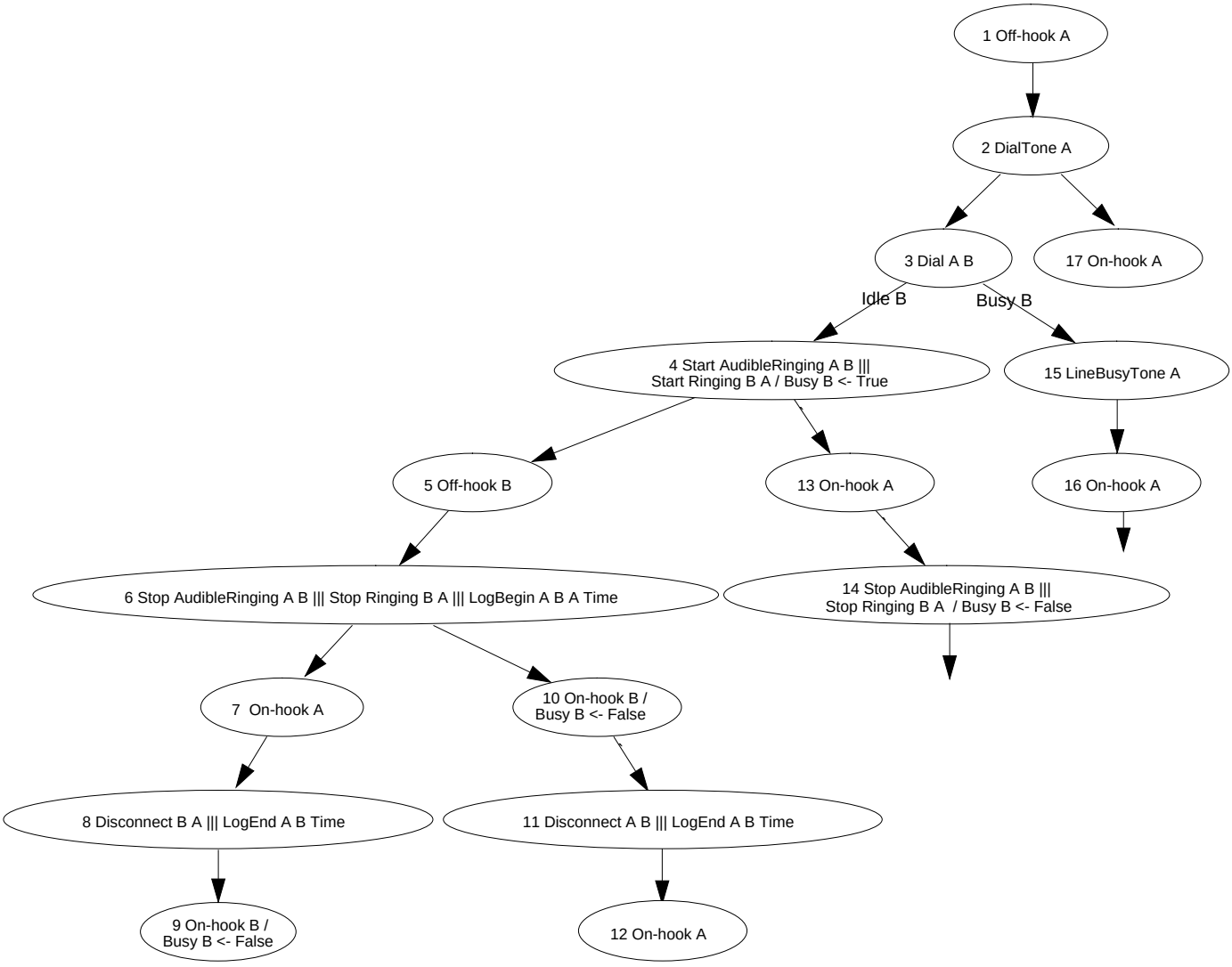
The **Charge Call** feature allows a caller to be automatically charged on a different telephone number than the calling number. The feature is invoked by dialing a code (as defined in the sequence diagram, a prefix "0" to the called telephone number. The caller is then prompted to dial the telephone number to be charged and a PIN. If the PIN is correct, the caller can proceed with the call.

Sequence Diagram: Figure 10, Appendix

POTS

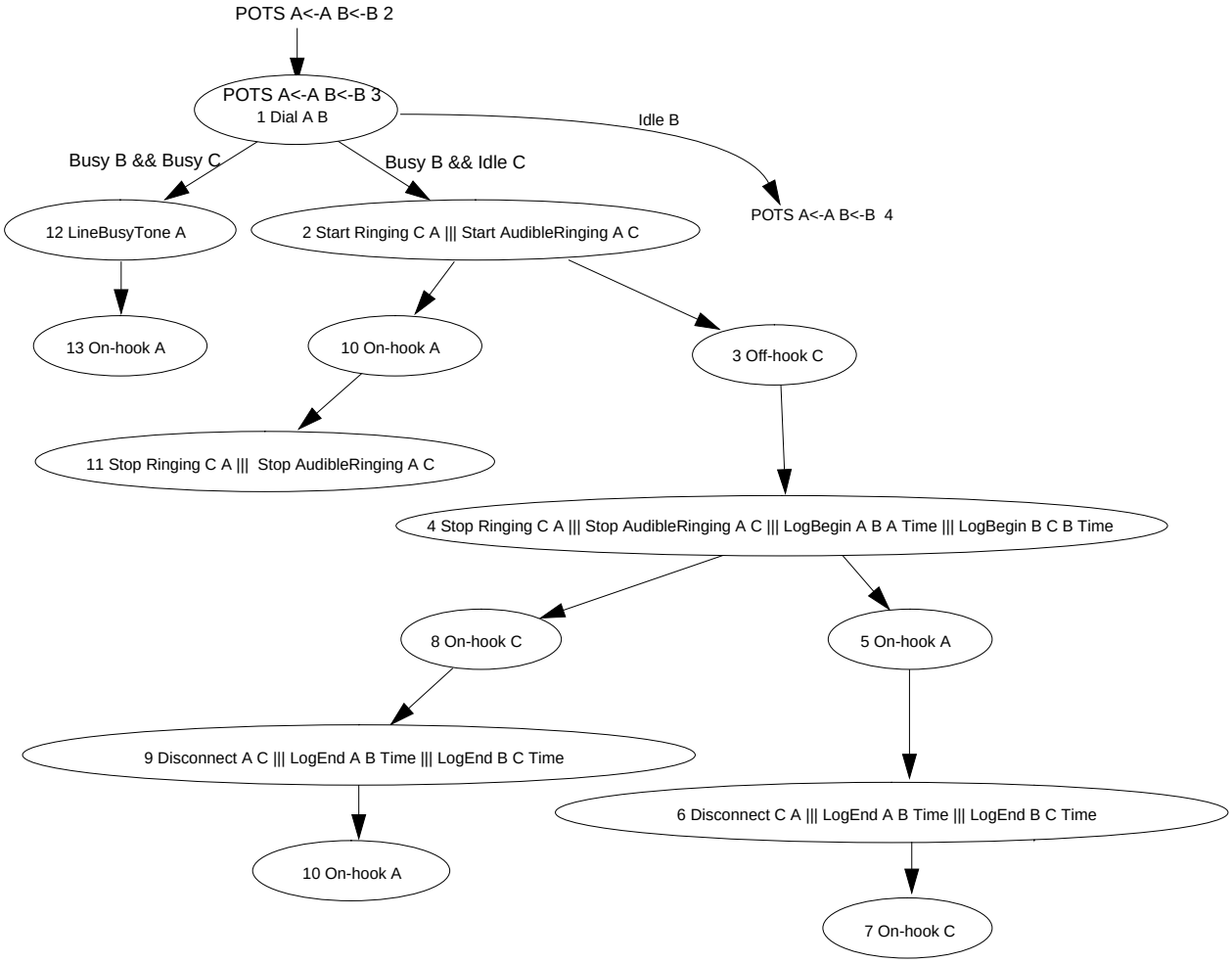
This is the Basic Call Model.

Variables:
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 POTS event sequences start and end with Busy A = False (Idle A = True).
The values of Busy B in the diagram have been given in the preceding rules, but for illustration we show the value changes, in nodes 4, 9, 10, and 14.



1. Call Forwarding Busy Line

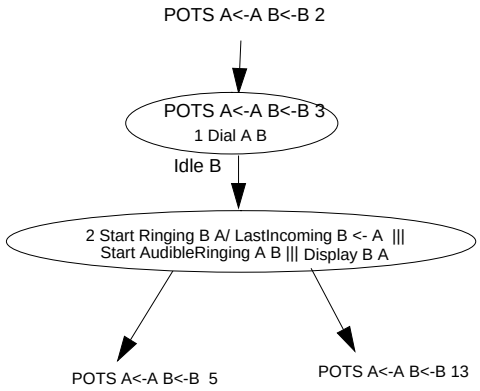
This feature permits a subscriber to have incoming calls that encounter a busy condition to be re-directed.
New Variables: BLForward B is the line to which incoming calls will be redirected if
they encounter a busy condition. If B subscribes to CFBL, this variable is defined and unchanging.
In the sequence diagram below, C = BLForward B.
All of the event sequences in this diagram end with Busy A = False (Idle A = True).



2. Calling Number Delivery

This feature enables the subscriber's telephone to receive and display the number of the originating party on an incoming call.

New Variables:
LastIncoming A is the address of the originator of the last call to the subscriber A.
It is always undefined in POTS. For a subscriber to Calling Number Delivery, the value is changed at each Start Ringing event..

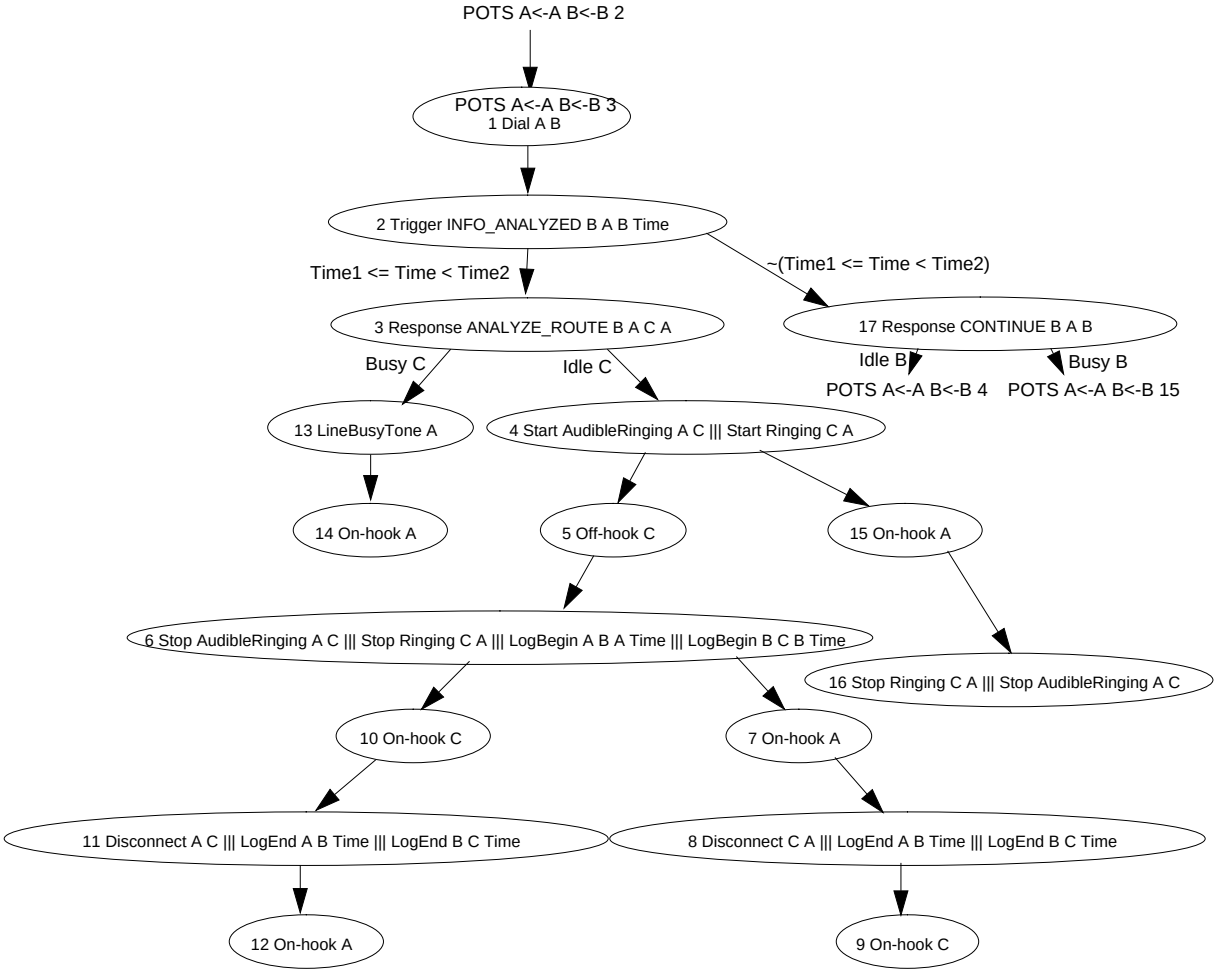


3. IN Freephone Billing

This feature allows the subscriber to be charged for incoming calls. Freephone normally includes routing as well, but we define that as a separate feature in Figure 4.
New Variables: None.

4. IN Freephone Routing

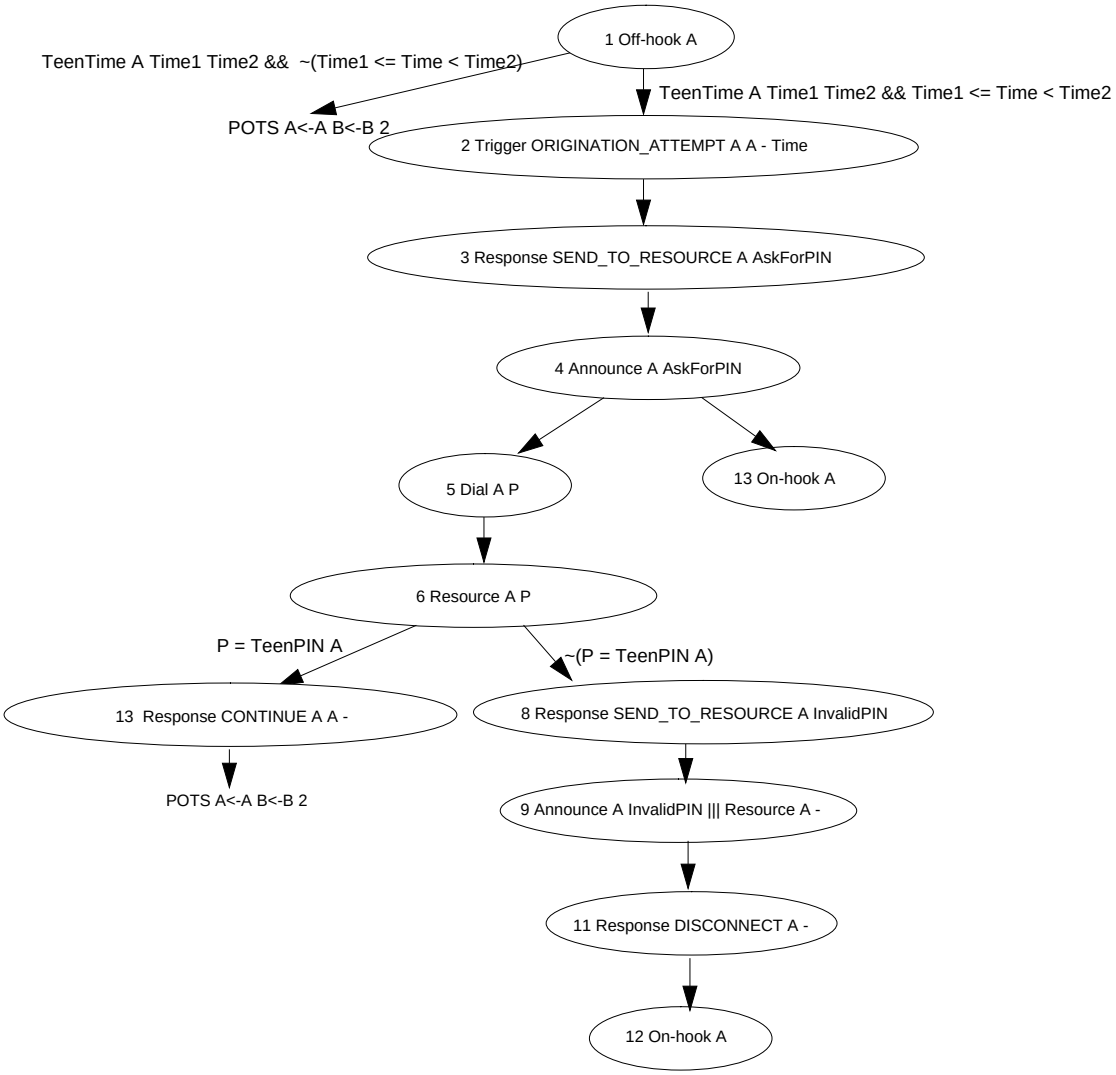
This feature allows the subscriber to redirect a call to various telephones, potentially using all or part of the calling number and the time of day.
New Variables:
Redirect A B Time1 Time2 is the line to which a call is redirected when the subscribed line (B) is called from line A between Time1 and Time2. It is undefined for POTS. In the following sequence diagram,
C=Redirect A B Time1 Time2



5. IN Teen Line

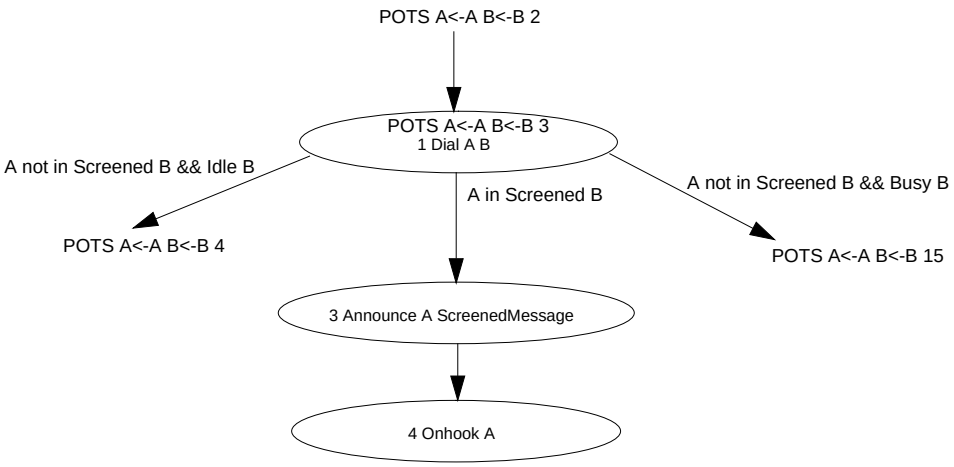
This feature restricts outgoing calls based on the time of day. The restriction can be over-ridden by entering the correct PIN.

New Variables:
TeenPIN A is the valid teen-line PIN for subscriber A. It is undefined for POTS.
TeenTime A Time1 Time2 means that the PIN must be used from Time1 to Time2 in order to initiate a call.
These variables are defined and unchanging for subscribers to Teen Line.



6. Terminating Call Screening

This feature allows a subscriber to screen calls based on the originating number.
New Variables:
Screened B is a set of lines from which subscriber B does not accept calls.
This variable is undefined in POTS, defined and unchanging for subscribers to Terminating Call Screening.



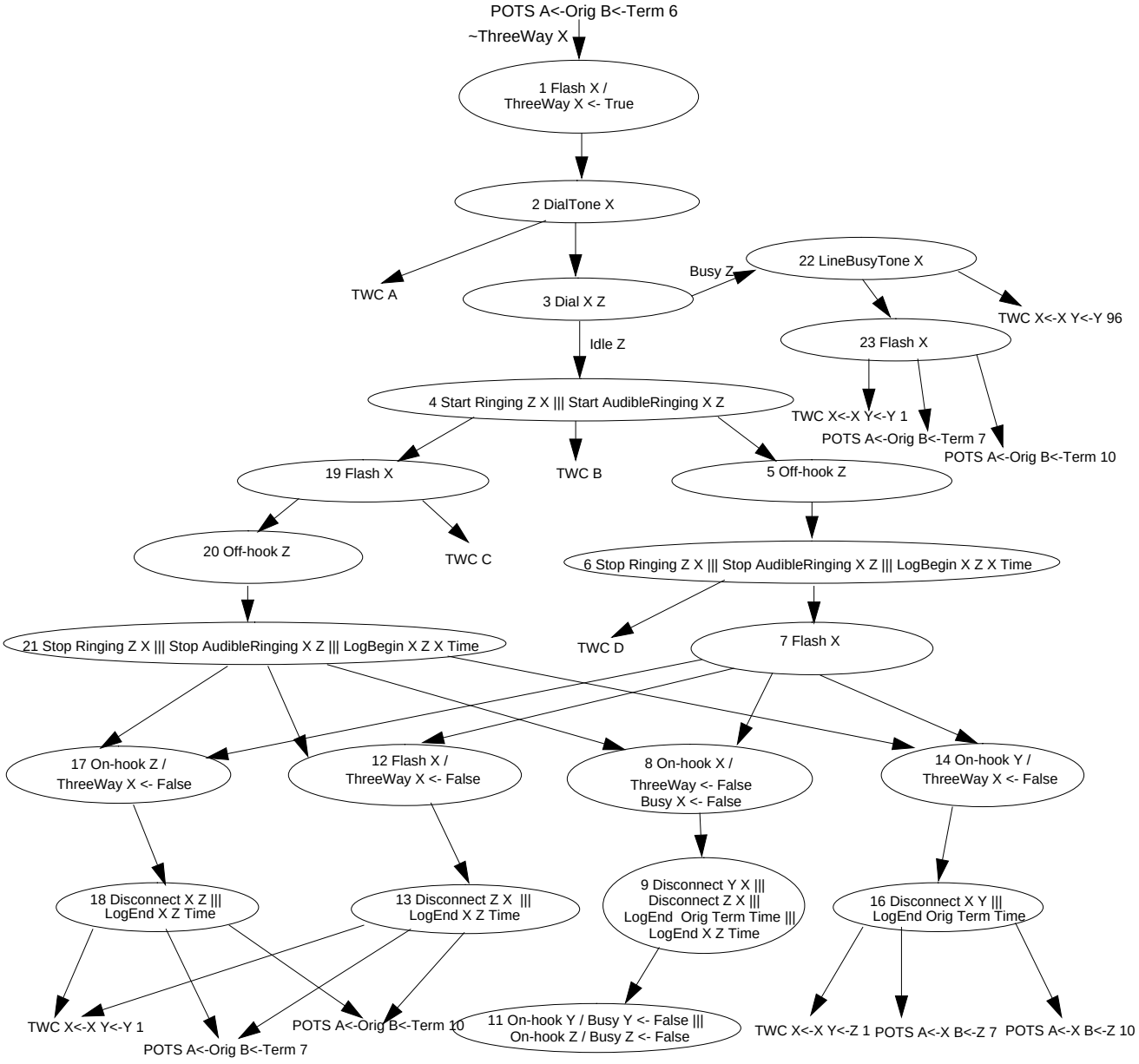
7. Three-Way Calling

.

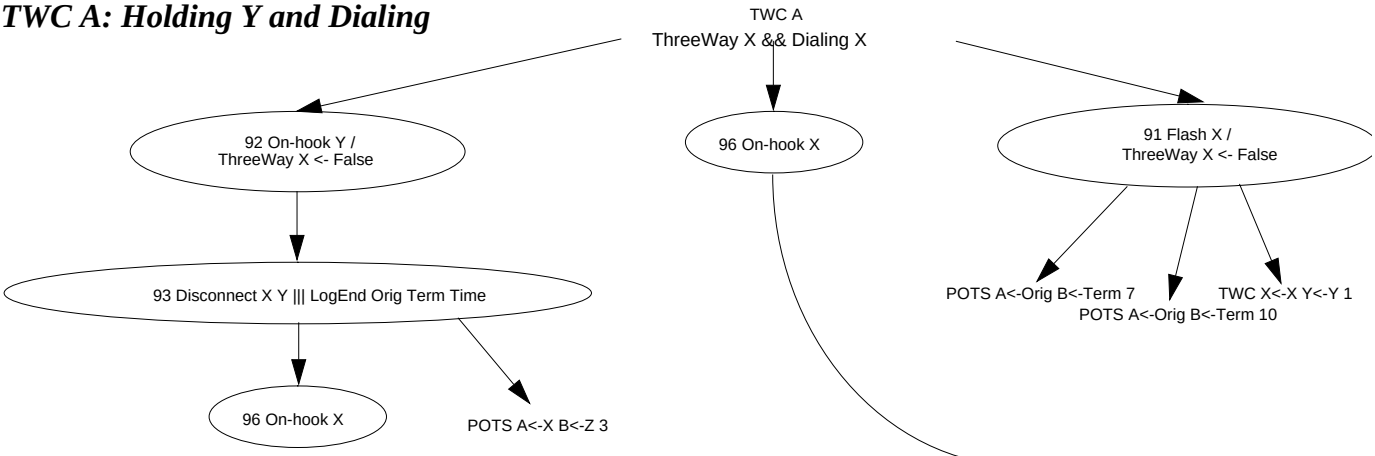
This feature allows the connection of three parties in a call. The following diagram represents two different cases, one with X as the originating party and one with X as the terminating party. The symbols “Orig” and “Term” stand for X and Y respectively when X is the originating party, and for Y and X respectively when X is the terminating party. Each use of this diagram when constructing an event sequence introduces a new Z; if it were expanded, we might write Z1, Z2, ... The symbols TWC A, TWC B, ... direct you to another part of the diagram on a subsequent page.

New Variables:

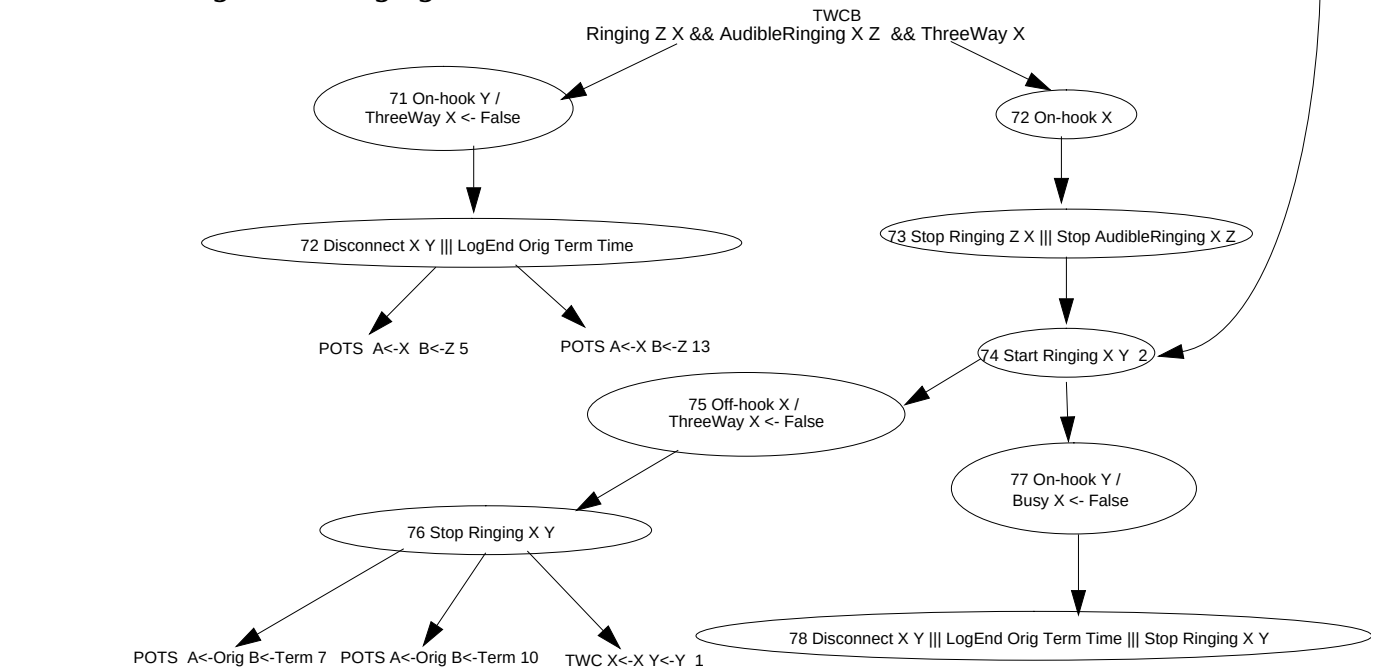
ThreeWay X is true when the Three-Way Calling feature is controlling call behavior. Changes in its value are shown in the diagrams. Busy X is changed somewhat by this feature, so its changes are also given in the sequence diagram



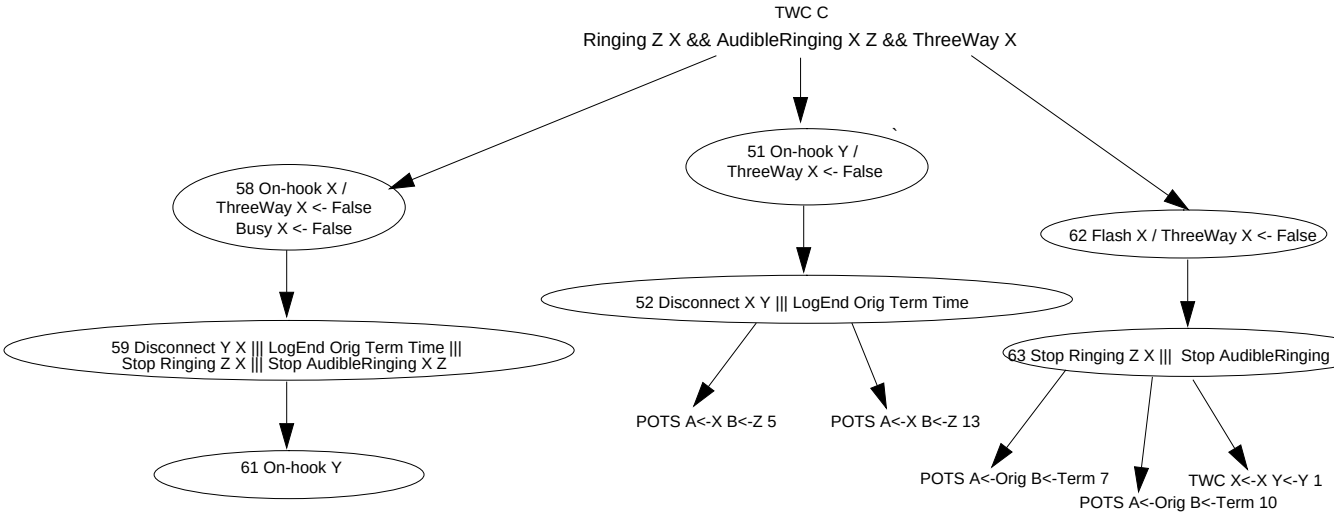
TWC A: Holding Y and Dialing



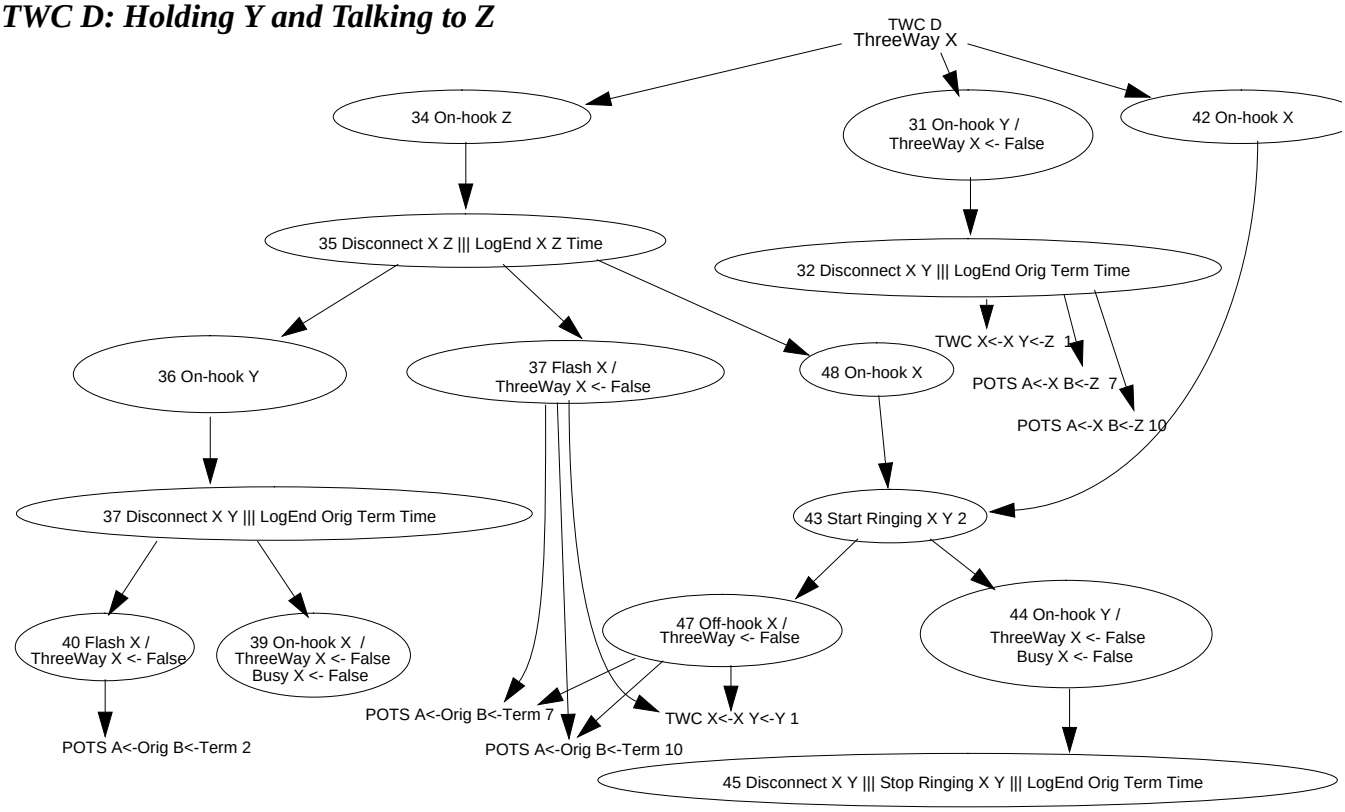
TWC B: Holding Y and Ringing Z



TWC C: Talking to Y and Ringing Z

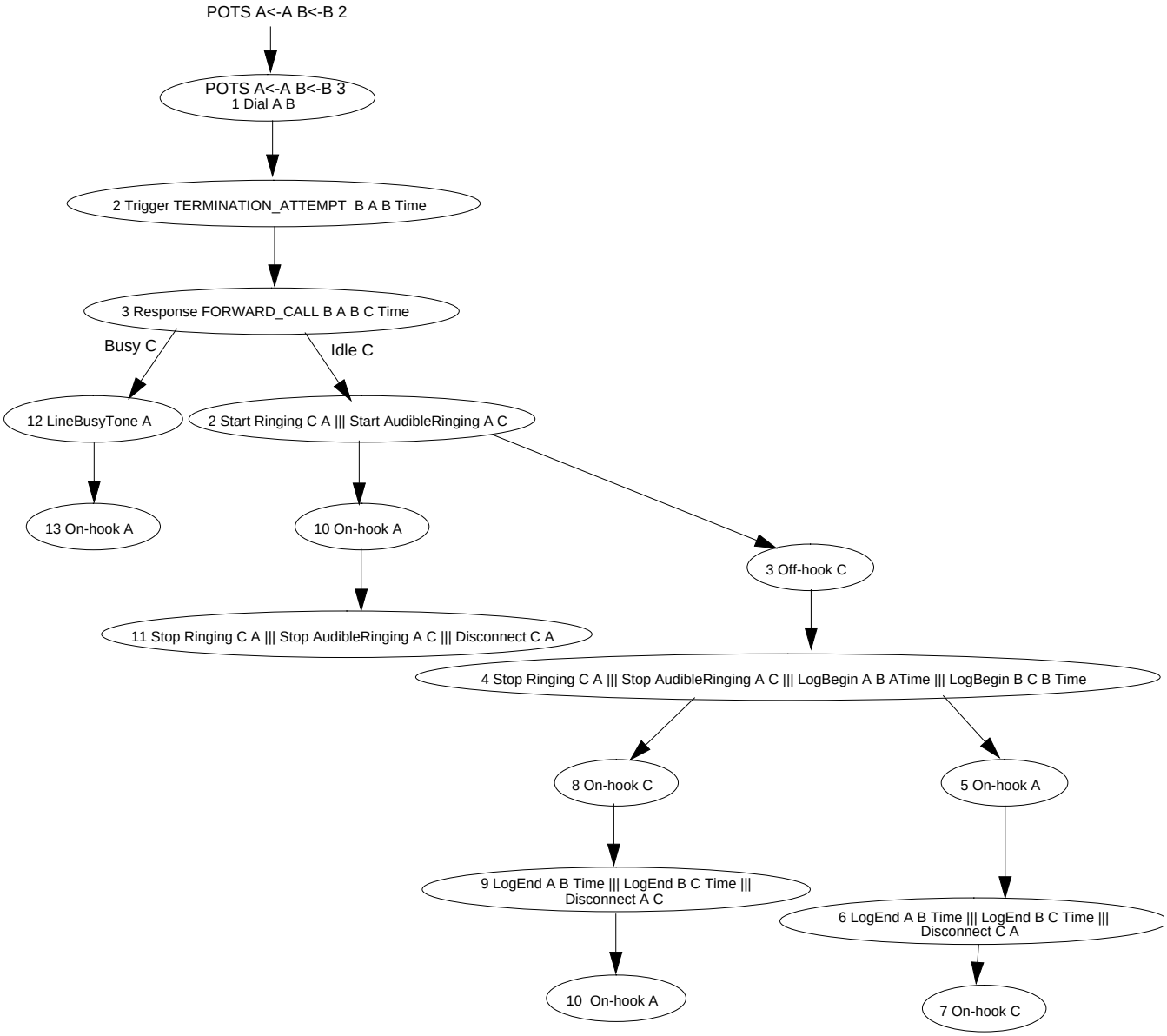


TWC D: Holding Y and Talking to Z



8. IN Call Forwarding

This feature permits a subscriber to have all calls forwarded. The subscriber pays the charges on the forwarded leg.
New Variables: ForwardeToB is the address of the forward-to line for Call Forwarding subscriber B's calls.
It is defined and unchanging for subscribers B to IN Call Forwarding. In the following diagram, C=ForwardTo B.

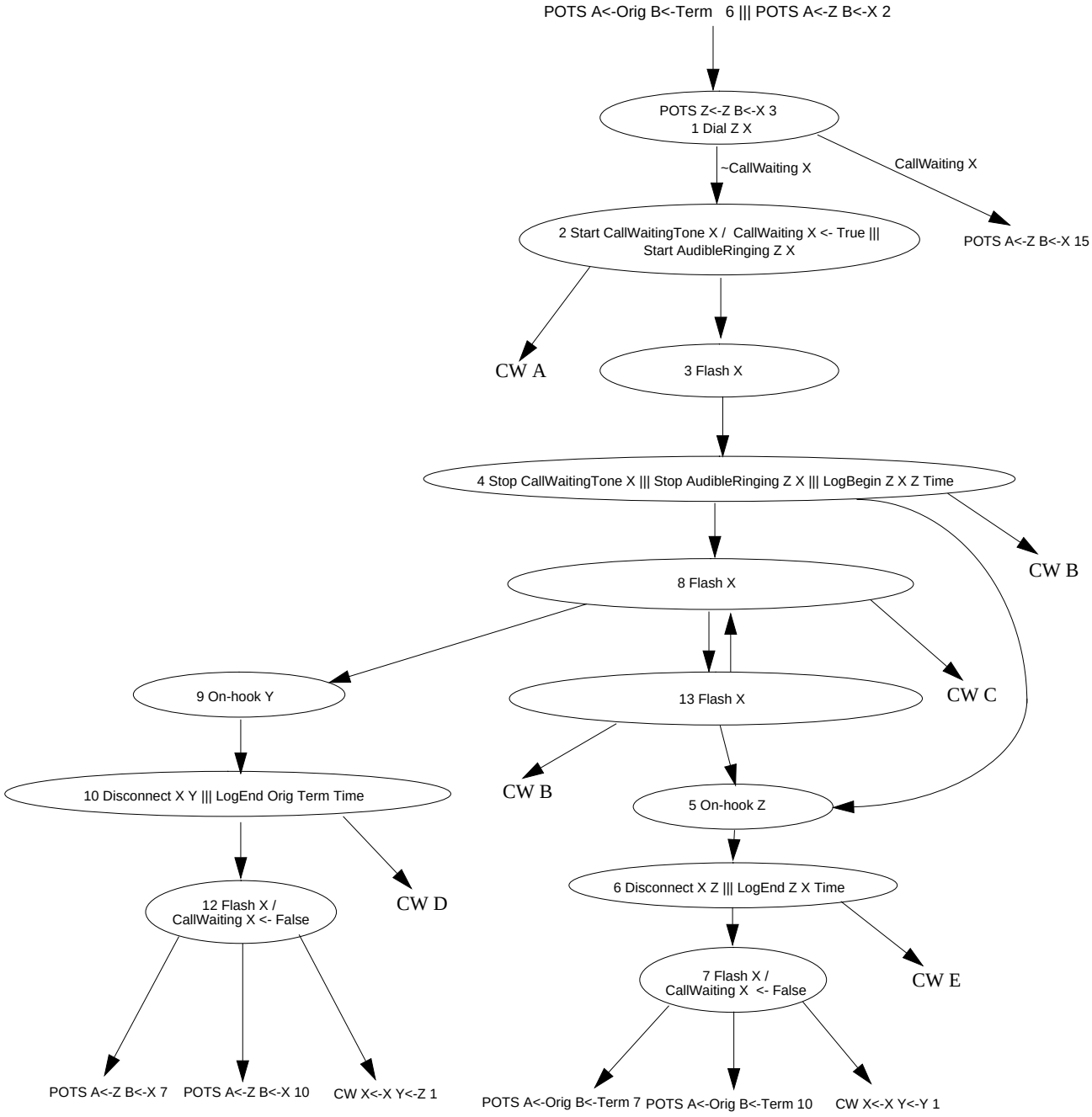


9. Call Waiting

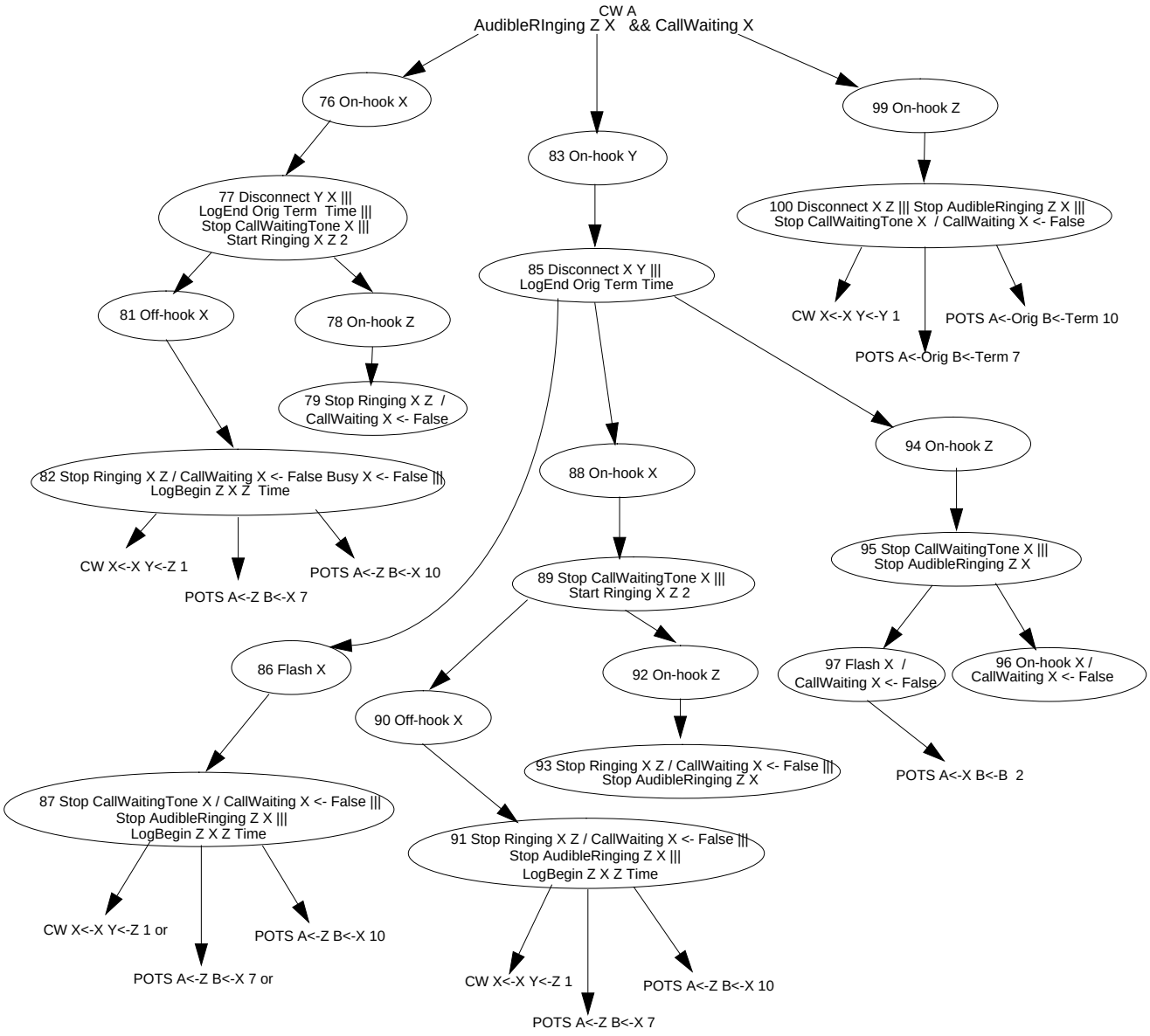
The call waiting feature permits the subscriber to accept a second call when the telephone is already in use. The drawing on this page includ just the "sunny day scenario." Less usual user behaviors are shown in diagrams CW A-E.

As in the Three-Way Calling diagram, the following diagram represents two different cases, one with X as the originating party and one with X as the terminating party. The symbols “Orig” and “Term” stand for X and Y respectively when X is the originating party, and for Y and X respectively when X is the terminating party. Each use of this diagram when constructing an event sequence introduces a new Z.

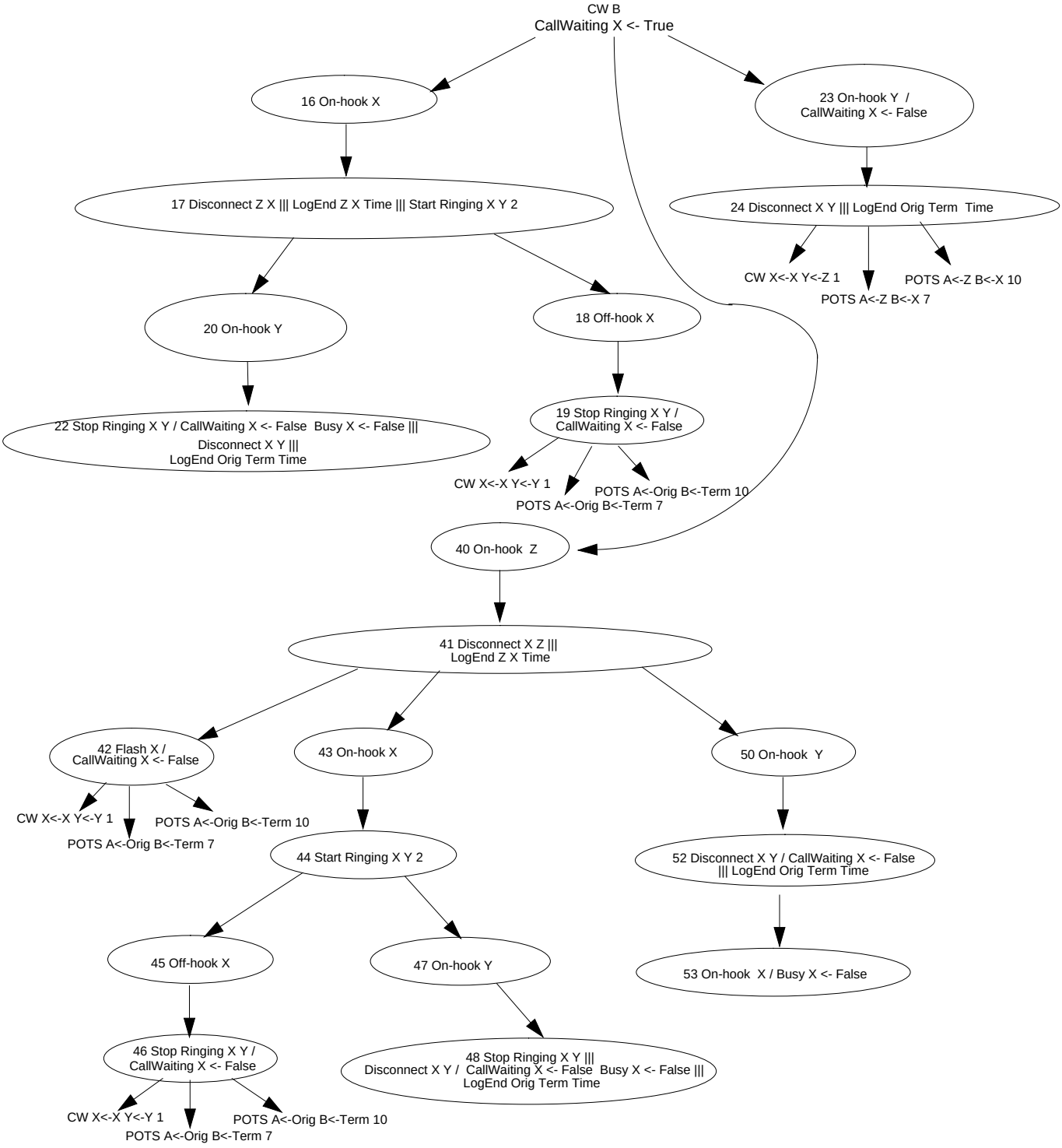
New Variables:
CallWaiting X is true when the Call Waiting feature is controlling the call processing for subscriber X.
Busy X is redefined for Call Waiting, so changes are shown in the diagram.



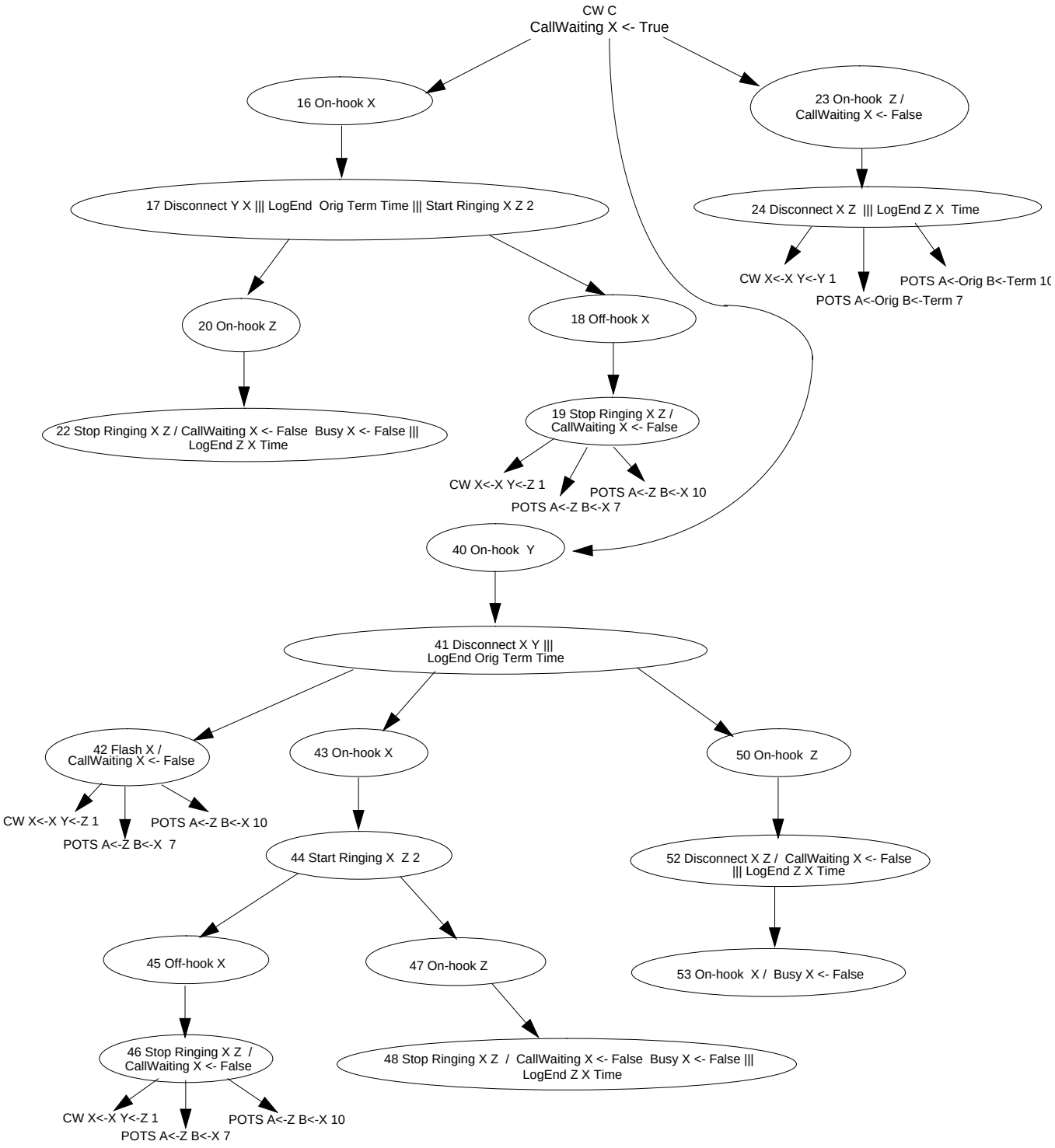
CW A: Call Waiting - Talking to Y, Waiting Z



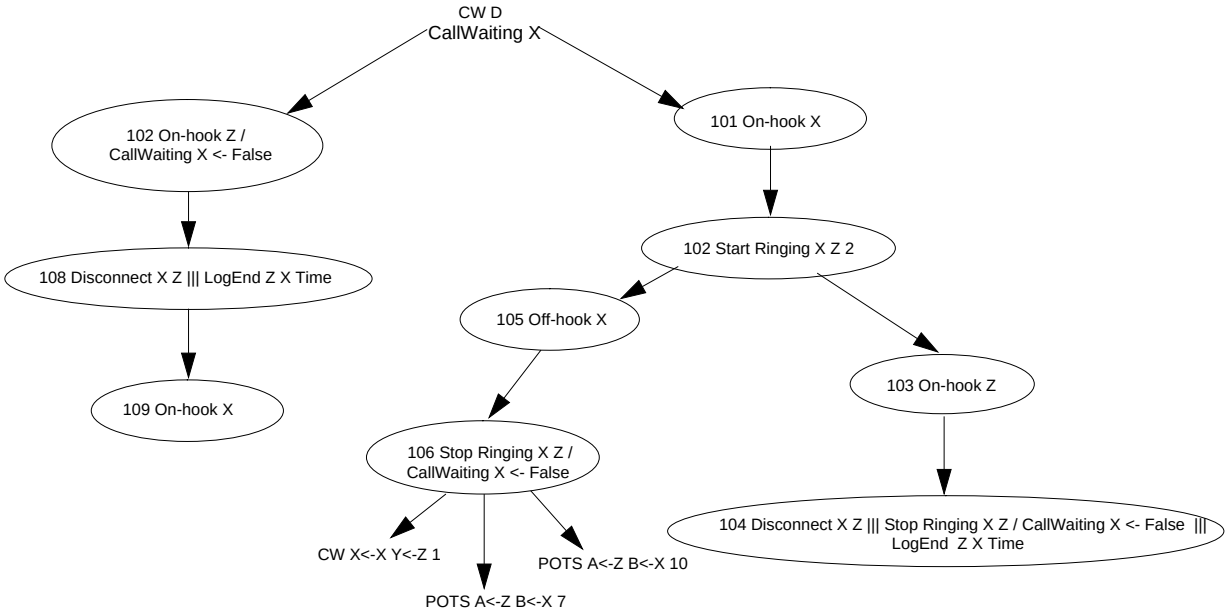
CW B. Call Waiting - Talking to Z, Holding Y



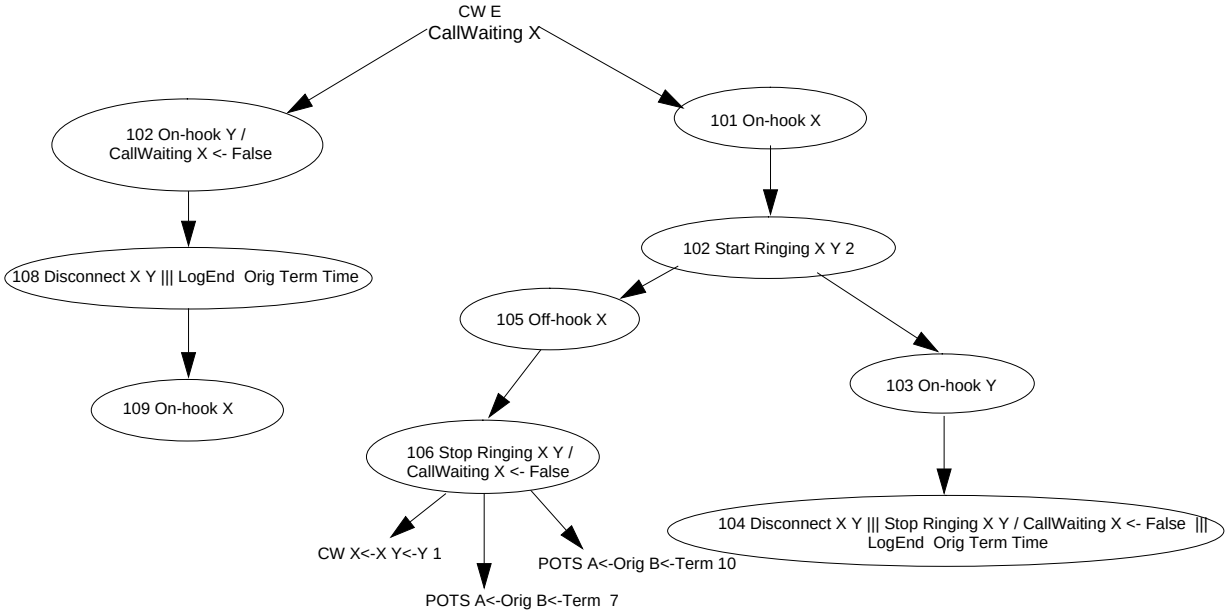
CW C. Call Waiting - Talking to Y, Holding Z



CW D. Call Waiting - Holding Z



CW E. Call Waiting - Holding Y

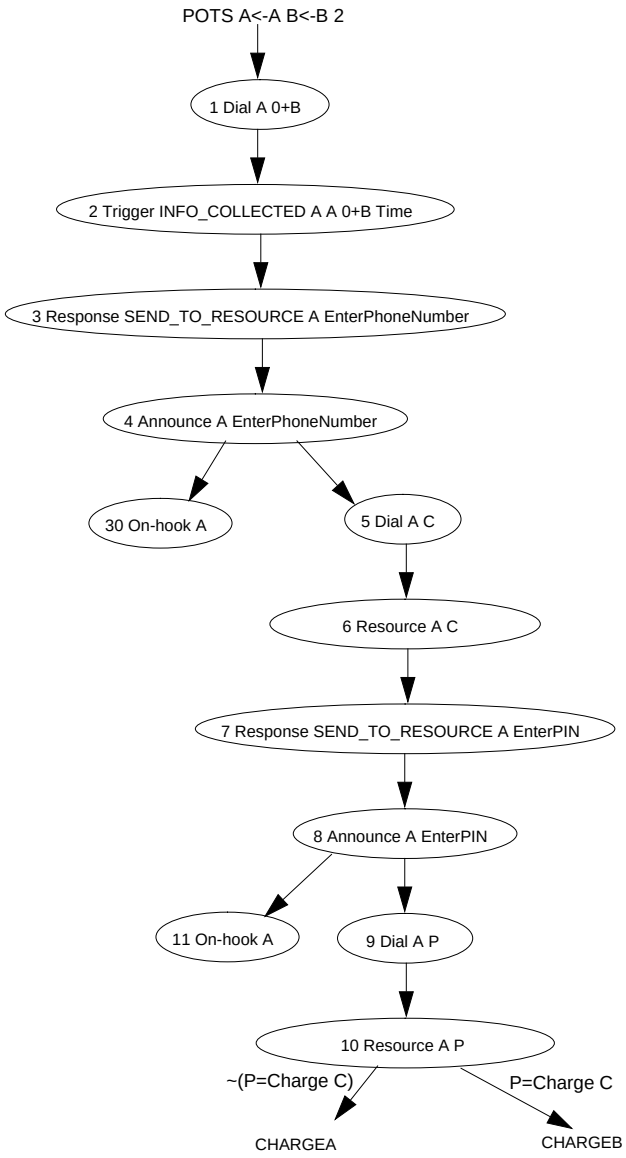


10. Charge Call

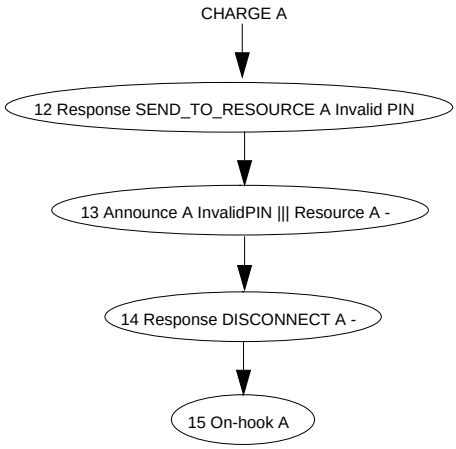
This feature permits a subscriber to charge a call to a different address than the originating address, if the correct PIN is entered

New Variables:

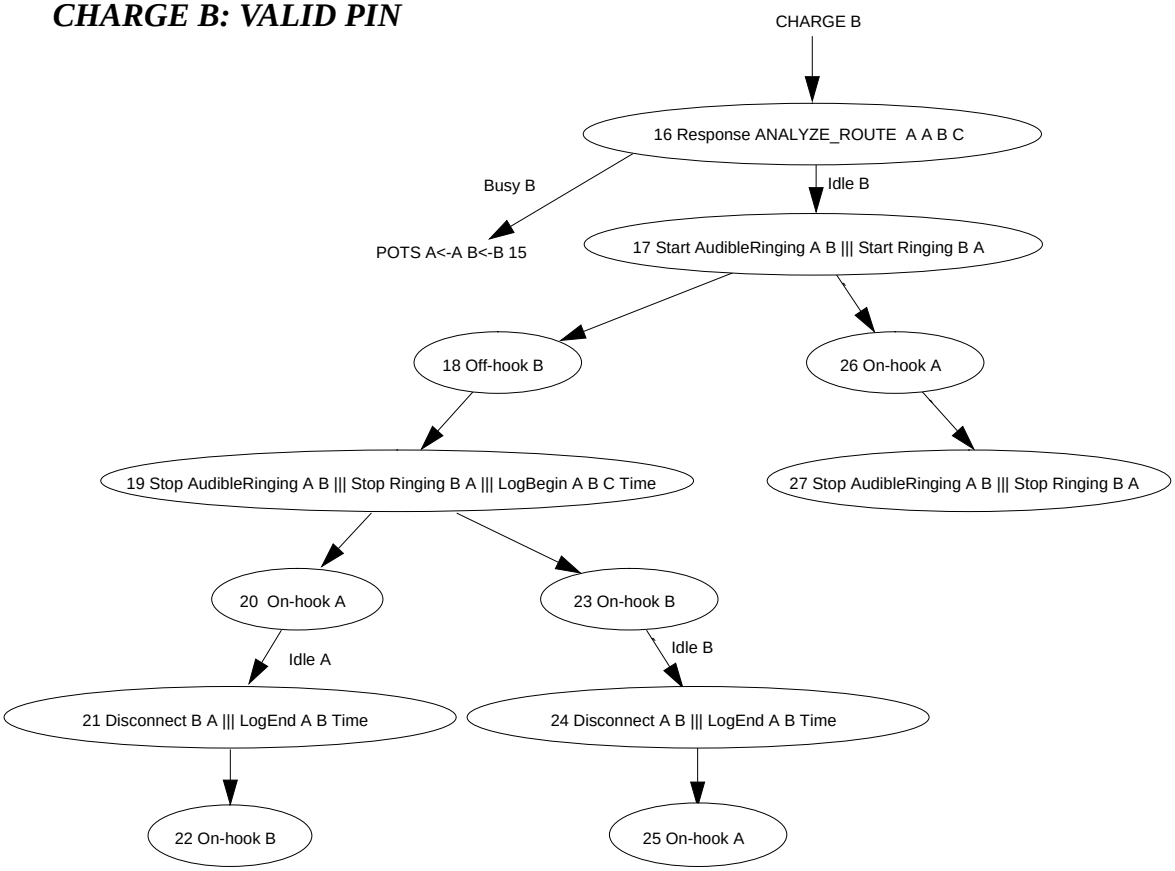
Charge C is the PIN for C



CHARGE A: INVALID PIN

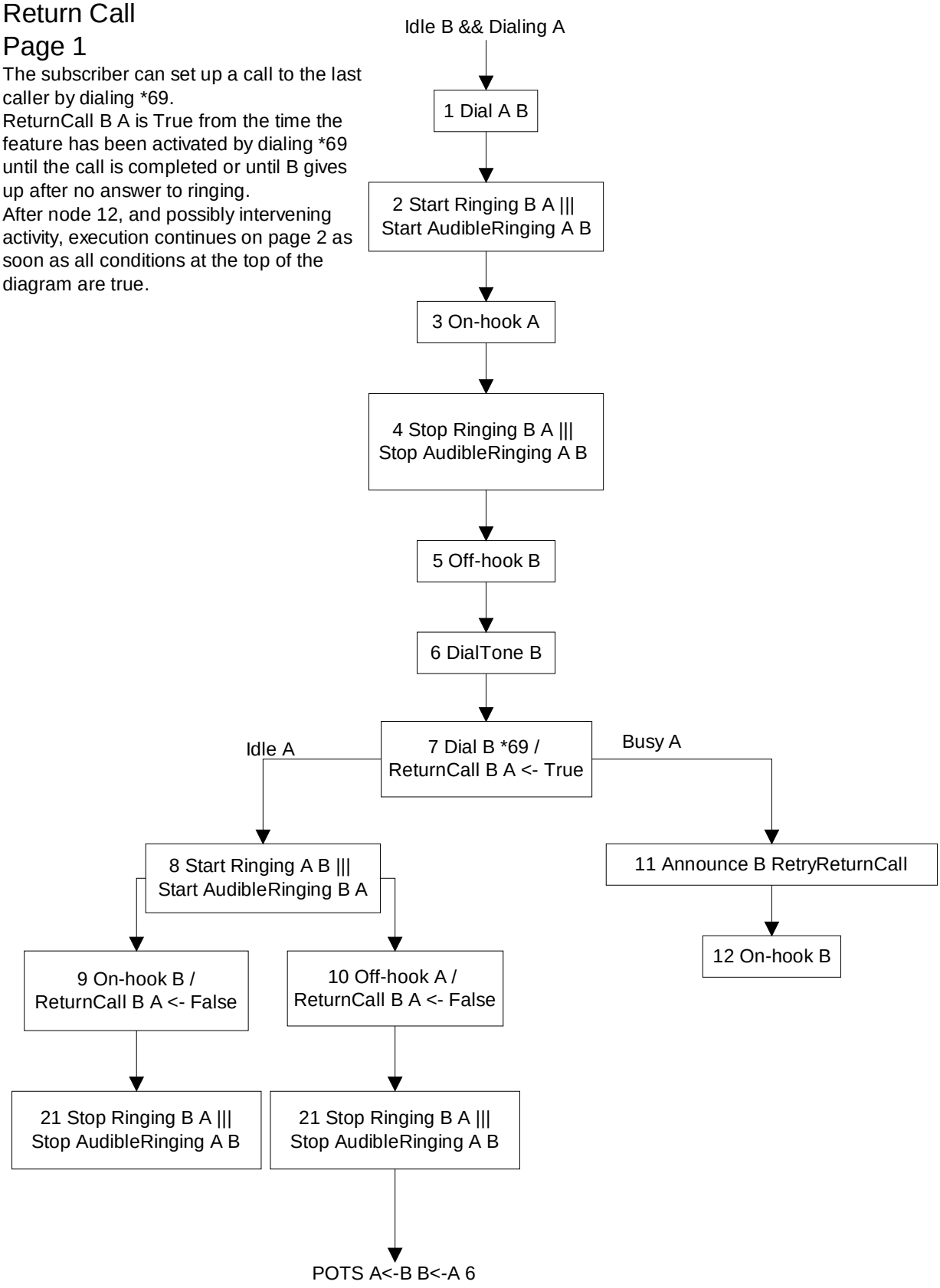


CHARGE B: VALID PIN



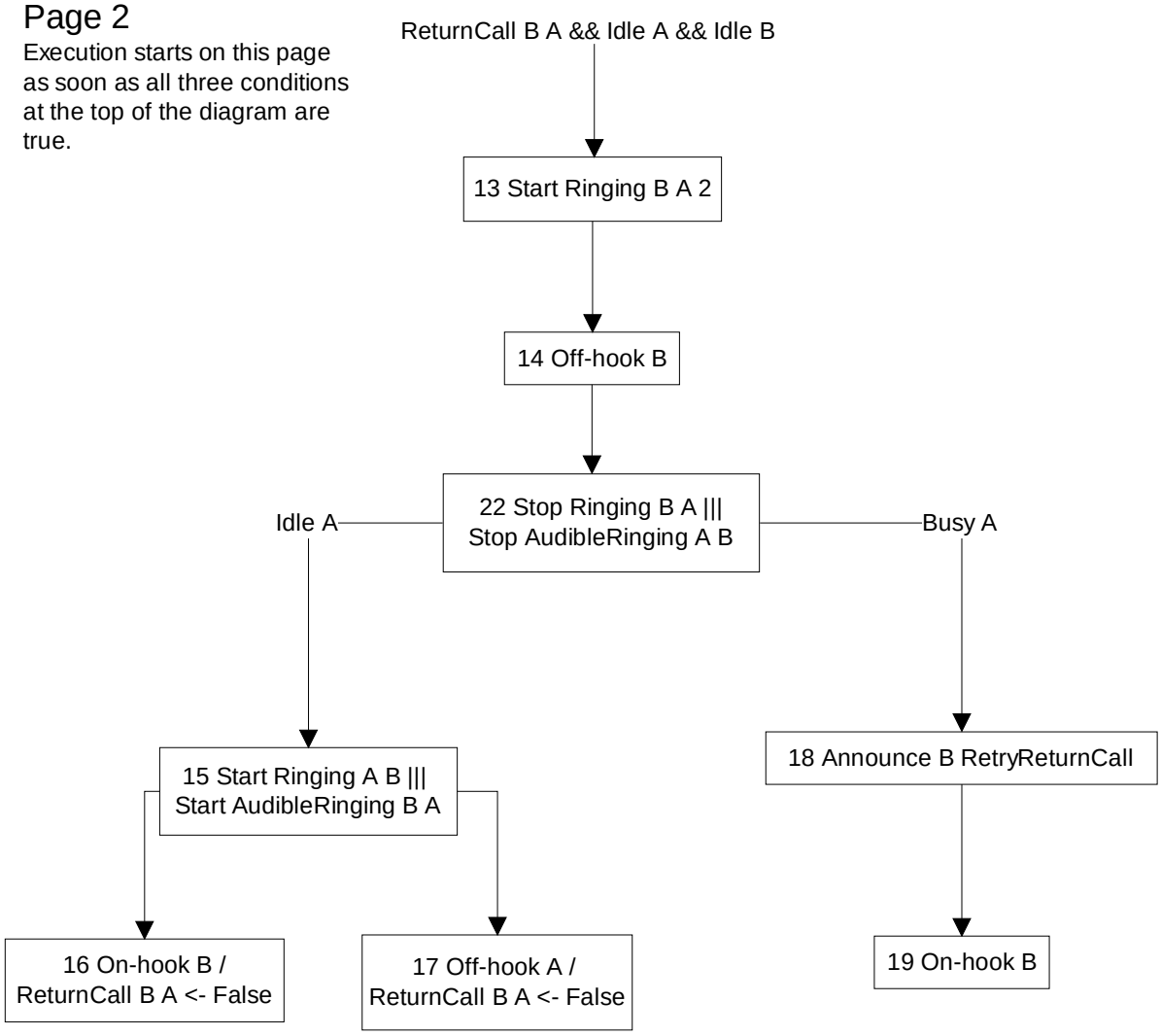
Return Call
Page 1

The subscriber can set up a call to the last caller by dialing *69.
ReturnCall B A is True from the time the feature has been activated by dialing *69 until the call is completed or until B gives up after no answer to ringing.
After node 12, and possibly intervening activity, execution continues on page 2 as soon as all conditions at the top of the diagram are true.



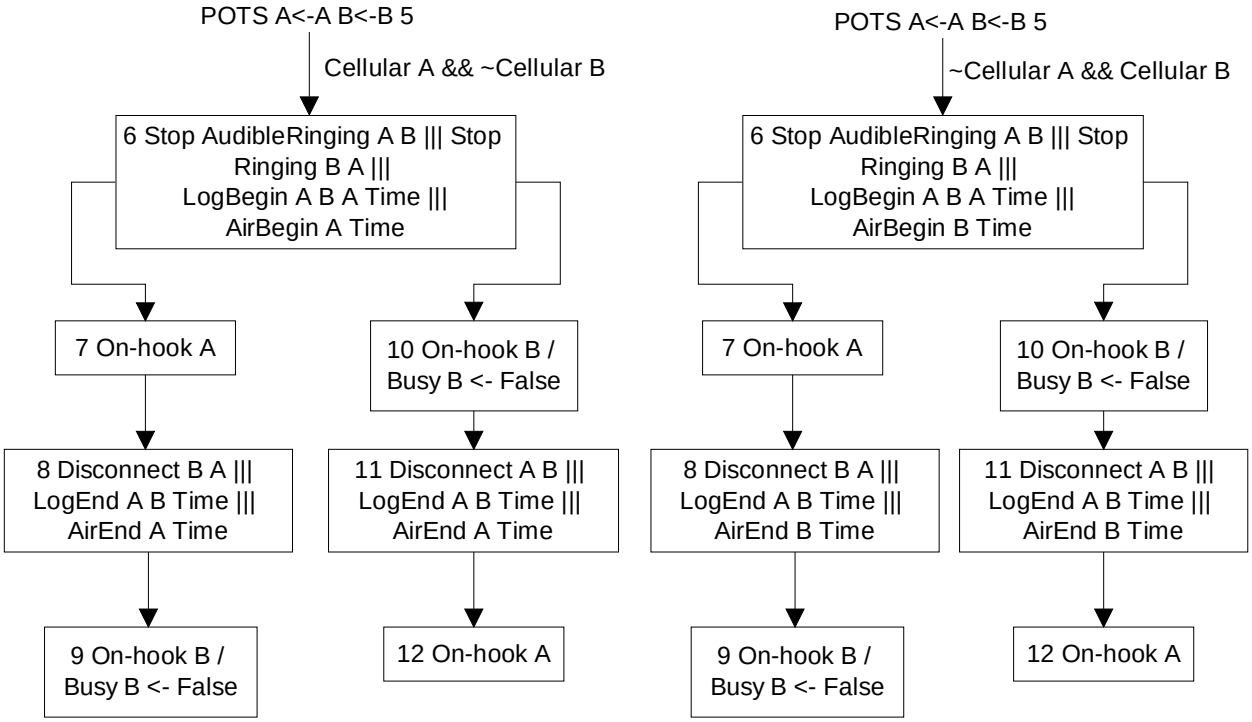
Return Call
Page 2

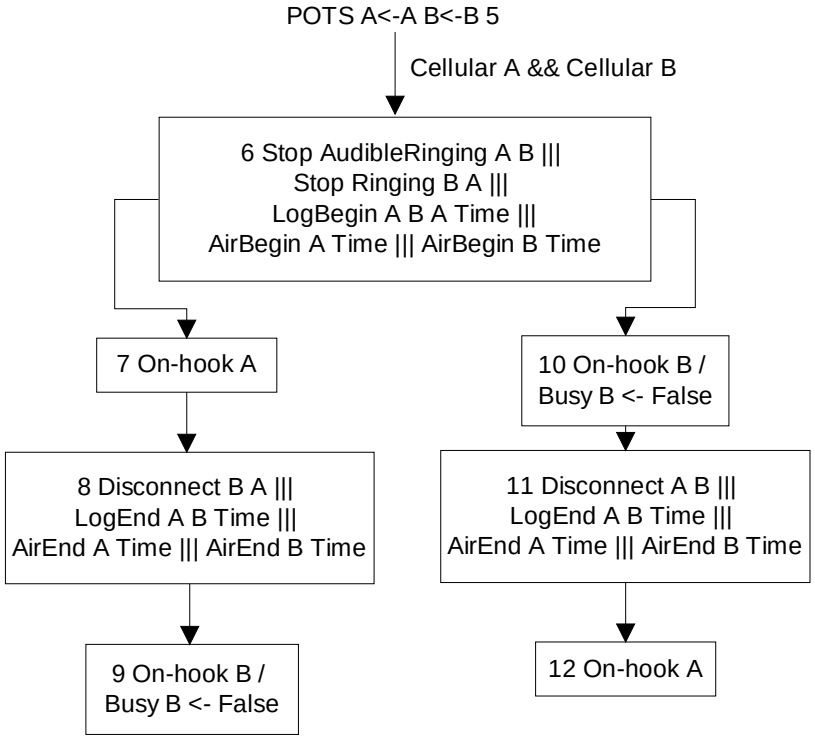
Execution starts on this page as soon as all three conditions at the top of the diagram are true.



Cellular

With cellular service, the cellular subscriber pays for air-time, possibly in addition to normal charges. In our simple model, acellular subscriber pays a fixed fee for each minute (or fraction of a minute) that a call is in progress. Two extra events are introduced: AirBegin <subscriber> <time> and AirEnd <subscriber> <time>, to record the subscriber and the beginning and ending time for each call.
Variables: Cellular A = true if A is a cellular subscriber





Annexe C

Énoncé du deuxième concours de détection d'interactions (FIW'2000)

Second Feature Interaction Contest

Contest Committee
Mario Kolberg[1], Evan H. Magill[1] (chair), Dave Marples[2] and Stephan Reiff[3]

[1]University of Strathclyde,
Department of Electronic and Electrical Engineering,
204 George Street, Glasgow G1 1XW, Scotland, UK
{m.kolberg, e.magill}@eee.strath.ac.uk

[2]Telcordia Technologies
445 South Street 1J226B, Morristown, New Jersey 07960
dmarples@research.telcordia.com

[3]University of Glasgow,
Department of Computing Science,
8-17 Lilybank Gardens, Glasgow G12 8RZ, Scotland, UK
sreiff@dcs.gla.ac.uk

Abstract. Following the First Feature Interaction Detection Contest held in 1998, a Second Feature Interaction Contest is held in conjunction with the Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems. The goal is to provide a simple comparison of different automated tools for feature interaction detection. A preferable outcome will be a catalog of interactions. In addition to the usual two-way interactions, participants are encouraged to consider interactions between three features.

This document contains the contest instructions. The Appendixes A, B, and C contain the Basic Call and feature definitions as state diagrams.

1 Introduction

Many feature interaction handling approaches have been proposed during the previous years. Many of these approaches have been implemented into tools. However in order to show the applicability of the approaches, people had to use known interactions. Also because different people use different features to test their tools a comparision of tools is difficult. The goal of this Feature Interaction Contest is to help people to test their tools more widely and also to provide a benchmark of feature interactions which should ease to compare tools from different sources. Also, the contest provides a means to test tools against the original need behind the research - to detect *unknown* interactions. This is because the features are provided by a different source than the tools.

Usually experiments only focus on finding interactions between two features, that is only pairwise interactions are targeted. In order to progress work towards finding interactions between a multiple set of features (more than two), participants are encouraged to also consider interactions between three features (or three instances of features). It is hoped that this will help to show the true potential of a number of tools and the respective approaches.

As with the previous contest, this will be split in two phases. For phase one, ten features are made available on August 23rd, 1999. The results for phase one need to be submitted by January 28th, 2000. For phase two, specifications of two more features are made available on January 31st, 2000. You are asked to find the additional interactions and hand in your results by February 14th, 2000.

2 The Network

The employed model of the network is depicted in figure 1. The central element is the *Message Switching*. All the customer end devices (telephones in our case) are connected to it. Each line has associated *Control Software* which can be split into *Basic Call Software* and *Features*. All these instances of *Control Software* are connected to the *Message Switching*. Finally, there is a *Billing System* connected to the *Message Switching*. As needed for the billing, a global clock provides the current time which is accessed through a variable *time*.

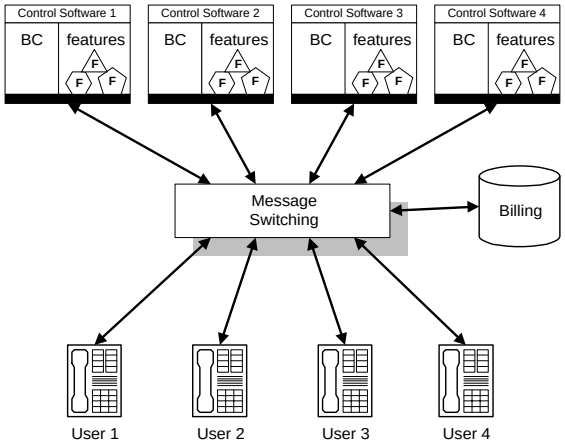


Figure 1: Network Model

All the events and signals exchanged between components of the model are passed through the Message Switching. The following sections describes these signals and their handling in more detail.

3 Events used in the System

The events used in the model can be split into three groups. Firstly, all messages prefixed with an "*o_*" are sent by a Control Software to the Message Switching and the ones prefixed with an "*i_*" are received by a Control Software from the Message Switching. Secondly, events whose names start with "*billing_*" are sent to the Billing Machine. Thirdly, all other messages are used between a user (that is the associated telephone) to the corresponding Control Software. The next three subsections are focussing on the various events.

To obtain a unique message format, all messages consist of four parts: the event, the origin, the destination and an argument. A "-" denotes a constant value for an unneeded argument. The origin and destination might be the same, in which case the message is communicated between the user equipment and the associated Control

Software. The billing messages do not use the origin/destination parameter assignment, details are shown in the following sections.

For the explanations below three variables, X, Y and Z are used. These symbolise three users. However, these variables do not reflect roles of users like originating or terminating party. In other words, X is not necessarily the originating party nor is Y the terminating one.

3.1 User Messages (no Prefix)

A telephone is used as device which provides the interface between the user and the network model. It is important to note that these events are not changed in any way by the Message Switching. All events belonging to this group are listed and explained below, whereby we distinguish between messages initiated by the user and the those initiated by the control software.

User initiated messages:

(offhook, X, X, -) means that subscriber X has gone offhook.

(dial, X, number(Y)) means that subscriber X is dialling the number of subscriber Y.

(onhook, X, X, -) means that subscriber X has gone onhook.

(flash, X, X, -) means that subscriber X flashes, that is goes onhook briefly and then offhook again. Usually this behaviour is created by pressing the flush button on a phone.

Control Software initiated messages:

(dial_tone, X, X, -) means that subscriber X receives a dialtone.

(ringtone, X, X, -) means that subscriber X receives a ringtone.

(busytone, X, X, -) means that subscriber X receives a busytone.

(timeout_tone, A, A, -) means that subscriber X receives a timeout_tone.

(disconnect_tone, X, X, -) means that subscriber X receives a disconnect_tone.

(connect, X, X, -) means that subscriber X's telephone connects to the other party.

(stop_alert, X, X, -) means that subscriber X's telephone stops ringing.

(alert, X, X, -) means that subscriber X's telephone starts ringing.

(announce, X, X, message) means that some announcement is played to X. For instance, X could be notified that the called party is not available but he can leave a message.

(cwtone, X, X, -) means that X receives a Call Waiting tone signalling that the call is currently on hold or that another call attempt is being made (depending on the situation of the user).

(store_msg, X, X, -) means that the message (voice) from the calling party is storred in X's mailbox.

(store_read, X, X, msg) means that the message (voice) from X's mailbox is transmitted to X's telephone.

(store_clear, X, X, -) means that the message stored in X's mailbox is removed.

3.2 Control Software Messages

As said earlier messages commencing with "*o_*" are send by an instance of the control software, those starting with "*i_*" are received. The reader will see that all of these messages come in pairs, which is due to the fact that they are the same message, just with a different direction. The Message Switching converts "*o_*"-messages into "*i_*"-messages.

(o_alert, X, Y, -), (i_alert, X, Y, -) is exchanged between X and Y to notify Y of an call attempt being made. The special case of *(o_alert, X, Y, anonymous)* is used by the calling number delivery blocking and will be converted into *(i_alert, anonymous, Y)* by the message switching to keep the originator private. The message switching keeps track of the association between the anonymous and the return address needed for further message exchange. That is, also the billing machine is always sent the real identity of users and not the *anonymous*.

(o_stopalert, X, Y, -), (i_stopalert, X, Y, -) shows that X is no longer trying to connect to Y

(o_disconnect, X, Y, -), (i_disconnect, X, Y, -) tells the other party about the termination of a call (that was connected)

(o_connect, X, Y, -), (i_connect, X, Y, -) tells the originating party that the called party answers the call

(o_timeout, X, Y, -), (i_timeout, X, Y, -) tells the originating party that the called party does not answer, and that the system timed out the connection attempt.

(o_busy, X, Y, -), (i_busy, X, Y, -) informs about the busy status, in this case the called party is busy.

(o_free, X, Y, -), (i_free, X, Y, -) informs about the busy status, in this case the called party is free.

(o_inform, X, Y, information), (i_inform, X, Y, information) is used to communicate messages. It is used here for features that expect an announcement or other notification to be made at the other side of the call.

(o_msg, X, Y, msg), (i_msg, X, Y, msg) is used for the voicemail feature to transmit messages to be stored.

(o_notify, X, Y, Z), (i_notify, X, Y, Z) is used to inform Y that his call with X has been forwarded to Z.

3.3 Billing Messages

Billing events are sent from the Control Software of one user. In the Message Switching these events are directed to the Billing Machine. Typically, billing related events carry three parameters. The first two specify two users who are involved in some activity for which a charge is created. The global time is passed in as third parameter so that the duration of some activity can be measured and a corresponding charge levied.

(billing_start, X, Y, time) means that user X is started to be charged for a call to user Y. However, these roles can be changed by other events (e.g. billing_reverse which have been sent before). The global time is passed in as a parameter so that the billing system can work out the duration of the call upon reception of the billing_stop event.

(billing_stop, X, Y, time) is the counterpart to the billing_start event. Upon reception of this event the duration of a call can be calculated and user X be charged with the correct amount. Please note that this fixed role of X being charged can be changed by other events which are always be sent before the billing_start event.

(billing_forwarded, X, Y, Z) indicates to the billing system that a call from user X to user Y has been forwarded to user Z. Hence the forwarding can be taken into account for billing.

(billing_reverse, X, Y) notifies the billing system that for the next event (billing_start, X, Y, time) not user X but user Y is charged. Also a surcharge might be added to user Y's bill.

(billing_split, X, Y, factor) notifies the billing system that for the next event (billing_start, X, Y, time) users X and Y are charged according to the factor passed in as third parameter. For instance, user Y will pay 30% of the charge if the factor is 30. In addition the *factor* can be used to code policies like the caller only pays a local call charge and the subscriber the rest etc.

(billing_onhook, X, X, time) means that the billing system gets notified of user X going onhook.

(billing_offhook, X, X, time) means that the billing system gets notified of user X going offhook.

In case the billing system does not receive a *billing_start/billing_end* pair after any of the signals *(billing_forwarded, X, Y, Z)*, *(billing_reverse, X, Y)* or *(billing_split, X, Y, factor)*, and before the next *billing_onhook* the concerned call never took place. In this case the block of billing instructions starting from the *billing_offhook* until the *billing_onhook* is erased from the database in the Billing Machine.

3.4 Some Simplifying Assumptions

For the sake of simplicity of both the model and the instructions some assumptions on the functioning of the overall system have been made. These are listed below.

1. The above messages are all there are.

2. There are no network busy conditions.

3. There is no provisioning or de-provisioning of features. Also, there is no feature activation or de-activation. A subscriber either has a feature or has not.

4. The end-user equipment has a button for "Flash".

5. Transitions between states take no time.

6. All transistions without triggering events do not consume time and happen without time delay. That is, no external events can occur in these states and hence such events are not handled.

7. Tones are stopped with the next triggering event.

4 Notation used for Defining POTS and Features

The Basic Call Model and all features are defined as a set of events on interfaces between network components (telephones, Control Software, Message Switching and Billing Machine). The definitions are expressed in the form of state transition diagrams. Nodes in these diagrams represent states. State transitions are triggered by an incoming event. During the transition some events may be sent to various network elements. The trigger event is typed in *italics* in the diagrams. Outgoing events are typed in normal font. Event sequences involving multiple calls can be interleaved to define global system activity.

As a base a basic two-party POTS diagram is given (cf. appendix A). The diagram consists of two half call models, one representing the originating and the other the terminating side of the call. Beside their names, all states have a unique number assigned (BC 1 - BC 12). These numbers are referred to in the features in order to link to the Basic Call Model. All state names prefixed with an "*o_*" belong to the originating half of the call model whereas the states prefixed with "*t_*" belong to the terminating half. The state *idle* is shared between both half models.

All nodes are connected by directed edges. An edge represents a transition from one state two the next. Clearly, a transition which corresponds to an incoming event can be performed. In addition, in some states transitions are triggered by state internal triggers. An example is the *o_timeout* edge from BC 10 to BC 1, where some kind of timer is assumed that triggers the transition after a certain time has elapsed.

Each feature modifies POTS somehow. As stated previously we do not model service provisioning or activation, so if a subscriber has a feature, that feature is always active. Feature diagrams originate and terminate at some Basic Call node. That is, the behaviour of a feature extends the behaviour of the Basic Call Model. The link from the Basic Call Model to a feature is done by introducing a new transition from a Basic Call state or by *replacing* one or more transitions. That is for the latter, only the transition which is used to link the BCM and the feature is replaced. All other transitions which might be possible from the BCM state are still valid. For instance in the feature *Ringback when Free* the transition triggered by the *(onhook, A, A, -)* event from state BC 5 to BC 1 is *replaced* by a transition to the state RB 1 (check RBlist). In other words the original transition from the Basic Call is *not valid* with that feature. On the other hand, all other transitions possible from state BC 5 are still valid with this feature. The (i_inform, B, A, "ringback") transition from BC 5 to BC 8 introduced

in the Ringback when Free feature is additional to the transitions defined in the BCM for state BC 5.

5 An Outline of the Phase 1 Features

5.1 Call Forwarding on Busy

With this feature all calls to the subscriber's line are redirected to a predetermined number when the subscriber's line is busy. The Billing Machine is notified of the forwarding and it is assumed that the subscriber pays the charge for the forwarded call from his location to the location where the call has been forwarded to (this assumption is part of the Billing Machine and as such the implementation is not considered here). The definition can be found in appendix B.1.

5.2 Teenline

During a pre-set time of day, this feature restricts all outgoing calls from the subscriber's telephone. To place an outgoing call during that time a PIN is required. The definition of the feature can be found in appendix B.2.

5.3 Terminating Call Screening

By this feature the originators of all incoming calls to the subscriber's telephone are screened against a screening list. If an originator of an incoming call matches an entry in the list an announcement is played to that line and the call is cancelled. The definition of the feature can be found in appendix B.3.

5.4 Call Waiting

This feature allows the subscriber to be notified of an incoming call while he is busy in a conversation and to accept the new call be putting the originating call on hold. The subscriber is then able to toggle between these two calls (cf. appendix B.4).

5.5 Three Way Calling

Three way calling allows a user already connected to another user to bring a third partner into the call. Any side of the first call (as long as its the subscriber to 3WC) can setup a connection to the new party, by putting the current partner on hold, connecting to the third side and then joining both lines. The three way call is terminated with any party going onhook. The feature can be found in appendix B.5.

5.6 Reverse Charging

Reverse charging is also known as freephone billing, and allows the subscriber to be charged for all calls in which the subscriber is the terminating party. The definition of the feature can be found in appendix B.6.

5.7 Calling Number Delivery Blocking

Usually the callers identity is available at the terminating side, for evaluation by the callee if required (e.g. caller number display or terminating call screening). This feature blocks the provision of the callers number at the terminating side. The definition of the feature can be found in appendix B.7.

5.8 Ringback when Free

When a call attempt is made to a busy line with this feature subscribed, the caller is informed that he will be called back, as soon as the other person becomes available. Once the subscriber terminates his/her current call a connection to the stored numbers will be established. The definition of the feature can be found in appendix B.8.

5.9 Voice Mail

Voice mail works similar to an answering machine, by offering the possibility to leave messages if the called user is not answering. The stored messages can be listened to by the subscriber. The definition of the feature can be found in appendix B.9.

5.10 Split Billing

Split billing allows to share costs between the partners in a call. A company might provide local call charge lines to customers as a service, in which case the customer (and originator of a call) pays the local charges and the company the rest. The definition of the feature can be found in appendix B.10.

6 An Outline of the Phase 2 Features

6.1 Call Transfer

Call Transfer allows the subscriber to transfer the current call to a 3rd Party. That is, while being in a call the subscriber can put the second party on hold and can setup a call to a 3rd Party. Once the subscriber goes onhook the second party of the first call and the 3rd Party are connected. This is effectively a mid-call call transfer. The feature supports both, the subscriber being the caller or callee in the first call. The definition of the feature can be found in appendix C.1.

6.2 Group Ringing

This feature allows for an incoming call to ring at three phones. The phone which goes offhook first is connected to the calling party. The remaining two phones stop ringing. The definition of the feature can be found in appendix C.2.

7 Submission of Results

This section summarises the deliverables expected from the contest participants. We would like to support participants in focussing on the real work and hence provide some templates which should be filled in by participants. We aimed at keeping the amount

of writing to a minimum while trying to give participants of the workshop maximum gain of the contest results.

7.1 Submissions

At the end of each, Phase 1 and Phase 2, participants are supposed to submit a summary of their results in form of tables, as outlined in section 7.2. Please identify in the tables where interactions occurred, who subscribes to the feature (terminator or originator) and also which states the respective features have been in. The latter is simplified by a consistent numbering of states across the features, and should help us retrace the interaction.

For three way interactions we are only interested in **true** three way interactions, meaning those where the interaction does no longer exist once one feature is removed. In other words, the *same* interaction cannot be occur between any pair of the involved features. For both phases a two to three page document should accompany the table, describing your method, tool and optionally selected interaction scenarios in more detail.

Further, in order to stimulate a lively discussions between workshop delegates, participants are requested to create a poster for exhibition on the conference, giving an overview of the approach, tool and results. For details on posters please refer to the FIW'00 calls for papers and participation.

The participation in the contest does not restrict the contestants in any way to submit papers to the workshop itself. This includes papers on their approach and tool used. This is not part of the contest and will have no impact on the valuation of the results. Please refer to the guidelines given in the call for papers for FIW.

Summarising, there is one table to be filled in (in case you do both three way and two way interactions, there are two), three pages to write and a poster to create. This keeps the submissions to a minimum and allows the participants to spent their time on the more important issue of detecting interactions.

7.2 Table Outlines for Concise Submission of Results

Suggestions for the tabular representation of the results are given for both two-way interactions (Table reftab2way) and three-way interactions (Table 7.2). The table outline for three-way interactions is not complete and is intended to provide only the general outline.

	CFB	CW	3WC	TL	TCS	RC	CNDB	RBF	VM	SB
CFB										
CW										
3WC										
TL										
TCS										
RC										
CNDB										
RBF										
VM										
SB										

Table 1: Suggested Table for Two-Way Interactions

		CFB	CW	3WC	TL	TCS	RC	CNDB	RBF	VM	SB
CFB	CFB										
	CW										
	3WC										
	TL										
	TCS										
	RC										
	CNDB										
	RBF										
	VM										
	SB										
		CFB	CW	3WC	TL	TCS	RC	CNDB	RBF	VM	SB
CW	CFB										
	CW										
	3WC										
	TL										
	TCS										
	RC										
	CNDB										
	RBF										
	VM										
	SB										
...	...										
		CFB	CW	3WC	TL	TCS	RC	CNDB	RBF	VM	SB
VM	CFB										
	CW										
	3WC										
	TL										
	TCS										
	RC										
	CNDB										
	RBF										
	VM										
	SB										
		CFB	CW	3WC	TL	TCS	RC	CNDB	RBF	VM	SB
SB	CFB										
	CW										
	3WC										
	TL										
	TCS										
	RC										
	CNDB										
	RBF										
	VM										
	SB										

Table 2: Suggested Table for Three-Way Interactions

7.3 Reminder of Deadlines

There are three deadlines for submission for contest participants; the results of phase one should be submitted by 28th January 2000 and the results of phase two by 14th February 2000. Posters are due on 17th April 2000. For details on these deadlines please refer to Section 7.1.

A Basic Call Definition

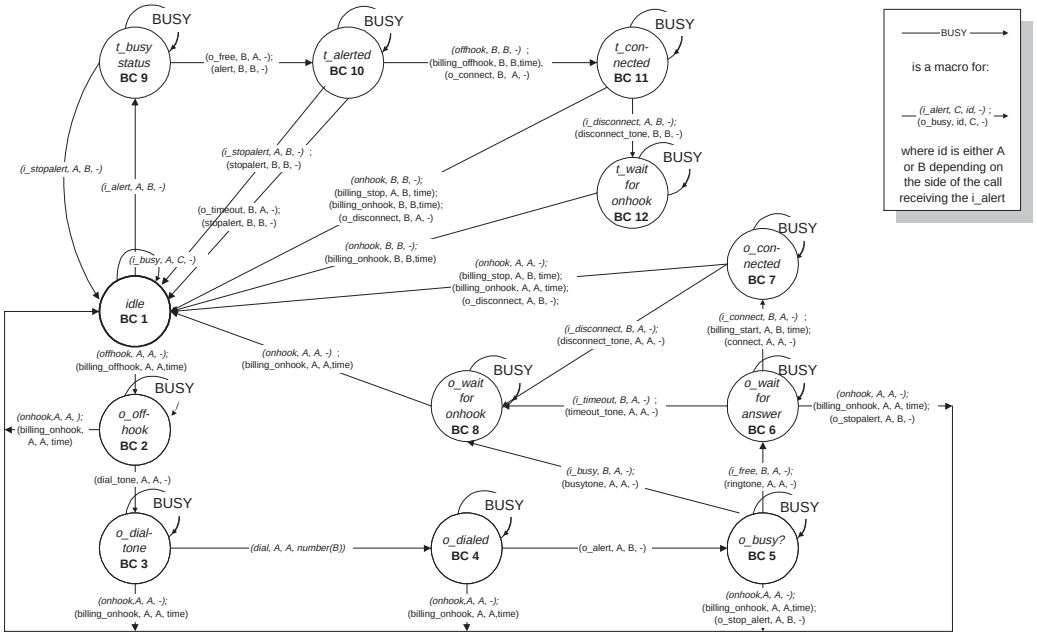


Figure 2: Basic Call Model

B Phase One Feature Definitions

B.1 Call Forwarding on Busy

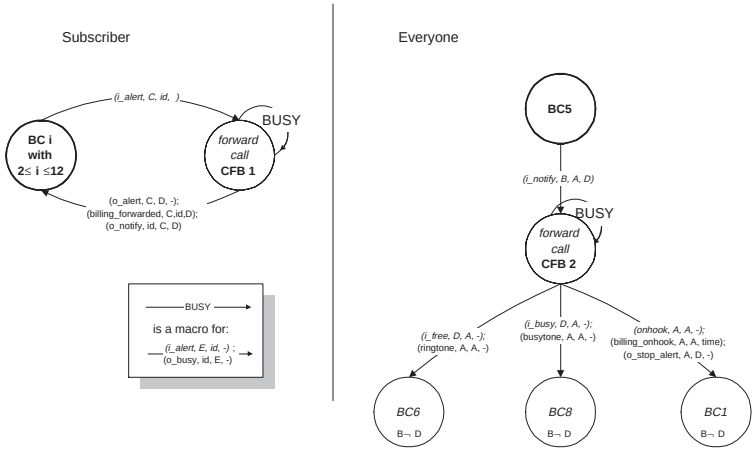


Figure 3: Call Forwarding on Busy Model

B.2 Teenline

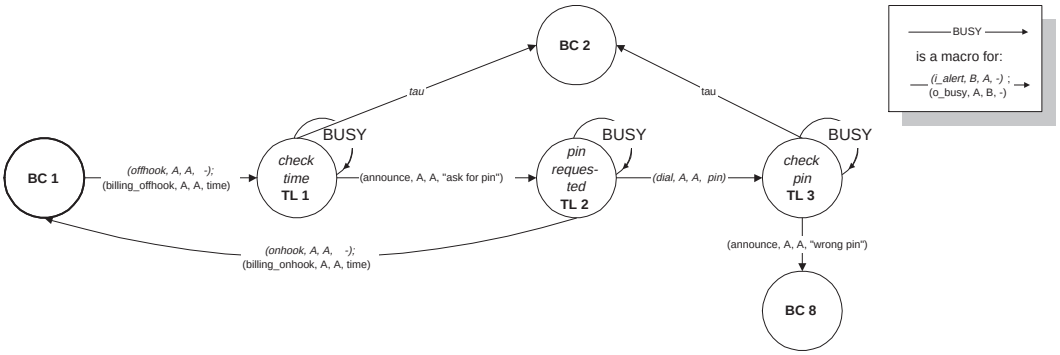


Figure 4: Teenline Model

B.3 Terminating Call Screening

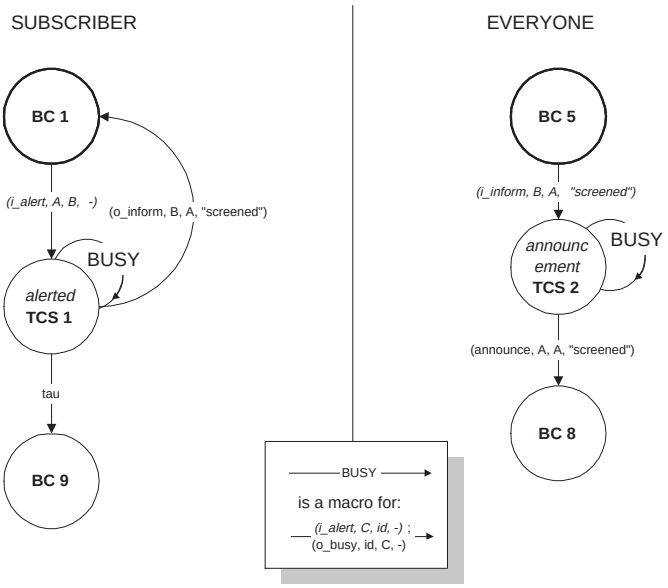


Figure 5: Terminating Call Screening Model

B.4 Call Waiting

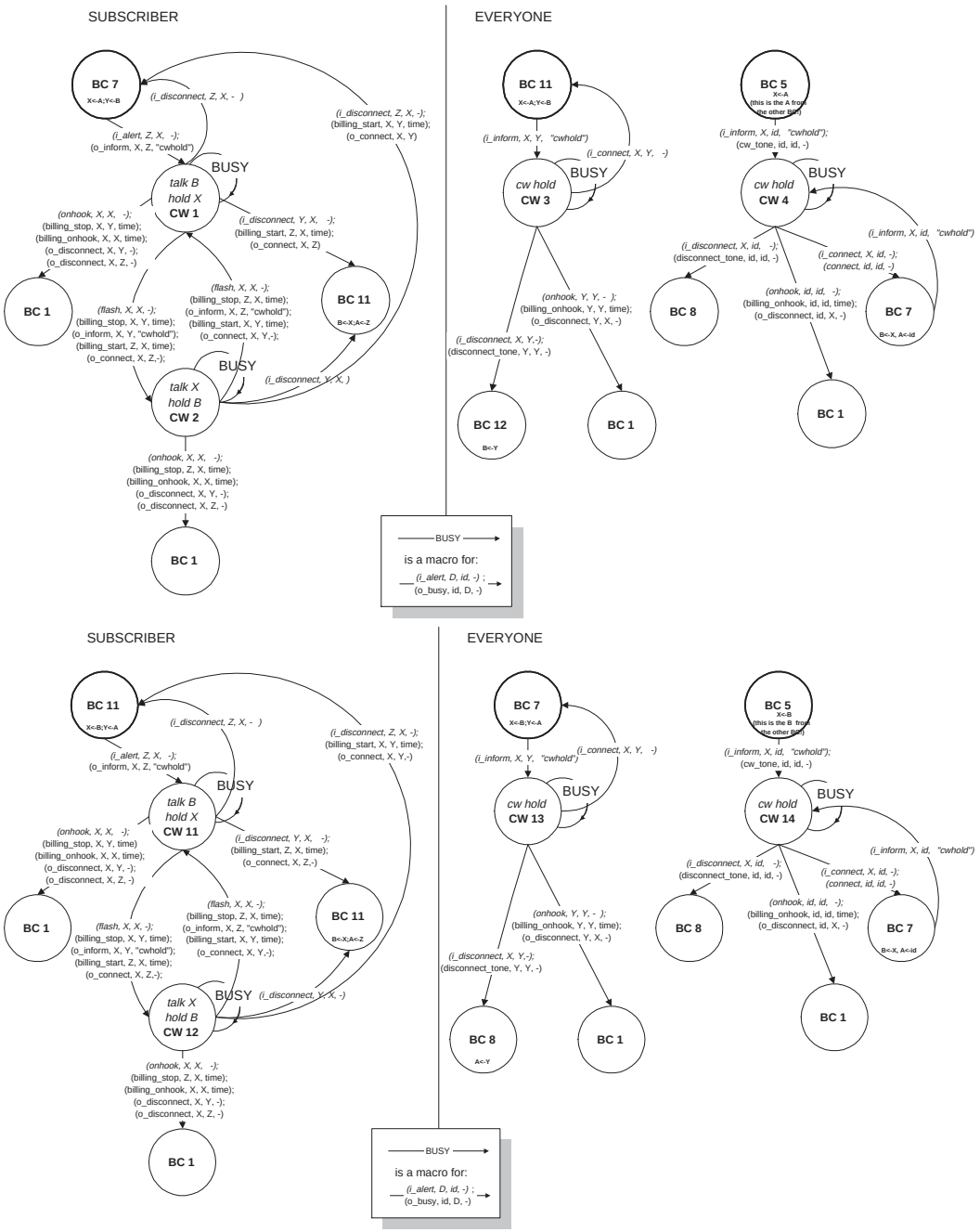


Figure 6: Call Waiting Model

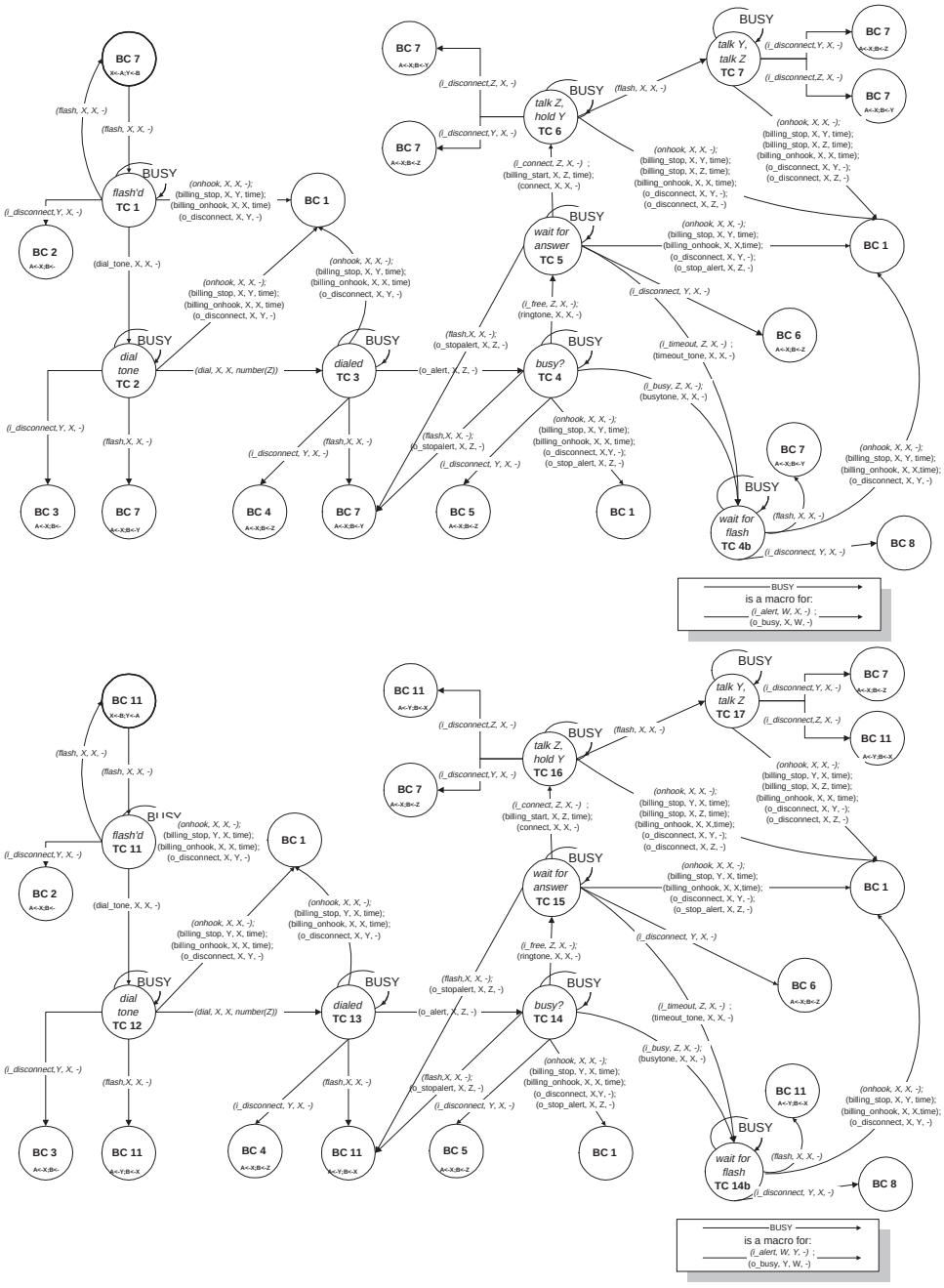


Figure 7: Three Way Calling Model

B.6 Reverse Charging

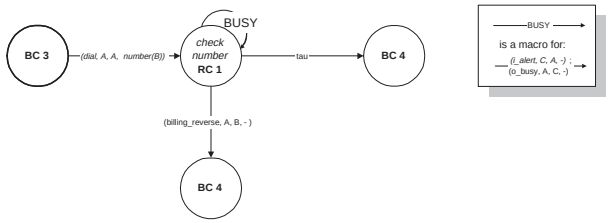


Figure 8: Reverse Charging Model

B.7 Calling Number Delivery Blocking

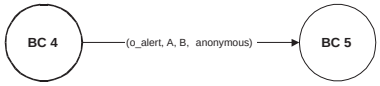


Figure 9: Calling Number Delivery Blocking Model

B.8 Ringback when Free

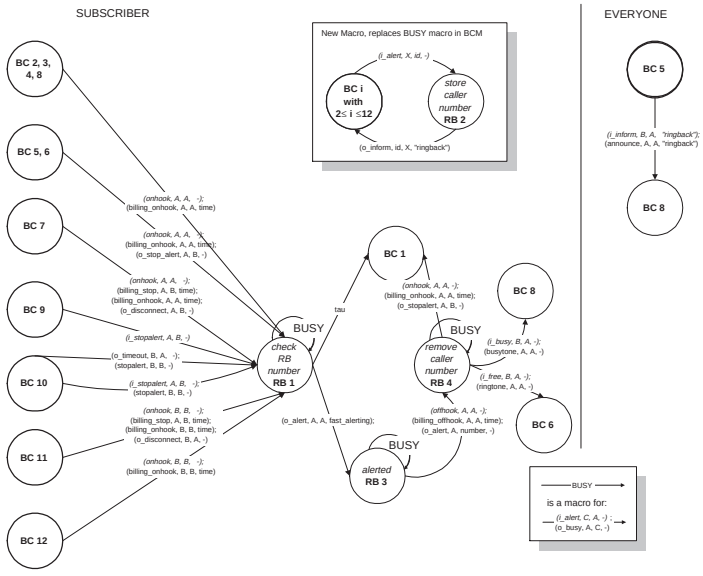


Figure 10: Ringback when Free Model

B.9 Voice Mail

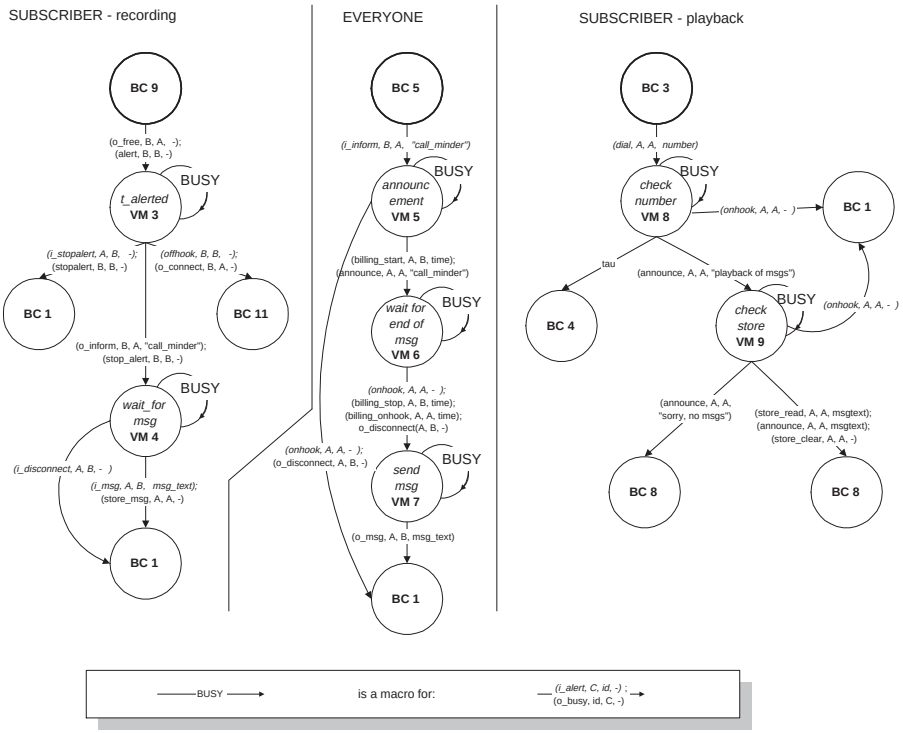


Figure 11: Voice Mail Model

B.10 Split Billing

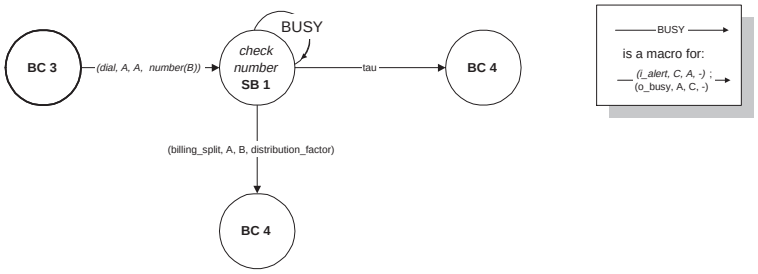


Figure 12: Split Billing Model

C Phase Two Feature Definitions

C.1 Call Transfer

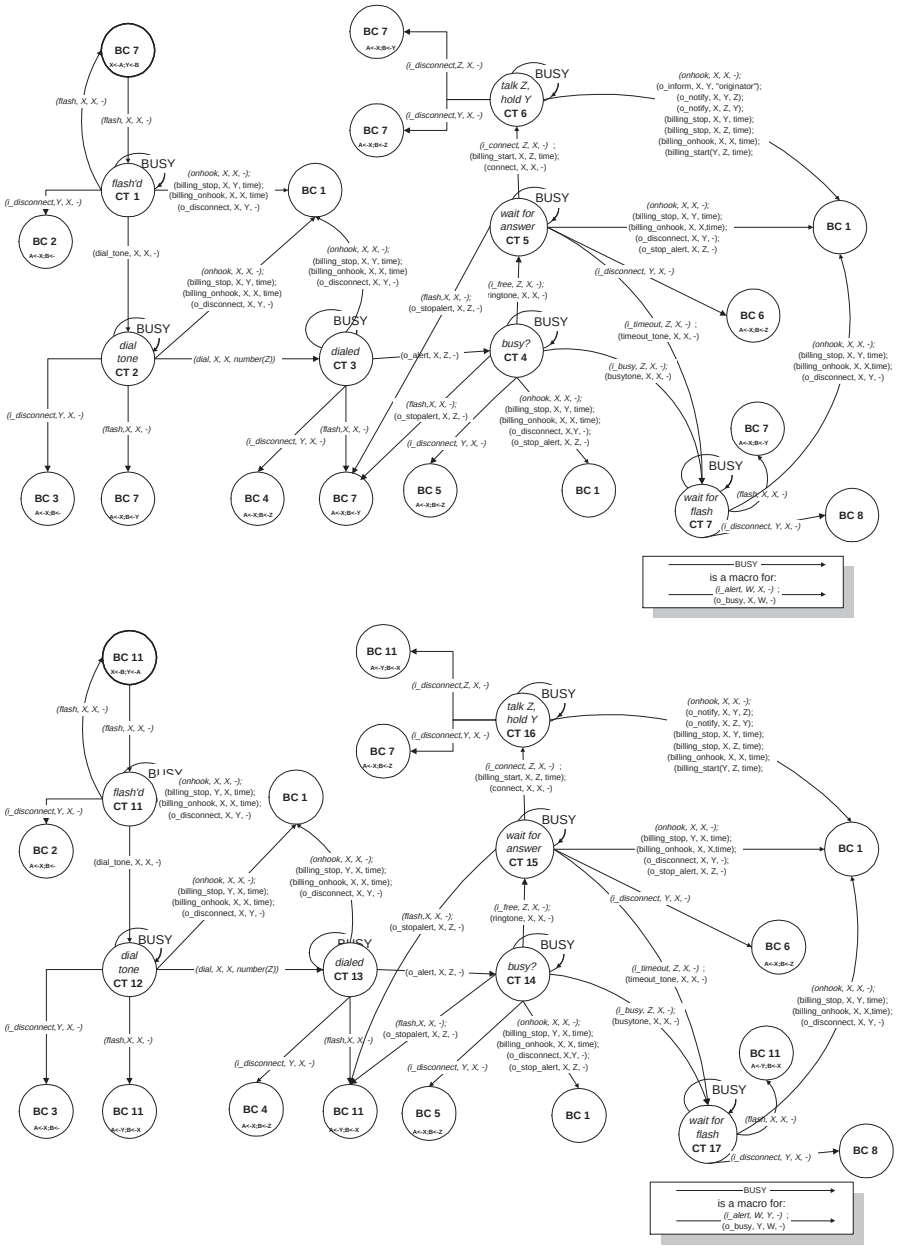


Figure 13: Call Transfer Model - Subscriber

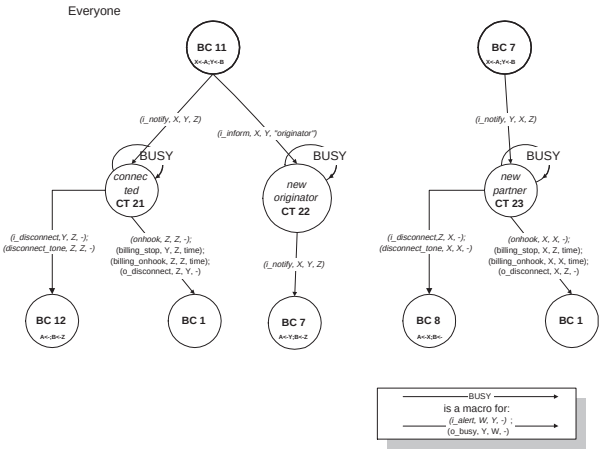


Figure 14: Call Transfer Model - Everyone

C.2 Group Ringing

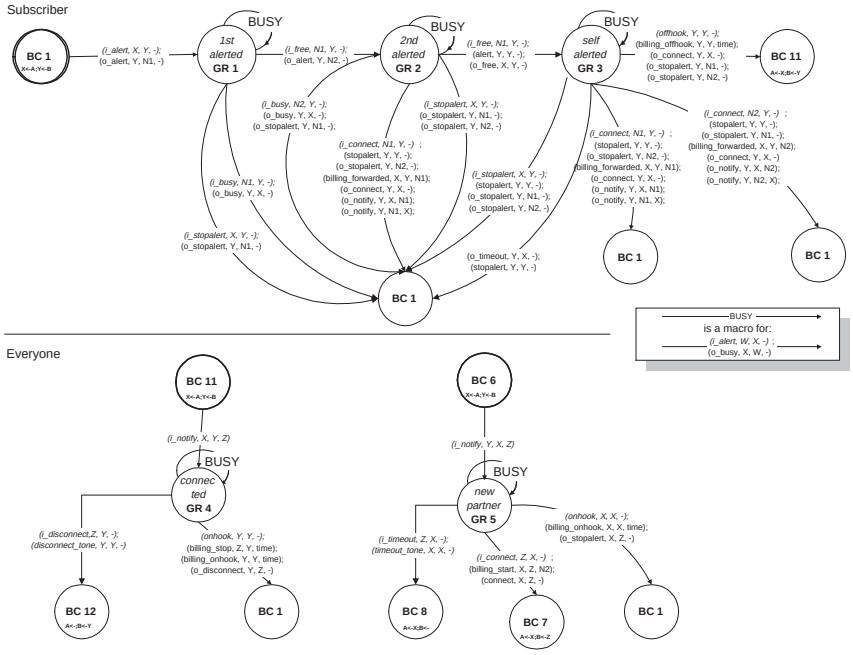


Figure 15: Group Ringing Model

Annexe D

Propriétés de services du concours FIW'98

Cette annexe est un extrait traduit du rapport relatant notre contribution au premier concours de détection d'interactions, organisé dans le cadre de la conférence *Feature Interaction Workshop 1998*[1]. Nous décrivons ici l'ensemble des attentes de l'utilisateur que nous avons répertorié et utilisé afin de valider fonctionnellement les services.

Définition d'une *attente de l'utilisateur*
Les attentes sont un moyen de décrire ce que l'usager est en droit d'espérer d'un service lorsqu'il s'y abonne, en termes de fonctionnement satisfaisant et correct. Dans la pratique, ces attentes sont extraites essentiellement de la description informelle fournie pour chaque service, et une large part est laissée à l'interprétation de celle-ci. Elles sont donc relativement subjectives, et peuvent être discutées. L'objectif est d'illustrer par leur biais les souhaits généraux des usagers.

Les attentes de l'usager sont exprimées dans notre modèle par des propriétés de logique temporelle écrites en LUSTRE [21]. Cependant, pour en faciliter la lecture, le formalisme adopté est un pseudo-Lustre. En particulier, nous évitons l'utilisation de la notation préfixée (e.g., implies(A,B), once_from_to(A,B,C)) et préférons un style infixé, plus intuitif (A $\Rightarrow$ B, ONCE A FROM B TO C).

Dans la suite, x, y, z sont des variables désignant un usager quelconque. "-" signifie que la valeur de la variable est sans importance (*don't care* value).

Les prédicats mentionnés mais non-explicités ici sont issus de l'énoncé du concours FIW'98. Rappelons tout de même que les prédicats indiquant l'émission d'une sonnerie sur un poste disposent de deux paramètres : le premier indique le poste sur lequel est émis la sonnerie, tandis que le second désigne le poste à l'origine de la sonnerie.

1. La version originale du rapport cité, ainsi que les soumissions des "finalistes" du concours sont disponibles à l'URL suivante : http://www-db.research.bell-labs.com/user/nancyg/Contest/Submissions.html

D.1 Définitions préliminaires

Nous donnons ici quelques définitions utiles au lecteur pour comprendre la description des attentes.

Opérateurs temporels

▷ *pre* est un opérateur LUSTRE , tel que *pre lustre-predicate* désigne la valeur de *lustre-predicate* à l'instant précédent (cf. section 3.1).

▷ Opérateurs usuels de logique temporelle

```
node between(A, B : bool) returns (X : bool)
let
   X =  A -> (pre X and not B) or A;
tel

node once_from_to(A, B, C : bool) returns (X : bool);
let
   X = implies(C, once_since(A,B));
tel

node once_since(A, B : bool) returns (X : bool);
let
   X = if B then A else (true -> (A or pre X) );
tel

node always_since (A, C: bool) returns (X: bool);
let
   X = if C then A
       else if after(C) then (true -> (A and pre(X)))
       else true;
tel

node always_from_to (A, B, C : bool) returns (X: bool);
let
   X = implies (after(B), always_since(A,B) or once_since(C,B));
tel

node implies(A, B: bool) returns (X: bool);
let
   X = if A then B else true;
tel

node after(X : bool) returns (A : bool);
let
```

```
   A = false -> pre(X or A);
tel
```

▷ Opérateur "THEN_PROVIDED"
Par souci de lisibilité, nous utilisons dans les pages qui suivent une méta-opération "THEN_PROVIDED" afin d'indiquer une relation temporelle entre plusieurs prédicats LUSTRE.
"THEN_PROVIDED(A,B,C)" (ou "A THEN B PROVIDED C") prend la valeur *vrai* si B a été observé après A, à la condition que C ait été satisfait continuellement entretemps.

```
node THEN_PROVIDED(A,B,C : bool) returns (res : bool)
let
   res= B and between(A,pre B) and not once_since(not C,A);
tel
```

Lorsque la donnée de la condition d'intervalle C n'est pas indispensable à la compréhension de la propriété, nous l'ignorons et écrivons simplement "A THEN B". Il faut néanmoins noter que cette expression est un abus de notation, destiné seulement à faciliter la lecture des propriétés.

▷ Définition du prédicat *ConnectRequest*
Nous avons constaté que la notion de demande de connexion est une donnée très importante pour l'énoncé de nombreuses propriétés. En effet, beaucoup de services se déclenchent ou deviennent actifs lorsqu'une connexion est demandée. Cette notion n'étant pas présente dans la spécification originelle, nous avons choisi de définir un prédicat l'explicitant. *ConnectRequest(A,B)* sera *vrai* lorsqu'il existe une tentative de connexion entre A et B, quelle que soit la suite donnée à cette tentative.

{ ConnectRequest(x,y) ⇔ x requests a connection to y. }

▷ ConnectRequest(x,y) = between(StartRinging(y,x),StopRinging(y,x)))
or (Dial(x,y) and between(DialTone(x),On-hook(x) or LineBT(x)))

D.2 Propriétés du service de base

Note de rappel : Il existe deux niveaux d'interaction dans notre modèle. Le premier entre services supplémentaires inclut les interactions néfastes qu'il faut révéler et corriger. Le second existe entre un service supplémentaire et le service de base. Les interactions présentes à ce niveau sont pour la plupart volontaires et désirées.

Il est néanmoins utile de les révéler, afin de s'assurer que le service supplémentaire affecte bien le service de base de la manière prévue. Les propriétés du service de base (ou *propriétés du système*), décrites ci-dessous, sont utilisées dans ce but.
Ces propriétés ont également leur utilité lorsqu'il s'agit de confronter plusieurs services supplémentaires. Dans cette situation, elles ont pour rôle de garantir la cohérence de la manière suivante : si chacun des services satisfait isolément les propriétés du service de base, mais que la composition de ces services conduit à la mise en défaut de l'une d'elles, une interaction

sera révélée, quand bien même les propriétés spécifiques de chaque service sont satisfaites (cf. section 2.8).

- L'appelant est facturé pour les appels qu'il initie.
  (POTS1)▷ LogBegin(x,y,z) ⇒ x=z

- La facturation d'un appel prend fin dès qu'un des correspondants raccroche.
  (POTS2)▷ LogBegin(x,y,-) THEN (On-hook(x) or On-hook(y)) ⇒ LogEnd(x,y)

- Au plus une sonnerie ou tonalité est émise à la fois sur chaque poste.

- Au plus un message est affiché à la fois sur chaque poste.

- Une demande de connexion vers un poste libre doit conduire ce poste à sonner.
  (POTS3)▷ ConnectRequest(y,x) and pre Idle(x) ⇒ StartRinging(x,y)

- Les variables de correspondance sont cohérentes : si la variable de correspondance de X est valide (i.e., désigne un usager), la variable de correspondance de l'usager désigné est valide également.
  (POTS4)▷ Valid($party_A$) ⇒ Valid($party_{party_A}$)

- La facturation est cohérente : à chaque début de facturation correspond une fin de facturation.
  (POTS5)▷ not ONCE(LogBegin(x,-,-) or LogBegin(-,y,-)) FROM LogBegin(x,y,-) TO LogEnd(x,y)

D.3 Propriétés des services

Pour chaque service, nous donnons ici une description informelle des attentes de l'usager ainsi que la description LUSTRE correspondante.

L'état physique de chaque poste est représenté dans le modèle par un ensemble de variables globales à la signification intuitive :

- *Idle.* Le combiné est raccroché, le poste ne sonne pas ;

- *Dialing.* Le combiné est décroché, le poste émet la tonalité invitant à composer ;

- *Alerting.* Le poste a effectué une demande de connexion et attend que son correspondant réponde ;

- *Ringing.* Le poste reçoit une demande de connexion ;

- *Talking.* L'usager du poste est en communication avec un correspondant ;

- *Exception.* Le poste est dans un état d'erreur, par exemple après avoir demandé une connexion vers un poste déjà occupé.

À tout instant, il est donc possible d'avoir la connaissance de l'état de n'importe quel téléphone, ce qui permet de décrire simplement les attentes associées à chaque service.

Nous représentons par *State(x)*, la condition selon laquelle le poste de l'usager x est dans l'état *State*.

Call Forwarding Busy Line (CFBL)
(CFBL1) *Le service CFBL permet de rediriger les appels parvenant à son souscripteur lorsque celui-ci est occupé.*
(CFBL2) *Lorsqu'un appel est redirigé, la facturation de l'appelant ne change pas. En revanche, le souscripteur redirigeant l'appel est facturé pour la redirection.*

Définition de prédicats:

- BLForward(x) identifie le poste vers lequel les appels vers x sont redirigés lorsque ce dernier est occupé.
- CFBLsub(x) $\Leftrightarrow$ x est un souscripteur au service CFBL.
- La définition de ConnectRequest est complétée comme suit :
  ConnectRequest(x,y) = (ConnectRequest(x,z) and BLForward(z)=y) or (CRpots(x,y))
  où CRpots est la définition de base de ConnectRequest.

Propriétés:
(CFBL1) $\triangleright$ CFBLsub(y) and ConnectRequest(x,y) and pre Busy(y) $\Rightarrow$ ConnectRequest(x,z) and not StartRinging(y,x)
(CFBL2) $\triangleright$ (CFBLsub(y) and Dial(x,y) and pre Busy(y) and not pre Busy(BLForward(y)) THEN Off-hook(BLForward(y))) $\Rightarrow$ LogBegin(x,y,x) and LogBegin(y,BLForward(y),y)

Calling Number Delivery (CND)
(CND1) *Le service CND permet à son souscripteur de voir affiché sur son poste l'identité de l'usager appelant.*
(CND2) *Le souscripteur au service CND est informé du numéro de l'appelant dès que son poste sonne.*

Définition de prédicats:
CNDsub(x) $\Leftrightarrow$ x est un souscripteur au service CND.

Propriétés:
(CND1) $\triangleright$ CNDsub(x) and ConnectRequest(z,x) and pre Idle(x) $\Rightarrow$ Display(x,z)
(CND2) $\triangleright$ CNDsub(x) $\Rightarrow$ Display(x,z) $\Leftrightarrow$ StartRinging(x,z)

Freephone Billing (FB)
(FB1) *Le service FB permet à son souscripteur de prendre en charge le coût des appels lui parvenant ; l'appelant n'est plus facturé pour un tel appel.*

Définition de prédicats:
FBsub(x) $\Leftrightarrow$ x est un souscripteur au service FB.

Propriétés:
(FB1) $\triangleright$ FBsub(y) and StopAudibleRinging(y,x) and Talking(x,y) $\Rightarrow$ LogBegin(x,y,y) and not LogBegin(x,y,x)

La propriété ci-dessus indique que lorsque y répond à une demande de connexion de x, l'appel de x vers y est facturé à ce dernier.

Freephone Routing (FR)
(FR1) *Le service FR permet de rediriger les appels parvenant à son souscripteur vers divers postes suivant l'heure de la journée.*
(FR2) *Le souscripteur paie la partie de l'appel correspondant à la redirection, tandis que l'appelant est facturé normalement pour son appel.*
(FR3) *Le poste du souscripteur ne sonne pas lorsque les conditions de redirection sont satisfaites.*

Définition de prédicats:

- FRsub(x) $\Leftrightarrow$ x est un souscripteur au service FR.
- T est une variable représentant l'heure de la journée.
- y=FRredirect(x,t1,t2) $\Leftrightarrow$ les appels vers x sont redirigés vers y entre les heures t1 et t2.
- La définition de ConnectRequest est complétée comme suit :
  ConnectRequest(x,y) = (CRpots(x,z) and FRredirect(x,z,t1,t2)=y and t1$\leq$T<t2) or (CRpots(x,y))
  où CRpots est la définition de base de ConnectRequest.

Propriétés:
(FR1) $\triangleright$ FRsub(y) and ConnectRequest(x,y) and FRredirect(x,y,t1,t2)=z and not z=noAddress and pre not Busy(z) and (t1$\leq$ T<t2) $\Rightarrow$ ConnectRequest(x,z) and not StartRinging(y,x)
(FR2) $\triangleright$ FRsub(y) and FRredirect(x,y,t1,t2)=z and Dial(x,y) THEN StopAudibleRinging(x,z) and Off-hook(z) $\Rightarrow$ LogBegin(x,y,x) and LogBegin(y,z,y).
(FR3) $\triangleright$ FRsub(y) and FRredirect(x,y,t1,t2)=y and t1<= T <t2 $\Rightarrow$ not (StartRinging(y,x)

IN Teen Line (TL)
(TL1) *Un souscripteur au service TL ne peut initier d'appel lorsque certaines conditions horaires sont remplies, à moins de composer un code personnel secret correct (PIN) après avoir décroché.*
(TL2) *Durant la plage horaire de restriction des appels, le souscripteur TL doit entendre une annonce "AskForPin" après avoir décroché alors que son poste ne sonnait pas.*
(TL3) *Le souscripteur TL ne doit être facturé pour aucun appel durant la plage horaire de restriction, à moins que le code correct ait été composé au moment opportun.*
Cette dernière attente est un exemple de subjectivité : la spécification du service ne l'exprime pas explicitement, mais un usager peut la supposer implicite lorsqu'il s'abonne au service.

Définition de prédicats :

- TLsub(x) $\Leftrightarrow$ x est un souscripteur au service TL.
- IsTeenTime(x,t) $\Leftrightarrow$ t appartient à l'intervalle de temps de restriction d'appel pour l'usager x, défini par le prédicat TeenTime(x,t1,t2) donné par la spécification.
- y=TeenPin(x) $\Leftrightarrow$ y est le code correct pour l'usager x.
- T est une variable représentant l'heure de la journée.
- AnnounceAskForPin(x) $\Leftrightarrow$ x entend l'annonce "AskForPin".

Propriétés:
(TL1) $\triangleright$ TLsub(x) $\Rightarrow$ not (IsTeenTime(T,x) and StartAudibleRinging(x,y))
or ONCE Dial(x,TeenPin) SINCE Off-hook(x)
(TL2) $\triangleright$ TLsub(x) and IsTeenTime(T,x) and pre Idle(x) and Off-hook(x)
$\Rightarrow$ AnnounceAskForPin(x)
(TL3) $\triangleright$ TLsub(x) and IsTeenTime(T,x) $\Rightarrow$ not LogBegin(-,-,x)
or ONCE Dial(x,TeenPin) SINCE Off-hook(x)

Terminal Call Screening (TCS)
(TCS1) *Le service TCS permet de filtrer les appels parvenant à son souscripteur en fonction de l'identité de l'appelant.*
(TCS2) *Les appels provenant de postes apparaissant dans une liste de numéros filtrés sont refoulés en étant redirigés vers un message vague et poli.*

Définition de prédicats:

- TCSsub(x) $\Leftrightarrow$ x est un souscripteur au service TCS.
- IsScreenedBy(x,y) $\Leftrightarrow$ x est filtré par le service TCS de y.
- AnnounceScreenedMsg(x) $\Leftrightarrow$ x entend l'annonce de filtrage.

Propriétés:
(TCS1) $\triangleright$ TCSsub(y) and IsScreenedBy(x,y) $\Rightarrow$ not StartRinging(y,x)
(TCS2) $\triangleright$ TCSsub(y) and ConnectRequest(x,y) and IsScreenedBy(x,y)
$\Rightarrow$ AnnounceScreenedMsg(x)

ThreeWay Calling (TWC)
(TWC1) *Un souscripteur TWC peut initier un deuxième appel durant une communication.*
(TWC2) *Le souscripteur peut ensuite joindre les deux appels en un seul. Les trois correspondant font alors partie d'une unique communication.*

Définition de prédicats:

- TWCsub(x) $\Leftrightarrow$ x est un souscripteur au service TWC.
- ThreeWay(x) $\Leftrightarrow$ x a activé son service TWC.

Propriétés:
(TWC1) $\triangleright$ TWCsub(x) and pre Talking(x,y) and not ThreeWay(x) and Flash(x)
$\Rightarrow$ DialTone(x)
(TWC2) $\triangleright$ (((TWCsub(x) and pre Talking(x,y) and not ThreeWay(x) THEN Flash(x)) THEN Dial(x,z) and pre Idle(z)) THEN Off-hook(z)) THEN Flash(x)
$\Rightarrow$ Talking(x,z) and Talking(x,y)

IN Call Forwarding (CF)
(CF1) *Le service CF permet de rediriger de manière inconditionnelle tous les appels destinés à son souscripteur.*
(CF2) *Le poste d'un souscripteur au service ne sonne jamais.*
(CF3) *Le souscripteur est facturé pour le surcoût lié à la redirection, tandis que l'appelant est normalement facturé.*

Définition de prédicats:

- CFsub(x) $\Leftrightarrow$ est un souscripteur au service CF.
- (y=ForwardTo(x)) $\Leftrightarrow$ y est le poste vers lequel CF redirige les appels destinés à x.
- La définition de ConnectRequest est complétée comme suit :
  ConnectRequest(x,y) = (CRpots(x,z) and ForwardTo(z)=y) or (CRpots(x,y))
  où CRpots est la définition de base de ConnectRequest.

Propriétés:
(CF1) $\triangleright$ CFsub(x) and ConnectRequest(y,x) and not pre Busy(ForwardTo(x))
$\Rightarrow$ ConnectRequest(y,ForwardTo(x))
(CF2) $\triangleright$ CFsub(x) $\Rightarrow$ not StartRinging(x,z)
(CF3) $\triangleright$ CFsub(x) and Dial(y,x) and not pre Busy(ForwardTo(x)) THEN Off-hook(ForwardTo(x))
$\Rightarrow$ LogBegin(y,x,y) and LogBegin(x,ForwardTo(x),x)

Call Waiting (CW)
(CW1) *Un usager qui appelle un souscripteur au service CW alors que celui-ci est occupé entend la tonalité AudibleRinging au lieu de LineBusy, sauf si le souscripteur a déjà un usager en attente.*
(CW2) *Le service CW permet à son souscripteur d'accepeter un deuxième appel alors qu'il est déjà en communication.*
(CW3) *Busy(x) est redéfini par le service CW.*
(CW4) *Une fois activé, le service permet à son souscripteur de passer d'un correspondant à l'autre.*

Définition de prédicats:

- CWsub(x) $\Leftrightarrow$ x est un souscripteur au service CW.
- NoHeldCall(x) $\Leftrightarrow$ not CallWaiting(x), i.e. CW n'est pas actif pour x, et aucun appel n'est en attente.
- y=HeldCall(x) $\Leftrightarrow$ y est mis en attente par le service CW de x.
- Talking(x,y) $\Leftrightarrow$ x et y sont en communication.

Propriétés:
(CW1) $\triangleright$ CWsub(x) and Talking(x,y) and ConnectRequest(z,x) and NoHeldCall(x)

⇒ StartAudibleRinging(z,x)
(CW2) ▷ CWsub(x) and NoHeldCall(x) and Dial(z,x) and Talking(x,y) THEN Flash(x)
⇒ Talking(x,z) and HeldCall(x)=y
(CW2-bis) ▷ CWsub(x) and ConnectRequest(y,x) and NoHeldCall(x) ⇒ StartAudibleRinging(y,x)
(CW3) ▷ CWsub(x) and NoHeldCall(x) ⇒ not busy(x)
(CW4) ▷ CWsub(x) and not NoHeldCall(x) and Talking(x,y) and Flash(x)
⇒ Talking(x,pre HeldCall(x))

Charge Call (CC)
Ce service permet à son souscripteur de facturer un appel sur un compte différent que celui associé au poste appelant, à condition qu'un code d'identification correct pour ce compte soit fourni.
(CC1) *Si un souscripteur A au service CC compose le numéro de B préfixé par "0", puis compose le numéro de C, suivi du code d'identification correct pour C, un appel sera initié entre A et B.*
(CC2) *Dans les conditions citées ci-dessus, l'appel de A à B sera facturé à C.*
(CC3) *Si le code entré pour C est incorrect, l'appel n'aura pas lieu.*

Définition de prédicats:

- CCsub(x) ⇔ x est un souscripteur au service CC.
- Charge(x) est le code d'identification pour que x soit facturé pour un appel.
- ConnectRequest est modifié par le service, comme indiqué par CC1.

Propriétés:
(CC1) ▷ Dial(X,0+Y) THEN Dial(X,Z) THEN Dial(X,T) and T=Charge(Z)
⇒ ConnectRequest(X,Z)
(CC2) ▷ Dial(X,0+Y) THEN Dial(X,Z) THEN Dial(X,Charge(Z)) and Idle(Y)
THEN Off-hook(Y) ⇒ LogBegin(X,Y,Z)
(CC3) ▷ Dial(X,0+Y) THEN Dial(X,Z) THEN Dial(X,T) and not T=Charge(Z)
⇒ Announce(X,InvalidPIN) and IsDisconnected(X)

Cellular (CELL)
Ce service permet de facturer les appels destinés à ou initiés par un téléphone mobile.
(CELL1) *Lorsqu'un souscripteur au service CELL initie ou répond à une demande de connexion, il est facturé pour le temps de transmission aérienne.*
(CELL2) *La facturation perdure le temps que le souscripteur demeure en communication.*

Définition de prédicats:
CELLsub(x) $\Leftrightarrow$ x est un souscripteur au service CELL.

Propriétés:
(CELL1) $\triangleright$ CELLsub(X) and ((StopAudibleRinging(X,Y) and Off-hook(Y)) or (StopRinging(X,Y) and Off-hook(X))) $\Rightarrow$ AirBegin(X)
(CELL2) $\triangleright$ CELLsub(X) and ((Disconnect(X,Y) and On-hook(Y)) or (Disconnect(Y,X) and On-hook(X))) $\Rightarrow$ AirEnd(X)

Return Call (RC)
Ce service permet à son souscripteur de retourner un appel au dernier appelant auquel il n'a pas été répondu.
(RC1) *Le fait de composer le code d'activation du service RC conduit à l'alternative suivante : si le dernier appelant est occupé, le souscripteur obtient un message Retry, dans le cas contraire, un appel est initié vers cet usager.*
(RC2) *Lorsque le service RC est activé, le poste de son souscripteur sonne dès que l'usager à rappeler est libre, et que le souscripteur l'est également.*
(RC3) *Si le souscripteur décroche alors que son poste sonne du fait du service RC, il obtient un message Retry si le dernier appelant est occupé, ou un appel est initié vers cet usager dans le cas contraire.*

Définition de prédicats:

- RCsub(x) $\Leftrightarrow$ x est un souscripteur au service RC.
- RCcode = *69 est le code d'activation du service RC.
- ReturnCall(x,y) est une variable définie dans les spécifications pour indiquer que x tente de retourner un appel vers y.
- LastCall(x,y) est *vrai* lorsque y est le dernier appelant de x.
  LastCall(x,y) = between(StartRinging(x,y),StartRinging(x,z) and z$\neq$y)
- AnnounceRetryReturnCall(x) $\Leftrightarrow$ x reçoit une annonce indiquant que le service RC a été activé.
- IsRCRinging(x) $\Leftrightarrow$ between(StartRinging(x,y,2), Off-hook(x)).
- ConnectRequest est modifié comme suit :
  Soit (RC1-ant) (resp. RC3-ant) la prémice de la propriété (RC1) (resp. (RC3)).
  (RC1-ant) or (RC3-ant) $\Rightarrow$ ConnectRequest(x,y)

Propriétés:
(RC1) $\triangleright$ RCsub(x) and LastCall(x,y) and Dial(x,RCcode)
$\Rightarrow$ ((pre Busy(y) and AnnounceRetryReturnCall(x)) or (pre Idle(y) and StartRinging(y,x) and StartAudibleRinging(x,y)))
(RC2) $\triangleright$ RCsub(x) and ReturnCall(x,y) and pre Idle(x) and pre Idle(y) $\Rightarrow$ StartRinging(x,y,2)
(RC3) $\triangleright$ RCsub(x) and pre IsRCRinging(x) and Off-hook(x)
$\Rightarrow$ ((pre Busy(y) and AnnounceRetryReturnCall(x)) or (pre Idle(y) and StartRinging(y,x) and StartAudibleRinging(x,y)))

Résumé

Ce travail aborde le problème de la validation de spécifications de services téléphoniques et notamment la recherche d'interactions entre services. Une interaction correspond à la modification du comportement d'un ou plusieurs services, du fait de la coexistence des services. L'interaction est un obstacle majeur au développement de l'offre de services de télécommunications.

La validation de spécifications requiert une modélisation des services et du réseau sous-jacent. Nous proposons une méthode de spécification formelle et de validation de services. Celle-ci est basée sur l'utilisation d'un formalisme synchrone pour la modélisation et la spécification, et sur la mise en oeuvre de méthodes de test pour la validation.

Nous avons à cette fin proposé une méthode de test adaptée au problème. Cette méthode a été intégrée à Lutess, un environnement de test fonctionnel de systèmes réactifs synchrones, reposant sur un principe de génération de données dynamique et aléatoire. Elle est basée sur la notion de "guidage par schémas". Un schéma représente une classe de comportements de l'environnement du système sous test, comportements que l'on souhaite tester principalement, soit parce qu'ils sont réalistes, soit parce qu'ils conduisent à une situation estimée critique.

Cette méthode a été formalisée, puis validée expérimentalement dans plusieurs études de cas conséquentes, en particulier lors du premier concours de détection d'interaction proposé en marge de la conférence "Feature Interaction Workshop", qui a consacré Lutess "Meilleur outil pour la détection d'interactions".

Abstract

This work is a proposal regarding the validation of telecommunication feature specifications. It is more precisely concerned with the detection of feature interactions, which is a problem that hinders the rapid introduction of new features in telecommunication systems. A feature interaction may occur when several features, developed in an independent manner are simultaneously available: this coexistence may alter the behavior of the features.

The validation of specifications requires modeling both the services and the underlying network. The proposed solution relies on a methodology for formally specifying and validating features. It is based on the use of a synchronous formal language for the specification part and of some specific testing methods for the validation part.

The thesis describes a new testing method called "pattern-guided testing", which we embedded in Lutess. Lutess is a testing tool for synchronous software, whose basic principle lies on a dynamic and random but constrained generation of test data.
Our testing method extends the notion of constraints in Lutess with guiding patterns. A pattern describes a class of environment behaviors which one wishes to favor. In the context of Lutess, patterns are seen as loose constraints: they influence the generation so as the behaviors they represent are more often produced.

This method has been formalized then validated on several case studies. The most important one has been conducted in the context of the "First Feature Interaction Detection Contest", held during the 5th Feature Interaction Workshop in 1998. On that occasion, Lutess won the Best Tool Award.

Mots clefs : Validation de services téléphoniques, génération de données de test, spécifications formelles, approche synchrone.

LSR-IMAG, UMR 5526 681 rue de la Passerelle, BP 72, 38402 Saint Martin d'Hères Cedex