N° d'ordre : 3103

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Ismaïl Berrada**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Modélisation, Analyse et Test des Systèmes Communicants à Contraintes Temporelles : Vers une Approche Ouverte du Test

---

Soutenue le : 14 décembre 2005

Après avis des rapporteurs :

Jean-Claude Fernandez	Professeur
Olivier Roux	Professeur

Devant la commission d'examen composée de :

Richard Castanet	Professeur	Directeur de Thèse
Patrick Félix	Maître de Conférences	Encadrant
Jean-Claude Fernandez	Professeur	Rapporteur
Francine Krief	Professeur	Examinatrice
Olivier Roux	Professeur	Rapporteur
Bertrand Tavernier	CRIL Technology	Examinateur

– 2005 –

Remerciements

À Richard Castanet

À Patrick Félix

Au membre du jury

À mes amis

À ceux qui .... oui Antoine, ...

À Aziz

À Nazha

À Pilar

À mes parents pour leur amour et leur soutien
je leur dois beaucoup

Table des matières

1 Introduction **1**
1.1 Méthodes formelles 1
1.2 Développement d'un système 2
1.3 Systèmes temporisés 3
1.4 Approches et contributions de cette thèse 5
1.4.1 Modélisation des systèmes communicants 5
1.4.2 Analyse des systèmes temporisés 6
1.4.3 Test des systèmes communicants 7
1.4.4 Outils et étude de cas 8
1.5 État de l'art 8
1.6 Organisation de ce document 11
1.7 Cadre de cette thèse 12

I Formalismes **13**

2 Préliminaires **18**
2.1 Graphes 19
2.1.1 Définitions élémentaires 19
2.1.2 Chemin 20
2.2 Espaces denses 20
2.2.1 Valuation 20
2.2.2 Polyèdre 21
2.2.3 Opérations sur les polyèdres 22

3 Modèles pour les systèmes non-temporisés **24**

3.1 Alphabet. 24
3.2 Machine à états finis 25
3.3 Système de transitions 27
3.3.1 Système de transitions étiqueté (LTS) 27
3.3.2 Composition synchrone des LTSs 28
3.3.3 Notations standards des LTSs 29
3.4 Système de transitions à entrée/sortie. 29
3.5 Réseau de Petri 31
3.6 Terme d'une algèbre de processus 32

4 Modèles pour les systèmes temporisés **36**
4.1 Alphabet et notion de temps 37
4.2 Système de transitions temporisé 37
4.3 Automate temporisé 39
4.3.1 Variable horloge 39
4.3.2 Automate temporisé d'Alur et Dill 40
4.3.3 Sémantique d'un TA 41
4.3.4 Computation 42
4.3.5 Composition synchrone des TAs 43
4.4 Automate temporisé avec invariants 44
4.5 Automate temporisé avec urgence 45
4.5.1 Urgence à l'Uppaal [96] 45
4.5.2 Urgence à la Kronos [33] 46
4.6 Automate temporisé à entrée/sortie 47
4.7 Automate temporisé étendu 48
4.8 Automate des régions 49
4.8.1 Région d'horloges 49
4.8.2 Automate des régions 50
4.9 Autres modèles 51

II Modélisation des Systèmes Communicants **53**

5 Modèle des systèmes communicants **57**

5.1 Modèle CS 58
5.1.1 Topologie de communication 58
5.1.2 Systèmes communicants 59
5.1.3 Sémantique d'un CS 61
5.1.4 CS et algèbre de processus 63
5.1.5 Comparaison 63
5.2 Exemples 64
5.2.1 Consommateur-Producteur 64
5.2.2 L'algorithme de Peterson 66
5.2.3 Protocole CSMA/CD 71
5.3 Communication synchrone/asynchrone 72
5.3.1 Système distribué synchrone 73
5.3.2 Système distribué asynchrone 75
5.4 Conclusion 77

III Analyse des Systèmes Temporisés 79

6 Graphe, polyèdre et valuation 84
6.1 Graphe positif et minimal 84
6.1.1 Préliminaires 85
6.1.2 Transformation b2m() 86
6.1.3 Transformation $R_{i\to *}()$ 87
6.1.4 Transformation $R_{*\to i}()$ 88
6.1.5 Récapitulation 90
6.2 Extraction de valuations à partir d'un polyèdre 90
6.2.1 Préliminaires 90
6.2.2 Forme canonique d'un polyèdre 91
6.2.3 Valuation de bornes 94
6.2.4 k-complétude d'une valuation 98
6.2.5 Conclusion 98
6.2.6 Ce que nous allons faire 99

7 Diagnostic temporisé 100

7.1 Diagnostic temporisé basé sur le polyèdre de contraintes 101
7.1.1 Diagnostic temporisé de bornes 101
7.1.2 Inclusion des traces 103
7.1.3 Conclusion 104
7.2 Diagnostic temporisé basé sur une analyse symbolique 104
7.2.1 État symbolique 104
7.2.2 Analyse symbolique d'un chemin 107
7.2.3 k-complétude d'une computation 110
7.2.4 Diagnostic temporisé de bornes 112
7.2.5 Inclusion des traces 113
7.3 Comparaison avec des travaux similaires 113
7.3.1 Travaux d'Alur et al. [6] 113
7.3.2 Travaux de Tripakis [145, 146] 113
7.3.3 Autres travaux 114
7.3.4 Nos contributions 114

IV Test des Systèmes Communicants 117

8 Introduction au test 123

8.1 Norme ISO 9646 [81] 123
8.2 Relations de conformité 125
8.2.1 Concept de la relation de conformité 125
8.2.2 Relation $\leq_{tr}$ 126
8.2.3 Relation ioconf 127
8.2.4 Relation ioco 127
8.3 Objectif de test 129
8.4 Verdict, cas de test et exécution 131
8.4.1 Verdict 131
8.4.2 Cas de test 132
8.4.3 Exécution 132
8.4.4 Propriétés des tests 133
8.5 Interopérabilité 134

8.5.1 Définitions de l'interopérabilité 135
8.5.2 Activités et types du test d'interopérabilité 136
8.5.3 Architectures de test 137
8.5.4 Méthodologie pour l'interopérabilité 139

9 Cadre formel pour le test d'interopérabilité 141
9.1 Préliminaires 141
9.2 Définitions de l'interopérabilité 142
9.3 Dérivation de cas de test d'interopérabilité 145
9.3.1 Hypothèses de test 146
9.3.2 Algorithme de génération 146
9.4 Génération de cas de test d'interopérabilité à partir d'outils de conformité 147
9.5 Comparaison 148

10 Cadre formel pour le test de conformité 151
10.1 Introduction 152
10.2 Définitions de la conformité 152
10.2.1 Rappels 152
10.2.2 Relation $\leq_{ttr}$ 153
10.2.3 Relation $ioconf_t$ 154
10.2.4 Relation $ioco_t$ 155
10.3 Cas de test et testeurs 156
10.3.1 Test adaptatif et statique 157
10.3.2 Formalisme de cas de test 157
10.4 Dérivation de cas de test abstraits et squelettes 159
10.4.1 Hypothèses de test 159
10.4.2 Automate de simulation 160
10.4.3 Cas de test abstraits et squelettes 161
10.5 Dérivation de tests digitaux 162
10.6 Dérivation de tests analogiques 165
10.7 Comparaison 168
10.7.1 Définition de la conformité 168
10.7.2 Dérivation de tests 168

11 Test ouvert **171**
11.1 Couverture structurelle 172
11.1.1 Critères de couverture 172
11.1.2 Terminologies et définitions 174
11.1.3 Coloriage comme critère de couverture 175
11.1.4 Coloriage ordonné comme critère de couverture 176
11.1.5 Comment définir un coloriage et un coloriage ordonné 178
11.1.6 Conclusion 179
11.2 Modélisation uniforme du test 179
11.2.1 Méthodologie des algorithmes génériques de génération 180
11.2.2 Modélisation de l'architecture de test 181
11.2.3 Modélisation de l'approche passive du test 182
11.2.4 Modélisation de l'approche active du test 186
11.2.5 Comparaison 188
11.2.6 Test ouvert 189

V Implémentation et Outils **191**

12 Implémentation **193**
12.1 Représentation symbolique 193
12.1.1 Préliminaires 193
12.1.2 Représentation symbolique des polyèdres : DBMs 194
12.1.3 Implantation des opérations sur les polyèdres 195
12.2 Trace symbolique 198
12.2.1 Préliminaires 198
12.2.2 Trace symbolique 199

13 L'outil TGSE **202**
13.1 Architecture du générateur de test de TGSE 202
13.2 Algorithme de génération gga 208
13.3 TGSE et le projet Calife 210
13.4 Étude de Cas : CSMA/CD 211

VI Conclusions et Perspectives 213

14 Conclusion 215

A Preuves de la partie III 229

B Preuves de la partie IV 234

Index 236

Table des figures

1 Vérification. 3
2 Test. 4
3 Opérations sur les polyèdres (1). 22
4 Opérations sur les polyèdres (2). 23
5 Machines à états finis. 26
6 Machine de Mealy. 27
7 IOLTS. 30
8 Réseau de Petri marqué. 32
9 Sémantique opérationnelle de $\mathcal{PA}$ en transitions. 35
10 Système de transitions temporisé. 38
11 Comportement d'une horloge. 39
12 Automate temporisé. 41
13 Automate temporisé avec invariants. 44
14 Urgence à l'Uppaal. 46
15 Urgence à la Kronos. 47
16 Automate temporisé étendu. 49
17 Région d'horloges. 50
18 Automate des régions. 51
19 Topologie de communication. 59
20 Modèle CS. 60
21 Consommateur-producteur. 64
22 Consommateur-producteur en CS. 65
23 Algorithme de Peterson. 70
24 Modélisation CSMA/CD. 71
25 Topologie de communication de CSMA/CD. 72

26 Utilisation d'un canal par un protocole. 73
27 Modélisation de la latence. 74
28 Modélisation d'une file de taille 2. 77
29 Graphes complets et étiquetés sur $\mathbb{T} = \mathbb{Z}$. 85
30 Transformations $R_{2\to*}$ et $R_{*\to 2}$. 87
31 Transformations et résultats sur les graphes complets. 90
32 Graphes de contraintes. 92
33 Automate A_ρ. 101
34 Polyèdre de contraintes 102
35 Successeur temporel. 105
36 Prédécesseur temporel. 106
37 Complexités. 114
38 Relation $\leq_{tr}$. 126
39 Relation ioco. 128
40 Exemple d'objectif de test. 130
41 Exemple de cas de test. 133
42 Relation interop. 144
43 Algorithme de génération de cas de test d'interopérabilité. 146
44 Exemple d'application. 148
45 Exemple de génération. 149
46 Relations $\leq_{ttr}$ et $ioconf_t$. 153
47 Relation $ioco_t$. 155
48 Test analogique et digital. 156
49 Décomposition de test. 159
50 Algorithme de génération des tests digitaux. 163
51 Spécification et cas de test squelette. 164
52 Cas de test temporisé. 165
53 Algorithme de génération des tests adaptatifs. 166
54 Couverture. 173
55 Coloriage de A. 176
56 Coloriage ordonné. 177
57 Architecture de test. 183

58 Test passif (1) 185
59 Test passif (2) 186
60 Test actif d'une composante 188
61 Activités du test ouvert. 189
62 Mise en forme canonique : algorithme de Floyd-Warshall. 195
63 Procédure SymbolicTrace. 200
64 Procédure UpdatePredicates. 201
65 Procédure UpdateContext. 201
66 Architecture logicielle 203
67 Algorithme de génération *gga*. 209
68 Interfaces de TGSE avec Calife. 211
69 Modélisation CSMA/CD. 212
70 Expérimentation. 212

Chapitre 1

Introduction

1.1 Méthodes formelles

L'évolution technologique a conduit au développement de systèmes informatiques complexes, dont l'impact socio-économique est devenu très fort, dans la mesure où ils occupent des places de plus en plus stratégiques au sein des organisations. De tels systèmes intègrent de nombreux composants logiciels et matériels et interagissent avec des environnements complexes. Ces systèmes sont devenus critiques tant par les conséquences de leur utilisation que par la complexité de leur développement et de leur évolution. Une classe importante des systèmes critiques est celle des systèmes réactifs, systèmes interagissant de façon continue avec un environnement.

Les *méthodes formelles* fournissent un cadre mathématique permettant de décrire de manière précise et stricte les systèmes et les programmes conçus. Elles ont prouvé leur efficacité dans la validation des systèmes. Les exemples de leur utilisation ne sont pas restreints. Les domaines du succès de leur application incluent l'industrie automobile [129, 101], l'industrie aérospatiale [51, 53] et bien d'autres domaines. Parmi les méthodes formelles, nous pouvons citer :

- La *vérification.* Il s'agit d'une procédure automatique qui permet de tester algorithmiquement si un modèle donné, le système lui-même ou une abstraction du système, satisfait une spécification logique, généralement formulée en termes de logique temporelle [50, 139, 34].

- La *preuve.* À partir d'un système et d'une propriété exprimée dans un langage de spécification, elle permet de prouver, en utilisant des règles de déduction comme pour prouver un théorème mathématique, la validité de la propriété [74, 128].

- La *génération de cas de test.* Les techniques de génération des séquences de test

consistent à extraire, à partir d'une spécification formelle du système sous test et un critère de sélection, un ensemble de "scénaris" à appliquer sur le système réel en vue de sa validation [143, 41].

La preuve ainsi que la vérification ne visent pas à certifier n'importe quel système ou programme. Elles s'appliquent à des modèles et non à des systèmes réels. Ces modèles ne permettent pas toujours de représenter tous les systèmes réels. La raison est que ces derniers dépendent, en général, d'un environnement non contrôlable alors que les modèles de ces systèmes ne peuvent refléter qu'une partie du comportement de cet environnement. Le test des systèmes réels vient compléter ces méthodes : il permet de mettre en conditions réelles les systèmes [1].

1.2 Développement d'un système

Comme toute activité de fabrication, le développement d'un logiciel requiert plusieurs phases : la conceptualisation, la réalisation puis celle où l'on s'assure que le produit final correspond aux exigences. Le *génie logiciel* est l'ensemble des méthodes mises en oeuvre pour le développement d'un logiciel ; il comprend en outre les mesures prises au cours du *cycle de développement* d'un logiciel pour garantir sa qualité. Ce cycle découpe le développement en différentes phases clés, généralement successives, qui s'appuient chacune sur le résultat de la phase précédente. Il est constitué des phases suivantes :

1. *Analyse des besoins.*
   Une spécification informelle des besoins, des exigences et des fonctionnalités du logiciel est rédigée : c'est le cahier des charges établi au début du processus de développement.
2. *Modélisation.*
   Cette phase consiste à *modéliser* le système, i.e. construire un objet dans un cadre formel bien défini qui "reproduit" aussi fidèlement que possible le comportement du système réel (le cahier des charges), pour obtenir ce qu'on appelle la *spécification*. Le choix du formalisme dépend des exigences du système réel et du niveau d'abstraction et de détail que l'on souhaite faire apparaître.
3. *Vérification du modèle.*
   Une étape préliminaire de la vérification consiste à modéliser la propriété à vérifier dans un langage de spécification. La vérification consiste alors à vérifier que le modèle construit pour le système réel vérifie la propriété exprimée dans le langage précité. Pour pouvoir réaliser cela, il est bien entendu nécessaire de disposer d'algorithmes de vérification qui parcourent exhaustivement le modèle du système.
4. *Implantation.*
   Dans cette phase, le système réel est développé sous forme de code exécutable. La

[1] Ce qui n'est pas le cas avec la vérification et la preuve automatique qui interviennent sur les modèles.

génération du code est basée sur la spécification et peut être automatique.

5. *Test.*
   Il consiste à soumettre l'implantation du système à une série d'épreuves pour s'assurer de sa validité. C'est la seule phase où le système est mis en conditions réelles. Contrairement à la vérification qui s'intéresse aux propriétés du modèle du système, le test vise à analyser les propriétés du système réel. Différents types de test existent. Le choix d'un type dépend de la validation que l'on souhaite effectuer.

Les méthodes formelles s'appliquent en particulier dans les phases de vérification et de test. Le schéma général de la vérification (resp. du test) est donné dans la Fig.1 (resp. Fig.2).

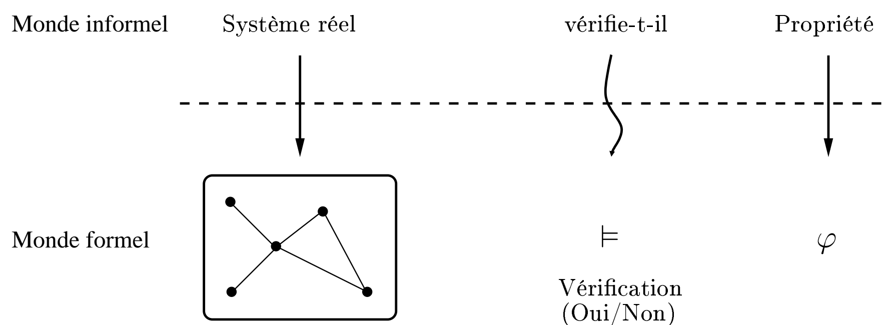


Fig. 1: Vérification.

Étant données une spécification formelle S et une propriété φ du système réel, les algorithmes de vérification répondent "oui" dans la cas où S vérifie φ et en général donnent une (ou plusieurs) trace d'exécution correcte [2]. Dans le cas contraire, ils génèrent une trace d'exécution incorrecte.

Dans le cas du test, un algorithme de génération est défini en tenant compte d'une ou plusieurs propriétés du système. Une propriété peut être la couverture de certains états ou transitions de la spécification ou encore un comportement particulier du système. Les cas de test générés sont exécutés sur l'implantation réelle du système. Un verdict est associé à l'exécution d'un cas de test. Le verdict "Pass" signifie que l'implantation satisfait le comportement testé. Dans le cas d'un verdict "Fail", l'implantation est déclarée erronée.

Dans ce document, nous nous concentrons sur l'étude des méthodes formelles pour la phase du test en particulier et pour une partie de la phase de vérification

1.3 Systèmes temporisés

L'analyse formelle des systèmes considère plusieurs aspects :

[2]Le cas d'une propriété d'accessibilité.

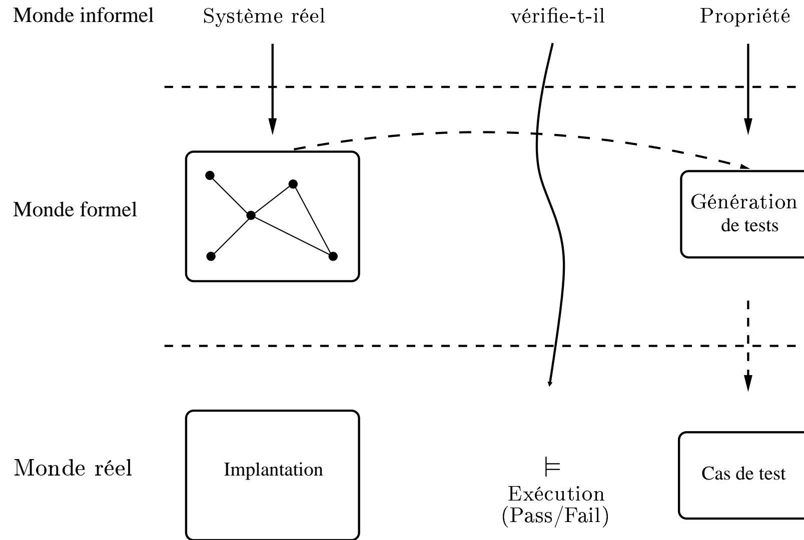


FIG. 2: Test.

1. *Contrôle.* L'aspect de base de la construction d'un système est l'obtention d'un flux de contrôle, i.e. une dépendance causale convenable entre les événements. L'ordre temporel du déroulement des événements est un élément crucial pour l'exactitude d'un système. Par exemple, il est évident qu'un consommateur ne peut pas utiliser un produit si ce dernier n'a pas été produit auparavant. L'aspect contrôle est essentiel dans l'analyse d'un système et ne peut être négligé.

2. *Données.* De nombreux systèmes sont fortement basés sur la manipulation des données. Les données interviennent sur le flux de contrôle pour ces systèmes. Un système de gestion d'une base de données est un exemple de tels systèmes. Pour d'autre systèmes, comme c'est le cas pour la plupart des protocoles, cette dépendance est moins cruciale et les données peuvent être abstraites dans l'analyse formelle.

3. *Temps.* Un système temporisé est un système contraint par des besoins portants sur les occurrences des événements par rapport au temps (réel). Ces conditions temporelles peuvent représenter les performances acceptables du système ou les délais qui doivent être respectés.

Dans ce document, nous étudions et développons des formalismes pour l'analyse des systèmes dans lesquels le *temps* est un facteur critique et important.

Le cadre formel adopté dans ce document, pour les systèmes temporisés, se compose des éléments suivants :

Sémantique du modèle : temps continu (dense)

Nous considérons des systèmes qui évoluent continuellement dans le temps, ainsi le domaine du temps est l'ensemble des réels positifs. En dehors des propriétés qualitatives renseignant sur l'ordre relatif des événements (i.e. l'événement a se produit avant l'événement b), ce modèle peut aussi exprimer des propriétés quantitatives du temps, renseignant sur les délais entre les occurrences des événements (i.e. a se produit 4 unités de temps avant b). Les délais peuvent être exacts, bornés ou non-bornés. Le modèle doit être indépendant des unités du temps ou des granulations utilisées.

Langage de description des systèmes : automates temporisés

Nous décrirons un système temporisé en utilisant les automates temporisés (TA) d'Alur-Dill [4], qui sont des automates finis munis d'un ensemble fini d'horloges à valeurs réelles. Un TA alterne entre deux modes d'exécutions : le passage du temps dans les états et la réalisation des transitions discrètes (changement d'état).

1.4 Approches et contributions de cette thèse

Les travaux que nous présentons portent sur la modélisation, l'analyse et le test des systèmes communicants. Ces travaux peuvent être divisés en quatre parties.

1.4.1 Modélisation des systèmes communicants

L'utilisation des automates comme modèle et cadre théorique pour décrire les systèmes communicants a l'avantage d'offrir des facilités de définition d'algorithmes applicables dans un outil de validation. Cependant, les modèles existants basés sur les automates présentent deux limitations au niveau de la description des aspects de contrôle et de données. Pour l'aspect contrôle, la majorité des modèles existants ne permet que la modélisation des communications binaires ou sur des ensembles d'événements communs. Pour l'aspect données, une modélisation du partage des ressources entre processus est requise. Il est clair que les algèbres de processus se présentent comme un bon modèle, très expressif. Cependant, l'utilisation des algèbres de processus présente une difficulté de définition d'algorithmes applicables dans un outil de validation.

Nous introduisons le modèle CS comme un modèle de description des systèmes communicants. Une description dans le modèle CS considère deux aspects des systèmes communicants : les flux (contrôle, données et temps) propres aux entités du système et les flux (contrôle et données) partagés entre les entités. La modélisation des flux propres aux entités est basée sur les automates. Le flux de données partagé est modélisé par des variables et des paramètres. Le flux de contrôle partagé est modélisé par une topologie

de communicant qui explicite les différentes communications possibles dans un état du système. La séparation entre les flux propres et les flux partagés d'une part et une sémantique en automate d'autre part, procurent au modèle CS l'aspect graphique des automates et l'aspect hiérarchique des algèbres de processus.

1.4.2 Analyse des systèmes temporisés

Un diagnostic temporisé est une trace du système contenant des informations sur les changements discrets du système et les délais exacts entre deux transitions discrètes. La vérification des automates temporisés est réalisée à travers l'exploration d'espaces d'états abstraits. Les diagnostics qui peuvent être générés directement à partir d'une telle exploration sont aussi abstraits. En particulier, de tels diagnostics manquent souvent d'information sur le temps, ce qui peut être crucial dans la compréhension des raisons de satisfaction ou de non satisfaction d'une propriété. Nous examinons les deux problèmes suivants : l'extraction des diagnostics temporisés d'un chemin temporisé et la vérification de l'inclusion des traces entre deux chemins temporisés.

Extraction des diagnostics temporisés. Nous introduisons deux approches pour extraire des diagnostics temporisés. La première approche est basée sur le polyèdre des contraintes associé à un chemin. La deuxième approche considère un calcul symbolique. Pour les deux approches, nous montrons comment extraire des diagnostics temporisés. De plus, nous identifions des diagnostics temporisés dits *de bornes maximales* et *minimales*. L'intérêt de ces derniers est qu'ils donnent une représentation finie de l'espace des traces d'un chemin temporisé. Leur nombre varient entre 1 et $2 \times (n+1)$ diagnostics pour un chemin de longueur n.

Vérification d'inclusion des traces. Le problème d'inclusion des traces considère deux chemins temporisés ρ et ρ' de taille n tels que ρ est connu (les différentes contraintes et mises à jour de ses transitions sont données) et ρ' n'est pas connu mais seulement ses traces temporisés de taille n ($TTrace(\rho', n)$) sont connues. La problématique consiste à donner les conditions nécessaires et suffisantes pour montrer que $TTrace(\rho, n) \subseteq TTrace(\rho', n)$. Ce problème trouve une application intéressante dans le domaine du test boîte noire. En effet, pour une implantation I d'une spécification S, vérifier que les comportements de S ont été bien implémentés par I consiste à vérifier que les traces de S sont incluses dans celles de I. Nous établirons les conditions nécessaires et suffisantes, basées sur les diagnostics temporisés de bornes, pour montrer cette inclusion.

À notre connaissance, les diagnostics de bornes et le problème d'inclusion des traces sont des résultats originaux.

1.4.3 Test des systèmes communicants

Durant la dernière décennie, plusieurs recherches ont été menées dans le domaine du test et ont donné différentes méthodes basées sur différents modèles formels et différentes applications possibles. Toutes ces méthodes génèrent avec succès des cas de test, mais la plupart d'entre elles ne reposent pas sur des cadres formels bien définis et souffrent de l'explosion du nombre des cas de test générés.

Test de conformité. Le test de conformité compare les comportements de l'implantation aux comportements de la spécification. Le problème majeur à résoudre dans le cas des systèmes temporisés est l'explosion combinatoire des nombres de cas de test générés due au caractère dense du temps. Nous proposons un cadre formel pour le test de conformité des systèmes temporisés. Ce cadre englobe une définition de la conformité ainsi que des méthodes de génération de cas de test. Pour la définition, nous proposons une extension temporisée de la relation de conformité *ioco* de Tretmans [144]. Pour les méthodes de génération, elle repose sur l'utilisation de l'automate de simulation introduit par Tripakis [145] et l'application des résultats du problème d'inclusion des traces. En effet, les tests générés, que ce soit pour un testeur digital ou analogique, ne sont pas arbitraires mais correspondent aux diagnostics temporisés de bornes. Ainsi, notre approche ne souffre pas de l'explosion combinatoire du nombre de cas de test générés tout en couvrant l'espace des états accessibles.

Test d'interopérabilité. Ce type de test considère plusieurs entités communicantes et vise à tester leur aptitude à interagir et à échanger l'information en vue de rendre des services attendus. D'une part, nous formalisons la notion d'interopérabilité par le biais d'une relation d'interopérabilité reliant les différentes entités à leurs implantations et paramétrée par les interfaces accessibles des implantations. D'autre part, nous montrons comment utiliser des outils de génération des cas de test de conformité pour dériver des cas de test d'interopérabilité. L'intérêt de notre approche réside dans le fait que le système global n'est pas construit. La dérivation se porte uniquement sur les entités du système.

Modélisation uniforme du test. Selon la composition d'un système, l'accessibilité de ses composantes et les propriétés attendues de ce dernier, plusieurs types (conformité, interopérabilité, $\cdots$), approches (active et passive) et architectures (boîte blanche, noire, $\cdots$) de test peuvent être définies. Nous montrons qu'il est possible d'appréhender les différents types, approches et architectures du test au sein du même modèle et avec la même méthodologie. En d'autre termes, nous proposons une modélisation uniforme de ces différents aspects du test. Cette modélisation est basée d'une part, sur le modèle des systèmes communicants (CS) et d'autre part, sur la méthodologie des algorithmes génériques de génération (GGA). Nous montrerons que le modèle CS est un modèle générique pour le test dans le sens qu' (i) il offre des mécanismes pour modéliser différents types de communications et d'architectures de test et (ii) il permet l'application du même algorithme générique de génération pour différents types et approches de test. L'intérêt

majeur de cette modélisation est la possibilité de réalisation d'outils de génération de test supportant différents types et approches du test. Ainsi, avec le même outil, il est possible 1) de modéliser différentes architectures de test, 2) de générer des cas de test de conformité, d'interopérabilité, etc, et 3) de vérifier l'appartenance d'une trace (de l'implantation) à la spécification (approche passive de test). Comme conséquence de ce résultat, nous introduisons le concept du test *ouvert* qui consiste à vérifier que des implantations satisfassent les spécifications selon une propriété P.

1.4.4 Outils et étude de cas

Outils TGSE. Les résultats présentés dans ce de document ont été implémentés dans l'outil TGSE (Génération de Test, Simulation et Émulation). TGSE représente la contribution pratique de cette thèse. TGSE est un ensemble d'outils qui permettent de générer des cas de test pour les systèmes temporisés paramétrés, de simuler l'exécution d'un cas de test sur une implantation et un émulateur (qui joue aussi le rôle de générateur automatiquement le code exécutable (en langage C)) qui permet d'exécuter les différentes composantes du système en parallèle. Dans sa version actuelle, TGSE supporte la génération automatique des cas de test (test actif) et la vérification de l'appartenance d'une trace à une spécification donnée (test passif). Ceci pour une ou plusieurs composantes et différentes architectures de test. TGSE a été incorporé dans la plate-forme Calife réalisée dans le cadre des projets RNRT Calife et le projet RNTL Avérroes. Le but de cette plate-forme est de procurer une interface unique pour une large gamme d'outils de vérification et de test.

Protocole CSMA/CS. CSMA/CD est pris comme une étude de cas dans ce document. D'une part, la modélisation de CSMA/CD fait intervenir des paramètres et des horloges, aspects traités dans ce document et d'autre part, il présente un bon exemple pour tester la capacité et les performances de l'outil TGSE.

1.5 État de l'art

L'état d'art présenté dans cette partie n'a pas la vocation d'être exhaustif. Il se porte essentiellement sur les travaux relatifs aux diagnostics temporisés et sur le test de conformité pour les systèmes non temporisés. Les travaux qui s'approchent de nos axes de recherche feront l'objet d'une présentation plus détaillée dans les chapitres concernés.

Diagnostics temporisés

Le problème d'extraction d'une exécution temporisée d'un chemin de longueur (nombre de transitions) n a été traité dans [6]. Les auteurs proposent un algorithme efficace pour générer l'exécution du cumul temporel minimal. Tripakis [145] propose une extension de ce problème par l'étude de la k-complétude d'une valuation.

Test de conformité non-temporisé

Les méthodes formelles de génération de tests se décomposent en plusieurs familles. Les deux grandes familles que nous présentons ici ont une base commune : elles partent de la donnée d'une spécification formelle du système.

La première famille, la plus ancienne, est celle des méthodes fondées sur les automates. Elle est issue des méthodes de test de circuits et des problèmes de reconnaissance de machines. Il existe plusieurs méthodes basées sur le même principe et qui diffèrent par la partie identification de l'état d'arrivée. Cette identification s'effectue par une séquence d'entrées/sorties qui doit permettre d'identifier d'une façon certaine l'état d'arrivée. La manière la plus simple d'identifier l'état d'arrivée est de supposer l'existence d'une entrée *statut* à laquelle le système répond par son état courant. C'est l'hypothèse faite pour la méthode dite de *tour de transition* (TT) [113]. La deuxième méthode appelée *séquence de distinction* (DS) [64] suppose l'existence d'une séquence d'entrée DS capable de différencier toute paire d'états. La troisième méthode appelée *entrée/sortie unique* (UIO pour Unique Input Output) suppose que pour chaque état, il existe une séquence d'entrées UIO [131, 98, 99] qui la différencie de tout autre état. Finalement, la méthode appelée *W* [47] fait l'hypothèse que toute paire d'états peut être différenciée par une séquence W. DS et UIO sont très proches. Pour une comparaison des tailles des séquences générées par ces méthodes, ainsi que d'autres méthodes comme la méthode H, se référer à [153].

La deuxième famille est fondée sur les systèmes de transitions. On distingue ici deux grandes approches. La première approche est basée sur les LTSs. Brinksma [40] introduit la relation de conformité **conf** qui limite l'observation aux traces de la spécification plutôt que de considérer toutes les traces observables possibles. Il en résulte la notion du *testeur canonique*, construit à partir de la spécification du système. Ce dernier, composé avec la spécification, permet de tester "exhaustivement" la relation conf. La deuxième approche est basée sur la distinction entre les entrées et les sorties dans le modèle [144, 123, 57, 84].

Quelques outils de vérification et de test

HyTech [69]. L'outil HyTech (Hybrid Technology) a été développé à l'université de Berkeley par Tom Henziger, Pei-Hsin Ho et Howard Wong-Toi. Cet outil vise à vérifier des réseaux d'automates hybrides très généraux. Il permet même d'analyser des systèmes paramétrés. Il vérifie principalement des propriétés d'accessibilité et de sûreté.

Uppaal [96]. L'outil Uppaal a été développé conjointement par l'université d'Aalborg au Danemark et l'université d'Uppsala en Suède. Les principales personnes ayant participé à son développement sont Wang Yi, Kim G. Larsen, Paul Pettersson, Johan Bengtsson, Gerd Behrmann et Kåre I. Kristoffersen. Il permet de vérifier des réseaux d'automates temporisés avec des variables entières bornées, actions urgentes, etc. Il vérifie principalement des propriétés d'accessibilité et de vivacité ou d'états bloquants. La logique utilisée est un

fragment de TCTL. L'algorithme implémenté est essentiellement un algorithme d'analyse en avant. Uppaal possède une interface graphique qui permet la saisie du système (des automates) et son exécution d'une façon interactive. Notons finalement qu'il est possible de générer des cas de test en utilisant Uppaal [70, 63].

Kronos [156]. L'outil Kronos a été développé au laboratoire Vérimag à Grenoble principalement par Sergio Yovine, Alfredo Olivero, Conrado Daws et Stravros Tripakis. Cet outil vise à vérifier des réseaux d'automates temporisés. Il vérifie principalement des propriétés énoncées dans la logique TCTL. Plusieurs algorithmes sont implémentés : l'analyse en avant et l'analyse en arrière.

CMC [93]. L'outil CMC (Compositional Model-Checking) a été développé au Laboratoire Spécification et Vérification à l'ENS de Cachan par François Laroussinie. Cet outil vise à vérifier des réseaux d'automates temporisés. Il utilise comme langage de spécification la logique TCTL.

IF [37]. L'outil IF (Interchange Format) a été développé au laboratoire Vérimag à Grenoble principalement par Jean-Claude Fernandez, Laurent Mounier, Marius Bozga, Lucian Ghirvu, Susanne Graf et Jean-Pierre Krimm. IF est tout d'abord un langage de modélisation des systèmes temps-réel asynchrones. Il est devenu le lien d'un ensemble d'outils de validation, désormais appelé "IF validation environment". L'environnement de validation IF se découpe en trois niveaux de représentation de programme : le niveau spécification (SDL, OBJECTGEODE,$\cdots$), le niveau intermédiaire IF (SDL2IF, simulation, $\cdots$) et enfin le niveau du modèle de sémantique LTS (CADP, LTS, vérification, $\cdots$). Signalons finalement que les outils de test TTG [92] (Timed Test Generation) et TGV [57] (Test Generation using Verification techniques) sont connectés à l'environnement de validation IF.

TGV [57]. L'outil TGV (Test Generation using Verification techniques) a été développé conjointement par le laboratoire Vérimag à Grenoble et le laboratoire Irisa à Rennes. Les principales personnes ayant participé à son développement sont Jean-Claude Fernandez, Claude Jard, Thierry Jéron et César Viho. La génération de tests en TGV est orientée objectif de test, i.e. on cherche à générer des cas de test pour un comportement donné de la spécification exprimée en SDL. TGV utilise l'outil commercial GEODE (VERILOG) pour générer un graphe (à états finis) de comportement et l'outil Aldebaran de la boîte-à-outils CADP [56] pour minimiser le graphe ainsi obtenu. La génération des arbres de tests en TGV est basée sur un produit synchrone comme dans l'algèbre des processus CCS. Durant le parcours du produit synchrone, plusieurs calculs sont effectués. L'algorithme calcule la consistance entre la spécification et l'objectif de test (au cours d'une phase de descente). Un squelette est synthétisé durant la phase de remontée du graphe. Ce graphe contient des séquences du produit synchrone menant à un état d'acceptation de l'automate.

Les transitions du squelette généré sont alors décorées avec le verdict. Finalement, TGV teste la relation de conformité *ioco* de Tretmans [144].

TorX [15]. L'outil TorX a été développé à l'université de Twente. Les principales personnes ayant participé à son développement sont Ed. Brinksma et Jan Tretmans. TorX est basé sur la même théorie du test que TGV : modèles similaires et même relation de conformité. L'algorithme de génération est différent puisqu'au lieu d'utiliser des objectifs de test pour la sélection des cas de test, TorX fait un tirage aléatoire sur les entrées de la spécification ou demande à l'utilisateur d'en choisir une. D'une part, cette génération simultanée a pour conséquence que TorX ne parcourt qu'une séquence d'actions observables de la spécification sans avoir besoin de mémoriser les états. D'autre part, pour chaque système testé, il est nécessaire de développer une couche d'adaptation spécifique pour passer du niveau des tests abstraits aux tests exécutables. Une version de de TorX a été développée dans l'outil SPIN pour des spécifications Promela.

1.6 Organisation de ce document

Ce document est organisé en six parties.

La première partie présente le fond de ce document. Le chapitre 2 introduit les graphes et les polyèdres, utilisés le long de ce document. Dans le chapitre 3, nous présentons quelques formalismes de description des systèmes sans contraintes temporelles, spécialement les systèmes de transitions étiquetés. Dans le chapitre 4, le modèle des automates temporisés est présenté ainsi que d'autres formalismes connexes.

La deuxième partie représente la partie modélisation de ce document. Le chapitre 5 introduit notre modèle CS ainsi que trois exemples de description dans CS : producteur-consommateur, exclusion mutuelle et le protocole CSMA/CD.

La troisième partie constitue la partie analyse de ce document et correspond au corps théorique relatif au test des systèmes temporisés. Dans le chapitre 6, nous introduisons quelques propriétés des polyèdres par le biais des graphes. Le chapitre 7 étudie le problème d'extraction des diagnostics temporisés ainsi que la vérification d'inclusion des traces de deux chemins temporisés.

La quatrième partie de ce document dresse le problème de génération des tests. Le chapitre 8 est une introduction au test. Le cadre formel, comprenant la définition de l'interopérabilité et la génération des cas de test d'interopérabilité, est présenté dans le chapitre 9. Basé sur les résultats du chapitre 7, le chapitre 10 présente un cadre formel relatif au test de conformité des systèmes temporisés. Dans le chapitre 11, la modélisation uniforme des différents types, approches et architectures de test est présentée. Cette modélisation est basée, d'une part sur la formalisation d'un critère de couverture sous forme de coloriage et d'autre part sur la méthodologie des algorithmes génériques de génération. Comme conclusion de chapitre, nous introduisons le concept du test ouvert.

La cinquième partie introduit les contributions pratiques de ce document. Le chapitre 11 est une implantation des approches présentées dans la deuxième et la troisième parties. Le chapitre 12 introduit l'outil TGSE développé au cours de cette thèse ainsi que l'étude de cas CSMA/CD.

Finalement, la dernière partie est consacrée à la conclusion et les perspectives.

Concernant la lisibilité de ce document, chaque partie dépend de la partie I. Le chapitre 10 dépend aussi du chapitre 7. Au delà de ces dépendances, chaque chapitre est supposé (espérons le) auto-dépendant. Un environnement spécial est utilisé pour les définitions et les mots, symboles et opérateurs clés apparaissent en *italique* lors de leur définition et peuvent être retrouvés dans l'index.

1.7 Cadre de cette thèse

Cette thèse s'inscrit dans le cadre du projet RNTL Avérroes et du projet européen Marie Curie RTN TAROT (MCRTN 505121).

Première partie

Formalismes

Introduction

Probablement, la façon la plus simple de décrire le comportement d'un système consiste à le représenter par un automate. Un *automate* est un graphe contenant des noeuds et des arcs étiquetés. Un noeud représente un état possible du système et un arc (ou transition) l'activité liant deux noeuds : l'un est l'état avant "l'exécution" de l'action et l'autre est l'état atteint après l'action. Les automates ont été étendus de plusieurs façons et utilisés pour différents buts. La sémantique de plusieurs langages de spécification est donnée en terme d'automates. L'utilisation des automates procure un cadre mathématique solide, éliminant toute confusion ou ambiguïté pour la validation des systèmes. Bien que les automates offrent une représentation graphique agréable pour la description des systèmes, cette représentation s'avère inexploitable pour des systèmes complexes incluant des descriptions larges et détaillées.

Pour les systèmes complexes, il est crucial de disposer d'une approche structurelle et d'une méthodologie systématique qui permettent la construction de grands systèmes à partir de la composition de systèmes de tailles réduites. Les algèbres de processus ont été conçues pour une spécification *hiérarchique* des systèmes. Une *algèbre de processus* est une algèbre au sens mathématique qui vise la description des processus. Chaque élément d'une algèbre de processus représente un comportement du système de la même façon qu'un automate le fait. De plus, elle procure des opérations qui permettent la composition des systèmes dans le but de construire un système plus complexe. A l'exemple de chaque algèbre, une algèbre de processus satisfait des axiomes et des lois, ce qui permet d'une part, la compréhension des concepts introduits par l'algèbre et d'autre part, l'analyse des systèmes d'une façon mathématique.

Au-delà des programmes séquentiels qui prennent des données en entrée et fournissent des résultats en sortie, beaucoup de systèmes informatiques sont *parallèles*, i.e. composés de plusieurs processus qui s'exécutent simultanément et s'échangent des informations. Le parallélisme est dit *synchrone* lorsque le système possède une horloge globale qui, à intervalles réguliers, pilote l'exécution des processus. Dans le cas contraire, le parallélisme est dit *asynchrone* : chaque processus est libre d'évoluer à son propre rythme, mais doit se synchroniser avec d'autres processus lorsqu'il veut accéder à des ressources partagées, afin d'assurer une cohérence globale. L'étude du parallélisme asynchrone trouve de nombreuses et importantes applications dans les domaines du logiciel, du matériel et des télécommunications. En règle générale, les systèmes asynchrones sont plus difficiles à concevoir et à mettre au point que les systèmes séquentiels ou synchrones. En effet, l'amélioration continue des méthodologies et des langages de programmation permet aujourd'hui de construire des systèmes séquentiels complexes à peu près corrects. Le niveau de correction atteint est essentiellement déterminé par un problème de coût et d'expertise grâce notamment à des concepts informatiques tels que le typage, la programmation structurée, les modules, les objets, l'analyse statique, les preuves formelles, etc. De même, pour les systèmes synchrones, l'apport des langages dédiés [66] et des outils pour la génération de code et la vérification permet de répondre aux exigences de sécurité pour des applications critiques. En revanche,

la situation des systèmes asynchrones est beaucoup moins avancée, ceci pour plusieurs raisons :

- Les systèmes asynchrones sont intrinsèquement *indéterministes*, i.e. qu'une exécution donnée peut ne pas être prédicable ni reproductible, d'où des difficultés pour la mise au point, la vérification et le test [3].
- Les systèmes asynchrones n'ont pas de propriétés évidentes de compositionnalité, i.e. qu'il n'est pas aisé de construire un système correct par assemblage de sous-systèmes plus simples et prouvés corrects.

Selon Hubert Garavel [61], un formalisme d'avenir devrait satisfaire quatre critères essentiels :

- *Expressivité.* Le formalisme doit permettre de modéliser simplement, sans contorsions inutiles, les caractéristiques essentielles des systèmes asynchrones.
- *Universalité.* Le formalisme doit être utilisable dans tous les domaines d'activité où intervient le parallélisme asynchrone (logiciel, matériel, télécommunications) et ne pas être étroitement dépendant d'un domaine particulier (comme ESTELLE [80] et SDL [82] l'ont été pour les télécommunications).
- *Exécutabilité.* Le formalisme doit avoir un caractère exécutable afin que les modélisations des systèmes asynchrones ne servent pas seulement de documentation, mais puissent être traitées par des outils de simulation (pour effectuer la mise au point), de prototypage rapide (pour produire du code exécutable) et de génération automatique de tests.
- *Vérifiabilité.* Certains systèmes asynchrones sont suffisamment critiques pour que leur correction doive être garantie par des techniques de vérification ou de preuve. Or celles-ci ne peuvent être appliquées que sur des modélisations faites dans des formalismes dont la sémantique est rigoureusement définie et possède de bonnes propriétés de compositionnalité et d'abstraction permettant de repousser les limites de l'explosion d'états.

Motivations

Depuis le milieu des années 70 (cf [106, 125]) jusqu'à nos jours, de nombreuses études théoriques relatives à la modélisation ont été menées et de nombreux modèles ont été proposés. Il est quasiment impossible de faire une étude portant sur l'ensemble des modèles proposés. Le but de cette partie est plutôt d'essayer de présenter les plus couramment utilisés dans la vérification et le test [4]. Rappelons que nous ne considérons que des modèles ayant un fondement mathématique et par conséquent une sémantique correcte. Le choix d'un modèle dépend ensuite du type de système que l'on souhaite développer, les conditions dans lesquelles ce système est censé évoluer et le mode de synchronisation considéré.

[3] En général, une combinatoire exponentielle limite fortement les possibilités de vérification et de test automatisées.

[4] Pour une comparaison des différents modèles, se référer à [61, 36].

Organisation

Cette partie introductive se compose de trois chapitres. Le chapitre 2 fournit un ensemble de notations et de définitions utilisées le long de ce document. Le chapitre 3 présente quelques modèles de description dans le cadre non-temporisé. Finalement, le chapitre 4 présente essentiellement les modèles temporisés basés sur les automates.

Chapitre 2

Préliminaires

Sommaire

2.1 Graphes **19**
2.1.1 Définitions élémentaires 19
2.1.2 Chemin 20
2.2 Espaces denses **20**
2.2.1 Valuation 20
2.2.2 Polyèdre 21
2.2.3 Opérations sur les polyèdres 22

Notations Générales. Dans la suite de ce document, $\mathbb{R}$ dénote l'ensemble des réels, $\mathbb{Z}$ l'ensemble des entiers relatifs et $\mathbb{N}$ l'ensemble des naturels. Pour un domaine $\mathbb{T}$ ($\mathbb{R}$ ou $\mathbb{Z}$), $\mathbb{T}^{\geq 0}$ dénote l'ensemble $\{x \,|\, x \geq 0, x \in \mathbb{T}\}$. Le symbole $+\infty$ (resp. $-\infty$) dénote l'infinité positive (resp. négative) telle que : pour tout $t \in \mathbb{T}$, $-\infty \leq t \leq +\infty$, $t+(+\infty) = (+\infty)+t = +\infty$ et $t + (-\infty) = (-\infty) + t = -\infty$. $\overline{\mathbb{T}}$ dénote $\mathbb{T} \cup \{+\infty, -\infty\}$.

Si X et Y sont deux ensembles, les opérations d'intersection, union, complément, la différence d'ensembles et le produit cartésien sont respectivement notées par $X \cap Y$, $X \cup Y$, $\overline{X}$, $X \backslash Y$ et $X \times Y$. L'ensemble vide est noté $\emptyset$. L'inclusion et l'inclusion stricte sont notées par $X \subseteq Y$ et $X \subset Y$. L'appartenance d'un élément x à X est notée par $x \in X$. Pour un ordre donné sur X, $min(X)$ et $max(X)$ dénotent respectivement le plus petit et le plus grand élément de X. 2^X dénote l'ensemble des sous-ensembles de X. $card(X)$ dénote la cardinalité de X si X est fini.

Les opérateurs logiques "et", "ou" et "non" sont respectivement notés par $\wedge$, $\vee$ et $\neg$. L'implication et l'équivalence sont respectivement notées par $\Rightarrow$ et $\equiv$ et sont définies de la façon habituelle. Finalement, le symbole $=$ teste l'égalité de deux expressions et $:=$ affecte une valeur à une variable.

2.1 Graphes

Dans cette section, nous rappelons quelques notions fondamentales des graphes.

2.1.1 Définitions élémentaires

Définition 1 *Un graphe G est un couple (N, E), où N est un ensemble fini d'éléments $\{n_1, n_2, \cdots, n_k\}$ appelés noeuds et E est un ensemble de paires d'éléments distincts de N, appelées arêtes.* □

Définition 2 *Un graphe orienté G est un couple (N, E), où N est un ensemble fini d'éléments $\{n_1, n_2, \cdots, n_k\}$ appelés noeuds et E est un ensemble de couples d'éléments distincts du produit cartésien $N \times N$, appelés arcs. Le couple $(n_i, n_j) \in E$, noté $n_i \rightarrow n_j$, représente l'arc de source n_i et de destination n_j.* □

Soit G un graphe orienté. Nous définissons les opérations suivantes :

- $src : E \mapsto N$ définie par : $src(n_i \rightarrow n_j) = n_i$.
- $dst : E \mapsto N$ définie par : $dst(n_i \rightarrow n_j) = n_j$.
- $out : N \mapsto 2^E$ définie par : $out(n_i) = \{e \in E \,|\, src(e) = n_i\}$.
- $in : N \mapsto 2^E$ définie par : $in(n_i) = \{e \in E \,|\, dst(e) = n_i\}$.
- $^- : E \mapsto E$ définie par : $\overline{e} = n_j \rightarrow n_i$ avec $e = n_i \rightarrow n_j$.

$src()$ (resp. $dst()$) retourne la source (resp. destination) d'un arc et $out()$ (resp. $in()$) calcule l'ensemble des arcs de source (resp. destination) un noeud donné. Finalement, pour un arc $e = n_i \rightarrow n_j$, $\overline{e}$ dénotera l'arc $n_j \rightarrow n_i$ appelé *conjugué* de e.

Définition 3 *Un graphe G (orienté ou non) est étiqueté par l'alphabet L s'il existe une fonction d'étiquetage λ_G de E vers L. Dans ce cas, G est noté $G = (N, L, E)$.* □

Par abus de notation et lorsque G est orienté, nous utiliserons $n_i \xrightarrow{l} n_j \in E$ pour dénoter $(n_i, n_j) \in E$ et $\lambda_G((n_i, n_j)) = \lambda_G(n_i \rightarrow n_j) = l$. Lorsque l'ensemble L est égal à $\mathbb{T}$, l est appelé le poids de l'arc $n_i \rightarrow n_j$ dans G. Dans ce cas, la fonction d'étiquetage λ_G est notée w_G.

Les graphes considérés dans ce document sont tous orientés.

L'ensemble des graphes orientés et étiquetés de noeuds N sur l'ensemble L est noté $\overrightarrow{G}(N, L)$. Un graphe de $\overrightarrow{G}(N, L)$ sera simplement dit étiqueté.

Définition 4 (Graphe complet) *Un graphe est complet si pour toute paire de noeuds distincts $\{n_i, n_j\}$, les arcs $n_i \to n_j$ et $n_j \to n_i$ existent.* □

L'ensemble des graphes complets et étiquetés sur L est noté $\overrightarrow{G}_c(N, L)$ ($\overrightarrow{G}_c(N, L) \subseteq \overrightarrow{G}(N, L)$).

Remarque 1 *Remarquons que si $G = (N, L, E)$ et $G' = (N, L, E')$ sont deux graphes de $\overrightarrow{G}_c(N, L)$, alors G et G' sont identiques s'ils sont vus comme de simples graphes, mais ils diffèrent sur les étiquettes des arcs.* □

Les graphes étiquetés considérés dans ce document sont définis sur $\mathbb{T}$.

2.1.2 Chemin

Un *chemin* p (fini ou infini) est une séquence d'arcs $e_1.e_2 \cdots e_n(\cdots)$ (les e_i sont des arcs). Un chemin de *longueur* n est un chemin de n arcs. p *traverse* le noeud n_k s'il existe $i \in [1, n-1]$ tel que $dst(e_i) = src(e_{i+1}) = n_k$. Ainsi, un chemin de source n_k n'est pas forcement un chemin qui traverse n_k. Soit e un arc. Alors $path_G(e)$ dénote l'ensemble des chemins dans G de source $src(e)$ et de destination $dst(e)$. Un *cycle de racine* n_k est un chemin de n_k à lui même. Un cycle *élémentaire* est un cycle qui ne visite aucun noeud deux fois à l'exception du noeud racine. Le terme *e-cycle* sera utilisé comme abréviation de cycle élémentaire. Un *2-cycle* est e-cycle qui passe exactement par deux noeuds distincts (i.e. $n_i \to n_j \to n_i$).

Les fonctions $src()$, $dst()$ et $w_G()$ sont étendues aux chemins de la façon suivante : si $p = e_1.e_2 \cdots e_n$ est un chemin alors $src(p) = src(e_1)$, $dst(p) = dst(e_n)$ et $w_G(p) = \sum_{i \in [1,n]} w_G(e_i)$. Par la suite, l'indice $_G$ sera omis dans un contexte sans confusion.

2.2 Espaces denses

Soit $V = \{v_1, v_2, \cdots, v_n\}$ un ensemble de variables à valeurs dans $\mathbb{R}^{\geq 0}$.

2.2.1 Valuation

Une *valuation* ν sur V est une fonction :

$$\nu : V \mapsto \mathbb{R}^{\geq 0}$$

associant une valeur réelle positive à toute variable de V. $\mathcal{V}(V)$ est l'ensemble des valuations de V. Soient $X \subseteq V$, $d \in \mathbb{R}^{\geq 0}$ et $\nu \in \mathcal{V}(V)$. Alors, $\nu[X := 0]$, $\nu + d$, $\nu - d$ et $d.\nu$ sont les

valuations définies, respectivement, par :

- $\nu[X := 0](x) = \nu(x)$ si $x \notin X$ et $\nu[X := 0](x) = 0$ autrement.
- $(\nu + d)(x) = \nu(x) + d$ pour toute variable $x \in V$.
- $(\nu - d)(x) = \nu(x) - d$ pour toute variable $x \in V$.
- $(d.\nu)(x) = d \times \nu(x)$ pour toute variable $x \in V$.

Intuitivement, $\nu[X := 0]$ affecte à chaque variable de X la valeur 0 et laisse le reste des variables inchangé ; $\nu + d$ est obtenue à partir de ν en rajoutant à chaque variable la valeur d. D'une façon similaire, $d.\nu$ est la valuation ν' telle que $\forall x \in V$, $\nu'(x) = d \times \nu(x)$

c-équivalence [145]. Soit $c \in \mathbb{N}$. Deux valuations ν et ν' sur V sont dites *c-équivalentes* si :

- pour toute variable $x \in V$, $\nu(x) = \nu'(x)$ ou ($\nu(x) > c$ et $\nu'(x) > c$).
- pour tout couple $(x, y) \in V^2$, $\nu(x) - \nu(y) = \nu'(x) - \nu'(y)$ ou ($|\nu(x) - \nu(y)| > c$ et $|\nu'(x) - \nu'(y)| > c$).

2.2.2 Polyèdre

Contrainte de variables. Une *contrainte atomique* compare une seule variable ou la différence de deux variables avec une constante naturelle. Une contrainte atomique sur V est une expression de la forme :

$$x \bowtie n \textit{ ou } x - y \bowtie m \quad \textit{avec } (x, y) \in V^2,\ (n, m) \in \mathbb{N}^2 \textit{ et} \bowtie \in \{\leq, \geq\}.$$

Une contrainte de la forme $x - y \bowtie m$ est dite *diagonale*. L'ensemble des formules qui sont conjonctions finies de contraintes atomiques est noté $\Phi(V)$.

Polyèdres. Les éléments de $\Phi(V)$ sont appelés des *polyèdres* sur V [1]. Nous écrirons **false** pour le polyèdre correspondant à l'ensemble vide $\emptyset$, **true** pour le polyèdre $\bigwedge_{\forall x \in V} x \geq 0$ et **zero** pour le polyèdre $\bigwedge_{\forall x \in V} x \geq 0 \wedge \bigwedge_{\forall x \in V} x \leq 0$.

Soient $\nu \in \mathcal{V}(V)$ et $Z \in \Phi(V)$. Alors, ν *satisfait* Z, notée $\nu \in Z$, si ν satisfait toutes les contraintes de Z. Z est *borné* s'il existe un entier $d \in \mathbb{N}$ tel que pour toute $\nu \in Z$, $\nu + d \notin Z$. Z est *équivalent* à Z', noté $Z \sim Z'$, si pour tout couple $(\nu, \nu') \in Z \times Z'$, on a $\nu \in Z'$ et $\nu' \in Z$ (Z et Z' représentent la même portion d'espace, i.e. les mêmes valuations).

[1] Notons que tout polyèdre sur V peut être vu comme un sous-ensemble de $\mathcal{V}(V)$.

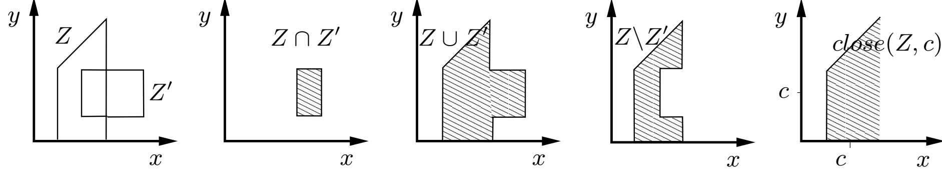


FIG. 3: Opérations sur les polyèdres (1).

Remarque 2 *Notons qu'un polyèdre Z est un ensemble convexe : pour toutes les $\nu_1, \nu_2 \in Z$, pour tout $0 < d < 1 \Rightarrow d.\nu_1 + (1-d).\nu_2 \in Z$.* □

2.2.3 Opérations sur les polyèdres

Par définition, les opérations d'intersection $\cap$ et de complément '$\backslash$' sont bien définies sur $\Phi(V)$. La FIG.3 illustre ces opérations.

c-clôture [145]. Étant donné un polyèdre Z, $close(Z, c)$ est le plus grand polyèdre contenant Z qui vérifie : pour toute $\nu' \in Z'$, il existe $\nu \in Z$ telle que ν et ν' sont c-équivalentes. Intuitivement, Z' est obtenu à partir de Z en ignorant les contraintes invoquant des constantes plus grandes que c. La FIG.3 illustre un exemple de $close(Z, c)$. Finalement, Z est *c-close* si $close(Z, c) = Z$.

Lemme 1 *Soient $c, c' \in \mathbb{N}$.*

1. *Si Z est c-close, alors Z est c'-close, pour tout $c' > c$.*
2. *Si Z et Z' sont c-close, alors $Z \cap Z'$ et $Z \cup Z'$ sont c-close.*
3. *Pour tout Z, il existe c tel que Z est c-close.*
4. *Pour toute constante c, l'ensemble des polyèdres de $\Phi(V)$ c-close est fini.* □

Par la suite, $c_{max}(Z)$ dénotera la plus petite constante c tel que Z est c-close.

Preuve. La preuve du lemme est donnée dans [145]. □

Soient $Z \in \Phi(V)$, $X \subseteq V$ et $d \in \mathbb{R}^{\geq 0}$. Nous notons par :

$$Z[X := 0] = \{\nu[X := 0] \mid \nu \in Z\} \qquad [X := 0]Z = \{\nu \mid \nu[X := 0] \in Z\}$$

$$Z^{\uparrow} = \{\nu + d \mid \nu \in Z, d \in \mathbb{R}^{\geq 0}\} \qquad Z^{\downarrow} = \{\nu - d \mid \nu \in Z, d \in \mathbb{R}^{\geq 0}\}$$

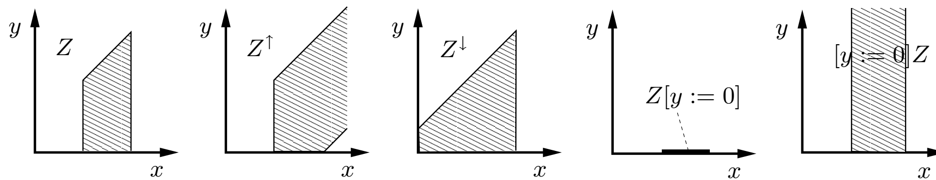


FIG. 4: Opérations sur les polyèdres (2).

Intuitivement, $Z[X := 0]$ contient toutes les valuations obtenues à partir des valuations de Z, en affectant 0 aux variables de X. $[X := 0]Z$ contient les valuations qui, après avoir affecté 0 aux variables de X, conduisent à une valuation de Z. $Z^{\uparrow}$ contient les valuations obtenues après dilatation des valuations de Z. $Z^{\downarrow}$ est l'opération inverse. La FIG.4 illustre les différentes opérations.

Le lemme suivant découle directement des définitions des différentes opérations.

Lemme 2 *Soient $Z \in \Phi(V)$ et $X \subseteq V$. Alors $Z[X := 0]$, $[X := 0]Z$, $Z^{\uparrow}$ et $Z^{\downarrow}$ sont des polyèdres de $\Phi(V)$.* □

Chapitre 3

Modèles pour les systèmes non-temporisés

Sommaire

3.1 Alphabet. **24**
3.2 Machine à états finis **25**
3.3 Système de transitions **27**
3.3.1 Système de transitions étiqueté (LTS) 27
3.3.2 Composition synchrone des LTSs 28
3.3.3 Notations standards des LTSs 29
3.4 Système de transitions à entrée/sortie. **29**
3.5 Réseau de Petri **31**
3.6 Terme d'une algèbre de processus **32**

Dans le cadre non-temporisé, plusieurs modèles de description ont été introduits. Dans ce chapitre, nous rappelons les concepts de base relatives aux modèles : machines à états finis, systèmes de transitions, réseaux de Petri et algèbres de processus. Le modèle des systèmes de transitions sera le modèle de base de nos travaux relatifs aux systèmes non-temporisés.

3.1 Alphabet.

Un *alphabet* Σ est un ensemble fini de symboles appelés *actions* ou *événements*. L'ensemble des séquences finies et non vides de Σ est noté Σ^+. La séquence vide est noté ϵ. Σ^* dénote l'ensemble des séquences de Σ, i.e. l'ensemble $\Sigma^+ \cup \{\epsilon\}$. De plus, τ désigne une action qui n'appartient pas à Σ. τ est dite action *silencieuse* ou action *interne*. L'ensemble $\Sigma \cup \{\tau\}$ est noté Σ_τ.

Une *séquence* est une suite finie d'actions que l'on note simplement par juxtaposition :

$$\sigma = a_1 a_2 \cdots a_n, \ a_i \in \Sigma$$

La *longueur* d'une séquence σ, notée $|\sigma|$, est le nombre d'actions qui la composent. σ_i dénote la *ième* action de σ (l'indice i est compris entre 0 et $|\sigma|$). Le *produit* de concaténation de deux séquences $\sigma_1 = a_1 a_2 \cdots a_n$ et $\sigma_2 = b_1 b_2 \cdots b_m$ est la séquence σ :

$$\sigma = a_1 a_2 \cdots a_n b_1 b_2 \cdots b_m$$

Bien entendu, on a $\epsilon.\sigma = \sigma.\epsilon = \sigma$ et $|\sigma_1.\sigma_2| = |\sigma_1| + |\sigma_2|$.

Exemple 3.1 *Les gènes sont des séquences sur l'alphabet $\{A, C, G, T\}$. Les entiers naturels, écrits en base 10, sont des séquences sur l'alphabet des dix chiffres décimaux. Le code d'entrée d'un immeuble (le "digicodes") est une séquence écrite sur un alphabet à 12 symboles.* □

Soit Σ un alphabet et L une partie de Σ ($L \subseteq \Sigma$). Pour toute séquence $\sigma = a_1 \cdots a_n \in \Sigma^*$, $\sigma_{|L}$ est la séquence projection de σ sur L définie par :

- Pour tout $a \in \Sigma$: $a_{|L} = a$ si $a \in L$ et $a_{|L} = \epsilon$ si $a \notin L$,
- Pour tout $\sigma = a_1 a_2 \cdots a_n$: $\sigma_{|L} = a_{1|L} a_{2|L} \cdots a_{n|L}$.

3.2 Machine à états finis

Les machines à états finis (en anglais FSM pour Finite State Machine) sont l'un des modèles les plus anciens. Elles sont issues de la modélisation des circuits logiques. Une FSM est une machine ayant une quantité finie de mémoire pour représenter les états et qui distingue entre une sortie (signal émis) et une entrée (signal reçu).

Il existe deux catégories de machines à états finis séquentielles : machine de Moore et machine de Mealy. Dans une machine de Moore, les sorties dépendent des états seulement, tandis que dans une machine de Mealy les sorties dépendent des états et des entrées. Par conséquence, une machine de Moore a plus d'états et moins de logiques à la sortie et une machine de Mealy a moins d'états et plus de logiques à la sortie. Une théorie complète sur FSM peut être trouvée dans [62]. Nous rappelons ici les notions de base relatives aux machines de Mealy.

Machine de Mealy. C'est un graphe décrivant le comportement de la spécification. Les noeuds du graphe décrivent les états internes du système. Les arcs (transitions) sont étiquetés par un ensemble de symboles d'entrée et un ensemble de symboles de sortie. Les sorties changent immédiatement en réaction à un changement d'entrée de manière

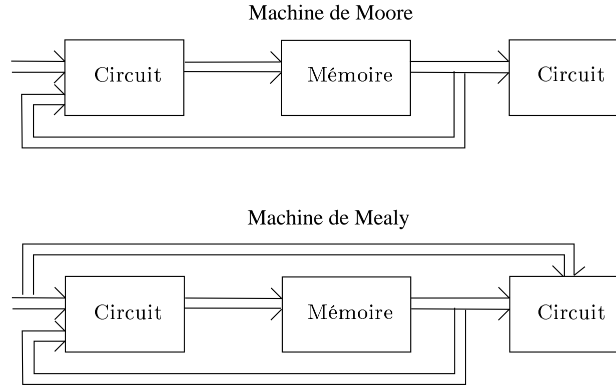


FIG. 5: Machines à états finis.

asynchrone (contrairement aux sorties d'une machine de Moore qui sont synchronisées). Seuls les changements d'états sont synchrones. Formellement,

Définition 5 (FSM) *Une machine de Mealy est un 6-uplet* $(s_0, S, I, O, \delta, \gamma)$ *où :*

1. s_0 *est l'état initial,*
2. S *est un ensemble fini d'états,*
3. I *est un alphabet fini d'actions d'entrée,*
4. O *est un alphabet fini d'actions de sortie,*
5. $\delta : S \times I \mapsto S$ *est la fonction de changement d'états, qui associe à une entrée* i *dans un état* s*, l'état destination* $s' = \delta((s, i))$*,*
6. $\gamma : S \times I \mapsto O$ *est la fonction de sortie, qui associe à une entrée* i *dans un état* s*, la sortie correspondante* $o = \delta((s, i))$*.* □

Par convention, une transition est notée $s \xrightarrow{i/o} s'$. Si l'entrée i est reçue alors que le système est dans l'état s, la sortie o est produite et le nouvel état du système est s'. Dans ce cas, i est aussi appelé le *déclencheur* (trigger). Si une entrée est reçue et qu'aucune transition n'est définie pour cette entrée, il ne se passe rien (l'entrée est ignorée).

Exemple 3.2 *La* FIG.*6 illustre un exemple simple d'une machine de Mealy. L'alphabet d'entrée et de sortie de cette machine est* $\{1, 0\}$*.* □

Remarque 3 *Remarquons que (1) les FSMs ne fournissent pas une définition hiérarchique, (2) un état représente l'état complet du système, (3) le système est dans un seul état à la fois et finalement (4) une transition est atomique (elle ne peut être décomposée).* □

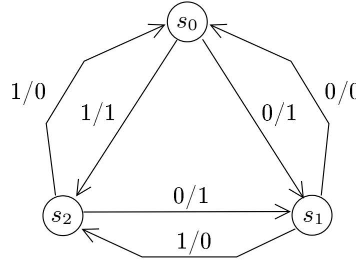


Fig. 6: Machine de Mealy.

Une machine de Mealy $M = (s_0, S, I, O, \delta, \gamma)$ est dite *déterministe* si pour tous les états $s, s', s'' \in S$, pour toutes les entrées $i, i' \in I$ et toutes les sorties $o, o' \in O$, si $s \xrightarrow{i/o} s'$, $s \xrightarrow{i'/o'} s''$ et $s' \neq s''$ alors $i \neq i'$.

Sémantique d'une machine de Mealy. La sémantique d'une machine de Mealy $M = (s_0, S, I, O, \delta, \gamma)$ est la fonction $g_M \;\; I^+ \mapsto O$ définie par les équations récursives suivantes, avec d_M une fonction auxiliaire, $i \in I$ et $t \in I^*$.

$$d_M(\epsilon) = s_0, \quad d_M(t.i) = \delta((d_M(t), i)) \quad g_M(t.i) = \gamma((d_M(t), i))$$

Finalement, le modèle de base FSM a été (1) étendu de plusieurs façons : modèle de communications, canaux (CFSM) et (2) enrichi par les variables, les paramètres et les prédicats (EFSM) [62].

3.3 Système de transitions

Tout système réel (par exemple un programme) peut se décrire par un système de transitions, i.e. un ensemble quelconque d'états et de transitions étiquetées par des actions entre les états. Lorsque l'on se trouve dans l'un des états du système de transitions, il est possible de changer d'état en effectuant une action qui étiquette l'une des transitions sortantes de cet état. Une exécution dans un système de transitions est alors une séquence d'actions $(a_i)_{i \in [1,n]}$ notée $a_1 \cdots a_n$.

3.3.1 Système de transitions étiqueté (LTS)

Il est défini en terme d'états et de transition étiquetées entre états, où les étiquettes indiquent le comportement produit durant la transition. Formellement,

Définition 6 (LTS) *Un LTS [10, 11] est un quadruple* $(Q, q_0, \Sigma, \rightarrow)$ *tel que :*

- *Q est un ensemble dénombrable d'états,*
- *$q_0 \in Q$ est l'état initial,*
- *Σ est un alphabet dénombrable d'actions,*
- *$\rightarrow \subset Q \times \Sigma_\tau \times Q$ est un ensemble de transitions. Un élément (s, a, s') de $\rightarrow$ sera simplement noté $s \xrightarrow{a} s'$.* □

Une action d'un LTS peut prendre plusieurs formes : une entrée, une sortie, un appel de méthode, etc. Les états sont souvent une abstraction des états du système (habituellement, il est impossible de représenter tous les états du système). Les actions de Σ sont dites actions *observables*. Une transition peut être étiquetée soit par une action observable, ou par une action interne (inobservable, ou encore silencieuse) τ.

Remarque 4 *La différence majeure entre un automate et un LTS est que l'ensemble des états et des actions d'un automate est fini, ce qui peut ne pas être le cas pour un LTS.* □

3.3.2 Composition synchrone des LTSs

Il est difficile de représenter le parallélisme dans le cas des LTSs, mais la composition synchrone entre deux LTSs peut s'exprimer au moyen d'un opérateur de composition parallèle [102, 108, 115], noté $\|$. Nous verrons dans la section 4.3.5 une définition plus générale de la composition des LTSs.

Définition 7 (Composition) *Soient $A = (Q^A, q_0^A, \Sigma^A, \rightarrow_A)$ et $B = (Q^B, q_0^B, \Sigma^B, \rightarrow_B)$ deux LTSs. La composition synchrone de A et B, notée $A\|B$, est LTS $C = (Q^C, q_0^C, \Sigma^C, \rightarrow_C$) défini par :*

1. *$q_0^C = (q_0^A, q_0^B)$,*
2. *$Q^C = \{(q_1, q_2) \mid q_1 \in Q^A, q_2 \in Q^B\}$,*
3. *$\Sigma^C \subset \Sigma^A \cup \Sigma^B$,*
4. *$\rightarrow_C$ est obtenue par l'application de l'une des règles suivantes :*
   (a) $q_1 \xrightarrow{\mu}_A q_1'$, $q_2 \xrightarrow{\mu}_B q_2'$, $\mu \in \Sigma^A \cap \Sigma^B \quad \Rightarrow \quad (q_1, q_2) \xrightarrow{\mu}_C (q_1', q_2')$
   (b) $q_1 \xrightarrow{\mu}_A q_1'$, $\mu \in \Sigma_\tau^A$, $\mu \notin \Sigma^B \quad \Rightarrow \quad (q_1, q_2) \xrightarrow{\mu}_C (q_1', q_2)$
   (c) $q_2 \xrightarrow{\mu}_B q_2'$, $\mu \in \Sigma_\tau^B$, $\mu \notin \Sigma^A \quad \Rightarrow \quad (q_1, q_2) \xrightarrow{\mu}_C (q_1, q_2')$. □

Dans la première règle, A et B exécutent simultanément l'action de synchronisation. Les deux autres règles correspondent à une évolution indépendante de A et de B.

3.3.3 Notations standards des LTSs

Nous introduisons par la suite quelques notations standards des LTSs.

Soient $M = (Q, q_0, \Sigma, \rightarrow)$ un LTS, $\mu_{(i)} \in \Sigma_\tau$ un ensemble d'actions, $\alpha_{(i)} \in \Sigma$ un ensemble d'actions observables, $\sigma \in \Sigma^*$ une séquence d'actions observables et $q, q' \in Q$ deux états.

- $q \xrightarrow{a} \quad \triangleq \quad \exists\, q' \,|\, q \xrightarrow{a} q'$.
- $q \xrightarrow{\mu_1 \ldots \mu_n} q' \quad \triangleq \quad \exists\, q_0 \ldots q_n \,|\, q = q_0 \xrightarrow{\mu_1} q_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} q_n = q'$.

Le comportement observable de M est décrit par la relation $\Rightarrow$:

- $q \overset{\epsilon}{\Rightarrow} q' \quad \triangleq \quad q = q'$ ou $q \xrightarrow{\tau \ldots \tau} q'$.
- $q \overset{\alpha}{\Rightarrow} q' \quad \triangleq \quad \exists\, q_1, q_2 \,|\, q \overset{\epsilon}{\Rightarrow} q_1 \xrightarrow{\alpha} q_2 \overset{\epsilon}{\Rightarrow} q'$.
- $q \overset{\alpha_1 \ldots \alpha_n}{\Longrightarrow} q' \quad \triangleq \quad \exists\, q_0 \ldots q_n \,|\, q = q_0 \overset{\alpha_1}{\Rightarrow} q_1 \overset{\alpha_2}{\Rightarrow} \ldots \overset{\alpha_n}{\Rightarrow} q_n = q'$.
- $q \overset{\sigma}{\Rightarrow} \quad \triangleq \quad \exists\, q' \,|\, q \overset{\sigma}{\Rightarrow} q'$.
- $q\, after\, \sigma \quad \triangleq \quad \{q' \in Q \,|\, q \overset{\sigma}{\Rightarrow} q'\}$, l'ensemble des états atteignables à partir de q par σ. Par extension, M after $\sigma \triangleq q_0$ after σ.
- $Traces(q) \quad \triangleq \quad \{\sigma \in \Sigma^* \,|\, q \overset{\sigma}{\Rightarrow}\}$, l'ensemble des séquences observables tirables à partir de q. L'ensemble des séquences observables de M est $Traces(M) \triangleq Traces(q_0)$.

3.4 Système de transitions à entrée/sortie.

La distinction entre les entrées et les sorties nécessite un modèle plus riche. Plusieurs modèles ont été proposés dont les automates à entrée/sortie (IOA pour Input Output Automata) de Lynch [104], les automates à entrée/sortie (IOSM pour Input Output State Machines) de Phalippou [123] et les systèmes de transitions à entrée/sortie (IOLTS Input Output Transition Systems) de Tretmans [144]. Contrairement aux machines de Mealy dont les transitions portent à la fois une entrée et une sortie, les transitions de ces modèles sont soit des entrées, soit des sorties, soit des actions internes. Ces modèles sont tous similaires ainsi que les travaux qui en ont découlé. Nous nous intéressons ici principalement au modèle des IOLTSs.

Système de transitons à entrée/sortie. Les IOLTSs sont simplement des LTSs où l'on distingue deux types d'actions observables : les entrées et les sorties. Formellement,

Définition 8 (IOLTS) *Un IOLTS $M = (Q, q_0, \Sigma, \rightarrow)$ est un LTS dont l'alphabet Σ est partitionné en deux ensembles $\Sigma = \Sigma_o \cup \Sigma_i$, avec Σ_o l'alphabet de sortie et Σ_i l'alphabet d'entrée.* □

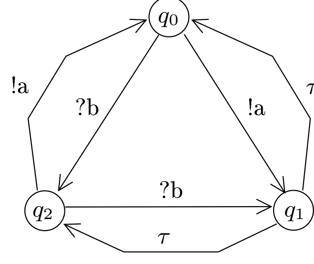


FIG. 7: IOLTS.

L'ensemble des IOLTSs définis sur Σ est noté $\mathcal{IOLTS}(\Sigma)$ ou encore $\mathcal{IOLTS}(\Sigma_i, \Sigma_o)$. Une entrée $a \in \Sigma_i$ est notée $?a$ et une sortie $a \in \Sigma_o$ est notée $!a$.

Exemple 3.3 *La* FIG. *7 illustre un exemple d'un IOLTS. Il est composé de trois états et six transitons. L'alphabet Σ correspond à $\{!a, ?b\}$ avec $\Sigma_i = \{?b\}$ et $\Sigma_o = \{!a\}$.* □

Notations des IOLTSs. En plus des notations standards des LTSs, nous utiliserons les notations suivantes pour les IOLTSs :

- $Out(q) \triangleq \{a \in \Sigma_o \mid q \stackrel{a}{\Rightarrow}\}$, l'ensemble des sorties possibles de q.
- $Out(P) \triangleq \{Out(q) \mid q \in P\}$, l'ensemble des sorties possibles de $P \subseteq Q$.
- $Out(M, \sigma) \triangleq Out(M \text{ after } \sigma)$.

Exemple 3.4 *Prenons encore la* FIG. *7.*
- $q_0 \stackrel{!a}{\Rightarrow} q_1 \triangleq q_0 \stackrel{!a}{\rightarrow} q_1,\ q_0 \stackrel{?b}{\Rightarrow} q_2 \triangleq q_0 \stackrel{?b}{\rightarrow} q_2,$
- $q_1 \stackrel{!a}{\Rightarrow} q_0 \triangleq q_1 \stackrel{\tau}{\rightarrow} q_2 \stackrel{!a}{\rightarrow} q_0,\ q_1 \stackrel{!a}{\Rightarrow} q_1 \triangleq q_1 \stackrel{\tau}{\rightarrow} q_0 \stackrel{!a}{\rightarrow} q_1,\ q_1 \stackrel{?b}{\Rightarrow} q_1 \triangleq q_1 \stackrel{\tau}{\rightarrow} q_2 \stackrel{?b}{\rightarrow} q_1,$
- $q_2 \stackrel{!a}{\Rightarrow} q_0 \triangleq q_2 \stackrel{!a}{\rightarrow} q_0,\ q_2 \stackrel{?b}{\Rightarrow} q_1 \triangleq q_2 \stackrel{?b}{\rightarrow} q_1,$

Ainsi, on peut déduire que :

- $Out(q_0) = Out(q_1) = Out(q_2) \triangleq \{!a\},$
- $Out(M, !a) = Out(M, ?b) \triangleq \{!a\},$
- $Out(M, ?b.!a) = Out(M, !a.?b) \triangleq \{!a\}.$ □

Soit $M = (Q, q_0, \Sigma, \rightarrow)$ un IOLTS. M est dit :

- *déterministe* si aucun état n'admet plus d'un successeur par une action observable. Formellement, pour tout $a \in \Sigma$, pour tout $q \in Q$, si $q \stackrel{a}{\Rightarrow} q_1$ et $q \stackrel{a}{\Rightarrow} q_2$ alors $q_1 = q_2$
- *observable* si aucune transition n'est étiquetée par τ. Formellement, si $q \stackrel{a}{\rightarrow}$ alors $a \neq \tau$.

– *input-complet* s'il accepte toute entrée a dans chaque état. Formellement, pour tout $a \in \Sigma_i$, pour tout $q \in Q$, alors $q \stackrel{a}{\Rightarrow}$.

Exemple 3.5 *Il est facile de voir que l'IOLTS de la* FIG.*7 est :*
– *indéterministe (vu que* $q_1 \stackrel{!a}{\Rightarrow} q_1$ *et* $q_1 \stackrel{!a}{\Rightarrow} q_0$*),*
– *inobservable (*$q_1 \stackrel{\tau}{\rightarrow} q_0$*),*
– *input-complet.* □

3.5 Réseau de Petri

Les systèmes informatiques ou de commande d'automatismes logiques sont conçus comme des ensembles de sous-systèmes. De plus en plus, ces sous-systèmes seront autorisés à fonctionner en parallèle (le partage des ressources). Les problèmes dus au parallélisme et sa modélisation ont conduit à des nombreux travaux sur le plan théorique (parallélisme, communication, synchronisation). Tant par l'étendue de leurs résultats théoriques que par la diversité et le nombre de leurs applications (informatique, logiciel ou matériel), les réseaux de Petri constituent, aujourd'hui, le modèle formel le plus avancé et le plus complet pour la description logique des structures du contrôle de parallélisme et du contrôle de l'information.

Réseaux de Petri (RdP). Les *réseaux de Petri* sont un formalisme mathématique de modélisation et d'analyse des comportements dynamiques d'un système discret, introduit par le mathématicien allemand Carl Adam Petri [120, 121] durant sa thèse en 1962. Ils ont été développés au MIT (Massachusetts Institute of Technology) pour spécifier, modéliser et comprendre les systèmes dans lesquels plusieurs processus sont interdépendants. Un réseau de Petri (RdP) se compose de quatre objets :
– Les places (graphiquement représentées par des cercles)
– Les transitions (graphiquement représentées par des rectangles)
– Les flux (graphiquement représentés par des flèches)
– Les jetons (graphiquement représentés par des disques noirs)

L'ensemble des places permet de représenter les états du système. L'ensemble des transitions permet de représenter l'ensemble des événements dont l'occurrence provoque la modification des états du système. Formellement,

Définition 9 (RdP) *Un RdP est un quadruplet* (P, T, F, B) *tel que :*
– P *est un ensemble fini de places,*
– T *est un ensemble fini de transitions, disjoint de* P*,*
– B *est la fonction d'incidence arrière (backward function), reliant des places à des transitions,* $B : P \times T \mapsto \mathbb{N}$*,*

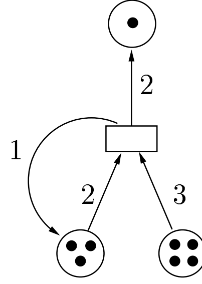


FIG. 8: Réseau de Petri marqué.

– *F est la fonction d'incidence avant (forward function), reliant des transitions à des places, $F : P \times T \mapsto \mathbb{N}$.* □

Un *marquage* de RdP est une fonction M de l'ensemble des places dans $\mathbb{N}$. Un RdP marqué est un couple (N, M_0) tel que

– N est un réseau de Petri,
– $M_0 : P \mapsto \mathbb{N}$ est un marquage *initial.*

Contrairement aux automates, une transition est segmentée en arcs entrants et sortants. Chaque arc sortant est valué par un nombre entier représentant le nombre de jetons que la place de départ doit avoir pour que la transition puisse être franchie. Dans ce cas, la place de départ perd ce nombre de jetons. De même, chaque place d'arrivée reçoit un nombre de jetons égal à la valuation de l'arc entrant. Un exemple d'un RdP marqué est donné dans la FIG.8.

3.6 Terme d'une algèbre de processus

Une algèbre de processus contient un certain nombre de processus élémentaires et un certain nombre d'opérations qui, appliquées à des processus, construisent d'autres processus. À chaque processus d'une algèbre de processus, il est possible de construire un système de transitions représentant son comportement. Cette propriété très générale est largement utilisée dans l'étude des algèbres de processus. Dans cette section, nous allons seulement illustrer un exemple simple d'une algèbre de processus, notée $\mathcal{PA}$ inspirée de CCS de Milner [108] et LOTOS [39, 102].

L'ingrédient basique d'une algèbre est une action/événement élémentaire. Une action décrit une activité effectuée par le système. Cette action est supposée être atomique et ainsi ne peut être interrompue lors de son exécution. Nous supposons aussi qu'il existe un ensemble A d'actions élémentaires. Nous utiliserons les opérations suivantes :

Préfixage. Si p est un processus et a un élément de A, alors $a;p$ est un processus

qui exécute a puis se comporte comme p (ou qui attend l'événement a et lorsqu'il s'est produit se comporte comme p). Nous supposons que l'algèbre $\mathcal{PA}$ contient un processus particulier, noté NIL, qui ne fait rien.

Renommage. Si p est un processus qui exécute a puis se comporte comme p', alors $[p]$ est un processus qui exécute $[a]$ puis se comporte comme $[p']$. $[]$ est une fonction qui renomme chaque action en une autre action.

Alternative gardée ou Choix. Si p et q sont deux processus et a et b deux actions de A, alors $a; p + b; q$ est un processus qui exécute a puis se comporte comme p ou qui exécute b puis se comporte comme q.

Composition parallèle. Si p et q sont deux processus et $B \subseteq A$ un ensemble d'actions, alors $p \parallel_B q$ est un processus qui peut avoir trois types de comportements :

- Si p peut exécuter $a \notin B$ puis se comporter comme p', alors $p \parallel_B q$ peut exécuter a puis se comporter comme $p' \parallel_B q$.
- Si q peut exécuter $b \notin B$ puis se comporter comme q', alors $p \parallel_B q$ peut exécuter b puis se comporter comme $p \parallel_B q'$.
- Si p peut exécuter $a \in B$ puis se comporter comme p' et q peut exécuter $a \in B$ puis se comporter comme q', alors $p \parallel_B q$ peut exécuter a puis se comporter comme $p' \parallel_B q'$.

Restriction. Si p est un processus et a un élément de A, alors $p \backslash a$ est un processus ; si p peut exécuter $b \neq a$ puis se comporter comme p', alors $p \backslash a$ peut exécuter b puis se comporter comme $p' \backslash a$.

Récursion. Si X est une variable représentant des processus et p est un processus, l'équation récursive $X = p$ définit la valeur de X comme celle qui résout cette équation. Si p peut exécuter a et se comporter comme p' alors X peut exécuter a et se comporter comme p'. Ainsi, $X = a; X$ est un processus qui exécute infiniment souvent a [1].

Définition 10 ($\mathcal{PA}$) *Le langage de $\mathcal{PA}$ est défini par la grammaire suivante :*

$$p \;:=\; NIL \;\mid\; a;p \;\mid\; p\backslash a \;\mid\; p+p \;\mid\; p \parallel_B p \;\mid\; [p] \;\mid\; X$$

avec a le nom d'une action appartenant à l'ensemble des noms des actions A et $B \subseteq A$ un ensemble de synchronisation. □

Exemple 3.6 *Supposons que $A = \{a, b, c\}$ et considérons le processus suivant décrit dans*

[1]En CCS, la récursion est réalisée par p **where** $(x_1 = p_1, ..., x_n = p_n)$ où p_i est un processus et x est une variable de processus.

l'algèbre $\mathcal{PA}$:

$$p = ((x = a; x + b; NIL) \parallel_{\{a\}} (y = a; y + c; NIL))\backslash a$$

D'après les règles fixant la sémantique opérationnelle de $\mathcal{PA}$, p peut exécuter b dans son composant x et se trouver transformé en :

$$p_1 = ((NIL) \parallel_{\{a\}} (y = a; y + c; NIL))\backslash a$$

ou exécuter c dans son composant y et se trouver transformé en :

$$p_2 = ((x = a; x + b; NIL) \parallel_{\{a\}} (NIL))\backslash a.$$

La seule action que peut exécuter p_1 est l'action c dans son second composant et il est transformé en :

$$p_3 = (NIL \parallel_{\{a\}} NIL)\backslash a.$$

De même, p_2 ne peut exécuter que b et se trouver transformer aussi en p_3. □

Exemple 3.7 *Nous proposons de modéliser, en processus, la désactivation de l'écran de veille d'un terminal par un utilisateur. Lorsqu'un utilisateur n'active aucune touche du clavier ou de la souris, l'écran de veille est activé automatiquement après un temps fixe. À tout instant, l'écran peut être soit actif, soit inactif et dans ce cas c'est l'écran de veille qui est actif. L'utilisateur peut être modélisé par un processus simple qui consiste à activer un périphérique :*

$$User = on; User$$

Le processus de l'écran de veille est le suivant :

$$ScreenSaverOff = off; ScreenSaverOn$$

$$ScreenSaverOn = on; ScreenSaverOn + off; ScreenSaverOff.$$

Le système complet est décrit par l'interaction entre l'utilisateur et l'écran de veille sur l'action "on" :

$$System = User \parallel_{on} ScreenSaverOff$$ □

La FIG.9 donne la sémantique opérationnelle de $\mathcal{PA}$. A partir de cette sémantique, il est possible d'associer un système de transitions à un processus.

Pour plus de détails sur les algèbres de processus se référer à : ACP de Bergstra et al. [21], CSP de Tony Hoare [72], CCS de Robin Milner [108, 111], MEIJE de Simone [138] et LOTOS [39, 102].

$$a;p \xrightarrow{a} p \qquad \frac{p \xrightarrow{b} p'}{p\backslash a \xrightarrow{b} p'\backslash a}\ a \neq b \qquad \frac{p \xrightarrow{a} p'}{p+q \xrightarrow{a} p'} \qquad \frac{p \xrightarrow{a} p'}{q+p \xrightarrow{a} p'}$$

$$\frac{p \xrightarrow{a} p'}{p \parallel_B q \xrightarrow{a} p' \parallel_B q}\ a \notin B \qquad \frac{p \xrightarrow{a} p'}{q \parallel_B p \xrightarrow{a} q \parallel_B p'}\ a \notin B$$

$$\frac{p \xrightarrow{a} p' \quad q \xrightarrow{a} q'}{p \parallel_B q \xrightarrow{a} p' \parallel_B q'}\ a \in B \qquad \frac{p \xrightarrow{a} p'}{X \xrightarrow{a} p'}\ X = p \qquad \frac{p \xrightarrow{a} p'}{[p] \xrightarrow{[a]} [p']}$$

Fig. 9: Sémantique opérationnelle de $\mathcal{PA}$ en transitions.

Chapitre 4

Modèles pour les systèmes temporisés

Sommaire

4.1 Alphabet et notion de temps **37**
4.2 Système de transitions temporisé **37**
4.3 Automate temporisé **39**
4.3.1 Variable horloge 39
4.3.2 Automate temporisé d'Alur et Dill 40
4.3.3 Sémantique d'un TA 41
4.3.4 Computation 42
4.3.5 Composition synchrone des TAs 43
4.4 Automate temporisé avec invariants **44**
4.5 Automate temporisé avec urgence **45**
4.5.1 Urgence à l'Uppaal [96] 45
4.5.2 Urgence à la Kronos [33] 46
4.6 Automate temporisé à entrée/sortie **47**
4.7 Automate temporisé étendu **48**
4.8 Automate des régions **49**
4.8.1 Région d'horloges 49
4.8.2 Automate des régions 50
4.9 Autres modèles **51**

Dans le chapitre précédent, nous avons présenté quelques formalismes de description des systèmes non-temporisés. Ce chapitre est consacré à la présentation des extensions temporisés des systèmes de transitions et des automates, ainsi que d'autres formalismes connexes. Les automates temporisés seront à la base de nos travaux relatifs aux systèmes temps-réel.

4.1 Alphabet et notion de temps

Soit Σ un alphabet. Une action *temporisée* (ou événement temporisé) sur Σ est un couple $u = (a, d)$ avec $a \in \Sigma$ et $d \in \mathbb{T}^{\geq 0}$ [1]. Si a est interprété comme l'occurrence de l'événement a alors d est interprété comme l'instant d'occurrence de a. Pour une action temporisée u, $event(u)$ dénote l'événement associé à u et $time(u)$ le réel associé à u. Par exemple, si $u = (a, d)$ alors $event(u) = a$ et $time(u) = d$.

Une séquence temporisée $\sigma = (a_1, d_1)...(a_n, d_n)$ sur Σ est un élément de $(\Sigma \times \mathbb{T}^{\geq 0})^*$ tel que la suite $(d_i)_{i \in [1,n]}$ est croissante. Par exemple, $\sigma = (a_1, 3).(a_2, 5)$ est une séquence temporisée. Cependant, $\sigma' = (a_1, 3).(a_2, 2)$ ne l'est pas. Le temps passé dans la séquence σ, noté $delay(\sigma)$, est l'instant d'occurrence du dernier événement temporisé de σ : $delay(\sigma) = time(\sigma_{|\sigma|})$, avec $delay(\epsilon) = 0$. Finalement, l'ensemble des séquences temporisées sur Σ est noté $TW(\Sigma)$.

4.2 Système de transitions temporisé

Les systèmes de transitions *temporisés* sont des systèmes de transitions particuliers dans lesquels deux types d'actions peuvent être distinguées :

- des actions d'événement qui correspondent à l'occurrence d'un événement.
- des actions d'attente qui correspondent à un écoulement du temps et non à une action visible. Une action attente de d unités sera notée $\epsilon(d)$.

Une sémantique particulière est donnée aux exécutions dans les systèmes de transitions temporisés. Une exécution va être décrite par une séquence temporisée $(a_1, d_1) \cdots (a_n, d_n)$. d_i est le réel qui représentant la date à laquelle l'action a_i a été effectuée, ce qui correspond à la somme des attentes avant l'action a_i. Par exemple, l'exécution $1.2\,a\,2.2\,b$ va être représentée par la séquence temporisée $(a, 1.2).(b, 3.4)$.

Système de transitions temporisé (TLTS)

Définition 11 (TLTS) *Un système de transitions étiqueté et temporisé [94, 34] sur l'alphabet Σ et le domaine de temps $\mathbb{T}$ est un système de transitions $(Q, q_0, \Gamma, \rightarrow)$ avec $\Gamma = \Sigma \cup \{\epsilon(d) \mid d \in \mathbb{T}\}$. La relation de transition $\rightarrow$ vérifie les propriétés suivantes :*

- *Déterminisme temporel : pour tous les états q, q', q'' de Q et pour tout $d \in \mathbb{T}$, si $q \xrightarrow{\epsilon(d)} q'$ et $q \xrightarrow{\epsilon(d)} q''$ alors $q' = q''$.*
- *Additivité du temps : pour tous les états q, q'' de Q et pour tous les $d, d' \in \mathbb{T}$, si $q \xrightarrow{\epsilon(d+d')} q''$ alors il existe un état $q' \in Q$ tel que $q \xrightarrow{\epsilon(d)} q'$ et $q' \xrightarrow{\epsilon(d')} q''$.*

[1] Une action temporisée est un élément de $\Sigma \times \mathbb{T}^{\geq 0}$.

– *Attente 0 : pour tous les états q, q' de Q, $q \xrightarrow{\epsilon(0)} q'$ si et seulement si $q = q'$.* □

Remarquons que cette définition n'impose pas que les TLTSs aient un nombre fini d'états. Soit S un TLTS. Une exécution de S est une suite de transitions consécutives de l'état initial q_0 :

$$q_0 \xrightarrow{\alpha_1} q_1 \xrightarrow{\alpha_2} q_2 \cdots \xrightarrow{\alpha_n} q_n$$

où $q_{i-1} \xrightarrow{\alpha_i} q_i$ est une transition de S, pour tout $i \in [1, n]$ ($n \in \mathbb{N}$). Une exécution d'attente de S est une suite de transitions consécutives de l'état initial q_0 :

$$q_0 \xrightarrow{\alpha_1} q_1 \xrightarrow{\alpha_2} q_2 \cdots \xrightarrow{\alpha_n} q_n$$

telle que pour tout $i \in [1, n]$, $\alpha_i = \tau$ ou $\alpha_i = \epsilon(d_i)$ pour un certain $d_i \in \mathbb{T}$. Rappelons que τ correspond à une action invisible.

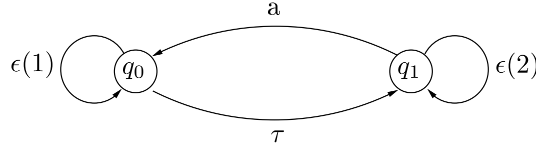


FIG. 10: Système de transitions temporisé.

Exemple 4.1 *Considérons le système de transitions $S = (\{q_0, q_1\}, q_0, \{a, \tau, \epsilon(1), \epsilon(2)\}, \rightarrow)$ de la* FIG.*10. Une exécution de ce système est :*

$$q_0 \xrightarrow{\epsilon(1)} q_0 \xrightarrow{\tau} q_1 \xrightarrow{\epsilon(2)} q_1 \xrightarrow{a} q_0 \xrightarrow{\epsilon(1)} q_0 \xrightarrow{\epsilon(1)} q_0.$$

Une exécution d'attente de S est :

$$q_0 \xrightarrow{\epsilon(1)} q_0 \xrightarrow{\tau} q_1 \xrightarrow{\epsilon(2)} q_1.$$

□

Notations des TLTSs

Nous définissons, par la suite, quelques notations pour les TLTSs. Soient $S = (Q, q_0, \Gamma, \rightarrow)$ un système de transitions temporisé sur l'alphabet Σ, q, q' deux états, $d \in \mathbb{T}$ et $a \in \Sigma$. Alors :

- $q \overset{a}{\Rightarrow} q'$ $\quad\triangleq\quad$ il existe $q'' \in Q$ tel que $q \xrightarrow{\tau}{}^{*} q'' \xrightarrow{a} q'$.
- $q \overset{\epsilon(d)}{\Rightarrow} q'$ $\quad\triangleq\quad$ il existe une exécution d'attente $q = q_0 \xrightarrow{\alpha_1} q_1 \xrightarrow{\alpha_2} q_2 \cdots \xrightarrow{\alpha_n} q_n = q'$, tel que $d = \sum\{d_i \mid \alpha_i = \epsilon(d_i)\}$.

où la relation $\xrightarrow{\tau}^{*}$ représente la clôture réflexive de $\xrightarrow{\tau}$. Remarquons que la relation $q \overset{a}{\Rightarrow} q$ abstrait uniquement les actions silencieuses qu'il est possible de faire avant l'action a et que $\xrightarrow{\tau}^{*}$ correspond à $\overset{\epsilon(0)}{\Rightarrow}$. Finalement, pour une séquence temporisée $\sigma = (a_1, d_1)...(a_n, d_n) \in \mathcal{TW}(\Sigma)$:

- $q \overset{\sigma}{\Rightarrow} q' \quad \triangleq \quad$ il existe une exécution $q = q_0 \overset{\epsilon(d'_1)}{\Rightarrow} q'_1 \overset{a_1}{\Rightarrow} q_1 \cdots \overset{\epsilon(d'_n)}{\Rightarrow} q'_n \overset{a_n}{\Rightarrow} q_n = q'$ telle que $d_i = \sum_{1 \leq j \leq i} d'_j$, pour tout $i \in [1, n]$.

4.3 Automate temporisé

Si l'on comparait les modèles à des langages de programmation, nous dirions que les systèmes de transitions temporisés sont des langages de "bas niveau" pour la modélisation des systèmes temporisés. Dans le cadre temporisé, des modèles de "haut niveau" ont été proposés en adjoignant aux modèles précédents (systèmes de transitions, réseaux de Petri et algèbres de processus) des notions de temps. Dans cette section, nous intéressons au modèle des automates temporisés [2, 4], l'extension temporisé du modèle des automates finis.

4.3.1 Variable horloge

Les automates temporisés ont été introduits par R. Alur et D. Dill [2, 4] dans les années 90. La notion du temps intervient via des variables réelles appelées *horloges*. Ces horloges ont toutes une pente égale à 1, i.e. elles avancent toutes à la même vitesse (celle du temps universel souvent implicite dans le modèle). Leurs valeurs peuvent être comparées à des constantes ou entre elles (grâce aux contraintes de variables définies dans la section 2.2.2) et elles peuvent être remises à zéro. Par la suite et lorsque les variables sont des horloges, nous parlerons de *contrainte d'horloges* pour designer une contrainte de variables.

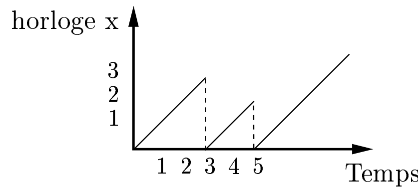


FIG. 11: Comportement d'une horloge.

Le comportement d'une horloge peut être décrit par le schéma de la FIG.11. Dans ce schéma, l'horloge x a été remise à zéro en deux occasions : la première après 3 unités de temps et la deuxième après deux nouvelles unités de temps. Les automates temporisés peuvent alors être décrits comme des automates finis auxquels ont été rajoutées des horloges. Il y a alors deux types d'évolution possible pour un tel système :

1. Soit le temps s'écoule alors que le système reste dans le même état. Dans ce cas, toutes les horloges avancent d'une valeur égale à la durée d'attente.
2. Soit il est possible de franchir une transition et ainsi d'effectuer l'action indiquée sur celle-ci. Au préalable, il faut vérifier que les valeurs des horloges satisfont la contrainte d'horloges qui est placée sur la transition, puis, après que l'action est réalisée, il faut remettre à zéro les horloges spécifiées sur la transition.

Les automates temporisés ont été très étudiés. Dès leur définition dans [2], une preuve de la décidabilité de l'accessibilité d'un état a été donnée, ainsi que bon nombre d'autres résultats théoriques. Ensuite, viendront des algorithmes permettant de vérifier qu'un automate donné vérifie une propriété logique et/ou temporelle et des outils comme Uppaal, Kronos, CMC. Forts de ces succès, beaucoup de variantes et extensions ont été proposées au modèle initial qui ne comportait que des gardes sur les transitions. On a vu ajouter une distinction entre actions urgentes et non urgentes [33, 96] comme dans l'algèbre de processus CCS, ainsi que l'introduction des invariants sur les états pour forcer l'évolution du système. Les gardes des transitons ont été elles aussi étendues (automates temporisés avec mises à jour [34]). Finalement, les systèmes hybrides, dans lesquels les différentes horloges n'évoluent pas forcément au même rythme, ont été proposés [5]. Ces derniers sortent du cadre de ce document. Pour un historique très clair des différents modèles se référer à [122].

4.3.2 Automate temporisé d'Alur et Dill

Rappelons qu'une horloge est une variable qui permet d'enregistrer le passage du temps. Elle prend ses valeurs dans un domaine $\mathbb{T}$. Dans le modèle d'Alur et Dill [4], les affectations permises sur les horloges sont les remises à zéro de la forme $x := 0$.

Définition 12 *Un automate temporisé (TA) sur l'alphabet Σ est un 5-uplet $A = (S, s_0, \Sigma, C, \rightarrow)$ tel que :*

- *S est un ensemble fini d'états,*
- *s_0 est l'état initial,*
- *Σ est un alphabet,*
- *C est ensemble fini d'horloges,*
- *$\rightarrow \subseteq S \times \Phi(C) \times \Sigma_\tau \times 2^C \times S$ est un ensemble de transitions.* □

$t = (s, Z, a, r, s') \in \rightarrow$, notée $s \xrightarrow{Z,a,r} s'$, est la transition de source s et de destination s' sur l'occurrence de l'événement a gardée par la contrainte Z [2] et r est l'ensemble des horloges à remettre à zéro lors du franchissement de t. Remarquons que dans la définition d'un TA, l'ensemble des actions est $\Sigma_\tau = \Sigma \cup \{\tau\}$.

Exemple 4.2 ([4]) *La* FIG.*12 illustre un exemple d'un automate temporisé (TA). Cet automate utilise deux horloges x et y et l'ensemble des actions $\Sigma = \{a, b, c, d\}$. La garde*

[2] Z est dit la garde de la transition.

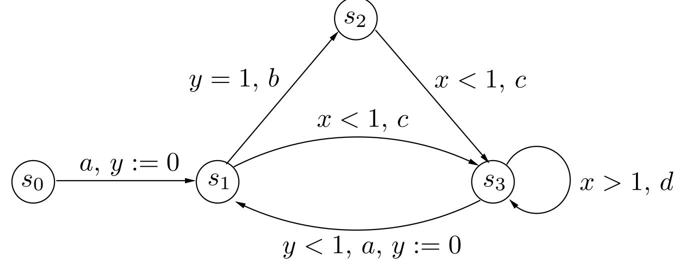


FIG. 12: Automate temporisé.

de la transition $s_0 \xrightarrow{a,\,y:=0} s_1$ *correspond à la contrainte true non représentée dans la figure. Le passage de* s_0 *à* s_1 *se fait par la réalisation de l'action* a *et la remise à zéro de* y. □

L'ensemble des automates définis sur Σ est noté $\mathcal{TA}(\Sigma)$. Nous définissons $c_{max}(A)$ comme étant le maximum des $c_{max}(Z)$ où Z est une garde dans A. Finalement, rappelons qu'un chemin d'un TA A est une suite de transitions de la forme :

$$s_0 \xrightarrow{Z_1,a_1,r_1} s_1 \xrightarrow{Z_2,a_2,r_2} s_2 \cdots \xrightarrow{Z_n,a_n,r_n} s_n.$$

4.3.3 Sémantique d'un TA

La sémantique d'un TA $A = (S, s_0, \Sigma, C, \rightarrow)$ est définie par le système de transitions temporisé $Q^A = (Q, q_0, \Gamma, \rightarrow_A)$. Les états Q sont des couples (s, ν) où s est un état de A ($s \in S$) et ν est une valuation sur C ($\nu \in \mathcal{V}(C)$). L'état initial q_0 correspond à (s_0, ν_0), avec $\nu_0 \in$ **zero** (la valuation nulle). L'alphabet Γ est inclus dans $\Sigma \cup \{\epsilon(d) \,|\, d \in \mathbb{T}\}$. On distingue deux types de transitions dans Q^A :

- *Transitions temporisées.* Le changement d'état est dû au passage du temps : pour un état (s, ν) et $d \in \mathbb{T}^{\geq 0}$, $(s, \nu) \xrightarrow{\epsilon(d)}_A (s, \nu + d)$.
- *Transitions discrètes.* Le changement d'état est dû à la réalisation des actions : pour un état (s, ν) et une transition $s \xrightarrow{Z,a,r} s'$, $(s, \nu) \xrightarrow{a}_A (s', \nu[r := 0])$ si $\nu \in Z$.

Les transitions discrètes sont supposées prendre zéro unité de temps. Soient $A = (S, s_0, \Sigma, C, \rightarrow)$ un TA et $Q^A = (Q, q_0, \Gamma, \rightarrow_A)$ sa sémantique. A est dit :

- *observable* si aucune transition n'est étiquetée par τ.
- *événementiellement déterministe* si $s \xrightarrow{Z,a,r} s'$ et $s \xrightarrow{Z',a,r'} s''$ implique que $s' = s''$ ($s, s', s'' \in S$).
- *déterministe* si $q \xrightarrow{a}_A q$ et $q \xrightarrow{a}_A q''$ implique que $q' = q''$ ($q, q', q'' \in Q$).

– *non-bloquant* si pour tout $q \in Q$, pour tout $d \in \mathbb{T}^{\geq 0}$, il existe une exécution $q \xrightarrow{\alpha_1}_A q_1 \cdots \xrightarrow{\alpha_n}_A q_n$ de Q^A tel que $d = \sum\{d_i \mid \alpha_i = \epsilon(d_i)\}$.[3]

Les automates temporisés observables sont nettement moins expressifs que les automates temporisés non-observables [17, 18]. La condition de non-bloquant considère que les systèmes sont supposés s'exécuter infiniment. En général, cette condition est modélisée par une boucle d'actions internes sur les états sans successeurs. Une autre propriété essentielle d'un système est l'absence de comportements *Zenon*, i.e. un automate ne peut pas exécuter une infinité d'actions en un temps fini. Ceci implique l'absence d'exécutions avec un ensemble infini d'actions et un cumul temporel fini. Les automates considérés dans ce document vérifient ces deux conditions.

4.3.4 Computation

Soient $A = (S, s_0, \Sigma, C, \rightarrow)$ un TA et $\sigma \in TW(\Sigma_\tau)$ une séquence temporisée telle que $|\sigma| = n$. Une *computation* r de A sur σ, notée $(\overline{s}, \overline{\nu})$, est une séquence finie de la forme :

$$r : (s_0, \nu_0) \xrightarrow{\sigma_1} (s_1, \nu_1) \;...\; (s_{n-1}, \nu_{n-1}) \xrightarrow{\sigma_n} (s_n, \nu_n)$$

avec $s_i \in S$ et $\nu_i \in \mathcal{V}(C)$ pour tout $i \in [0, n]$ satisfaisant les conditions suivantes :

1. *Initiation* : pour tout $x \in C$, $\nu_0(x) = 0$.
2. *Succession* : pour tout $i \in [1, n]$, il existe une transition $t_i = (s_{i-1}, Z_i, event(\sigma_i), r_i, s_i)$ de A telle que :
   – $\nu_{i-1} + (time(\sigma_i) - time(\sigma_{i-1})) \in Z_i$.
   – ν_i est égale à $(\nu_{i-1} + (time(\sigma_i) - time(\sigma_{i-1})))[r_i := 0]$.

Intuitivement, à l'état initial s_0 de A, les valeurs des horloges sont nulles. Lorsqu'une transition t_{i+1} de l'état s_i à l'état s_{i+1} est réalisée, on utilise les valeurs de la valuation $\nu_i + time(\sigma_{i+1}) - time(\sigma_i)$ pour vérifier la satisfaction de la contrainte d'horloges associée à t_{i+1}. Cependant, lors du franchissement de la transition t_{i+1}, les valeurs des horloges remises à zéro dans t_{i+1} sont nulles. Par convention, la date $time(\sigma_0)$ du début de la séquence temporisée est égale à zéro.

Exemple 4.3 *Considérons le TA A de la* FIG. *12 et la séquence temporisée :*

$$(a, 0.2)(c, 0.4)(a, 0.8).$$

La computation correspondante à cette séquence est donnée ci-dessous. Une valuation est donnée en listant les valeurs de x et y : $[x, y]$.

$$r \;:\; (s_0, [0, 0]) \xrightarrow{(a,0.2)} (s_1, [0.2, 0]) \xrightarrow{(c,0.4)} (s_3, [0.4, 0.2]) \xrightarrow{(a,0.8)} (s_1, [0.8, 0]).$$

[3]Voir la section 4.2 pour la définition d'une exécution d'un TLTS.

Cependant, A n'admet pas de computation sur la séquence temporisée :

$$(a, 2)(b, 3)(c, 3).$$

□

L'ensemble des séquences temporisées admettant une computation de A est noté $Run(A)$ et il est défini par :

$$Run(A) = \{\sigma \mid A \text{ admet une computation sur } \sigma \in TW(\Sigma_\tau)\}.$$

Finalement, les *traces temporisées* de A sont les projections sur Σ des séquences temporisées admettant une computation de A :

$$TTrace(A) = \{\sigma \mid \exists \sigma' \in Run(A),\, \sigma = \sigma'_{|\Sigma}\}.$$

et les traces temporisées de taille $n \in \mathbb{N}$ sont les séquences temporisées de taille n :

$$TTrace(A, n) = \{\sigma \mid |\sigma| = n,\ \sigma \in TTrace(A)\}.$$

4.3.5 Composition synchrone des TAs

Il est souvent pratique de décomposer le système à modéliser en sous-systèmes plus petits. Le problème ensuite est de faire communiquer entre eux les différents sous-systèmes pour obtenir un modèle du système original. Alors que dans le cadre non temporisé, des notions de composition de systèmes semblent d'être clairement définies, dans le cadre temporisé, la situation est complexe qui n'apparaissait pas par exemple avec des modèles à base de réseaux de Petri. En effet, dans les RdP, on considère généralement que la composition est faite par fusion de places et/ou de transitions. Le résultat de la composition de deux RdPs reste un RdP. S'il existe des conditions temporelles différentes sur les places et/ou transition que l'on fusionne, on laisse à l'utilisateur le choix de définir la nouvelle contrainte portée par le résultat de la fusion.

Dans le cas de la synchronisation d'automates, il est souvent courant de définir un opérateur de composition. En effet, par l'absence de parallélisme intrinsèque du modèle, la composition de deux automates crée un automate avec un nombre d'états très important (le nombre d'états de $A \| B$ est de l'ordre du nombre d'états de A multiplié par le nombre d'états de B), qui dissuade la construction à la main.

Nous présentons ici la composition selon des *vecteurs de synchronisation* introduite par André Arnold et Maurice Nivat [118, 9, 10, 11].

Définition 13 (Fonction de composition) *Soient* $(A^i = (S^i, s_0^i, \Sigma^i, C^i, \rightarrow_i))_{i \in [1,n]}$ *n automates temporisés et* Σ *un alphabet d'actions. Une fonction de composition est une fonction partielle* $f : \Sigma_\tau^1 \cup \{\bullet\} \times \cdots \times \Sigma_\tau^n \cup \{\bullet\} \mapsto \Sigma_\tau$, *avec* $\bullet$ *un symbole distingué de non-opération, n'appartenant pas à* Σ^i, *pour tout* $i \in [1, n]$.

Définition 14 (Composition selon f) *Soient* $(A^i = (S^i, s_0^i, \Sigma^i, C^i, \rightarrow_i))_{i \in [1,n]}$ *n automates temporisés,* Σ *un alphabet d'actions et* f *une fonction de composition. La composition de* $(A^i)_{i \in [1,n]}$ *selon* f *est l'automate temporisé* $A = (S, s_0, \Sigma, C, \rightarrow)$ *défini par :*

- $S = \{s = (s_1, ..., s_n) \mid s_i \in S^i, \forall i \in [1, n]\}$,
- $s_0 = (s_0^1, ..., s_0^n)$, $C = \bigcup_{i \in [1,n]} C^i$,
- $\rightarrow = \{(s_1, ..., s_n) \xrightarrow{Z,a,r} (s'_1, ..., s'_n) \mid \exists\, a = f(a_1, \cdots, a_n)$ *tel que* $\forall i \in [1, n]$, *soit* $((a_i = \bullet) \wedge (s_i = s'_i))$ *ou* $((a_i \neq \bullet) \wedge (s_i \xrightarrow{Z_i,a_i,r_i}_i s'_i))$ *avec* $Z = Z_1 \wedge \cdots \wedge Z_n$ *et* $r = r_1 \wedge \cdots \wedge r_n\}$. □

4.4 Automate temporisé avec invariants

Une variante du modèle présenté dans la section précédente est l'automate temporisé *avec invariants* (TAI). Dans un TAI, chaque état possède une contrainte d'horloges qui permet de forcer le changement d'état lorsque la contrainte n'est plus satisfaite. Les invariants sont choisis dans le sous-ensemble $\Phi_I(C) \subset \Phi(C)$ défini par les contraintes atomiques de la forme $x \bowtie n$ $(n \in \mathbb{N})$[4]. Formellement,

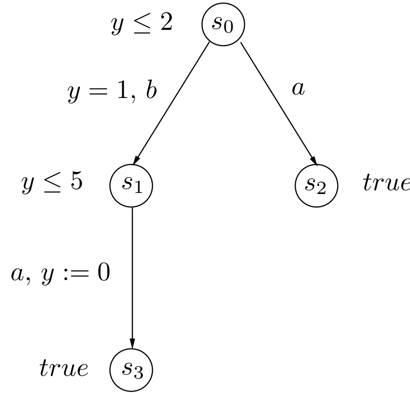


FIG. 13: Automate temporisé avec invariants.

Définition 15 (TAI) *Un automate temporisé avec invariants* A *sur* Σ *est un 6-uplet* $(S, s_0, \Sigma, C, I, \rightarrow)$, *avec :*

[4]Nous rappelons que $\bowtie \in \{\leq, \geq\}$

- $(S, s_0, \Sigma, C, \rightarrow)$ *un automate temporisé,*
- $I : S \mapsto \Phi_I(C)$ *une fonction qui affecte à chaque état un invariant (une contrainte dans* $\Phi_I(C)$*).* □

Exemple 4.4 *La* FIG.*13 illustre un exemple d'un TAI. L'invariant de l'état* s_0 *correspond à* $y \leq 2$*, ce qui implique que l'automate pourra rester dans* s_0 *au plus deux unités de temps bien que la transition* $s_0 \xrightarrow{a} s_2$ *ait une garde true. Les états* s_2 *et* s_3 *ont un invariant true et donc l'automate pourra rester indéfiniment dans ces états.* □

Sémantique

La sémantique d'un TAI $A = (S, s_0, \Sigma, C, I, \rightarrow)$ est aussi définie par un système de transitions temporisé $Q^A = (Q, q_0, \Gamma, \rightarrow_A)$. On distingue deux types de transitions dans Q^A :

- *Transitions temporisées.* Le changement d'état est dû au passage du temps : pour un état (s, ν) et $d \in \mathbb{T}^{\geq 0}$, $(s, \nu) \xrightarrow{\epsilon(d)}_A (s, \nu + d)$ si pour tout $0 \leq d' \leq d$, $\nu + d' \in I(s)$.
- *Transitions discrètes.* Le changement d'état est dû à la réalisation des actions : pour un état (s, ν) et une transition $s \xrightarrow{Z,a,r} s'$, $(s, \nu) \xrightarrow{a}_A (s', \nu[r := 0])$ si $\nu \in Z$ et $\nu[r := 0] \in I(s')$.

Ainsi, le passage du temps dans un état est conditionné par la satisfaction de l'invariant de l'état. De plus, la propriété de non-bloquant suppose que lorsque l'invariant d'un état n'est plus satisfait, au moins la garde d'une transition sortante de cet état est satisfaite.

4.5 Automate temporisé avec urgence

Le travail de vérification et de test réclame d'avoir des modèles facilitant la phase de modélisation, d'où la nécessité d'avoir des méthodes permettant de représenter aisément certains phénomènes. On s'intéresse ici à la modélisation des actions *urgentes*, i.e des actions où le système doit réagir “immédiatement”. Dans ce cadre, plusieurs solutions ont été proposées. On trouve :

- L'utilisation de la notion d'urgence [96] provenant de la synchronisation utilisée dans l'algèbre de processus CCS [111] étendue à TCCS [154].
- L'ajout d'un “deadline” sur chaque transition pour modéliser les actions urgentes [33].

4.5.1 Urgence à l'Uppaal [96]

Soit *Act* un ensemble d'actions. Celui-ci sera décomposé en deux ensembles disjoints, un ensemble classique Σ et un ensemble d'actions urgentes Σ_u. Nous supposerons que *Act*

est doté d'une fonction conjuguée $^-$: $Act \mapsto Act$ telle que pour tout $a \in Act$, $\overline{\overline{a}} = a$. Nous imposons que $\overline{a} \in \Sigma \Leftrightarrow a \in \Sigma$, pour s'assurer que l'émission correspondante à la réception d'un événement urgent est aussi urgente et inversement. La définition d'urgence selon Uppaal considère que les transitions sur les actions urgentes ont une garde *true*.

Définition 16 (TAU à l'Uppaal) *Un automate temporisé avec urgence (TAU) $A = (S, s_0, Act, C, \rightarrow)$ sur l'alphabet Act est un automate temporisé sur Act tel que :*

- *Urgence : si $s \xrightarrow{Z,a,r} s'$ est une transition de A telle que $a \in \Sigma_u$, alors Z est égal à true.* □

Exemple 4.5 *La* FIG.*14 illustre l'urgence à l'Uppaal. Dans ce cas, on a $\Sigma_u = \{a\}$ et $\Sigma = \{b\}$. Remarquons que la garde sur les transitions ayant a comme action est true.* □

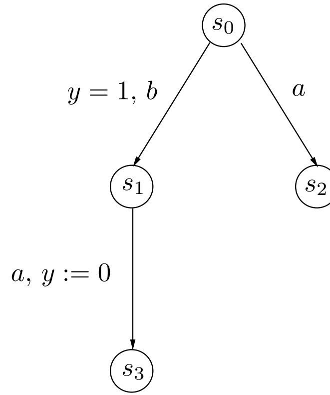


FIG. 14: Urgence à l'Uppaal.

La sémantique d'un TAU est la même qu'un TA. Dans le modèle Uppaal, la notion d'urgence n'intervient que lors de la mise en parallèle des automates ; lorsqu'une action a est urgente et qu'elle peut se synchroniser avec son complément $\overline{a}$, alors il n'est plus possible d'attendre. Notons aussi que Uppaal utilise la notion d'état "committed" pour forcer deux actions à s'effectuer séquentiellement de manière instantanée.

4.5.2 Urgence à la Kronos [33]

La solution de [33] pour modéliser les actions urgentes consiste à ajouter un "deadline" sur chaque transition pour modéliser les actions urgentes.

Définition 17 (TAU à la Kronos) *Soit $U = \{lazy, delayable, eager\}$ un ensemble de "deadline". Une automate temporisé avec urgence (TAU) sur l'alphabet Σ est un 6-uplet $(S, s_0, \Sigma, U, C, \rightarrow)$ tel que :*

- *S est un ensemble fini d'états,*
- *s_0 est l'état initial,*
- *C est un ensemble fini d'horloges,*
- *$\rightarrow \subseteq S \times \Sigma_\tau \times \Phi(C) \times 2^C \times U \times S$ est un ensemble de transitions.* □

Intuitivement, une transition "eager" doit être effectuée dès qu'il est possible de la faire et l'attente n'est pas permise ; une transition "lazy" n'impose aucune restriction sur le passage du temps alors qu'une transition "delayable" doit être réalisée avant qu'il ne soit plus possible de la faire.

Exemple 4.6 *Prenons l'exemple de la* FIG.*15. Dans l'état s_0, l'action a peut être réalisée sans contrainte temporelle et donc le système peut rester, à priori, indéfiniment dans cet état. Cependant, l'action b est "eager", ce qui oblige le système à la réaliser après une unité du temps. Par conséquent, dans l'état s_0, le système ne peut laisser le temps s'écouler plus d'une unité de temps. De même, dans l'état, s_1, le système ne peut rester plus de deux unités de temps du fait que l'action a est "delayable".* □

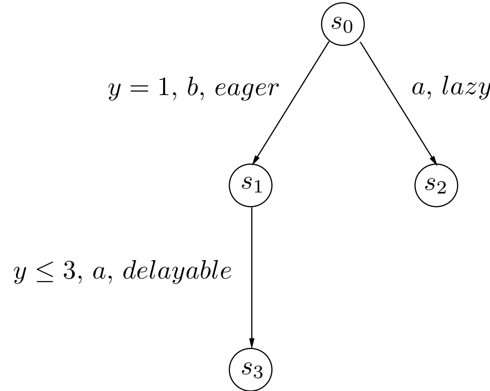


FIG. 15: Urgence à la Kronos.

Cette définition d'urgence à une conséquence sur le sémantique du TAU, en particulier sur les transitions temporisées qui sont, dans ce cas, définies par :

- $(s, \nu) \xrightarrow{\epsilon(d)}_A (s, \nu + d)$, avec $d \in \mathbb{T}^{>0}$ et il n'existe aucune transition $s \xrightarrow{Z,a,r,u} s'$ telle que : soit $u = delayable$ et il existe $0 \leq d_1 < d_2 \leq d$ tel que $\nu + d_1 \in Z$ et $\nu + d_2 \notin Z$; soit $d = eager$ et il existe $0 \leq d_1 < d$ tel que $\nu + d_1 \in Z$.

4.6 Automate temporisé à entrée/sortie

Le modèle des TAs de base ne permet pas de faire une distinction entre l'émission et la réception d'une action. Or, cette distinction s'avère parfois nécessaire, en particulier, il peut

être utile de savoir si le passage d'une transition se fera à l'initiative de l'environnement du système (une entrée) ou, si au contraire, le système lui même qui déclenchera le passage (une sortie). Pour exprimer cette information, une extension des automates temporisés a été définie : les automates temporisés *à entrée/sortie* [88] (TIOA).

Définition 18 (TIOA) *Un automate temporisé à entrée/sortie [88] (TIOA) est un automate temporisé sur l'alphabet Σ tel que Σ est divisé en deux ensembles disjoints :*
- *l'ensemble des actions d'entrée Σ_i.*
- *l'ensemble des actions de sortie Σ_o.* □

L'ensemble des TIOAs définis sur Σ est noté $\mathcal{TIOA}(\Sigma_i, \Sigma_o)$. Un élément de Σ_i (resp. Σ_o) est noté $?a$ (resp. $!a$). Soient $A \in \mathcal{TIOA}(\Sigma_i, \Sigma_o)$ et $Q^A = (Q, q_0, \Gamma, \rightarrow_A)$ sa sémantique. A est dit :
- *input-complet* si A accepte toute entrée à tout instant. Formellement, $\forall q \in Q$, $\forall a \in \Sigma_i$, $q \xrightarrow{a}_A$.

4.7 Automate temporisé étendu

Une autre variante des automates temporisés est les automates temporisés *étendus*. Ces derniers, en plus des variables continues (horloges) utilisent des variables discrètes et des paramètres.

Pour un ensemble d'horloges C, un ensemble de paramètres P et un ensemble de variables V, l'ensemble des contraintes d'horloge $\Phi(C, P, V)$ est défini par la grammaire suivante :

$$\phi := \phi \mid \phi \wedge \phi \mid x \leq f(P, V) \mid f(P, V) \leq x$$

avec x une horloge de C et $f(P, V)$ une expression linéaire de P et V.

Définition 19 (ETIOA) *Un automate temporisé étendu à entrée/sortie (ETIOA) est un 10-uplet $M = (S, s_0, \Sigma, C, P, V, V_0, Pred, Ass, \rightarrow)$ tel que :*
- *S est un ensemble fini des états.*
- *s_0 est l'état initial.*
- *Σ est un alphabet fini d'actions,*
- *C est un ensemble fini d'horloges.*
- *P est un ensemble fini de paramètres.*
- *V est un ensemble fini de variables.*
- *V_0 est un ensemble de valeurs initiales pour les variables de V.*
- *$Pred = \Phi(C, P, V) \cup \tilde{P}[P, V]$, $\tilde{P}[P, V]$ est un ensemble d'inégalités linéaires sur V et P.*
- *$Ass = \{x := 0 \mid x \in C\} \cup \{v := f(P, V) \mid v \in V\}$ est un ensemble de mises à jour sur les horloges et les variables.*

– $\rightarrow \subseteq S \times Pred \times \Sigma_\tau \times\ Ass \times S$ *est un ensemble de transitions.*

$t = (s, pred, a, ass, s')$ est la transition de l'état s à l'état s' sur l'occurrence du symbole a. $pred \subseteq Pred$ est une contrainte sur C et V et $ass \subseteq Ass$ est un ensemble des mises à jour de C et V.

Exemple 4.7 *Un exemple d'ETIOA est donné dans la* FIG.*16.*

- $S = \{s_0, s_1, s_2, s_3\}$ *et* s_0 *est l'état initial.*
- $L = \{!a, ?b, !c, ?d\}$, $C = \{x, y\}$, $P = \{\lambda\}$, $V = \{v1\}$ *et* $V_0 = \{2\}$.
- $Pred = \{y \geq \lambda, x \leq 1, v1 \leq 4\}$.
- $Ass = \{x := 0, y := 0, v1 := v1 + 1\}$.
- *La transition de source* s_2 *et de destination* s_3 *est :* $t = (s_2, \{x \leq 1\}, !c, \{v1 := v1 + 1\}, s_3)$.

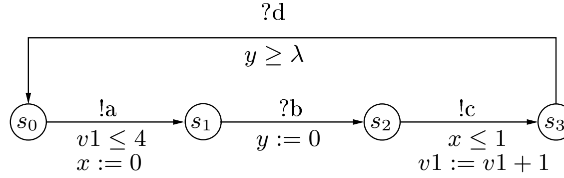


FIG. 16: Automate temporisé étendu.

4.8 Automate des régions

L'un des avantages d'un modèle formel est la possibilité de validation automatique. La démarche générale est la suivante : soit S la modélisation d'un système réel dans une théorie et soit ϕ une propriété exprimée dans une théorie appropriée. On se demande alors si le système S vérifie la propriété ϕ.

Un automate temporisé ayant par définition une sémantique infinie, l'analyse doit permettre de représenter le système de manière finie. Conjointement au modèle des automates temporisés, Alur et Dill [2, 4] proposent une technique pour prouver la décidabilité de l'accessibilité d'un état. Cette technique repose sur le partitionnement en classes d'équivalence de l'espace d'états. Ces classes d'équivalence (les régions) sont telles que chaque état d'une classe évoluera vers des états de la même classe.

4.8.1 Région d'horloges

Soit $A = (S, s_0, \Sigma, C, \rightarrow)$ un automate temporisé ayant des contraintes non diagonales.[5] Pour toute horloge $x \in C$, notons par c_x la plus grande constante telle qu'une contrainte

[5]Rappelons qu'une contrainte diagonale est de la forme $x - y \bowtie c$.

d'horloges $x \bowtie c$ apparaisse dans A. L'équivalence des valuations ν et ν', notée $\nu \equiv \nu'$, est définie par :

1. Pour toute horloge $x \in C$, soit $\lfloor \nu(x) \rfloor = \lfloor \nu'(x) \rfloor$ ou $\nu(x) > c_x$ et $\nu'(x) > c_x$.
2. Pour toutes les horloges $x, y \in C$ telles que $\nu(x) \leq c_x$ et $\nu'(x) \leq c_x$, $fr(\nu(x)) = fr(\nu(y))$ si et seulement si $fr(\nu'(x)) = fr(\nu'(y))$.
3. Pour toute horloge $x \in C$ telle que $\nu(x) \leq c_x$, $fr(\nu(x)) = 0$ si et seulement si $fr(\nu'(x)) = 0$.

où $\lfloor c \rfloor$ est la partie entière de c et $fr(c)$ est la partie fractionnaire de c. Une région d'horloges est une classe d'équivalence des valuations d'horloges induites par $\equiv$. Notons par $R(\nu)$ la région d'horloges dont fait partie ν.

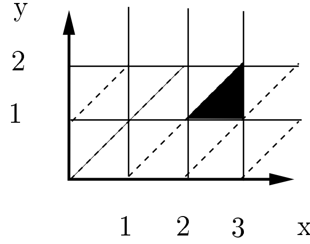


FIG. 17: Région d'horloges.

Exemple 4.8 *Considérons un automate temporisé à deux horloges x et y pour lequel $c_x = 3$ et $c_y = 2$. L'ensemble des régions de cet automate peut être décrit par le schéma de la* FIG.*17. La région en noir correspond à la contrainte :*

$$1 < y < 2 \wedge 2 < x < 3 \wedge y < x.$$

□

4.8.2 Automate des régions

Soit $A = (S, s_0, \Sigma, C, \rightarrow)$ un automate temporisé. L'automate des régions associé à A est l'automate $RG = (Q, q_0, \Sigma^{RG}, \rightarrow_{RG})$ défini par :

1. $\Sigma^{RG} = \Sigma \cup \mathbb{R}^{>0}$,
2. $Q = \{(s, R(\nu)) \mid s \in S, \nu \in \mathcal{V}(C)\}$,
3. $q_0 = (s_0, R(\nu_0))$, avec $\nu_0(x) = 0$ pour toute horloge x de C,
4. RG a une transition discrète $(s, R(\nu)) \xrightarrow{a} (s', R(\nu'))$ si et seulement s'il existe une transition $s \xrightarrow{Z,a,r} s'$de A telle que $\nu \in Z$ et $\nu' = \nu[r := 0]$,
5. RG a une transition temporisée $(s, R(\nu)) \xrightarrow{d} (s', R(\nu'))$ sur $d > 0$ si et seulement si $\nu' = \nu + d$.

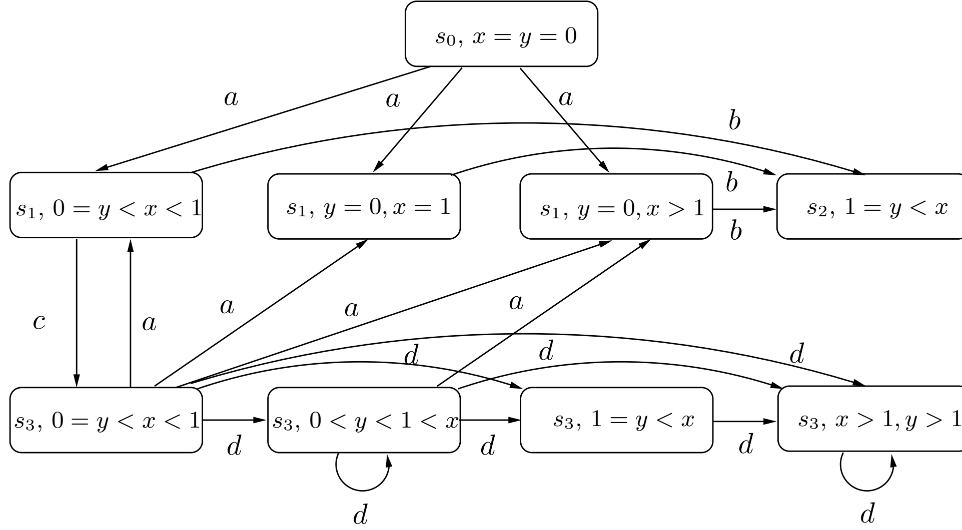


FIG. 18: Automate des régions.

Exemple 4.9 *La* FIG.*18 montre l'automate des régions de l'automate de la* FIG.*12. Le symbole d (*$0 < d < 1$*) est un réel représentant le passage du temps. Contrairement à la* FIG.*17, la* FIG.*18 ne présente que les régions d'horloges atteignables à partir de l'état initial* $(s_0, x = y = 0)$. □

L'automate des régions correspond au graphe d'accessibilité d'un automate temporisé, i.e. un état s est atteignable dans un automate A si et seulement s'il existe un état (s, R) dans l'automate des régions associé à A.

4.9 Autres modèles

De nombreux travaux ont été réalisés sur des extensions des automates temporisés, parmi lesquels :

- L'ajout des contraintes d'horloges du type $x + y \bowtie c$ [19].
- L'ajout de contraintes d'horloges périodiques (du type $x \bowtie c \, mod \, k$) [48].
- L'ajout de mises à jour de la forme $x :\bowtie c$ ou encore $x :\bowtie y + c$, avec $\bowtie \in \{<, \leq, =, \geq, >\}$ [34].
- L'ajout de paramètres dans les contraintes d'horloges [3, 20].
- Les systèmes hybrides : les variables utilisées ne sont plus uniquement des horloges mais peuvent avoir des pentes différentes de 1 [5].

D'autres modèles basés sur les réseaux de Pétri et les algèbres de processus ont été développés en parallèle pour tenir compte de l'aspect temporel des systèmes. On trouve :

- Les RdPs temporisés [107, 125, 137, 149],
- Les RdPs temporels [107],
- Les RdPs stochastiques temporisés [60].
- L'algèbre de processus temporisée TCCS comme extension temporisée de CCS [154].

Pour un état de l'art sur les extensions temporelles des RdPs se référer à la thèse de Marc Boyer [36].

Deuxième partie

Modélisation des Systèmes Communicants

Introduction

Un *système communicant* (centralisé ou distribué) est un système exploitant une architecture physique consistant en multiples et autonomes éléments, sans ou avec mémoire partagée, communicant à travers un réseau ou des canaux de communications. Les éléments du système communicant peuvent être situés sur le même site ou des sites différents. Un système communicant réagit alors avec son environnement pour réaliser une fonctionnalité donnée.

L'utilisation des automates comme modèle et cadre théorique pour décrire les systèmes communicants a l'avantage d'offrir des facilités de définition d'algorithmes applicables dans un outil de validation. Cependant, les modèles existants basés sur les automates présentent deux limitations au niveau de la description des aspects de contrôle et de données. Pour l'aspect contrôle, la majorité des modèles existants ne permet que la modélisation des communications binaires ou sur des ensembles d'événements communs. Or, les processus communicants ne possèdent pas en général les mêmes ensembles d'événements, i.e. les événements produits par les processus considérés n'appartiennent pas nécessairement à un ensemble commun. De plus, on assiste aujourd'hui à l'émergence des applications multimédias à caractère de diffusion, d'où le besoin de modéliser et de valider ce genre de communication. Il est clair que les algèbres de processus se présentent comme un bon modèle, très expressif. Cependant, l'utilisation des algèbres de processus présente une difficulté de définition d'algorithmes applicables dans un outil de validation. Pour l'aspect données, une modélisation du partage des ressources entre processus est requise. Une variable peut être commune à un ensemble de processus. Il est donc important de modéliser un tel partage par l'approche la plus simple possible.

Motivations

Les recherches entamées dans le cadre de ce document se veulent à la fois théoriques, par l'étude formelle de la modélisation et la validation et pratiques, par l'implantation des résultats théoriques. Nous nous intéressons à la modélisation des systèmes communicants *synchrones*, i.e. la communication se fait par *rendez-vous* : un message ne peut être envoyé par un processus que si le processus récepteur est capable de recevoir et de traiter ce message immédiatement. Une telle communication fait abstraction des couches sous-adjacentes, ainsi que du temps de traitement de ces couches et de la latence du réseau. La communication par rendez-vous peut concerner deux entités (*binaire/unicast*), ou plusieurs entités (un processus émetteur et des processus récepteurs comme dans le cas du *multicast* et du *broadcast*). Pour les protocoles assurant un service fiable, une telle abstraction est réaliste. Pour les systèmes asynchrones communicant par des files bornées de type *FIFO* (First In First Out), la modélisation de ces files de communication permet de se placer dans un cadre synchrone et ainsi, un modèle pour les systèmes synchrones peut être étendu aux systèmes *asynchrones*.

Comme nous venons de le souligner dans l'introduction, la réalisation d'un outil de

validation basé sur le modèle des automates ou des algèbres de processus ne répond pas aux différentes motivations, citées auparavant, relatives à la description des systèmes communicants. En effet, un modèle hybride combinant l'aspect graphique des automates et l'aspect hiérarchique des algèbres de processus s'adapterait mieux à un outil de validation.

Au delà des motivations relatives à la modélisation des systèmes et des communications inter-composantes, un autre point qui motive cette recherche est l'étude théorique de la possibilité d'une modélisation uniforme de l'ensemble des éléments intervenant dans le test (architectures de test, approches et types de test, séquences de test,...).

Contribution

La contribution majeure de cette partie est l'introduction du modèle CS, un modèle de description des systèmes communicants. Le modèle CS considère deux aspects des systèmes communicants : les flux (contrôle, données et temps) propres aux entités du système et les flux partagés (contrôle et données) entre entités. La modélisation des flux propres aux entités est basée sur les automates. Le flux de données partagé est modélisé par des variables et des paramètres. Le flux de contrôle partagé est modélisé par une topologie de communicant qui explicite les différentes communications possibles dans un état du système.

La séparation entre les flux propres et les flux partagés d'une part et une sémantique en automate d'autre part, procurent au modèle CS l'aspect graphique des automates et l'aspect hiérarchique des algèbres de processus. Au delà de l'aspect description des systèmes communicants, le modèle CS est au coeur de la modélisation uniforme, présentée dans la section 11.2 de la partie IV.

Organisation

Cette partie contient un seul chapitre. La section 5.1 du chapitre 5 introduit le modèle CS. La section 5.2 illustre des exemples de description en CS. Finalement, la section 5.3 est consacrée à la modélisation des systèmes synchrones et asynchrones.

Chapitre 5

Modèle des systèmes communicants

Sommaire

5.1 Modèle CS **58**
5.1.1 Topologie de communication 58
5.1.2 Systèmes communicants 59
5.1.3 Sémantique d'un CS 61
5.1.4 CS et algèbre de processus 63
5.1.5 Comparaison 63
5.2 Exemples **64**
5.2.1 Consommateur-Producteur 64
5.2.2 L'algorithme de Peterson 66
5.2.3 Protocole CSMA/CD 71
5.3 Communication synchrone/asynchrone **72**
5.3.1 Système distribué synchrone 73
5.3.2 Système distribué asynchrone 75
5.4 Conclusion **77**

Nous proposons dans ce chapitre d'utiliser les automates pour décrire la communication et le partage de ressources entre processus synchrones. Le modèle obtenu, nommé *modèle des systèmes communicants* ou le modèle *CS*, combine entre la représentation graphique des automates et la définition hiérarchique des algèbres de processus. Il définit un ensemble de ressources communes, un ensemble d'entités et une topologie de communication. Les entités représentent des processus. Elles sont modélisées par des automates temporisés étendus à entrée/sortie (ETIOAs). La topologie de communication décrit les différentes synchronisations possibles entre les entités dans un état global du CS. Les ressources communes représentent les différentes données partagées par les entités du CS, en l'occurrence les variables et les paramètres. Par la suite, nous utiliserons sans distinction les termes "composante" et "entité" pour référencer un élément d'un système communicant.

5.1 Modèle CS

5.1.1 Topologie de communication

Une *topologie de communication* **Top** d'un ensemble de processus est un modèle de synchronisation des différents processus. Elle décrit les configurations dynamiques des processus et les synchronisations possibles dans une configuration donnée. La définition de **Top** utilise les *vecteurs de synchronisation* d'Arnold et Nivat [9, 10, 11, 118] et elle est inspirée de celle de Barros et al. [13]. Elle définit un ensemble d'actions *globales* du système, un ensemble d'ensembles d'actions locales à chaque processus et un *traducteur* modélisé par un automate à entrée/sortie.

Définition 20 (Topologie) *Une topologie de communication Top d'un ensemble de n processus est un couple (Σ, Tr), avec $\Sigma = \{\Sigma^i\}_{1 \leq i \leq n}$ un ensemble fini d'ensembles d'actions et Tr (Traducteur) un automate à entrée/sortie $Tr = (S^{tr}, s_0^{tr}, \Sigma^{tr}, \rightarrow_{tr})$ tels que :*

1. *Il existe une fonction de composition* [1] *f de $\Sigma_\tau^1 \cup \{\bullet\} \times \cdots \times \Sigma_\tau^n \cup \{\bullet\} \mapsto \Sigma_\tau^{tr}$, avec $\bullet$ un symbole distingué de non-opération, n'appartenant pas à Σ^i, pour tout $i \in [1, n]$.*
2. *Pour tout élément a_g de Σ^{tr}, il existe $(a_i)_{i \in [1,n]}$, $a_i \in \Sigma^i \cup \{\bullet\}$ tel que $a_g = f(a_1, ..., a_n)$.*
$\square$

Un élément a_g de Σ^{tr} est appelé *une action globale de synchronisation*. Le *vecteur de synchronisation* induit par a_g, noté $\overrightarrow{v}$, est le $(n+1)$-uplet $\overrightarrow{v} =< a_g, a_1, ..., a_n >$ avec $a_g = f(a_1, ..., a_n)$. L'ensemble des vecteurs induits par Σ^{tr} est noté $\overrightarrow{\Sigma^{tr}}$. Un vecteur de synchronisation $\overrightarrow{v} =< a_g, a_1, ..., a_n >$ décrit l'action a_i que le processus i, $i \in [1, n]$, doit effectuer. La synchronisation des différentes actions donne lieu à l'action globale a_g.

Convention. Lors de la représentation graphique d'une topologie de communication (ou simplement topologie) $Top = (\Sigma, (S^{tr}, s_0^{tr}, \Sigma^{tr}, \rightarrow_{tr}))$, les actions de Σ^{tr} sont confondues avec les vecteurs de synchronisation de $\overrightarrow{\Sigma^{tr}}$ qu'elles induisent. De cette manière, nous confondons une topologie de communication avec son automate traducteur.

Exemple 5.1 *La* FIG.*19 représente trois processus A, B et C dont les états initiaux sont a_0, b_0 et c_0, respectivement. Une topologie de communication de ces trois processus est représentée par l'automate de la partie droite de la* FIG.*19. Les trois états de cet automate correspondent aux différents états des processus. Cette topologie illustre les différentes actions permises par les différents processus dans un état global du système S composé de A, B et C. Ainsi, dans l'état (a_0, b_0, c_0), seul le processus A peut émettre le message x à destination de B, ce qui correspond au vecteur $<!x, !x, ?x, \bullet>$ dans Top. Dans ce*

[1] Pour la fonction de composition, voir la définition 13 de la section 4.3.5.

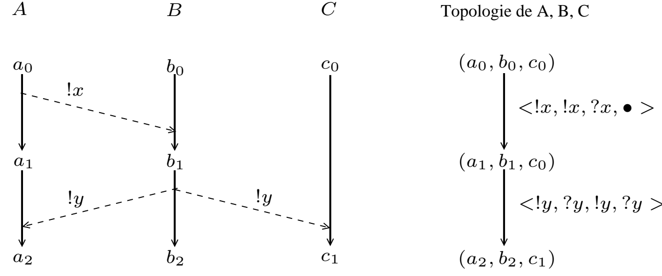


Fig. 19: Topologie de communication.

vecteur, le processus C ne change pas d'état et se met en attente (action interne d'attente). L'action globale visible de cette synchronisation est l'émission de x. Après cette action de synchronisation, le système S se trouve dans l'état (a_1, b_1, c_0). *Le processus B émet alors le message y en diffusion, ce qui correspond au vecteur* $<!y, ?y, !y, ?y>$ *dans Top. Notons dès maintenant que le choix de l'étiquette de l'action globale dépend de la signification qu'on lui attribue : d'une façon générale, il est possible d'utiliser l'étiquette de l'action d'émission, vu que l'action visible de l'émission et la réception du même événement est l'action d'émission.* □

Terminologie. Lorsqu'un vecteur de synchronisation $\overrightarrow{v} =< a_g, \bullet, ..., a_i, ..., \bullet >$ définit une seule action, seul le processus i exécute l'action a_i et change d'état. Dans ce cas, le vecteur $\overrightarrow{v}$ est dit *unitaire*. Si $\overrightarrow{v} =< a_g, \bullet, \cdots, a_i, \bullet, \cdots, a_j, \bullet, \cdots, \bullet >$, i.e. une synchronisation entre deux processus, alors $\overrightarrow{v}$ est dit *binaire*. Lorsque le nombre d'états du traducteur Tr est égal à 1, Top est dite *statique*. Elle est dite *libre* si tous ses vecteurs sont unitaires. Elle est dite *binaire* si tous ses vecteurs sont binaires.

L'écriture de la topologie de communication d'un système de processus est aussi un travail de modélisation. Elle contient donc une certaine part d'arbitraire. En particulier, lorsqu'on veut représenter un système, il est souvent possible d'en faire apparaître certains aspects, soit dans la modélisation des entités du système, soit dans la définition de la topologie. Signalons aussi que la topologie, grâce à la fonction de composition, offre la possibilité de modéliser des communications entre un, deux ou plusieurs processus : unicast, multicast et broadcast. Elle peut être utilisée, dans certains cas, comme une sorte de contrôleur sur les actions permises par les différents processus dans une configuration donnée du système global.

5.1.2 Systèmes communicants

Basé sur la définition de la topologie de communication, nous définissons le modèle des systèmes communicants comme étant un ensemble de variables et de paramètres partagés,

une topologie de communication et des entités.

Définition 21 (Système Communicant) *Un système communicant CS est un 5-uplet $(SP, SV, SV_0, (M^i)_{1\leq i\leq n}, Top)$ tel que :*
- *SP est un ensemble de paramètres partagés.*
- *SV est un ensemble de variables partagées.*
- *SV_0 est un ensemble de valeurs initiales pour les variables de SV.*
- *$Top = ((\Sigma^i)_{1\leq i\leq n}, Tr)$ est une topologie.*
- *$M^i = (S^i, s_0^i, L^i, C^i, P^i, V^i, V_0^i, Pred^i, Ass^i, \rightarrow_i)$ est un ETIOA [2] tel que : $\Sigma^i \subseteq L^i$, $\forall i \in [1, n]$.* □

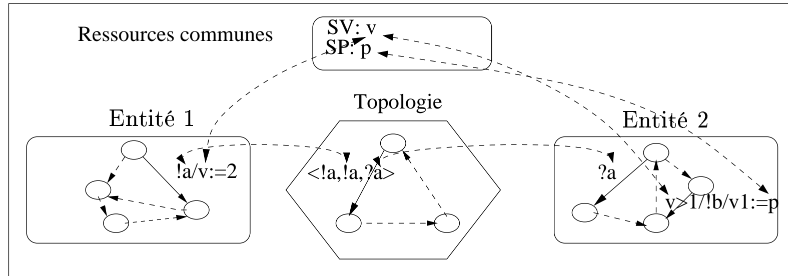


FIG. 20: Modèle CS.

La FIG.20 illustre graphiquement un CS. Les entités représentent des processus. Elles sont modélisées par des ETIOAs. La topologie de communication décrit les différentes synchronisations possibles entre les entités. Nous avons supposé dans la définition des entités que $\forall i \in [1, n]$, $\Sigma^i \subseteq L^i$. Ceci permet d'avoir des topologies partielles qui ne définissent que les synchronisations permises et ainsi ne considèrent que les comportements d'une partie du système global (dans la section suivante, nous donnerons des exemples de telles topologies). Les ressources communes représentent les différentes données partagées du système. Nous nous restreignons à des données de types variables et paramètres. Les paramètres (resp. les variables) peuvent être lus (resp. lues et modifiées) par les entités du CS. Ces paramètres et variables partagées peuvent apparaître dans la définition d'une transition d'une entité.

Convention. Nous utiliserons le terme "système communicant" pour référencer un ensemble d'entités communicantes et le terme "CS" pour référencer le modèle de description de la définition 21.

Remarque 5 *Pour un CS $(SP, SV, SV_0, (M^i)_{1\leq i\leq n}, Top)$, les variables et les paramètres associés à une entité M^i ont une porté locale restreinte à M^i [3], $i \in [1, n]$. Ainsi, deux entités M_i et M_j peuvent définir des variables ayant le même nom sans pour autant référencer*

[2] Voir section 4.7 pour la définition d'un ETIOA

[3] Si on suit la logique des langages de programmation, nous dirons que les variables et les paramètres de M_i sont des variables et des paramètres locaux.

la même donnée. Pour les variables SV et les paramètres SP, leur portée est globale dans toutes les entités [4]*. Dans le cas où une variable (resp. un paramètre) d'une entité M_i utilise le même nom qu'une variable (resp. un paramètre) partagée, alors ce nom fait référence dans M_i à la variable (resp. le paramètre) locale de M_i. Finalement, les ressources communes sont restreintes aux variables et paramètres. Cependant, le modèle peut être étendu par l'ajout du partage des horloges.*

5.1.3 Sémantique d'un CS

La *sémantique* d'un CS est définie en terme d'ETIOA. Pour simplifier, nous supposons que les variables et paramètres des entités du système sont toutes différentes et différentes de celles partagées dans le CS.

Définition 22 (Sémantique) *La sémantique d'un CS, $SC = (SP, SV, SV_0, (M^i)_{1\leq i\leq n}, Top)$, avec $M^i = (S^i, s_0^i, L^i, C^i, P^i, V^i, V_0^i, Pred^i, Ass^i, \rightarrow_i)$ et $Top = (\Sigma, (S^{tr}, s_0^{tr}, \Sigma^{tr}, \rightarrow_{tr}))$, est définie par l'ETIOA $\zeta(SC) = (S, s_0, L, C, P, V, V_0, Pred, Ass, \rightarrow)$, tel que :*

- $S = \{s = (s_{tr}, s_1, ..., s_n) \mid s_{tr} \in S^{tr}, s_i \in S^i, \forall i \in [1, n]\}$,
- $s_0 = (s_0^{tr}, s_0^1, ..., s_0^n)$,
- $L = \Sigma^{tr}$,
- $C = \bigcup_{i\in[1,n]} C^i$,
- $P = \bigcup_{i\in[1,n]} P^i \bigcup SP$,
- $V = \bigcup_{i\in[1,n]} V^i \bigcup SV$,
- $V_0 = \bigcup_{i\in[1,n]} V_0^i \bigcup SV_0$,
- $Pred = \bigcup_{i\in[1,n]} Pred^i$,
- $Ass = \bigcup_{i\in[1,n]} Ass^i$,
- $\rightarrow = \{(s_{tr}, s_1, ..., s_n) \xrightarrow{pred,a,ass} (s'_{tr}, s'_1, ..., s'_n) \mid \exists f(a_1, ..., a_n) = a \in \Sigma^{tr}$ *tel que* $s_{tr} \xrightarrow{a}_{tr} s'_{tr}$ *et* $\forall i \in [1, n]$ *soit* $((a_i = \bullet) \wedge (s_i = s'_i))$, *ou* $((a_i \neq \bullet) \wedge (s_i \xrightarrow{pred_i,a_i,ass_i}_i s'_i))$ *avec* $pred = pred_1 \wedge ... \wedge pred_n$ *et* $ass = ass_1 \wedge ... \wedge ass_n\}$. □

L'alphabet de $\zeta(SC)$ est l'ensemble Σ^{tr} des actions globales de Top. Un état de $\zeta(SC)$ est constitué d'un état de Top et des états de $(M^i)_{i\in[1,n]}$. Une transition $(s_{tr}, s_1, ..., s_n) \xrightarrow{pred,a,ass} (s'_{tr}, s'_1, ..., s'_n)$ de $\zeta(SC)$ est conditionnée par l'existence d'une transition de Top de s_{tr} à s'_{tr} sur un vecteur $\overrightarrow{v} =< a, a_1, ..., a_n >$ tel que pour tout $i \in [1, n]$, $s_i \xrightarrow{pred_i,a_i,ass_i}_i s'_i$. Finalement, d'après la sémantique définie ci-dessus, seules les actions de la topologie de communication, dans un état global du système, peuvent être appliquées.

Remarque 6 *Étant donné que la sémantique d'un CS est un ETIOA, ceci autorise la possibilité de synchronisation avec d'autres systèmes, ce qui permet une définition*

[4] Variables et paramètres globaux.

hiérarchique des CSs à l'exemple des algèbres de processus. Nous donnerons par la suite des exemples de telles compositions des CSs. Finalement, l'utilisation de la topologie de communication dans la définition du modèle CS permet d'étendre la composition des automates (selon une fonction) présentée dans la définition 14 de la section 4.3.5. □

Terminologie. Soit $SC = (SP, SV, SV_0, (M^i)_{1\leq i\leq n}, Top)$ un CS avec $M^i = (S^i, s_0^i, L^i, C^i, P^i, V^i, V_0^i, Pred^i, Ass^i, \rightarrow_i)$ et $Top = (\{\Sigma^i\}_{1\leq i\leq n}, Tr)$. SC est dit :

- *libre* si Top est statique, libre et $\forall i \in [1, n]$, $\Sigma^i = L^i$.
- *binaire* si Top est statique, binaire et $\forall i \in [1, n]$, $\Sigma^i = L^i$.

Intuitivement, dans un état global $s = (s_{tr}, s_1, ..., s_i, ..., s_n)$ d'un SC libre, chaque processus M^i se trouvant dans l'état s_i peut effectuer indépendamment des autres processus une transition t_i qui l'amènera dans un état s'_i. Le système se trouvera alors dans l'état global $s = (s'_{tr}, s_1, ..., s'_i, ..., s_n)$. Ainsi, un CS libre n'est que le *produit cartésien* des composantes. Un CS binaire correspond à une synchronisation binaire entre processus. Ainsi, la composition synchrone définie dans les sections 3.3.2 et 4.3.5 n'est autre qu'un CS binaire.

Soit $S = (SP, SV, SV_0, (M^i)_{1\leq i\leq n}, Top)$ un CS. Nous proposons de calculer le nombre d'états et de transitions de la sémantique de S. Pour un automate A et un état s de A, notons par :

- $t(A)$ le nombre de transitions (la taille) de l'automate A,
- $e(A)$ le nombre d'états de A,
- $out(s)$ le nombre de transitions de A ayant s comme source.

Le nombre d'états $e(\zeta(S))$ est majoré par le nombre des combinaisons d'états possibles des entités et de la topologie :

$$e(\zeta(S)) = O(\prod_{i\in[1,n]} e(M^i) \times e(Top)) \tag{1}$$

Le nombre de transitions de $\zeta(S)$ est :

$$t(\zeta(S)) = \sum_{s\in Top, s_i\in M^i, i\in[1,n]} out((s, s_1, \cdots, s_n)) \tag{2}$$

Comme dans un état global $(s, s_1, \cdots, s_n)$ de $\zeta(S)$, au plus les transitions de la topologie peuvent être exécutées, alors :

$$t(\zeta(S)) = O(\sum_{s\in Top, s_i\in M^i, i\in[1,n]} out(s)) = O(\sum_{s_i\in M^i, i\in[1,n]} \sum_{s\in Top} out(s)) \tag{3}$$

En conséquence :

$$t(\zeta(S)) = O(\prod_{i\in[1,n]} e(M^i) \times t(Top)) \tag{4}$$

Ou

$$t(\zeta(S)) = O(\prod_{i\in[1,n]} t(M^i) \times t(Top)) \tag{5}$$

On remarque que le nombre de transitions $t(\zeta(S))$ dans l'équation (4) ne dépend pas du nombre d'états $e(Top)$ de Top et que l'équation (5) est une majoration grossière de $t(\zeta(S))$. Finalement, la modélisation des synchronisations par la topologie n'ajoute pas un superflu au niveau de la taille de la sémantique mais il y a un coût lié à la modélisation du flux de contrôle associé à la communication entre entités.

5.1.4 CS et algèbre de processus

L'introduction de la topologie de communication dans les CSs, sous forme d'automate, n'est qu'une façon de dériver une composition parallèle. Une approche générique pour définir une composition parallèle de processus est celle indiquée dans [151], qui repose sur les travaux réalisés initialement dans [109, 110, 111]. Il a été démontré qu'il est possible de dériver n'importe quelle composition parallèle en utilisant les trois opérations, produit, restriction et renommage, introduites initialement par :

- Le produit peut être considéré comme étant une composition parallèle dans laquelle toutes les synchronisations imaginables sont possibles (voir la section 3.6).
- La restriction consiste, comme nous avons déjà vu, à masquer certains événements.
- Le renommage consiste, pour un processus, à changer les étiquettes de ses actions.

Le principe est de réaliser le produit des processus, d'appliquer la restriction pour éliminer les synchronisations non désirables et le renommage pour changer les étiquettes de la synchronisation. Cette approche est reprise dans [132]. Finalement, l'outil CMC [93] considère une composition synchrone des processus en conjonction avec le renommage.

5.1.5 Comparaison

La composition synchrone de systèmes de transitions a été introduite par Arnold et Nivat [9, 10, 11, 118] comme l'opération fondamentale dans la définition de la sémantique d'un système de processus interagissant. Cette composition est basée sur la définition des vecteurs de synchronisation qui restreignent l'ensemble de toutes les transitions globales que l'on veut voir apparaître dans le résultat.

Madelaine et al. [13] reprennent ce concept pour modéliser des objets *Java* distribués. Le modèle CS est étroitement lié aux modèles paramétrés présentés dans [13]. Ces modèles paramétrés se veulent généralistes et sont constitués de réseaux symboliques (pNets) de

systèmes de transitions paramétrés (pLTSs) capturant le comportement sémantique des objets ProActive (appels à distance de méthodes). Les modèles CS et pNets sont basés sur les vecteurs de synchronisation. Les principales différences entre les deux modèles sont : (i) dans les pNets, la communication inter-processus est modélisée par des actions paramétrées pour modéliser le passage d'arguments dans les appels à distance de méthodes ; CS utilise des variables et des paramètres partagés, (ii) Les pNets permettent la définition de synchronisations sur des événements autres que ceux définis par les processus ; CS oblige une définition partielle des synchronisations permises par les processus.

5.2 Exemples

Le but de cette section est de montrer, à travers des exemples, comment décrire les composantes habituelles d'un système de processus ou d'un programme dans le modèle CS. Comme nous allons le voir, il n'y a rien d'automatique dans la construction de ces descriptions. Il s'agit d'un travail de modélisation et il appartient au modélisateur de déterminer les aspects pertinents de l'objet qu'il veut faire apparaître dans le modèle et sous quelles formes il les fait apparaître. Dans la pratique, étant donnés un système de processus et des spécifications des interactions entre ces processus, on peut définir à priori et très naturellement une topologie de communication qui décrit exactement ces interactions, de sorte que la sémantique du système communicant obtenu représente, ipso facto, le comportement du système souhaité. Trois exemples de modélisation seront présentés : consommateur-producteur, algorithme d'exclusion mutuelle et enfin le protocole CSMA/CD.

5.2.1 Consommateur-Producteur

Principe. Soit un buffer B à n places ($n \in \mathbb{N}$) utilisé comme une file. L'exemple du consommateur-producteur considère p producteurs $P_1, ..., P_p$ et q consommateurs $C_1, ..., C_q$ communicants à travers le buffer B. B est initialement vide. Il est rempli par les p producteurs et vidé par les q consommateurs. Un processus P_i dépose un produit dans la file B si cette dernière n'est pas pleine. Un processus C_j retire un produit de B si cette dernière n'est pas vide. Nous définissons deux actions sur la file B : l'action *pop* qui consiste à retirer un produit de B et l'action *push* qui consiste à stocker un produit dans B.

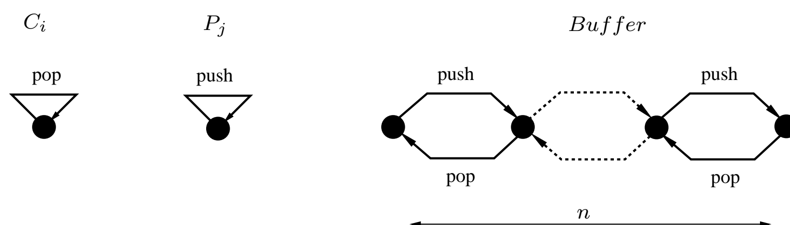


FIG. 21: Consommateur-producteur.

Modélisation. En premier lieu, nous proposons de modéliser l'exemple du consommateur-producteur sans utiliser le modèle CS (FIG.21). L'automate modélisant un processus C_i, $i \in [1, q]$, contient un seul état et une seule transition. Le processus C_i exécute l'action *pop* sur B. L'automate $Buffer$ modélisant la file B contient $n + 1$ états et $2 \times n$ transitions, où n est la taille du buffer B. Dans chaque état du $Buffer$ (sauf le dernier), la transition sortante est étiquetée par *push* et celle rentrante est étiquetée par *pop*. Cette modélisation simpliste du consommateur-producteur cache un problème de synchronisation lors de la construction du système global. En effet, les processus P_i (resp. C_j) ont des comportements symétriques et seul un processus P_i (resp. C_j) peut à la fois se synchroniser à un instant donné avec le $Buffer$ sur l'action *push* (resp. *pop*). La modélisation du $Buffer$ ne met pas en évidence l'ensemble des processus qui doivent se synchroniser sur une action donnée du $Buffer$ (accès concurrent). Une solution peut être alors le renommage des actions de P_i (resp. C_j) puis la définition du système global comme une composition basée sur une synchronisation binaire entre les p producteurs et le buffer d'une part et les q consommateurs et le buffer d'autre part. Dans un exemple plus complexe, une synchronisation sur une action peut dépendre du contexte du système, ce qui complique d'avantage la définition de la fonction de composition et de renommage.

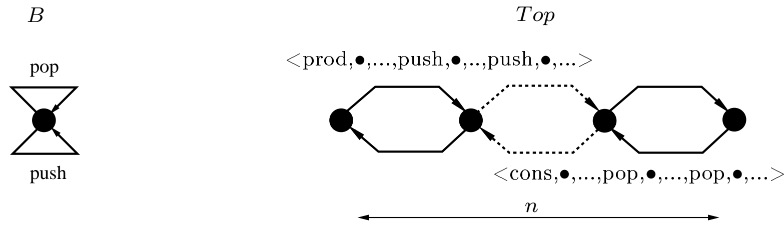


FIG. 22: Consommateur-producteur en CS.

Modélisation en CS. Nous proposons de modéliser l'exemple du consommateur-producteur par le CS $S = (\emptyset, \emptyset, \emptyset, (C_i)_{i\in[1,q]} \cup B \cup (P_j)_{j\in[1,p]}, Top)$ de la FIG.22. L'idée est de séparer entre le comportement de la file B (automate B) et la gestion de cette dernière (automate Top). L'automate de la file B contient deux transitions et un état. B peut stocker (action *push*) ou déstocker (action *pop*) un produit. Top contient $n+1$ états. Nous supposons que les états de Top sont ordonnés et nous utiliserons s_k pour référencer l'état d'indice k, $k \in [1, n + 1]$. Chaque état s_k (sauf s_{n+1}) définit p transitions sortantes sur un vecteur $\overrightarrow{v_i}$, $i \in [1, p]$, de la forme $< prod, \bullet, ..., push, \bullet, ..., push, \bullet, ... >$ [5] tel que :

- L'indice $2 + q + i$ de $\overrightarrow{v_i}$ vaut *push*.
- L'indice $2 + q$ de $\overrightarrow{v_i}$ vaut *push*.

Ces vecteurs sortants correspondent à une synchronisation binaire entre un processus P_i et la file B sur l'action *push*. L'action visible est *prod*.

[5] Ce qui correspond à l'action globale suivie de l'action de chaque C_j, puis l'action de la file et enfin l'action de chaque P_i.

De même, chaque état s_k (sauf s_{n+1}) définit q transitions entrantes sur un vecteur $\overrightarrow{v'_i}$, $i \in [1, q]$, de la forme $< cons, \bullet, ..., pop, \bullet, ..., pop, \bullet, ..., \bullet >$ tel que :

- L'indice $1 + i$ de $\overrightarrow{v'_i}$ vaut *pop*.
- L'indice $2 + q$ de $\overrightarrow{v'_i}$ vaut *pop*.

Ces vecteurs entrants correspondent à une synchronisation binaire entre un processus C_j et la file B sur l'action *pop*. L'action visible est *cons*. Cette modélisation explicite des synchronisations ne laisse pas de confusion lors de la construction du système global.

En conclusion, cette modélisation n'introduit pas un superflu vu que l'automate sémantique de $S = (\emptyset, \emptyset, \emptyset, (C_i)_{i \in [1,q]} \cup B \cup (P_j)_{j \in [1,p]}, Top)$ (FIG.22) est le même que l'automate $C_1 \| ... \| C_q \| Buffer \| P_1 ... \| P_p$ avec renommage d'actions (FIG.21) [6]. Finalement, comme cela a été montré à travers cet exemple, il est plus judicieux, dans la modélisation d'un système, de séparer le comportement nominal d'une entité de ses interactions (synchronisations) avec le reste du système (l'utilisation de la topologie).

5.2.2 L'algorithme de Peterson

Principe. L'algorithme de Peterson gère l'accès de deux processus P_0 et P_1 à une section critique. Il utilise quatre variables globales : deux variables booléennes a et b initialisées à *false*, une variable entière *round* qui ne peut prendre que les valeurs *0* ou *1* et une variable entière *ppid* contenant l'identifiant d'un processus. Nous considérons une version de Peterson basée sur l'algorithme présenté dans [10] dont nous proposons une traduction en langage C. Les processus P_0 (père) et P_1 (fils) exécutent la même fonction $peterson()$:

```
#include <sys/types.h>
#include <unistd.h>

#define FALSE 0
#define TRUE 1

typedef int bool;

/* Variables globales */
int round = 0;
bool a = FALSE, b = FALSE;
int ppid=0;             /* identifiant du processus père */


                        /* Algorithme de Peterson */
```

[6] $\|$ est la composition synchrone sur les actions de mêmes étiquettes.

```
/* Les deux processus exécutent une boucle tant que dans la fonction
   peterson() dont le code est le suivant. */


int peterson()
{
  while(TRUE)
  {
    section_non_critique();
    init();
    wait();
    section_critique();
    reset();
  }
}

section_non_critique()
{
  /*
  Code de la section non critique
  */
}

/* Initialisation des variables globales selon l'identifiant
   du processus courant. */

init()
{
  int v = getpid();     /* getpid() retourne le numéro du processus
                           exécutant la tâche courant */

  if(v == ppid)
  {
    a = TRUE;           /* Processus père */
    round = 0;
  }
  else
  {
    b = TRUE;           /* Processus fils */
    round = 1;
  }
}

/* Gestion de l'acces à la section critique : le père (resp. fils) ne
   peut entrer dans la section critique que si round vaut 1 (resp. 0) ou b
```

```
   (resp. a) vaut vrai (resp. faux). Ainsi, chaque processus procède à
   une attente active. */

wait()
{
  int v = getpid();    /* getpid() retourne le numero de processus
                          executant la tâche courant */

  if(v == ppid)
  {
    while(b == TRUE && round == 0)     /* Attente active du père */
     ;
  }
  else
  {
    while(a == TRUE && round == 1)    /* Attente active du fils */
     ;
  }

}

section_critique()
{
  /*
  Code de la section non critique
  */
}

/* Le processus sortant de la section critique libère
   l'acces à cette zone. */

reset()
{
  int v = getpid();    /* getpid() retourne le numero de processus
                          exécutant la tâche courant */
  if(v == ppid)
  {
    a = FALSE;

  }
  else
  {
    b = FALSE;
  }
}
```

```
/* Fonction principale : Initialisation de la variable ppid et le
   clonage du processus père (fonction standard fork()) pour créer un
   processus fils qui exécutera avec son père la fonction peterson(). */

int main()
{
  int fpid = 0;         /* identifiant du processus fils */

  ppid = getpid();      /* getpid() retourne l'identifiant du processus
                           appelant */

  fpid = fork();        /* fork() cree un processus fils */

  peterson();

}
```

Ce code contient six fonctions qui sont exécutées par P_0 et P_1 (la fonction main() n'est exécutée que par P_0) et qui seront notées :

- $P()$: fonction peterson().
- $SNC()$: fonction section_non_critique().
- $I()$: fonction init().
- $W()$: fonction wait().
- $SC()$: fonction section_critique().
- $R()$: fonction reset().
- $M()$: fonction main().

Nous proposons de modéliser ces fonctions par des automates. Pour modéliser une fonction $F()$, nous suivons la convention suivante :

- L'action BF représente l'appel (début) à la fonction $F()$. Ainsi, BW représente l'appel à la fonction $wait()$.
- L'action EF représente le retour (fin) de la fonction $F()$. Ainsi, EW représente le retour de la fonction $wait()$.

Remarquons que si $F()$ ne termine jamais, alors EF ne peut pas être représentée. Par exemple, la fonction $peterson()$ est exécutée infiniment à cause de la boucle “tant que” (while) et donc EP ne sera pas présentée.

La Fig.23 illustre une modélisation en automates de ces fonctions. Dans le souci de ne pas surcharger la figure, les fonctions $SNC()$ et $SC()$ ne sont pas modélisées et sont représentées par les actions SNC et SC, respectivement.

Modélisation en CS. Une modélisation du processus P_i, $i \in [0,1]$ est le CS $S_i = (\emptyset, \emptyset, \emptyset, (F_i)_{i\in[1,4]}, Top)$, avec F_i et Top les automates de la Fig.23. Nous avons omis de

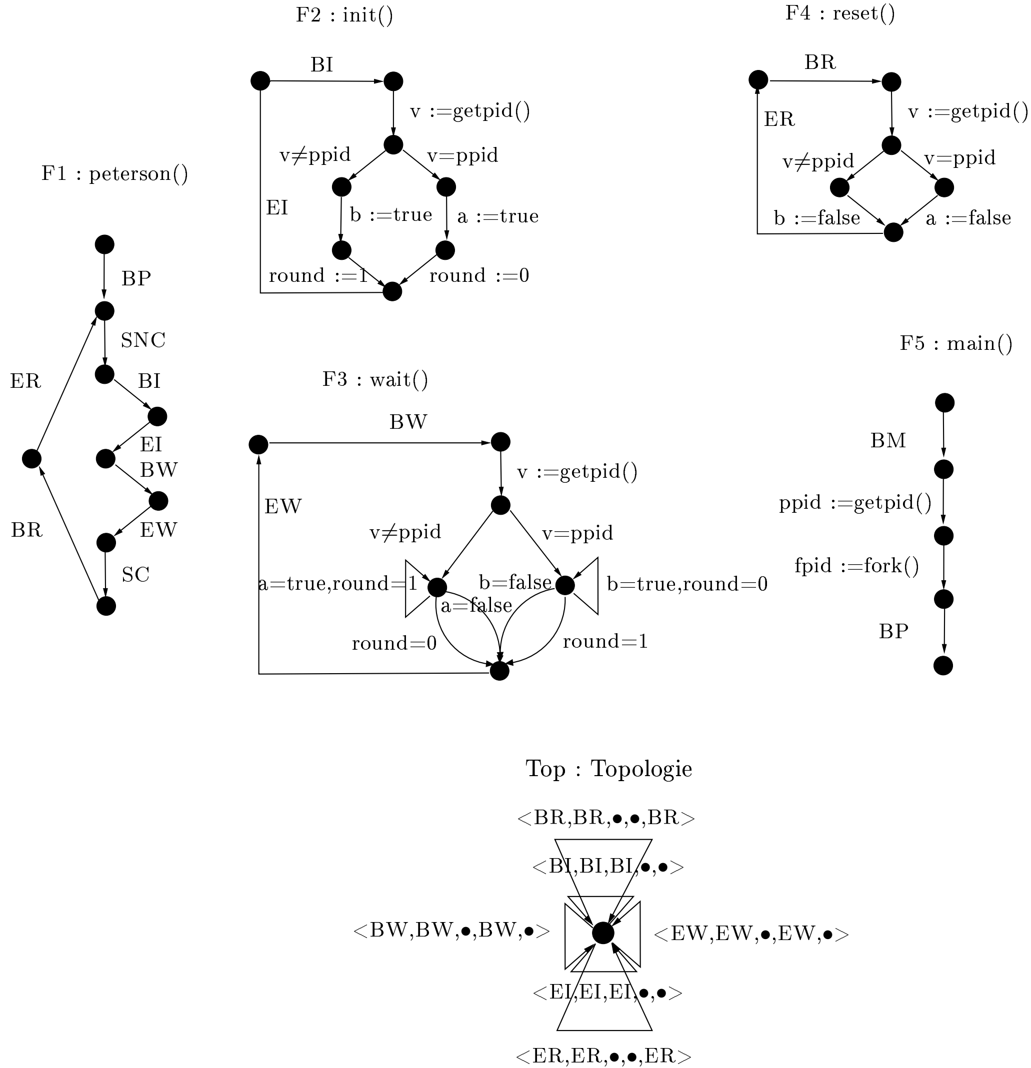


FIG. 23: Algorithme de Peterson.

représenter tous les vecteurs de synchronisation, mais nous supposons que tout vecteur unitaire sur toute autre action non représentée dans la figure de Top est permis. Dans cette modélisation, nous considérons que les deux processus ont déjà été créés [7]. Une modélisation du système composé de P_0 et P_1 est le CS libre (i.e. les composantes s'exécutent indépendamment les unes des autres) défini par $S = (\emptyset, V, V_0, , (S_i)_{i \in [0,1]}, Top_S)$, tels que $V = \{ppid, a, b, round\}$, $V_0 = \{0, false, false, 0\}$ et S_i le CS du processus P_i.

Remarquons que la fonction getpid() dépend du processus appelant et donc peut être modélisée par un paramètre ID qui sera dans ce cas un paramètre partagé de S ; les sections critiques et non critiques sont modélisées par des actions ; et enfin $v \neq ppid$ est équivalent

[7]Dans cette hypothèse, nous ne prenons pas en compte la modélisation de la fonction main(), ceci pour avoir des CS identiques pour P_0 et P_1, mais on peut toujours modéliser la fonction main() pour P_0.

à $v > ppid$ ou $v < ppid$. Finalement, le CS S tel qu'il est présenté n'est pas exploitable directement et une abstraction s'impose.

5.2.3 Protocole CSMA/CD

Principe. Le protocole CSMA/CD (Carrier Sense Multiple Access with Collision Detection) fait partie de la série des protocoles à détection de porteuse. Il permet aux stations d'adapter leurs décisions à l'activité en cours sur le canal d'émission. Quand une station veut expédier des données, elle commence par écouter le support de transmission pour savoir si une autre station n'est pas déjà en train de transmettre. Si c'est le cas, elle attend que le canal se libère. Quand il est de nouveau disponible, elle transmet sa trame. Lorsqu'une collision se produit, l'émetteur d'une trame observe un temps de pause aléatoire, puis recommence la procédure depuis le début. Si deux stations entament une transmission en même temps après s'être assurées que le canal est libre, elles provoqueront une collision qu'elles sauront toutefois immédiatement détecter. Dès qu'elles prennent connaissance de l'incident, elles interrompent sur le champ leurs activités. Ce protocole est largement déployé sur les réseaux locaux (LAN) dans la sous-couche MAC (Medium Access Control). Ethernet est une implantation de ce protocole.

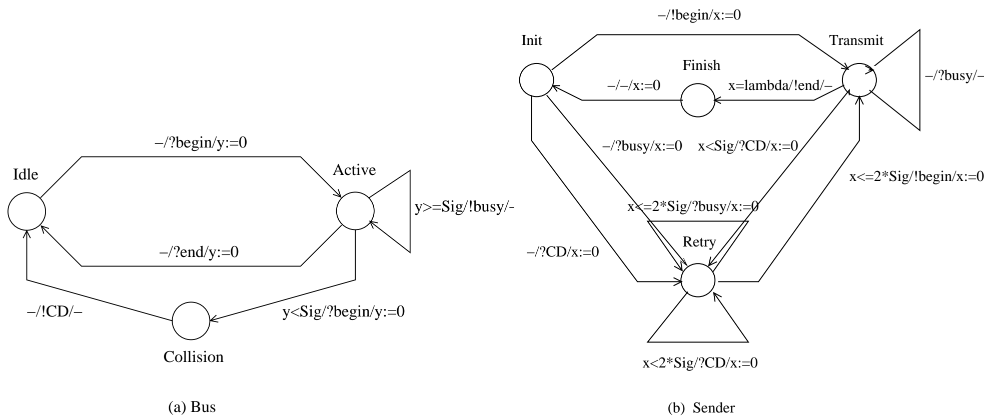


FIG. 24: Modélisation CSMA/CD.

Une modélisation d'un émetteur est donnée par l'automate *Sender* de la FIG.24 (b). L'émetteur est initialement dans l'état *Init*. Dans cet état, il se met à l'écoute du canal : il peut essayer de transmettre (!*begin*), il peut détecter une collision (?*CD*), ou il peut constater que le canal n'est pas libre (?*busy*). Dans l'état *Transmit*, l'émetteur est en train d'envoyer les données : *lambda* correspond au temps d'émission d'une trame. Dans cet état, l'émetteur peut terminer son émission (!*end*), ou recevoir les messages ?*busy* et ?*CD* dans le cas d'une station qui est en train d'émettre en même temps que lui. L'état *Retry* correspond à un état où l'émetteur a détecté que le canal est occupé ou lors d'une collision de trames. *Sig* correspond au temps de propagation entre les deux stations les plus éloignées du réseau.

Une modélisation du canal est donnée par l'automate *Bus* de la FIG.24 (a). Le canal est dans l'état initial *Idle*. Lorsqu'il reçoit une demande d'émission (*?begin*), le canal devient *Active*. Dans cet état, il informe tous les émetteurs de son état (*!busy*) et revient à l'état de repos après réception de *?end*. Cependant, si deux stations émettent en même temps, le canal reçoit deux demandes d'émission et dans ce cas une collision se produit. Le canal informe tous les émetteurs de la collision (*!CD*) et retourne à l'état de repos.

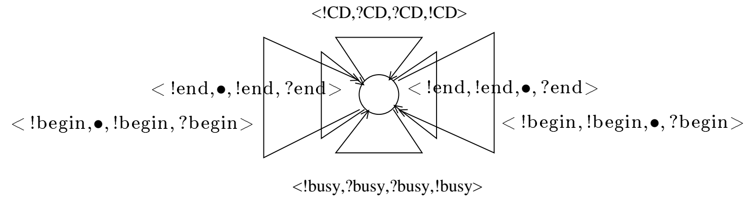


FIG. 25: Topologie de communication de CSMA/CD.

Modélisation en CS. Une modélisation contenant un bus et deux émetteurs est $S = (\{Sig, lambda\}, \emptyset, \emptyset, (Sender, Sender, Bus), Top)$, avec Top l'automate de la FIG.25 et *Sig* et *lambda* deux paramètres de S. Le vecteur de synchronisation $<!CD, ?CD, ?CD, !CD>$ consiste en une émission de $!CD$ par le bus, une réception de $?CD$ par les émetteurs (la collision est détectée par toutes les stations). L'action globale observable est une émission de $!CD$.

En conclusion, à travers ces trois exemples, la démarche suivie pour modéliser une application (programme ou protocole) consiste à séparer la modélisation des entités et le flux partagé entre ces entités, par l'utilisation de la topologie de communication, des paramètres et des variables partagées.

5.3 Communication synchrone/asynchrone

Jusqu'à maintenant, nous avons supposé que les processus communiquent d'une façon *synchrone*, i.e. un rendez-vous binaire ou multiple entre processus échangeant des messages. Cette abstraction implique que l'émission d'un message et sa réception est une action atomique qui prend zéro unité de temps. Or, en pratique, dans un système distribué synchrone temps-réel, l'échange de messages dépend du temps de traitement des couches inférieures et du temps de propagation du message dans le réseau, ce qui peut avoir des conséquences sur le comportement de ces systèmes. Nous proposons dans cette section, en premier lieu, de tenir compte dans la modélisation du temps de communication entre processus synchrones et en deuxième lieu, de modéliser les files d'attentes pour les systèmes asynchrones.

5.3.1 Système distribué synchrone

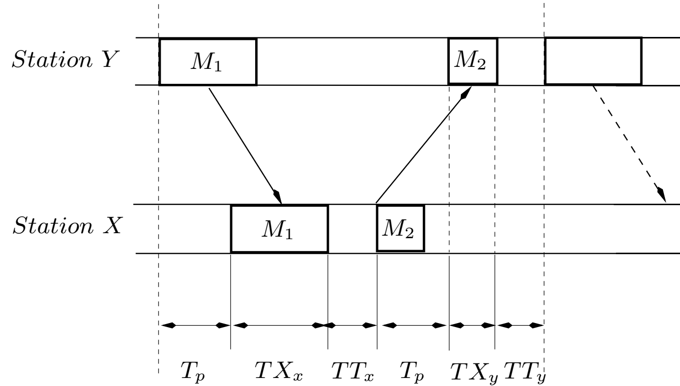


FIG. 26: Utilisation d'un canal par un protocole.

Considérer que les composantes d'un système distribué communiquent d'une façon synchrone est une abstraction, au premier regard, grossière. Cependant, il est possible de modéliser l'asynchrone avec du synchrone. En effet, si on considère un langage de programmation évolué, en l'occurrence le langage C ou $C++$, il est possible d'implanter des fonctions d'émission et de réception (*send*() et *rcv*()) bloquantes. D'ailleurs, il existe des librairies standards qui le font déjà. Le mécanisme derrière une fonction *send*() bloquante est que cette dernière ne termine pas (et donc le programme est bloqué) tant que le message n'a pas été bien reçu de l'autre coté, ce qui sous entend l'utilisation d'un mécanisme d'acquittement entre l'émetteur et le récepteur.

Différents temps relatifs à une fonction *send*() bloquante

Prenons le cas de l'envoi d'un message M_1 par une station Y vers une station X avec une fonction *send*() bloquante. Le déroulement événementiel simplifié du *send*() est le suivant : X envoie le message M_1 à Y, cette dernière acquitte ce message par un message M_2. Lors de la réception de M_2 par X, la fonction *send*() rend la main au programme qui peut continuer son exécution. Ici, nous avons supposé que le canal de communication entre X et Y est fiable, i.e. les messages ne sont ni corrompus ni perdus. La FIG.26 représente le schéma d'utilisation de cette fonction. Nous nous intéressons maintenant au temps T_{total} que la fonction *send*() met depuis son appel jusqu'à sa terminaison pour envoyer le message M_1 (comportement temporel). Les différents symboles utilisés dans la FIG.26 sont :

T_p = temps de propagation sur une liaison full-duplex entre X et Y,
TT_x = temps de traitement du message par la station X,
TT_y = temps de traitement du message par la station Y,
TX_x = temps de transmission du message par la station X,
TX_y = temps de transmission du message par la station Y.

Exemple 5.2 *À titre d'exemple, prenons le cas d'une station X se trouvant sur la terre et une station Y se trouvant au bord d'un satellite géo-stationnaire (à 36000km de la terre). La vitesse de propagation de la lumière est de $3 \times 10^8 m/s$. Si la station Y a un débit de 50kbps et la taille du message M_1 est 1000 bits, alors :*

$$TX_x = \frac{1000}{50000} = 0.02s, \ T_p = \frac{3\ 10^7}{3\ 10^8} = 0.12s,$$

Maintenant, supposons que $TX_x = TX_y$ et TT_x et TT_y soient négligeables, alors le temps total d'envoi du message M_1 par la fonction send() est :

$$T_{total} = 2 \times T_p + TX_x + TX_y + TT_x + TT_y \equiv 2 \times (TX_X + T_p) = 0.28s,$$

ce qui n'est pas négligeable pour des applications temps-réel. Par la suite $T_{total} = 2 \times T_p + TX_x + TX_y + TT_x + TT_y$ sera appelé le temps de communication entre Y et X et sera noté $T_{total}(Y, X)$. □

Prise en compte du temps de communication dans la modélisation

À travers l'exemple du *send*(), nous avons mis en évidence les différents temps à considérer pour une abstraction d'une communication synchrone pour les systèmes distribués temps-réel. Nous proposons de modéliser les différents temps de communication dans la topologie de communication étant donné qu'elle représente le flux de contrôle partagé par les différentes entités. Dans ce cas, la topologie n'est plus un simple automate mais un TIOA. Pour ce faire, nous représentons explicitement le temps de communication relatif à chaque vecteur de synchronisation. Intuitivement, un vecteur de synchronisation sera divisé (ou remplacé) par deux vecteurs. Le premier vecteur correspond à l'action d'émission et le deuxième correspond à l'action de réception. Par l'intermédiaire d'une horloge, l'action de réception ne peut être effectuée qu'après un certain temps correspondant au temps de communication du vecteur de synchronisation initial.

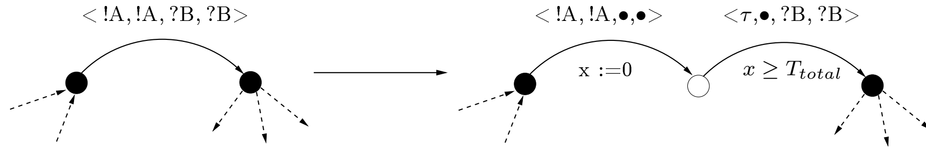


FIG. 27: Modélisation de la latence.

Soit $S = (SP, VP, V_0, (P_i)_{i\in[1,n]}, Top)$ un CS. Supposons que $t = (s_i, \overrightarrow{v}, s_j)$ est une transition dans Top avec $\overrightarrow{v} =< a_g, a_1, \cdots, a_n >$ un vecteur de synchronisation de n processus $(P_i)_{i\in[1,n]}$. Rappelons que les actions a_i des P_i se synchronisent et donnent lieu à une action globale a_g. Supposons que a_1 est une action d'émission et donc par définition les autres actions sont des réceptions. Notons par $T_{total}(\overrightarrow{v}) = max(\{T_{total}(P_1, P_i) \mid i \in [2, n]\})$,

le temps maximal pour l'émission de a_1. Considérons les deux transitions $t_1 = (s_i, \overrightarrow{v_1}, s)$ et $t_2 = (s, \overrightarrow{v_2}, s_j)$ avec $\overrightarrow{v_1} =< a_g, a_1, \bullet, \cdots, \bullet >$, $\overrightarrow{v_2} =< \tau, \bullet, a_2, \cdots, a_n >$ et s un état additionnel (ajouté à Top) urgent. Rappelons que dans un état urgent, le système doit quitter cet état dès que l'une des transitions sortantes est possible. L'idée est alors de remplacer la transition t dans Top par deux transitions t_1 et t_2 et d'utiliser une horloge x remise à zéro dans t_1 et contrainte par la garde $x \geq T_{total}(\overrightarrow{v})$ dans t_2. La FIG.27 illustre la transformation dans le cas $n = 3$ pour le vecteur $\overrightarrow{v} =<!A, !A, ?B, ?B>$. Le cercle blanc représente un état urgent. Dans cette transformation, le système global exécute le vecteur de synchronisation en deux phases : une phase d'émission de A suivie d'une attente, dans l'état urgent, de T_{total} unités de temps pour s'assurer que le message a été effectivement reçu et une deuxième phase de réception. On remarque que le nombre de transitions et d'états de Top est doublé et qu'il suffit d'une seule horloge additionnelle.

En conclusion, nous avons mis en évidence les différents temps à considérer dans la modélisation d'un système distribué synchrone temps-réel. A travers la topologie de communication, il est relativement simple de modéliser ces temps de communication en divisant les vecteurs de synchronisation et en introduisant une horloge supplémentaire. Dans ce cas, la topologie de communication est modélisée par un TIOA. Nous nous intéressons par la suite à la modélisation d'un canal de communication dans le cas d'un système distribué asynchrone.

5.3.2 Système distribué asynchrone

Dans un système distribué asynchrone, chaque entité dispose de canaux d'émission pour les messages sortants et de canaux de réception pour les messages entrants. Ces canaux sont en général finis et de types FIFO (First In First Out). Dans ce cas, chaque entité se comporte d'une façon indépendante des autres entités du système : l'émission (resp. la réception) d'un message consiste à le déposer (resp. le retirer) dans/de la file d'émission (resp. de réception). Nous proposons dans cette partie de modéliser un canal d'émission F de type FIFO et de taille $N \in \mathbb{N}$ (la modélisation d'une file de réception est identique).

Modélisation d'une file

Soit L un alphabet. Notons par L^i l'ensemble des séquences de taille i ($\sigma \in L^*$ tel que $|\sigma| = i$). Avant de présenter cette modélisation, considérons les deux transformations suivantes : $\rightarrow_{L_s^i}$ (resp. $\rightarrow_{L_p^i}$) associe à chaque séquence de L^i un suffixe (resp. préfixe) qui la transforme en une séquence de L^{i+1} ou ϵ.

$$\rightarrow_{L_s^i}: L^i \times L^{i+1} \mapsto L \cup \{\epsilon\}, \qquad \rightarrow_{L_s^i}(w, w') = \begin{cases} a & si\ w' = wa, \\ \epsilon & sinon \end{cases}$$

$$\rightarrow_{L_p^i}: L^i \times L^{i+1} \mapsto L \cup \{\epsilon\}, \qquad \rightarrow_{L_p^i}(w, w') = \begin{cases} a & si\ w' = aw, \\ \epsilon & sinon \end{cases}$$

Exemple 5.3 *Supposons que $L = \{a, b\}$ alors : $L^0 = \{\epsilon\}$, $L^1 = \{a, b\}$, $L^2 = \{aa, ab, bb, ba\}$.*

- $\rightarrow_{L_s^0}(\epsilon, a) = a$, $\rightarrow_{L_s^0}(\epsilon, b) = b$
- $\rightarrow_{L_p^0}(\epsilon, a) = a$, $\rightarrow_{L_s^0}(\epsilon, b) = b$
- $\rightarrow_{L_s^1}(a, aa) = a$, $\rightarrow_{L_s^1}(a, ab) = b$, $\rightarrow_{L_s^1}(b, bb) = b$, $\rightarrow_{L_s^1}(b, ba) = a$ *et pour les autres configurations possibles* ϵ.
- $\rightarrow_{L_p^1}(a, aa) = a$, $\rightarrow_{L_p^1}(a, ba) = b$, $\rightarrow_{L_p^1}(b, bb) = b$, $\rightarrow_{L_p^1}(b, ab) = a$ *et pour les autres configurations possibles* ϵ. □

Soient L un alphabet et $x \in L$. Notons par $\overline{x}$ le conjugué de x et par $\overline{L}$ l'ensemble des conjugués de L ($\overline{L} = \{\overline{x} \mid x \in L\}$). Par exemple, $\overline{x} = !a$ est le conjugué de $x = ?a$.

Définition 23 *Soit L un alphabet de cardinal $N \in \mathbb{N}$ tel que $L \cap \overline{L} = \emptyset$. L'automate à file associé à L est l'automate $A_F = (Q, q_0 = \epsilon, L \cup \overline{L}, \rightarrow)$ défini par :*

- $Q = \bigcup_{i \in [0,N]} L^i$ *est l'ensemble des séquences de taille inférieure ou égale à N.*
- $q \xrightarrow{a} q'$ *ssi il existe $i \in [0, N-1]$ tels que :*
  1. $a \in L \cup \overline{L}$ ($a \neq \epsilon$).
  2. $q \in L^i$, $q' \in L^{i+1}$ *et* $\rightarrow_{L_s^i}(q, q') = a$ *ou*
  3. $q' \in L^i$, $q \in L^{i+1}$ *et* $\rightarrow_{L_p^i}(q, q') = \overline{a}$. □

Les états de A_F sont les séquences de taille inférieure ou égale à N. Deux types de transitions sont définis dans A_F : transitions de réception sur l'alphabet L et transitions d'émission sur l'alphabet $\overline{L}$. Deux états q et q' sont reliés par une transition si la réception d'un événement transforme q en q', ou si l'émission d'un événement transforme q en q' et inversement. Il est facile de voir que le nombre d'états et de transitions de A_F est :

$$e(A_F) = \sum_{i \in [0,N]} card(L^i) = \frac{1 - card(L)^{N+1}}{1 - card(L)} \quad et \quad t(A_F) = (e(A_F) - 1) \times card(L)$$

Exemple 5.4 *La partie gauche de la* FIG.*28 représente l'automate à file associé à l'alphabet $L = \{a, b\}$.* □

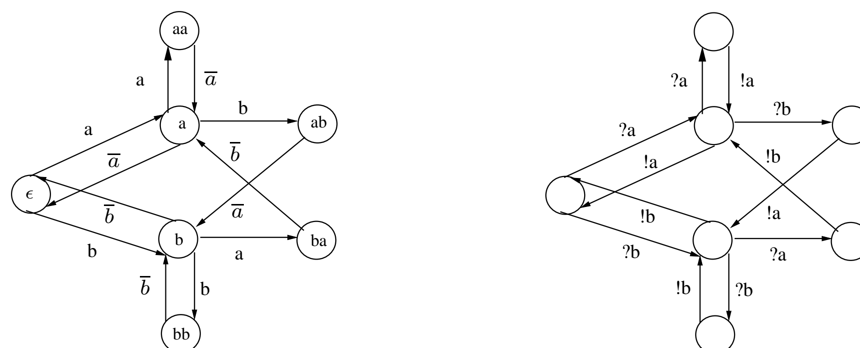


FIG. 28: Modélisation d'une file de taille 2.

Modélisation d'une file F de taille N

Maintenant, nous proposons de modéliser une file F de taille N. Supposons qu'une entité A envoie sur la file F un ensemble de messages qui forme un alphabet L. Par définition, les éléments de L sont des actions de réception du point de vue de la file ($L \cap \overline{L} = \emptyset$). Dès lors, l'automate à file associé à L est une modélisation de la file F.

Exemple 5.5 *La partie droite de la* FIG.*28 représente une modélisation d'une file F de taille 2 sur l'alphabet $L = \{?a, ?b\}$.* □

En conclusion, à travers la modélisation des canaux de communication par des automates, pour les systèmes asynchrones, il est possible de modéliser un système distribué asynchrone par un CSs. Dans ce cas, le canal est considéré comme une entité du système et la modélisation de la latence est identique à celle présentée dans le cas des systèmes distribués synchrones. Finalement, dans un système asynchrone, un message peut être destiné à plusieurs files sortantes et dans ce cas, l'utilisation de la topologie de communication permet de tenir compte de cette situation.

5.4 Conclusion

L'une des motivations de ce chapitre était la modélisation, basée sur une extension des automates, des systèmes de processus communicants en vue d'une validation de ces processus. Le choix des automates comme modèle de base est justifié par la possibilité de définition d'algorithmes applicables dans un outil de validation.

Notre contribution est l'introduction du modèle CS. Le modèle CS a été défini comme l'association de la description individuelle des processus (i.e. le flux de contrôle et de données de chaque processus), de la topologie de communication explicitant les synchronisations possibles entre processus (i.e. le flux de contrôle liant les processus) et les variables

et paramètres partagés représentant les ressources communes (i.e. le flux de données liant les processus). L'intérêt d'un tel modèle est la modularité de la construction du système global : séparation entre le flux de contrôle et de données propres à un processus et le flux de contrôle et de données partagés entre processus. Un autre point en faveur du modèle CS est la possibilité de modéliser dans un même système, différents types de communication : unicast sur un ensemble A de messages, multicast ou broadcast sur un ensemble B de messages, ce qui étend le champ d'application des CSs.

Le fait que la sémantique d'un CS soit définie en terme d'un automate permet, comme dans le cas d'une algèbre de processus, de construire des systèmes plus complexes à partir de composantes réduites (une définition hiérarchique). Par l'étude de la taille de la sémantique, nous avons montré qu'une modélisation en CS n'introduit pas un superflu, mais c'est un coût de modélisation. À travers les exemples du consommateur-producteur, l'algorithme de Peterson et la norme CSMA/CD, nous avons montré comment exprimer, en CS, des composantes habituelles, que ce soit un protocole ou un programme en langage évolué. Cette activité est loin d'être automatique et nécessite une certaine expertise.

La dernière section de ce chapitre était consacrée à des extensions du modèle CS en vue de prendre en compte les différents temps et mécanismes influant sur l'échange de messages dans un système.

Troisième partie

Analyse des Systèmes Temporisés

Introduction

Lors de la vérification d'une propriété sur un système, une simple réponse “oui/non” est en général insuffisante. Les *diagnostics* [8] (ou traces) sont des informations supplémentaires (par exemple états, exécutions, $\cdots$) qui aident l'utilisateur à comprendre pourquoi le système a échoué ou vérifie la propriété. Les diagnostics sont importants pour les raisons suivantes :

1. sans eux, aucune confiance dans le modèle du système ne peut être obtenue. Par exemple, dans le cas de la non satisfaction d'une propriété par le modèle du système, il est possible que l'erreur vienne du modèle du système et non du système lui-même.
2. même si le modèle du système est correct, une erreur du système ne peut pas être localisée sans aucun guidage.

Dans le cas des automates temporisés, il y a un besoin de *diagnostics temporisés*, contenant aussi bien des informations sur les changements discrets du système, que les délais exacts entre deux transitions discrètes. Ces délais peuvent être essentiels pour comprendre un comportement simple du système.

Motivations

Considérons le chemin $\rho = t_1 \cdots t_n$ (resp. $\rho' = t'_1 \cdots t'_n$) de taille $n \in \mathbb{N}$ d'un automate temporisé A (resp. B) :

$$\rho \ : \ s_0 \xrightarrow{Z_1,a_1,r_1} s_1 \xrightarrow{Z_2,a_2,r_2} s_2 \cdots \xrightarrow{Z_n,a_n,r_n} s_n$$

$$\rho' \ : \ s'_0 \xrightarrow{Z'_1,a'_1,r'_1} s'_1 \xrightarrow{Z'_2,a'_2,r'_2} s'_2 \cdots \xrightarrow{Z'_n,a'_n,r'_n} s'_n$$

tels que

$$\text{pour tout } i \in [1,n],\ a_i = a'_i$$

A_ρ (resp. $B_{\rho'}$) représentera l'automate temporisé induit par le chemin ρ (resp. ρ').

Nous nous intéressons dans cette partie aux deux problèmes suivants.

1. *Extraction des diagnostics temporisés.* Dans ce problème, on cherche à identifier quelques éléments de $TTrace(A_\rho, n)$[9].
2. *Inclusion des traces.* Dans ce problème, on cherche à vérifier l'inclusion des traces $TTrace(A_\rho, n) \subseteq TTrace(B_{\rho'}, n)$ des deux chemins ρ et ρ' sachant que :

[8] Dans ce document, nous préférons utilisé le terme diagnostic au lieu de trace.

[9] Rappelons que $TTrace(A, n)$ est l'ensemble des traces temporisées de A de taille n. Voir la section 4.3.4.

- A_ρ est connu : les différentes contraintes $(Z_i)_{i\in[1,n]}$ et remises à jour $(r_i)_{i\in[1,n]}$ de ρ sont données.
- $B_{\rho'}$ n'est pas connu ($(Z'_i)_{i\in[1,n]}$ et $(r'_i)_{i\in[1,n]}$ ne sont pas données) mais seulement l'ensemble $TTrace(B_{\rho'}, n)$ est connu.

Le problème d'extraction des diagnostics temporisés, comme nous l'avons dit auparavant, trouve une application intéressante dans la vérification des propriétés d'un système. Sachant que la majorité des algorithmes de vérification utilisent des graphes abstraits pour l'analyse exhaustive des comportements, l'énumération des différents états du système est en général impossible. Une conséquence directe est que ces algorithmes ne peuvent fournir un diagnostic direct que sous forme de chemins symboliques.

Le problème d'inclusion des traces est à notre connaissance, un problème nouveau. Au delà de la vérification des propriétés, ce problème trouve une application intéressante dans le test. En effet, dans le cas du test de conformité boîte noire des systèmes temporisés, le code de l'implantation est inconnu et seulement ses traces sont connues. Ainsi, vérifier la conformité de l'implantation par rapport à la spécification revient en général à une vérification d'inclusion des de l'impantation dans celles de la spécification.

Contributions

L'extraction des diagnostics temporisés a été introduit initialement par Alur et al. [6] qui proposent de calculer la trace temporisée de cumul temporel minimal. La méthode d'extraction proposée par Tripakis [145] consiste à choisir aléatoirement des diagnostics temporisés qui n'ont pas forcément des propriétés intéressantes comme la minimalité temporelle.

Les contributions de cette partie se portent sur l'extraction des diagnostics temporisés et l'étude du problème d'inclusion des traces. Par rapport aux travaux relatifs à l'extraction des diagnostics temporisés, la solution que nous proposons consiste à identifier des diagnostics temporisés ayant certaines propriétés. Ainsi, nous identifions *les diagnostics temporisés de bornes*. Ces derniers considèrent des comportements limites du système. Leur nombre varie entre 1 et $2 \times (n + 1)$ pour un chemin de longueur n. Le problème d'inclusion des traces se résume alors à une application directe de l'identification des diagnostics temporisés de bornes. Deux solutions sont proposées pour calculer ces diagnostics.

1. La première solution est basée sur le calcul du *polyèdre de contraintes* associé à un chemin. Le polyèdre de contraintes définit l'ensemble des contraintes que doivent vérifier une séquence temporisée σ pour pouvoir admettre une computation de l'automate.
2. La deuxième solution repose sur une analyse d'accessibilité en avant et en arrière. L'analyse d'accessibilité est l'une des techniques d'analyse les plus répandues, d'une part, du fait qu'elle a un pouvoir d'expressivité assez suffisant pour formuler un large

ensemble de propriétés telles l'invariance et la réponse bornée et d'autre part, elle n'est pas coûteuse au niveau implantation. A travers l'analyse d'accessibilité d'un état, nous apporterons une autre solution aux problèmes d'extraction de diagnostics temporisés basée sur la définition de deux opérateurs, un opérateur $post()$ de calcul des états successeurs d'un état symbolique donné et un opérateur $pred()$ faisant l'opération inverse.

Organisation

Cette partie se compose de deux chapitres. Le chapitre 6 est un chapitre préliminaire. Nous examinons dans ce chapitre l'extraction des valuations à partir d'un polyèdre. La solution proposée est basée sur la définition d'un ensemble de transformations sur le graphe associé à un polyèdre. Le chapitre 7 introduit nos deux approches pour extraire les diagnostics temporisés de bornes, ainsi qu'un état de l'art et une comparaison avec des travaux similaires.

Chapitre 6

Graphe, polyèdre et valuation

Sommaire

6.1 Graphe positif et minimal **84**
6.1.1 Préliminaires 85
6.1.2 Transformation b2m() 86
6.1.3 Transformation $R_{i\to *}()$ 87
6.1.4 Transformation $R_{*\to i}()$ 88
6.1.5 Récapitulation 90
6.2 Extraction de valuations à partir d'un polyèdre **90**
6.2.1 Préliminaires 90
6.2.2 Forme canonique d'un polyèdre 91
6.2.3 Valuation de bornes 94
6.2.4 k-complétude d'une valuation 98
6.2.5 Conclusion 98
6.2.6 Ce que nous allons faire 99

Dans ce chapitre préliminaire, nous montrons comment extraire des valuations à partir d'un polyèdre. Cette extraction consiste en l'identification des valuations de bornes et la k-complètude. Les graphes et quelques transformations sur ces derniers introduits dans la section 6.1 sont une base formelle utilisée pour démontrer l'existence des valuations de bornes. Les différents résultats de ce chapitre seront exploités par le suite dans le chapitre 7 pour extraire les diagnostics temporisés.

6.1 Graphe positif et minimal

Dans le reste de cette section, $N = \{n_1, n_2, \cdots, n_k\}$ représentera un ensemble de noeuds. Pour les définitions élémentaires relatives aux graphes, voir la section 2.1.

6.1.1 Préliminaires

Définition 24 (Graphe positif) *Soit $G = (N, \mathbb{T}, E) \in \overrightarrow{G}_c(N, \mathbb{T})$ un graphe complet, orienté et étiqueté. G est positif si et seulement si le poids de tout cycle de G est positif.* □

Formellement, G est positif si $\forall c$ cycle de G, alors $w(c) \geq 0$.

Propriété 1 *G est positif ssi les e-cycles (cycles élémentaires) sont positifs.* □

Preuve. Voir Annexe A. □

Définition 25 (Graphe minimal) *Soit $G = (N, \mathbb{T}, E) \in \overrightarrow{G}_c(N, \mathbb{T})$ un graphe complet et étiqueté. G est dit des plus courts chemins, ou simplement minimal, ssi pour tout arc e, son poids $w(e)$ est inférieur au poids $w(p)$ de tout chemin p de path(e).* □

Formellement, G est minimal ssi pour tout arc $e \in E$, pour tout chemin $p \in path(e)$ $\Rightarrow w(e) \leq w(p)$. L'ensemble des graphes minimaux de noeuds N sera noté $\overrightarrow{G}_m(N, \mathbb{T}) \subseteq \overrightarrow{G}_c(N, \mathbb{T})$

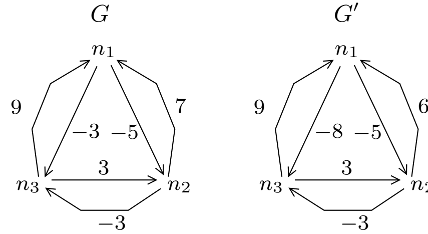


Fig. 29: Graphes complets et étiquetés sur $\mathbb{T} = \mathbb{Z}$.

Exemple 6.1 *La* Fig.*29 illustre l'exemple de deux graphes complets et étiquetés sur $\mathbb{Z}$ tels que $N = \{n_1, n_2, n_3\}$. D'après leurs définitions, entre deux noeuds n_i et n_j du graphe G (resp. G'), il existe un arc de n_i vers n_j et un arc de n_j vers n_i. Il est facile de remarquer que :*

- *G et G' sont tous les deux positifs. En effet, le poids de tout cycle élémentaire est positif dans G et G'.*
- *G n'est pas minimal. En effet, $w_G(n_2 \rightarrow n_1) = 7$, $w_G(n_2 \rightarrow n_3 \rightarrow n_1) = 6$ et par conséquence $w_G(n_2 \rightarrow n_1) > w_G(n_2 \rightarrow n_3 \rightarrow n_1)$*
- *G' est minimal du fait que le poids de tout arc e dans G' est inférieur au poids dans G' de tout chemin de même source et destination que e.* □

Propriété 2 *Soit G un graphe minimal. G est positif si et seulement si les 2-cycles de G sont positifs.* □

Intuitivement, si un graphe est minimal, pour montrer qu'il est positif, il suffit de montrer que les cycles, ayant exactement deux noeuds, sont positifs.

Preuve. Voir Annexe A. □

Par la suite, nous introduisons quelques transformations sur les graphes complets et étiquetés.

6.1.2 Transformation b2m()

Considérons la fonction $b2m()$, qui transforme un graphe complet et étiqueté G en un graphe minimal G', définie par :

$$\begin{array}{rrcl} b2m \,:\, & \overrightarrow{G}_c(N, \mathbb{T}) & \mapsto & \overrightarrow{G}_m(N, \overline{\mathbb{T}}) \\ & G & \rightarrow & G' \end{array}$$

telle que pour tout arc e :

$$w_{G'}(e) = min(\{w_G(p) \,|\, p \in path(e)\}).$$

Intuitivement, le poids d'un arc de source n_i et de destination n_j dans le graphe G' correspond au poids minimal dans G, de tout chemin de source n_i et de destination n_j. Ce poids minimal est soit atteint par un chemin, i.e. il existe un chemin $p \in path(e)$ tel que $w_{G'}(e) = w_G(p)$, soit égal à $-\infty$ dans le cas où l'ensemble $\{w_G(p) \,|\, p \in path(e)\}$ n'est pas minoré.

Remarque 7 *La transformation $b2m()$ est bien définie, i.e. G' tel qu'il est construit, appartient effectivement à $\overrightarrow{G}_m(N, \mathbb{T})$. En effet, supposons qu'il existe un chemin $p = e_1 \cdots e_k$ de $path(e)$ tel que $w_{G'}(e) \geq w_{G'}(p) = \sum_{i \in [1,k]} w_{G'}(e_i)$. D'après la construction de G', pour chaque arc e_i, il existe un chemin p_i de $path(e_i)$ tel que $w_G(p_i) = w_{G'}(e_i)$. Dans ce cas, le chemin p' obtenu par concaténation des chemins p_i ($p' = p_1 \cdots p_k$) est dans $path(e)$ et par conséquent $w_G(p') = \sum_{i \in [1,k]} w_G(p_i) = \sum_{i \in [1,k]} w_{G'}(e_i) = w_{G'}(p) \leq w_{G'}(e)$. Contradiction.* □

Exemple 6.2 *Prenons les graphes de la* FIG.*29. Il est facile de voir que $G' = b2m(G)$. En effet, G' diffère de G sur le poids des arcs $n_2 \rightarrow n_1$ et $n_1 \rightarrow n_3$: $w_{G'}(n_2 \rightarrow n_1) = w_G(n_2 \rightarrow n_3 \rightarrow n_1) \leq w_G(n_2 \rightarrow n_1)$ et $w_{G'}(n_1, n_3) = w_G(n_1 \rightarrow n_2 \rightarrow n_3) \leq w_G(n_1 \rightarrow n_3)$.* □

Lemme 3 *Soit $G \in \overrightarrow{G}_c(N, \mathbb{T})$. Alors, G est positif ssi $b2m(G)$ l'est aussi.* □

En autres termes, la transformation $b2m()$ préserve la positivité des cycles comme on peut le constater dans la FIG.29.

Preuve. L'idée de la démonstration est la suivante. Pour montrer que G est positif, on compare les poids des e-cycles dans les deux graphes (propriété 1). De même, pour montrer que $b2m(G)$ est positif, il suffit de montrer que ses 2-cycles sont positifs selon la propriété 2. Une preuve complète est donnée dans l'Annexe A. □

6.1.3 Transformation $R_{i\to *}()$

Rappelons que k est le nombre de noeuds ($k = card(N)$). Soit $i \in [1, k]$. Considérons la transformation $R_{i\to *}()$ définie par :

$$\begin{array}{rccc} R_{i\to *} : & \overrightarrow{G}_c(N, \mathbb{T}) & \mapsto & \overrightarrow{G}_c(N, \mathbb{T}) \\ & G & \to & G' \end{array}$$

telle que pour tout arc e :

$$w_{G'}(e) = \begin{cases} -w_G(\overline{e}) & si\ e \in in(n_i) \\ w_G(e) & sinon \end{cases}$$

Rappelons tout d'abord que $\overline{e}$ est l'arc conjugué de e. $R_{i\to *}$ fait correspondre à un graphe G le graphe G' tel que les poids des arcs de destination n_i dans G ont été changés dans G' de la façon suivante : pour chaque arc $e \in in(n_i)$, le poids $w_{G'}(e)$ de e dans G' est égal à l'opposé du poids $w_G(\overline{e})$ de l'arc $\overline{e}$ dans G.

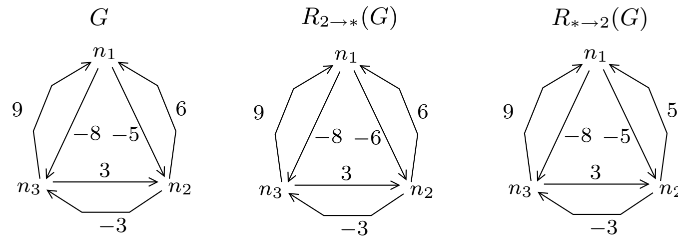


FIG. 30: Transformations $R_{2\to *}$ et $R_{*\to 2}$.

Exemple 6.3 *La* FIG.*30 illustre un exemple de cette transformation dans le cas $i = 2$ pour le graphe G. D'une part, le poids de $n_1 \to n_2$, dans G, est remplacé dans $R_{2\to *}(G)$ par -6 qui correspond à l'opposé du poids de $n_2 \to n_1$ dans G. D'autre part, le poids de*

$n_3 \rightarrow n_2$, dans G, est remplacé dans $R_{2\rightarrow *}(G)$ par 3 qui correspond à l'opposé du poids de $n_2 \rightarrow n_3$ dans G. Les autres arcs conservent le même poids dans G et $R_{2\rightarrow *}(G)$. □

Les lemmes suivants énoncent quelques propriétés de cette transformation relatives à la positivité et la minimalité du graphe transformé.

Lemme 4 *Soient $G \in \overrightarrow{G}_m(N, \mathbb{T})$ et $i \in [1, k]$. Si G est positif alors $R_{i\rightarrow *}(G)$ l'est aussi.* □

Intuitivement, le lemme établit que si un graphe est minimal et positif alors la transformation $R_{i\rightarrow *}()$ préserve la positivité des cycles. Cependant, comme nous allons le voir, $R_{i\rightarrow *}()$ ne préserve pas la minimalité.

Preuve. Voir Annexe A. □

Lemme 5 *Soient $G \in \overrightarrow{G}_m(N, \mathbb{T})$ et $i \in [1, k]$. Supposons que G est positif. Posons $G_1 = b2m(R_{i\rightarrow *}(G))$. Alors, pour tout arc e :*

$$w_{G_1}(e) = \begin{cases} w_G(e) & si\ e \in out(n_i) \\ -w_G(\overline{e}) & si\ e \in in(n_i) \\ -w_G(\overline{e}_1) + w_G(e_2) & sinon \end{cases}$$

avec $e_1 \in in(n_i)$, $e_2 \in out(n_i)$, $src(e_1) = src(e)$ et $dst(e_2) = dst(e)$. □

Ce lemme donne une méthode pour calculer le graphe minimal après la transformation $R_{i\rightarrow *}(G)$ en fonction des poids de G.

Preuve. Voir Annexe A. □

Complexité. Si G est minimal, alors le calcul du graphe minimal G_1 à partir de G se fait en $O(k)$.

6.1.4 Transformation $R_{*\rightarrow i}()$

Cette transformation est semblable à la transformation $R_{i\rightarrow *}()$. Elle est définie par :

$$\begin{array}{rccc} R_{*\rightarrow i} : & \overrightarrow{G}_c(N, \mathbb{T}) & \mapsto & \overrightarrow{G}_c(N, \mathbb{T}) \\ & G & \rightarrow & G' \end{array}$$

telle que pour tout arc e :

$$w_{G'}(e) = \begin{cases} -w_G(\overline{e}) & si\ e \in out(n_i) \\ w_G(e) & sinon \end{cases}$$

Intuitivement, la seule différence entre G et G', obtenu par la transformation $R_{*\to i}$, est sur le poids des arcs de source n_i. Pour tout arc $e \in out(n_i)$, le poids $w_{G'}(e)$ de e dans G' est égal à l'opposé du poids $w_G(\overline{e})$ de l'arc $\overline{e}$ dans G.

Exemple 6.4 *La* FIG.*30 illustre un exemple de cette transformation dans le cas $i = 2$ pour le graphe G. D'une part, le poids de $n_2 \to n_1$, dans G, est remplacé dans $R_{*\to 2}(G)$ par 5 qui correspond à l'opposé du poids de $n_1 \to n_2$ dans G. D'autre part, le poids de $n_2 \to n_3$, dans G, est remplacé dans $R_{*\to 2}(G)$ par -3 qui correspond à l'opposé du poids de $n_3 \to n_2$ dans G. Les autres arcs conservent le même poids dans G et $R_{*\to 2}(G)$.* □

Les deux lemmes qui suivent rapportent des propriétés de cette transformation semblables à celles de la transformation $R_{i\to *}(G)$.

Lemme 6 *Soient $G \in \overrightarrow{G}_m(N, \mathbb{T})$ et $i \in [1, k]$. Si G est positif alors $R_{*\to i}(G)$ l'est aussi.* □

Preuve. Une preuve similaire à celle du lemme 4. □

Lemme 7 *Soient $G \in \overrightarrow{G}_m(N, \mathbb{T})$ et $i \in [1, k]$. Supposons que G est positif. Posons $G_2 = b2m(R_{*\to i}(G))$. Alors, pour tout arc e :*

$$w_{G_2}(e) = \begin{cases} w_G(e) & si\ e \in in(n_i) \\ -w_G(\overline{e}) & si\ e \in out(n_i) \\ w_G(e_1) - w_G(\overline{e}_2) & sinon \end{cases}$$

avec $e_1 \in in(n_i)$, $e_2 \in out(n_i)$, $src(e_1) = src(e)$ et $dst(e_2) = dst(e)$. □

Preuve. Une preuve similaire à celle du lemme 5. □

Complexité. Si G est minimal, alors le calcul du graphe minimal G_2 à partir de G se fait en $O(k)$.

6.1.5 Récapitulation

Le schéma de la FIG.31 présente les différentes transformations et résultats introduits dans cette section.

1. La transformation $b2m()$ associe à G le graphe minimal (ou le graphe des plus courts chemins) G'. Nous avons démontré que G est positif ssi G' est positif.
2. Pour un noeud n_i, la transformation $R_{i\to *}()$ associe à G' le graphe G_1 dont les valeurs des poids d'arcs entrants au noeud n_i (les arcs de $in(n_i)$) ont été remplacées, pour chaque arc, par l'opposé du poids de l'arc conjugué. Nous avons démontré que si G' est positif alors G_1 l'est aussi.
3. Finalement, nous avons explicité comment calculer linéairement le graphe minimal G_3 (resp. G_4) associé à G_1 (resp. G_2) par $b2m()$.

Ces résultats généraux seront exploités par la suite pour identifier certaines propriétés des polyèdres.

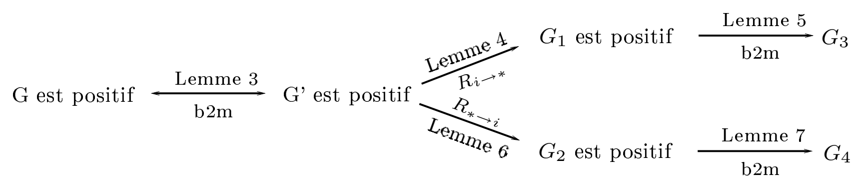


FIG. 31: Transformations et résultats sur les graphes complets.

6.2 Extraction de valuations à partir d'un polyèdre

Dans le reste de cette section, $V = \{v_1, ..., v_n\}$ dénotera un ensemble de variables à valeurs dans $\mathbb{R}^{\geq 0}$, $V_0 = V \cup \{v_0\}$ l'ensemble V muni d'une *variable fictive* v_0 qui vaut 0 tout le temps et Z un polyèdre de $\Phi(V)$. Nous confondrons les éléments de $\Phi(V)$ avec les éléments de $\Phi(V_0)$ et une valuation ν de V avec la valuation ν' de V_0 qui affecte 0 à v_0 et $\nu(v_i)$ à v_i, pour tout $i \in [1, n]$.

Le but de cette section est de montrer comment extraire des valuations à partir d'un polyèdre. Pour ce faire, nous introduisons la forme canonique d'un polyèdre.

6.2.1 Préliminaires

Par définition, Z est une conjonction de contraintes de la forme $v_i \leq c \,|\, v_i \geq c \,|\, v_i - v_j \leq c \,|\, v_i - v_j \geq c$. Il se peut que dans la définition de Z, deux ou plusieurs contraintes soient définies pour la même variable (*contraintes redondantes*), par exemple $Z = v_i \leq c \wedge v_i \leq c' \wedge Z'$ ou encore $Z = v_i - v_j \leq c \wedge v_i - v_j \leq c' \wedge Z'$, avec $Z' \in \Phi(V)$. Dans ce cas, Z est

équivalent à $v_i \leq min(c, c') \wedge Z'$ ou encore Z est équivalent à $v_i - v_j \leq min(c, c') \wedge Z'$. De plus, remarquons que Z peut être écrit sous forme de contraintes *atomiques diaphantiennes*, comme l'illustre l'exemple suivant.

Exemple 6.5 *Supposons que $V = \{v_1, v_2\}$ et considérons le polyèdre $Z = (v_1 \geq 3) \wedge (v_2 \leq 5) \wedge (v_1 - v_2 \leq 4)$. Souvent, dans la définition de Z, on ne représente pas les contraintes $v_1 \geq 0$, $v_2 \geq 0$, elles sont sous entendues.*

$$\begin{cases} v_1 \geq 3 \\ v_2 \leq 5 \\ v_1 - v_2 \leq 4 \end{cases} \Rightarrow \begin{cases} 0 - v_1 \leq -3 \\ v_1 - 0 \leq +\infty \\ v_2 - 0 \leq 5 \\ 0 - v_2 \leq 0 \\ v_1 - v_2 \leq 4 \\ v_2 - v_1 \leq +\infty \end{cases} \Rightarrow \begin{cases} v_0 - v_1 \leq -3 \\ v_1 - v_0 \leq +\infty \\ v_2 - v_0 \leq 5 \\ v_0 - v_2 \leq 0 \\ v_1 - v_2 \leq 4 \\ v_2 - v_1 \leq +\infty \end{cases}$$

En effet, une contrainte de la forme $v_i \leq c$ peut être écrite sous la forme de $v_i - 0 \leq c$ ou encore $v_i - v_0 \leq c$ (par définition v_0 vaut zéro tout le temps). De plus, chaque variable v_i est inférieure à l'infinité positive. Ainsi, s'il n'existe pas de contraintes sur v_i de la forme $v_i \leq c$ alors on peut toujours ajouter la contrainte $v_i - v_0 \leq +\infty$ sans pour autant changer la portion d'espace définie par Z. Les mêmes remarques restent valables dans le cas d'une contrainte diagonale ($v_i - v_j \leq c$). En résumé, on peut toujours écrire un polyèdre en utilisant des contraintes atomiques diaphantiennes comme c'est le cas pour Z qui s'écrit sous la forme :

$$v_0 - v_1 \leq -3 \wedge v_1 - v_0 \leq +\infty \wedge v_2 - v_0 \leq 5 \wedge v_0 - v_2 \leq 0 \wedge v_1 - v_2 \leq 4 \wedge v_2 - v_1 \leq +\infty.$$

□

Sans perte de généralité et dans un souci de simplicité, nous supposons que les polyèdres considérés dans le reste de ce chapitre ne contiennent pas de contraintes redondantes ($v_i \leq c$, $v_i \leq c'$ ou $v_i - v_j \leq c$, $v_i - v_j \leq c$). De plus, nous supposons que chaque polyèdre $Z \in \Phi(V)$ s'écrit syntaxiquement sous la forme :

$$Z = \bigwedge_{v_i, v_j \in V_0, v_i \neq v_j} (v_i - v_j \leq l_{ij}), \quad l_{ij} \in \overline{\mathbb{R}}.$$

6.2.2 Forme canonique d'un polyèdre

Deux polyèdres ayant des conjonctions de contraintes différentes peuvent représenter la même portion d'espace. Dès lors, il est primordial de définir une *forme canonique* d'un polyèdre. Cette forme canonique est définie à travers le graphe de contraintes associé à Z.

Définition 26 (Graphe de contraintes.) *Le graphe de contraintes associé à* $Z = \bigwedge_{v_i,v_j \in V_0, v_i \neq v_j}(v_i - v_j \leq l_{ij})$ *est le graphe complet et étiqueté* $G(Z) = (V_0, \overline{\mathbb{R}}, E)$ *défini par :*

$$v_j \xrightarrow{l_{ij}} v_i \in E \Longleftrightarrow v_i - v_j \leq l_{ij} \text{ est un terme de } Z.$$

□

Le graphe de contraintes est obtenu à partir de Z de la façon suivante : chaque contrainte $v_i - v_j \leq l_{ij}$ est représentée par l'arc $v_j \xrightarrow{l_{ij}} v_i$ [1].

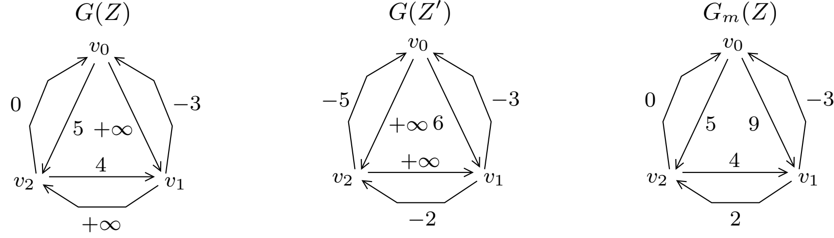


FIG. 32: Graphes de contraintes.

Exemple 6.6 *Prenons le polyèdre* $Z = (v_1 \geq 3) \wedge (v_2 \leq 5) \wedge (v_1 - v_2 \leq 4)$ *de l'exemple 6.5 qui s'écrit comme suivant :*

$$Z \;:\; v_0 - v_1 \leq -3 \wedge v_1 - v_0 \leq +\infty \wedge v_2 - v_0 \leq 5 \wedge$$
$$v_0 - v_2 \leq 0 \wedge v_1 - v_2 \leq 4 \wedge v_2 - v_1 \leq +\infty$$

Le graphe de contraintes associé à Z *est le graphe* $G(Z)$ *de la* FIG.*32.* □

Définition 27 (Polyèdre canonique) Z *est canonique si son graphe de contraintes* $G(Z)$ *est minimal.* □

Rappelons qu'un graphe est minimal (ou des courts plus chemins) si pour tous les noeuds v_i et v_j, le plus court chemin (en poids) de v_i à v_j est l'arête $v_i \rightarrow v_j$ [2]. Reprenons le polyèdre Z de l'exemple 6.6. Z n'est pas canonique vu que son graphe de contraintes $G(Z)$ (FIG.32) n'est pas minimal.

[1]Remarquons que l'on inverse l'ordre des variables dans le graphe.
[2]Voir la définition 25 de la section 6.1.1.

Maintenant, notons par $G_m(Z)$ le graphe minimal obtenu à partir de $G(Z)$ par la transformation $b2m()$ [3]. Le corollaire 1 découle de la définition 27 et de la définition de $G_m(Z)$.

Corollaire 1 *$Z = \bigwedge_{v_i,v_j \in V_0, v_i \neq v_j}(v_i - v_j \leq l_{ij})$ est canonique si et seulement si $v_j \xrightarrow{l_{ij}} v_i$ est une arête de $G_m(Z)$.* □

Le polyèdre *canonique* associé à Z est défini à travers le graphe minimal $G_m(Z)$ associé à au graphe de contraintes.

Définition 28 (Forme canonique) *Soient Z un polyèdre et $G_m(Z) = (V_0, \overline{\mathbb{R}}, E)$ le graphe de contraintes minimal de Z. La forme canonique de Z est le polyèdre $cf(Z)$ défini par :*

$$cf(Z) = \bigwedge_{\forall v_i,v_j \in V_0,\, v_i \neq v_j} (v_i - v_j \leq l_{ij}) \text{ tel que } v_j \xrightarrow{l_{ij}} v_i \in E.$$

□

Exemple 6.7 *Reprenons encore le cas où $Z = (v_1 \geq 3) \wedge (v_2 \leq 5) \wedge (v_1 - v_2 \leq 4)$. A partir de son graphe minimal $G_m(Z)$, donné dans la* FIG.*32, on déduit que :*

$$cf(Z) = (v_2 - v_0 \leq 5) \wedge (v_0 - v_2 \leq 0) \wedge (v_1 - v_0 \leq 9) \wedge$$
$$(v_0 - v_1 \leq -3) \wedge (v_1 - v_2 \leq 4) \wedge (v_2 - v_1 \leq 2).$$

□

Corollaire 2 *Soient Z et Z deux polyèdres. Alors [4] :*

1. *$Z \sim cf(Z)$.*
2. *$Z \sim Z'$ si et seulement si $cf(Z) = cf(Z')$ si et seulement si $G_m(Z) = G_m(Z')$.*
3. *$Z \subseteq Z'$ si et seulement si $cf(Z) \subseteq cf(Z')$.* □

Intuitivement, pour le premier point du corollaire, Z et sa forme canonique représentent la même portion d'espace. Pour le deuxième point, Z et Z' représentent la même portion de l'espace ssi ils ont le même graphe minimal ssi ils ont la même forme canonique. Finalement, l'espace induit par Z est inclus dans l'espace induit par Z' ssi $cf(Z)$ est inclus dans $cf(Z')$. Pour la preuve, il suffit de voir que $\sim$ est une équivalence.

[3]Rappelons que la transformation $b2m()$ calcule le graphe minimal d'un graphe donné, voir la section 6.1.2.

[4]Pour la définition de $\sim$, voir la section 2.2.

Théorème 1 (Floyd) $Z \not\leadsto$ **false** *si et seulement si* $G(Z)$ *est positif.* □

Intuitivement, un polyèdre Z ne représente pas une portion vide de l'espace ssi son graphe de contraintes ne contient pas de cycle négatif. Rappelons que selon le lemme 3 de la section 6.1.2, $G(Z)$ est positif si et seulement si $G_m(Z)$ est positif.

Preuve. La preuve est donnée dans [58]. □

Exemple 6.8 *Le polyèdre* Z *défini par :*

$$Z = (v_1 \geq 3) \wedge (v_2 \leq 5) \wedge (v_1 - v_2 \leq 4)$$

n'est pas vide car $G(Z)$ *est positif (voir* FIG.*32). Cependant,*

$$Z' = v_1 \geq 3 \wedge v_1 \leq 6 \wedge v_2 \geq 5 \wedge v_1 - v_2 \geq 2$$

est vide. Pour cela, il suffit de remarquer que le cycle $v_0 \xrightarrow{6} v_1 \xrightarrow{-2} v_2 \xrightarrow{-5} v_0$ *dans son graphe de contraintes* $G(Z')$*, donné dans la* FIG.*32, est négatif.* □

6.2.3 Valuation de bornes

Dans la section 6.2.2, nous avons défini la forme canonique d'un polyèdre à travers son graphe de contraintes (graphe complet) et dans la section 6.1, nous avons introduit quelques transformations sur les graphes complets. Ces résultats nous permettent de formuler le théorème suivant qui identifie les valuations de bornes.

Théorème 2 *Supposons que* $Z = \bigwedge_{v_i, v_j \in V_0,\, v_i \neq v_j} (v_i - v_j \leq l_{ij})$ *est canonique, borné et non vide* ($Z \not\leadsto false$)*. Alors, pour tout* $k \in [0, n]$ *:*

1. *Il existe une valuation* $\nu_k^M(Z)$ *de* Z *telle que : pour tout* $i \in [0, n]$*,* $i \neq k$*,* $\nu_k^M(Z)(v_i) - \nu_k^M(Z)(v_k) = l_{ik}$.
2. *Il existe une valuation* $\nu_k^m(Z)$ *de* Z *telle que : pour tout* $i \in [0, n]$*,* $i \neq k$*,* $\nu_k^m(Z)(v_k) - \nu_k^m(Z)(v_i) = l_{ki}$. □

Intuitivement, si Z est canonique, borné et différent de l'ensemble vide, alors pour toute variable $v_k \in V_0$, il existe une valuation $\nu_k^M(Z)$ (resp. $\nu_k^m(Z)$) qui atteint les bornes $(l_{ik})_{k \neq i, i \in [0,n]}$ (resp. $(l_{ki})_{k \neq i, i \in [0,n]}$) des contraintes diaphantiennes de Z, dont v_k est le membre droit (resp. gauche). En d'autres mots, pour tout $k \in [0, n]$, les polyèdres :

$$Z_k^M = \bigwedge_{v_i V_0,\, v_i \neq v_k} (v_i - v_k = l_{ik}) \wedge \bigwedge_{v_i, v_j \in V_0,\, v_i \neq v_j \neq v_k} (v_i - v_j \leq l_{ij})$$

et

$$Z_k^m = \bigwedge_{v_i V_0,\, v_i \neq v_k} (v_k - v_i = l_{ki}) \wedge \bigwedge_{v_i, v_j \in V_0,\, v_i \neq v_j \neq v_k} (v_i - v_j \leq l_{ij})$$

ou encore

$$Z_k^M = \bigwedge_{v_i V_0,\, v_i \neq v_k} (v_i - v_k \leq l_{ik} \wedge v_k - v_i \leq -l_{ik}) \wedge \bigwedge_{v_i, v_j \in V_0,\, v_i \neq v_j \neq v_k} (v_i - v_j \leq l_{ij})$$

et

$$Z_k^m = \bigwedge_{v_i V_0,\, v_i \neq v_k} (v_k - v_i \leq l_{ki} \wedge v_i - v_k \leq -l_{ki}) \wedge \bigwedge_{v_i, v_j \in V_0,\, v_i \neq v_j \neq v_k} (v_i - v_j \leq l_{ij})$$

ne sont pas vides ($Z_k^M \not\sim false$ et $Z_k^m \not\sim false$).

Remarque 8 *Nous avons supposé que Z est borné. Ceci est dans le but de garantir l'existence de $\nu_k^M(Z)$. Les $\nu_k^m(Z)$ existe même si Z n'est pas borné du fait que les variables de V sont à valeurs dans $\mathbb{R}^{\geq 0}$.* □

Preuve. Soient $G(Z)$ le graphe de contraintes de Z et $k \in [0, n]$. Z est canonique donc $G(Z)$ est minimal. Z est non vide donc, selon le théorème 1, $G(Z)$ est positif. Maintenant, il suffit de voir que le graphe de contraintes $G(Z_k^M)$ (resp. $G(Z_k^m)$) de Z_k^M (resp. Z_k^m) n'est autre que le graphe obtenu à partir de $G(Z)$ par la transformation $R_{k \to *}()$ (resp.$R_{* \to k}()$) définie dans la section 6.1.3 (resp. la section 6.1.4) : $G(Z_k^M) = R_{k \to *}(G(Z))$ et $G(Z_k^m) = R_{* \to k}(G(Z))$. Ainsi, d'après le lemme 4 (resp. le lemme 6), on déduit que $G(Z_k^M)$ (resp. $G(Z_k^m)$) est positif et par conséquent $Z_k^M = \not\sim false$ (resp. $Z_k^m = \not\sim false$). De plus, le lemme 5 (resp. le lemme 7) donne une méthode pour calculer la forme canonique de Z_k^M (resp. Z_k^m). □

Calcul de $\nu_k^M(Z)$ et $\nu_k^m(Z)$

Le théorème 2 établit en même temps l'existence de $\nu_k^M(Z)$ et $\nu_k^m(Z)$ et leur unicité. En effet, soit $k \in [0, n]$. D'après le premier point de ce théorème,

- Il existe une valuation $\nu_k^M(Z)$ de Z telle que : pour tout $i \in [0, n]$, $i \neq k$, $\nu_k^M(Z)(v_i) - \nu_k^M(Z)(v_k) = l_{ik}$.

Comme $v_0 = 0$ donc $\nu(v_0) = 0$ pour toute valuation de V. Selon k on peut avoir l'un des cas suivants :

1. Si k=0, alors

   pour tout $i \in [1, n]$,
   $$\nu_k^M(Z)(v_i) - \nu_k^M(Z)(v_k) = l_{ik}$$
   est équivalent à
   $$\nu_0^M(Z)(v_i) - \nu_0^M(Z)(v_0) = l_{i0}$$
   est équivalent à
   $$\nu_0^M(Z)(v_i) - 0 = l_{i0}$$
   ou encore
   $$\nu_0^M(Z)(v_i) = l_{i0}$$

2. Sinon,

   pour $i = 0$,
   $$\nu_k^M(Z)(v_i) - \nu_k^M(Z)(v_k) = l_{ik}$$
   est équivalent à
   $$\nu_k^M(Z)(v_0) - \nu_k^M(Z)(v_k) = l_{0k}$$
   est équivalent à
   $$0 - \nu_k^M(Z)(v_k) = l_{0k}$$
   ou encore
   $$\nu_k^M(Z)(v_k) = -l_{0k}$$
   Ainsi, on a calculé la valeur $\nu_k^M(Z)(v_k)$. Par conséquent,

   pour tout $i \in [0, n]$, $i \neq k$,
   $$\nu_k^M(Z)(v_i) - \nu_k^M(Z)(v_k) = l_{ik}$$
   est équivalent à
   $$\nu_k^M(Z)(v_i) = \nu_k^M(Z)(v_k) + l_{ik}$$
   ou encore
   $$\nu_k^M(Z)(v_i) = -l_{0k} + l_{ik}$$

D'où l'unicité de $\nu_k^M(Z)$. D'une façon similaire, on obtient l'unicité de $\nu_k^m(Z)$. L'unicité de l'existence de ces valuations est prouvée formellement dans les lemmes 5 et 7.

Définition 29 *Les $(\nu_k^M(Z))_{k\in[0,n]}$ et les $(\nu_k^m(Z))_{k\in[0,n]}$ sont appelées les valuations de bornes. Les $(\nu_k^M(Z))_{k\in[0,n]}$ (resp. $(\nu_k^m(Z))_{k\in[0,n]}$) sont appelées les valuations de bornes maximales (resp. minimales).* □

Complexité des calculs

Tout d'abord, le nombre des valuations de bornes varie entre 1 et $2 \times (n+1)$ valuations (n est le nombre des variables de V). D'après ce qui précède, si Z est canonique, le calcul de $\nu_k^M(Z)$ (resp. $\nu_k^m(Z)$), pour $k \in [0, n]$, se fait en temps $O(n)$.

Exemple 6.9 *Soit $Z = (v_1 \geq 3) \wedge (v_2 \leq 5) \wedge (v_1 - v_2 \leq 4)$. Z est borné. Sa forme canonique est $cf(Z) = (v_2 - v_0 \leq 5) \wedge (v_0 - v_2 \leq 0) \wedge (v_1 - v_0 \leq 9) \wedge (v_0 - v_1 \leq -3) \wedge (v_1 - v_2 \leq 4) \wedge (v_2 - v_1 \leq 2)$. Une application directe du théorème donne :*

1. *Il existe $\nu_0^M(Z) \in Z$, tel que $\nu_0^M(Z)(v_1) - \nu_0^M(Z)(v_0) = 9$ et $\nu_0^M(Z)(v_2) - \nu_0^M(Z)(v_0) = 5$*
2. *Il existe $\nu_0^m(Z) \in Z$, tel que $\nu_0^m(Z)(v_0) - \nu_0^m(Z)(v_1) = -3$ et $\nu_0^m(Z)(v_0) - \nu_0^m(Z)(v_2) = 0$.*
3. *Il existe $\nu_1^M(Z) \in Z$, tel que $\nu_1^M(Z)(v_0) - \nu_1^M(Z)(v_1) = -3$ et $\nu_1^M(Z)(v_2) - \nu_1^M(Z)(v_1) = 2$*
4. *Il existe $\nu_1^m(Z) \in Z$, tel que $\nu_1^m(Z)(v_1) - \nu_1^m(Z)(v_0) = 9$ et $\nu_1^m(Z)(v_1) - \nu_1^m(Z)(v_2) = 4$*
5. *Il existe $\nu_2^M(Z) \in Z$, tel que $\nu_2^M(Z)(v_0) - \nu_2^M(Z)(v_2) = 0$ et $\nu_2^M(Z)(v_1) - \nu_2^M(Z)(v_2) = 4$.*
6. *Il existe $\nu_2^M(Z) \in Z$, tel que $\nu_2^M(Z)(v_2) - \nu_2^M(Z)(v_0) = 5$ et $\nu_2^M(Z)(v_2) - \nu_2^M(Z)(v_1) = 2$.*

ou encore,

$$\nu_0^M(Z) = \begin{cases} v_1 - v_0 = 9 \\ v_2 - v_0 = 5 \end{cases} \quad \nu_0^m(Z) = \begin{cases} v_0 - v_1 = -3 \\ v_0 - v_2 = 0 \end{cases} \quad \nu_1^M(Z) = \begin{cases} v_0 - v_1 = -3 \\ v_2 - v_1 = 2 \end{cases}$$

$$\nu_1^m(Z) = \begin{cases} v_1 - v_0 = 9 \\ v_1 - v_2 = 4 \end{cases} \quad \nu_2^M(Z) = \begin{cases} v_0 - v_2 = 0 \\ v_1 - v_2 = 4 \end{cases} \quad \nu_2^m(Z) = \begin{cases} v_2 - v_0 = 5 \\ v_2 - v_1 = 2 \end{cases}$$

Par conséquent, on obtient les valuations suivantes de Z en listant les valeurs de v_1 et v_2, respectivement :

$$\nu_0^M(Z) = \begin{pmatrix} 9 \\ 5 \end{pmatrix} \begin{matrix} v_1 \\ v_2 \end{matrix} \quad \nu_1^M(Z) = \begin{pmatrix} 3 \\ 5 \end{pmatrix} \quad \nu_2^M(Z) = \begin{pmatrix} 4 \\ 0 \end{pmatrix}$$
$$\nu_0^m(Z) = \begin{pmatrix} 3 \\ 0 \end{pmatrix} \quad \nu_1^m(Z) = \begin{pmatrix} 9 \\ 5 \end{pmatrix} \quad \nu_2^m(Z) = \begin{pmatrix} 3 \\ 5 \end{pmatrix}$$

□

6.2.4 k-complétude d'une valuation

Définition 30 *Une k-incomplète valuation ν est une valuation sur $V_k = \{v_1, \cdots, v_k\}$. On dit que ν peut être complétée dans Z s'il existe une valuation v' sur V de Z telle que $\nu'(v_j) = \nu(v_j)$, pour tout $j \leq k$.* □

Notons que lorsque $k = 0$, compléter une 0-incomplète valuation revient à extraire une valuation de Z. Ce cas a été étudié dans la section précédente. Rappelons que n est le cardinal de V.

Lemme 8 *Supposons que Z est canonique et soit ν une k-incomplète valuation. La complexité en temps de compléter ou non ν dans Z est de $O(n^2)$.* □

Preuve. L'idée de la démonstration est la suivante. Supposons que $Z = \bigwedge_{v_i, v_j \in V_0,\, v_i \neq v_j} (v_i - v_j \leq l_{ij})$. Alors Z peut être écrit sous la forme

$$Z = \bigwedge_{v_i \in V_k} (v_i - v_0 \leq l_{i0} \wedge v_0 - v_i \leq l_{0i}) \wedge Z'$$

avec $Z' \in \Phi(V)$. Si maintenant nous considérons

$$Z_k = \bigwedge_{v_i \in V_k} (v_i - v_0 \leq \nu(v_i) \wedge v_0 - v_i \leq -\nu(v_i)) \wedge Z'$$

alors, ν peut être complétée dans Z si et seulement si Z_k n'est pas vide ($Z_k \not\sim false$). Par application directe du théorème 2 à $cf(Z_k)$ (dans le cas ou $Z_k \not\sim false$), nous aurons entre 1 et $(n+1)$ valuations qui complètent ν dans Z si Z_k n'est pas borné et entre 1 et $2 \times (n+1)$ valuations de Z dans le cas contraire. La complexité en temps de $O(n^2)$ vient du calcul de $cf(Z)$ à partir de Z. Pour plus de détails sur la preuve se reporter à l'Annexe A.□

6.2.5 Conclusion

Le tableau suivant résume les différentes contributions de ce chapitre relatives à l'extraction et la complétude d'une valuation.

Si $Z = \bigwedge_{v_i,v_j \in V_0,\, v_i \neq v_j}(v_i - v_j \leq l_{ij})$ est canonique, borné et non vide, alors pour tout $k \in [0, n]$,

1. La valuation $\nu_k^M(Z)$ définie par :
   - Si $k = 0$ alors $\nu_k^M(Z)(v_i) = l_{i0}$.
   - Sinon $\nu_k^M(Z)(v_i) = -l_{0k} + l_{ik}$, et $\nu_k^M(Z)(v_k) = -l_{0k}$

   pour tout $i \in [1, n]$, $i \neq k$, appartient à Z.
2. La valuation $\nu_k^m(Z)$ définie par :
   - Si $k = 0$ alors $\nu_k^m(Z)(v_i) = -l_{0i}$.
   - Sinon $\nu_k^m(Z)(v_i) = l_{k0} - l_{ki}$, et $\nu_k^m(Z)(v_k) = l_{k0}$

   pour tout $i \in [1, n]$, $i \neq k$, appartient à Z.

Le nombre total de ces valuations varie entre 1 et $2 \times (n + 1)$ valuations, (n est le nombre de variables de V).
Le calcul d'une valuation de bornes se fait en $O(n)$.
Finalement, compléter une k-incomplète valuation dans Z peut se faire en $O(n^2)$.

Il reste dans ce cas le calcul de la forme canonique d'un polyèdre. Ce calcul dépend des structures de données utilisées. Il est présenté dans la section 12.1.3.

6.2.6 Ce que nous allons faire

Dans ce chapitre préliminaire, nous avons proposé, d'une part une méthode en temps $O(n)$ qui permet d'extraire les valuations de bornes à partir d'un polyèdre canonique, et d'autre part une méthode en temps $O(n^2)$ pour compléter une k-incomplète valuation. Dans le chapitre suivant, nous montrons comment exploiter les résultats de ce chapitre pour extraire des diagnostics temporisés.

Chapitre 7

Diagnostic temporisé

Sommaire

7.1 Diagnostic temporisé basé sur le polyèdre de contraintes . . . 101
7.1.1 Diagnostic temporisé de bornes 101
7.1.2 Inclusion des traces 103
7.1.3 Conclusion 104
7.2 Diagnostic temporisé basé sur une analyse symbolique 104
7.2.1 État symbolique 104
7.2.2 Analyse symbolique d'un chemin 107
7.2.3 k-complétude d'une computation 110
7.2.4 Diagnostic temporisé de bornes 112
7.2.5 Inclusion des traces 113
7.3 Comparaison avec des travaux similaires 113
7.3.1 Travaux d'Alur et al. [6] 113
7.3.2 Travaux de Tripakis [145, 146] 113
7.3.3 Autres travaux 114
7.3.4 Nos contributions 114

Dans le reste de ce chapitre, $\rho = t_1 \cdots t_n$ (resp. $\rho' = t'_1 \cdots t'_n$) est un chemin de taille $n \in \mathbb{N}$ d'un automate temporisé A (resp. B) :

$$\rho \,:\, s_0 \xrightarrow{Z_1,a_1,r_1} s_1 \xrightarrow{Z_2,a_2,r_2} s_2 \cdots \xrightarrow{Z_n,a_n,r_n} s_n$$

$$\rho' \,:\, s'_0 \xrightarrow{Z'_1,a'_1,r'_1} s'_1 \xrightarrow{Z'_2,a'_2,r'_2} s'_2 \cdots \xrightarrow{Z'_n,a'_n,r'_n} s'_n$$

tels que

$$\text{pour tout } i \in [1,n],\ a_i = a'_i$$

A_ρ (reps. $B_{\rho'}$) dénotera l'automate temporisé induit par le chemin ρ (resp. ρ').

Dans ce chapitre, nous apportons des réponses aux interrogations suivantes :

1. Comment générer des diagnostics temporisés pour un chemin ρ ?
2. Comment montrer l'inclusion des traces de longueur n entre deux chemins temporisés ρ et ρ' sachant que seules les traces de ρ' sont connues ?

Nous allons procéder de la manière suivante. Comme nous l'avons fait pour les polyèdres, nous identifions, selon deux approches, les *diagnostics de bornes* pour les chemins. Dans la première approche, à un chemin ρ, nous associons un polyèdre de *contraintes* qui définit l'ensemble des contraintes que doivent satisfaire une séquence temporisée pour admettre une computation de A_ρ. Dès lors, les diagnostics de bornes correspondent aux valuations de bornes du polyèdre de contraintes. La deuxième approche est basée sur une analyse symbolique en avant et en arrière. Des opérateurs *post*() et *pred*() seront introduits.

7.1 Diagnostic temporisé basé sur le polyèdre de contraintes

7.1.1 Diagnostic temporisé de bornes

Soit $\sigma = (a_1, d_1) \cdots (a_n, d_n) \in TTrace(A_\rho, n)$. D'après la définition de $TTrace(A_\rho, n)$, les différents instants $(d_i)_{i \in [1,n]}$ vérifient un ensemble de contraintes relatives aux transitions de A_ρ comme nous le montrons dans l'exemple suivant.

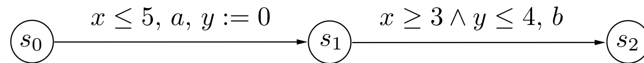


FIG. 33: Automate A_ρ.

Exemple 7.1 *Considérons l'automate de la* FIG. *33. Notons par v_i, $i \in [1,2]$ l'instant de franchissement, selon une horloge universelle h, de la transition t_i reliant l'état s_{i-1} à l'état s_i. Alors, l'état s_2 est atteignable si et seulement si le système de contraintes S,*

$$S = \begin{cases} 0 \leq v_1 \leq v_2 \\ v_1 \leq 5 \\ v_2 \geq 3 \\ v_2 - v_1 \leq 4 \end{cases}$$

admet une solution. Ceci est équivalent à dire que le polyèdre Z défini par :

$$Z = 0 \leq v_1 \wedge v_1 \leq v_2 \wedge v_1 \leq 5 \wedge v_2 \geq 3 \wedge v_2 - v_1 \leq 4$$

n'est pas vide. Par conséquent, $\sigma = (a, d_1).(b, d_2) \in TTrace(A_\rho, 2)$ *si seulement si la valuation* ν *définie par* $\nu(v_1) = d_1$, $\nu(v_2) = d_2$ *appartient à* Z. □

D'une façon générale, on peut associer à A_ρ un *polyèdre de contraintes* Z_ρ sur les variables $V = \{v_1, \cdots, v_n\}$ tel que : $\sigma \in TTrace(A_\rho, n)$ si et seulement si la valuation $\nu \in \mathcal{V}(V)$ définie par $\nu(v_i) = time(\sigma_i)$, pour tout $\forall i \in [1, n]$, appartient à Z_ρ. Nous allons proposer un algorithme calculant ce polyèdre de contraintes associé à A_ρ.ss

Polyèdre de contraintes

Soit $last_i^x$ l'indice de la transition inférieure à i ($i \in [0, n]$) où l'horloge x a été remise à zéro pour la dernière fois. Rappelons que $\bowtie \in \{\leq, \geq\}$ et $V_0 = \{v_0, \cdots, v_n\}$ et que toutes les horloges sont mises à zéro dans l'état initial de A_ρ. Par exemple, dans la FIG. 33, $last_2^y = 1$ et $last_1^x = last_1^y = last_2^x = 0$. La construction du polyèdre de contraintes Z_ρ est illustrée dans la FIG. 34.

Input : Un automate (filiforme) $A_\rho = t_1 \cdots t_n$ de taille n
Output : Le polyèdre de contraintes $Z_\rho \in \Phi(V)$
Début

/* Initialisation */
1. $Z_\rho := true$

/* $(v_i)_{i \in [0,n]}$ est croissante */
2. **Pour** i de 1 à n **Faire**
3. $Z_\rho := Z_\rho \wedge v_{i-1} \leq v_i$

4. **Pour** $x \in C$ **Faire**
5. **Si** la garde de t_i contient une contrainte de la forme $x \bowtie k$ **Alors**
6. $Z_\rho := Z_\rho \wedge v_i - v_j \bowtie k$ avec $j = last_i^x$.

7. **Si** la garde de t_i contient une contrainte de la forme $x - y \bowtie k$ **Alors**
8. $Z_\rho := Z_\rho \wedge v_p - v_q \bowtie k$ avec $q = last_i^x$ et $p = last_i^y$.

Fin

FIG. 34: Polyèdre de contraintes

Pour tout $i \in [0, n]$, v_i représente l'instant de franchissement de la transition t_i selon une horloge universelle. Par convention, nous supposons qu'il existe une transition t_0 où toutes les horloges sont mises à zéro en même temps à l'instant $v_0 = 0$. Le polyèdre de contraintes est initialisé à **true** (ligne 1). Les instants de franchissement v_i forment une suite croissante.

Ils sont ajoutés à Z_ρ (lignes 2 et 3). Dans l'étape $i \in [1,n]$ de l'algorithme, si la garde de la transition t_i contient une contrainte sur x de la forme $x \bowtie k$ (ligne 5) alors on ajoute à Z_ρ la contrainte $v_i - v_j \bowtie k$ tel que x est remise à zéro pour la dernière fois dans j (ligne 6). Par ailleurs, si la garde de la transition t_i contient une contrainte de la forme $x - y \bowtie k$ (ligne 7) alors on ajoute à Z_ρ la contrainte $v_p - v_q \bowtie k$ tel que $last_i^x = q$ et $last_i^y = p$ (ligne 8). En effet, le temps écoulé depuis la dernière remise à zéro de x (resp. y) dans la transition t_q (resp. t_p) est égal à $v_i - v_q$ (resp. $v_i - v_p$). Donc $x - y = (v_i - v_q) - (v_i - v_p) = v_p - v_q$.

Diagnostic temporisé de bornes

Comme nous venons de voir, il est possible d'associer à A_ρ un polyèdre de contraintes Z_ρ. Le problème de l'extraction d'un diagnostic temporisé pour ρ revient donc à extraire des valuations de Z_ρ. Soient $(\nu_i^M(cf(Z_\rho)))_{i\in[0,n]}$ et $(\nu_i^m(cf(Z_\rho)))_{i\in[0,n]}$ les valuations de bornes de la forme canonique de Z_ρ données par le théorème 2 de la section 6.2.3. Considérons les deux suites de séquences temporisées $(\sigma^{Mi})_{i\in[0,n]}$ et $(\sigma^{mi})_{i\in[0,n]}$ définies par :

- $\sigma^{Mi} = (a_1, \nu_i^M(cf(Z_\rho))(v_1)) \cdots (a_n, \nu_i^M(cf(Z_\rho))(v_n))$.
- $\sigma^{mi} = (a_1, \nu_i^m(cf(Z_\rho))(v_1)) \cdots (a_n, \nu_i^m(cf(Z_\rho))(v_n))$.

On remarque que, pour tout $i \in [0,n]$, $\sigma^{Mi} \in TTrace(A_\rho, n)$ et $\sigma^{mi} \in TTrace(A_\rho, n)$.

Définition 31 *Les séquences temporisées $(\sigma^{Mi})_{i\in[0,n]}$ et $(\sigma^{mi})_{i\in[0,n]}$ sont appelées des diagnostics temporisés de bornes associés au chemin ρ. σ^{Mi} (resp. σ^{mi}) est appelée diagnostic temporisé de bornes maximales (resp. minimales).* □

7.1.2 Inclusion des traces

D'une façon générale, montrer que $TTrace(A,n) \subseteq TTrace(B,n)$ est équivalent à montrer que $cf(Z_\rho) \subseteq cf(Z_{\rho'})$ et se fait en temps $O(n^2)$ [1]. Nous considérons ici le cas où Z_ρ est connu et seulement $TTrace(B_{\rho'}, n)$ est connu. Nous supposons que Z_ρ est borné et non vide.

Soient $(\sigma^{Mi})_{i\in[0,n]}$ et $(\sigma^{mi})_{i\in[0,n]}$ les diagnostics temporisés de bornes et $(\nu_i^M(cf(Z_\rho)))_{i\in[0,n]}$ et $(\nu_i^m(cf(Z_\rho)))_{i\in[0,n]}$ les valuations de bornes associées au chemin ρ.

Corollaire 3 *$TTrace(A_\rho, n) \subseteq TTrace(B_{\rho'}, n)$ si et seulement si pour tout $i \in [0,n]$, $\sigma^{Mi} \in TTrace(B_{\rho'}, n)$ et $\sigma^{mi} \in TTrace(B_{\rho'}, n)$.* □

Intuitivement, le corollaire donne les conditions nécessaires et suffisantes pour montrer $TTrace(A_\rho, n) \subseteq TTrace(B_{\rho'}, n)$. En effet, il suffit de montrer que les traces σ^{Mi} et σ^{mi} de $TTrace(A_\rho, n)$ sont aussi des traces de $TTrace(B_{\rho'}, n)$.

[1] Pour cela, il suffit de comparer les bornes des contraintes, voir la section 12.1.3.

Preuve. La démonstration découle du fait que $TTrace(A_\rho, n) \subseteq TTrace(B_{\rho'}, n)$ ssi $Z_\rho \subseteq Z_{\rho'}$. Comme $(\nu_i^M(Z_\rho))_{i\in[0,n]}$ et $(\nu_i^m(Z_\rho))_{i\in[0,n]}$ atteignent toutes les bornes de Z_ρ donc si $\nu_i^M(Z_\rho) \in Z_{\rho'}$ et $\nu_i^m(Z_\rho) \in Z_{\rho'}$ alors toutes les bornes de Z_ρ sont inférieures aux bornes de $Z_{\rho'}$. La densité et la connexité des ensembles Z_ρ et $Z_{\rho'}$ impliquent que toute $\nu \in Z_\rho$ est aussi dans $Z_{\rho'}$. □

7.1.3 Conclusion

Cette section peut être vue comme une application directe des résultats de la section 6.2. En effet, la solution apportée à l'extraction des diagnostics temporisés est basée sur le calcul du polyèdre de contraintes Z_ρ associé à un chemin, ce qui réduit cette extraction à une extraction de valuations dans Z_ρ. Nous avons aussi étudié le problème d'inclusion des traces entre deux chemins ρ et ρ' dans le cas où seules les traces de ρ' sont connues. Pour ce dernier point, la solution apportée se base sur les valuations de bornes associées à ρ qui permettent de réduire le problème d'inclusion à l'inclusion des valuations de bornes dans les valuations de ρ'. Ce dernier résultat, comme nous l'avons cité auparavant, sera exploité pour minimiser le nombre de cas de test temporisés considéré pour montrer l'inclusion des traces de l'implantation dans celles de la spécification. Dans la section 7.2, une autre solution basée sur une analyse en avant et une analyse en arrière sera développée. L'intérêt de l'analyse symbolique est le coût réduit de l'implantation des différentes opérations.

7.2 Diagnostic temporisé basé sur une analyse symbolique

Nous considérons dans cette section le problème d'accessibilité d'un état, à savoir, étant donné un état du système, existe-il une computation de l'état initial qui atteint cet état. A travers cette problématique, nous apporterons une autre solution, aux problèmes d'extraction de diagnostics temporisés et d'inclusion de traces, basée sur la définition de deux opérateurs, un opérateur *post*() de calcul des états *successeurs* d'un état *symbolique* donné et un opérateur *pred*() faisant l'opération inverse.

7.2.1 État symbolique

Avant de présenter notre approche, nous introduirons quelques notions de base. Considérons un TA $A = (S, s_0, L, C, \rightarrow)$.

Définition 32 (État symbolique) *Un état symbolique de* A *est un ensemble* $H = \{(s, \nu) \| \ \nu \in Z\}$ *avec* $s \in S$ *un état de* A*,* $Z \in \Phi(C)$ *un polyèdre et* $\nu \in \mathcal{V}(C)$ *une*

valuation. □

Pour simplifier, H sera simplement noté (s, Z) et sera appelé une *zone*. Les opérations d'inclusion et d'intersection sont étendues aux zones de la manière suivante. Soient $H = (s, Z)$ et $H' = (s', Z')$ deux zones. Alors :

1. $H \subseteq H'$ si et seulement si $s = s'$ et $Z \subseteq Z'$
2. $H \cap H' = \emptyset$ si $s \neq s'$ ou $Z \cap Z' = \emptyset$. Sinon $H \cap H' = (s, Z \cap Z')$.

Successeur temporel

Le premier opérateur que nous définissons est le *successeur temporel* d'une zone $H = (s, Z)$ par une transition $t = (s, Z', a, r, s')$ de A :

$$\mathbf{post}(H, t) = (s', (Z^{\uparrow} \cap Z')[r := 0])$$

Intuitivement, *post*() contient tous les états atteignables à partir des états de H, en laissant d'abord le temps s'écouler dans l'état s (la partie $Z^{\uparrow}$) tout en respectant la contrainte Z' de t (la partie $Z^{\uparrow} \cap Z'$), puis en faisant une transition discrète sur a pour atteindre s' (la partie $(.)[r := 0]$), comme l'illustre la FIG.35.

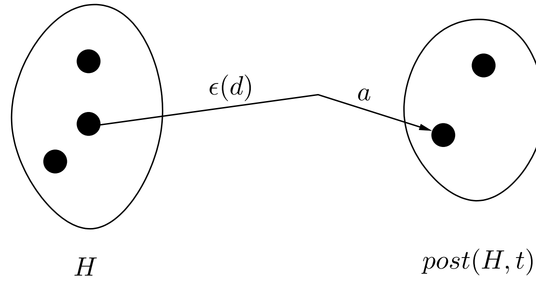


FIG. 35: Successeur temporel.

Exemple 7.2 *A titre d'exemple, si* $H = (s, 1 \leq x \leq 3)$, $t = s \xrightarrow{x=1,a,x:=0} s'$ *et* $t' = s \xrightarrow{x\leq 1,a} s'$, *alors* $post(H, t) = (s', x = 0)$ *et* $post(H, t') = (s', x = 1)$. □

Le corollaire suivant découle de la définition de *post*().

Corollaire 4 *Si H est une zone alors $post(H, t)$ est aussi une zone.* □

Prédécesseur temporel

D'une façon similaire, nous définissons le *prédécesseur temporel* d'une zone $H = (s, Z)$ par une transition $t = (s', Z', a, r, s)$:

$$\mathbf{pred}(H, t) = (s', ([r := 0]Z \cap Z')^{\downarrow})$$

Intuitivement, $pred()$ contient tous les états qui peuvent atteindre H, en laissant le temps s'écouler dans l'état s', tout en respectant la contrainte de Z' de t, puis en faisant la transition discrète sur a pour atteindre s, comme l'illustre la FIG.36.

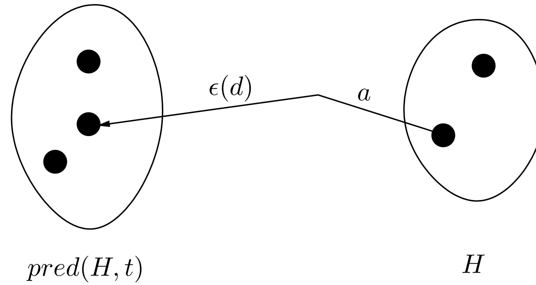


FIG. 36: Prédécesseur temporel.

Le corollaire suivant découle de la définition de $pred()$.

Corollaire 5 *Si H est une zone alors $pred(H, t)$ est aussi une zone.* □

Par convention, si $H = (s, false)$ alors H est dite vide et dans ce cas $post()$ et $pred()$ sont aussi vides.

post/pred-stabilité

Soient $H = (s, Z)$ une zone et $t_1 = (s_1, Z_1, a_1, r_1, s)$ et $t_2 = (s, Z_2, a_2, r_2, s_2)$ deux transitions. Supposons qu'il existe H_1 et H_2 tels que $H = post(H_1, t_1)$ et $H = pred(H_2, t_2)$. On a alors les propriétés suivantes :

- *pred-stabilité de post* : pour tout $q = (s, \nu) \in H$, il existe $q_1 = (s_1, \nu_1) \in H_1$ et $d_1 \geq 0$ tels que $\nu_1 + d_1 \in Z_1$ et $\nu = (\nu_1 + d_1)[r_1 := 0]$. H est dite *pred-stable* H_1 par t_1. q_1 est dit *état prédécesseur* de q dans H_1.
- *post-stabilité de pred* : pour tout $q = (s, \nu) \in H$, il existe $q_2 = (s_2, \nu_2) \in H_2$ et $d_2 \geq 0$ tels que $\nu + d_2 \in Z_2$ et $(\nu + d_2)[r_2 := 0] = \nu_2$. H est dite *post-stable* H_2 par t_2. q_2 est dit *état successeur* de q dans H_2.

7.2.2 Analyse symbolique d'un chemin

Supposons que le chemin ρ s'écrit comme $\rho : t_1 \cdots t_n$, avec $t_i = (s_{i-1}, Z_i, a_i, r_i, s_i)$, pour tout $i \in [1, n]$ et C l'ensemble des horloges de ρ.

Analyse en avant

Pour tout $i \in [0, n]$, posons :

$$\begin{cases} H_i = (s_i, zero) & si\ i = 0 \\ H_i = post(H_{i-1}, t_i) & sinon \end{cases}$$

H_0 est définie par le polyèdre *zero* ($\forall \nu \in zero, \forall x \in C, \nu(x) = 0$) et H_i est pred-stable H_{i-1} par t_i.

Exemple 7.3 *Considérons le chemin suivant :*

$$\rho : s_0 \xrightarrow{x \leq 2, a, y:=0} s_1 \xrightarrow{y \leq 1, b, x:=0} s_2.$$

Alors,

$$\begin{cases} H_0 = (s_0, x = y = 0) \\ H_1 = post(H_0, t_1) = (s_1, x \leq 2 \wedge y = 0) \\ H_2 = post(H_1, t_2) = (s_2, x = 0 \wedge y \leq 1) \end{cases}$$

On remarque que l'état s_2 est atteignable, i.e. $TTrace(A_\rho, 2)$ n'est pas vide. □

Le corollaire suivant découle de la définition de $post()$.

Corollaire 6 *L'état final s_n de ρ est atteignable si et seulement si H_n n'est pas vide.* □

Au chemin ρ, il est possible d'associer un chemin symbolique $S^+(\rho)$ dont les états sont les zones H_i. $S^+(\rho)$ est défini comme suivant :

$$S^+(\rho) : H_0 \xrightarrow{a_1} H_1 \cdots H_{n-1} \xrightarrow{a_n} H_n$$

Une transition $H_{i-1} \xrightarrow{a_i} H_i$ de $S^+(\rho)$ sera dite *pred-stable.*

Analyse en arrière

Pour tout $i \in [0, n]$, posons :

$$\begin{cases} H_i = (s_i, true) & si\ i = n \\ H_i = pred(H_{i+1}, t_{i+1}) & sinon \end{cases}$$

H_n est définie par le polyèdre *true* ($\forall \nu \in true, \forall x \in C, \nu(x) \geq 0$) et H_i est post-stable H_{i+1} par t_{i+1}.

Exemple 7.4 *Considérons le chemin :*

$$\rho \ : \ s_0 \xrightarrow{x \leq 2, a, y := 0} s_1 \xrightarrow{y \leq 1, b, x := 0} s_2.$$

Alors,

$$\begin{cases} H_2 = (s_2, x \geq 0 \wedge y \geq 0) \\ H_1 = pred(H_2, t_2) = (s_1, x \geq 0 \wedge y \leq 1) \\ H_0 = pred(H_1, t_1) = (s_0, x \leq 2 \wedge y \geq 0) \end{cases}$$

□

Le corollaire suivant découle de la définition de $pred()$.

Corollaire 7 *L'état final s_n de ρ est atteignable si et seulement si $H_0 \cap (s_0, zero)$ n'est pas vide.* □

En effet, il n'est pas suffisant que H_0 ne soit pas vide mais que $H_0 \cap (s_0, zero)$ le soit aussi, pour la simple raison que les horloges sont lancées simultanément à l'état initial. Par exemple, si au cours de l'analyse en arrière on obtient $H_0 = (s_0, x \geq 0 \wedge y = 1)$, l'état s_n n'est pas accessible du fait qu'à l'état initial $(s_0, x = y = 0)$, on ne peut atteindre H_0 en laissant le temps s'écouler.

Finalement, de la même façon, au chemin ρ, on associe un chemin symbolique $S_{-}(\rho)$ dont les états sont les zones H_i. $S_{-}(\rho)$ est défini comme suivant :

$$S_{-}(\rho) \ : \ H_0 \xrightarrow{a_1} H_1 \cdots H_{n-1} \xrightarrow{a_n} H_n$$

Une transition $H_{i-1} \xrightarrow{a_i} H_i$ de $S_{-}(\rho)$ sera dite *post-stable.*

Analyse en avant et en arrière

Le propriété de pred-stabilité de $post()$ garantit que tout $q \in H_i$ admet un état prédécesseur dans H_{i-1} . Cependant, elle ne garantit pas que tout $q \in H_i$ admet un état successeur dans H_{i+1}. Ce dernier point est pourtant garanti par la post-stabilité de $pred()$. L'idée est alors de combiner $post()$ et $pred()$ de la façon suivante :

Pour i de 0 à n

$$\begin{cases} H_i = (s_0, zero) & si\ i = 0 \\ H_i = post(H_{i-1}, t_i) & sinon \end{cases}$$

Puis pour i de n à 0

$$\begin{cases} H'_i = H_i & si\ i = n \\ H'_i = H_i \cap pred(H'_{i+1}, t_i) & sinon \end{cases}$$

De cette façon, H'_i vérifie la post/pred-stabilité, pour tout $i \in [0, n]$.

Exemple 7.5 *Supposons que ρ est le chemin défini par :*

$$\rho\ :\ s_0 \xrightarrow{a, y:=0} s_1 \xrightarrow{y=1 \wedge x \leq 3, b} s_2.$$

Par application de post() à ρ on obtient :

$$\begin{cases} H_0 = (s_0, x = y = 0) \\ H_1 = (s_1, x \geq 0 \wedge y = 0) \\ H_2 = (s_2, x \leq 3 \wedge y = 1) \end{cases}$$

On remarque que l'état $q_1 = (s_1, \nu_1) \in H_1$ défini par $\nu_1(x) = 3$ et $\nu_1(y) = 0$ possède un prédécesseur dans H_0 mais ne possède pas de successeur dans H_2. Maintenant, appliquons pred() à ρ :

$$\begin{cases} H'_2 = H_2 = (s_2, x \leq 3 \wedge y = 1) \\ pred(H_2, t_2) = (s_1, x \leq 3 \wedge y \leq 1 \wedge x - y \leq 2) \Rightarrow \\ H'_1 = H_1 \cap pred(H_2, t_2) = (s_1, x \leq 3 \wedge y = 0 \wedge x - y \leq 2) = (s_1, x \leq 2 \wedge y = 0) \\ H'_0 = H_0 \end{cases}$$

□

Corollaire 8 *Soit $i \in [0, n]$. Pour tout état $(s, \nu) \in H'_i$, il existe une computation $r\ :\ (\overline{s}, \overline{\nu})$ sur une séquence temporisée σ, telle que : $s = s_i$ et $\nu_i = \nu$.* □

Finalement, au chemin ρ, on associe un chemin symbolique $S^+_-(\rho)$ dont les états sont les zones H'_i. $S^+_-(\rho)$ est défini comme suivant :

$$S^+_-(\rho)\ :\ H'_0 \xrightarrow{a_1} H'_1 \cdots H'_{n-1} \xrightarrow{a_n} H'_n$$

Une transition $H'_{i-1} \xrightarrow{a_i} H'_i$ de $S^+_-(\rho)$ sera dite *post/pred stable*.

7.2.3 k-complétude d'une computation

Définition 33 *Une k-incomplète computation est un couple (s, ν) tel que s est un état et ν est une valuation de C. On dit que (s, ν) peut être complétée dans le chemin ρ, s'il existe une computation $r : (\overline{s}, \overline{\nu})$ de ρ sur une séquence temporisée σ telle que $\nu_k = \nu$ et $s_k = s$. Dans ce cas, r est dite une computation complétude (ou une k-complétude) de (s, ν).* □

La k-complétude d'une computation est similaire à la k-complétude d'une valuation. La différence est que l'on cherche, dans le premier cas, une computation du chemin ρ qui coïncide avec (s, ν) à l'état k alors que, dans le deuxième cas, on cherche à compléter ν dans un polyèdre.

D'après le corollaire 8, tout élément d'une zone du chemin symbolique $S^{+}_{-}(\rho)$ obtenu par une analyse en avant et en arrière admet une k-complétude. Par la suite, nous allons décrire une technique pour compléter une k-incomplète computation.

Passage en avant

Il est suffisant de décrire la construction pour une étape i, $i \in [0, n-1]$. D'une façon générale, étant donnée une transition post-stable $H_1 = (s_1, Z_1) \xrightarrow{a} H_2 = (s_2, Z_2)$ ($H_1 = pred(H_2, t)$ avec $t = (s_1, Z, a, r, s_2)$) et $q_1 = (s_1, \nu_1) \in H_1$, comment choisir $q_2 = (s_2, \nu_2) \in H_2$ et $d \geq 0$ tels que $\nu_1 + d \in Z$ et $(\nu_1 + d)[r := 0] = \nu_2$ (q_2 est un état successeur de q_1 dans H_2).

La recherche de d peut être faite par le calcul du délai minimum pour que ν_1 atteint Z, i.e. $d = min(\{d' \mid \nu + d' \in Z\}$. Si Z est canonique [2], cela revient à prendre $d = min(\{d_x \mid d_x = c_x - \nu_1(x), x \in C\}$ où c_x est la borne inférieure de la contrainte sur x dans Z. Dans ce cas là, ν_2 prend la valeur $(\nu_1 + d)[r := 0]$. Notons que la post-stabilité de la transition $H_1 \xrightarrow{a} H_2$ garantit l'existence de d.

Lemme 9 *Pour tout q_1 de H_1, calculer un état successeur de q_1 dans H_2 peut être fait en temps $O(p)$ (p est nombre d'horloges de C).* □

Remarque 9 *Si on considère $q_1 = (s_1, \nu_1) \in H_1$ comme une zone, i.e. on confond ν_1 avec le polyèdre qui contient uniquement ν_1, alors $H = post(q_1, t) \cap H_2$ contient tous les successeurs de q_1 dans H_2. Dans ce cas, il suffit d'extraire un élément de H qui sera q_2. Cependant, la complexité du calcul change.* □

[2] Lors de l'implantation des opérations *post*() et *pred*(), on considère la forme canonique des polyèdres.

Passage en arrière

De la même façon, étant donnée une transition pred-stable $H_1 = (s_1, Z_1) \xrightarrow{a} H_2 = (s_2, Z_2)$ ($H_2 = pred(H_1, t)$ avec $t = (s_1, Z, a, r, s_2)$) et $q_2 = (s_2, \nu_2) \in H_2$, comment choisir $q_1 = (s_1, \nu_1) \in H_1$ et $d \geq 0$ tels que $\nu_1 + d \in Z$ et $(\nu_1 + d)[r := 0] = \nu_2$ (q_1 est un état prédécesseur de q_2).

Sans perte de généralité, supposons que les j horloges non remises à zéro dans r sont $C_j = \{x_1, \cdots, x_j\}$. Soit ν une valuation sur C_j telle que pour tout $i \in [1, j]$, $\nu(x_i) = \nu_2(x_i)$. ν est une j-incomplète valuation de C. Soient v' la valuation de C qui complète ν dans $Z_1^{\uparrow} \cap Z$ et $d \geq 0$ tel que $\nu' - d \in Z_1$ et pour tout $d' > d$, $\nu' - d' \notin Z_1$. Dans ce cas, il suffit de prendre $\nu_1 = \nu' - d$. Notons que la pred-stabilité de la transition $H_1 \xrightarrow{a} H_2$ garantit l'existence de ν' et d.

Lemme 10 *Pour tout q_2 de H_2, calculer un état prédécesseur de q_2 dans H_1 peut être fait en temps $O(p^2)$ (p est nombre d'horloges de C).* □

En effet, la complexité d'une k-incomplète valuation est de $O(p^2)$ [3].

Remarque 10 *Si on considère $q_2 = (s_2, \nu_2) \in H_2$ comme une zone, i.e. on confond ν_2 avec le polyèdre qui contient uniquement ν_2, alors $H = pred(q_2, t) \cap H_1$ contient tous les prédécesseur de q_2 dans H_1. Dans ce cas, il suffit d'extraire un élément de H qui sera q_1.* □

Passage en avant et en arrière

Corollaire 9 *Soit $k \in [0, n]$. Pour tout état $q = (s, \nu) \in H'_k$ du chemin symbolique $S_-^+(\rho)$, calculer une k-complètude computation de q peut se faire en temps $O(n \times p^2)$ (n est la longueur de ρ et p est le nombre d'horloges de C).* □

En effet, les H'_k sont post/pred stables. Donc le calcul d'un successeur de $q \in H'_k$ dans H'_{k+1} se fait en temps $O(p)$ et le calcul d'un prédécesseur de q dans H'_{k-1} se fait en $O(p^2)$. Ainsi, le calcul d'une computation de ρ qui coïncide avec q se fait en temps $O(n \times p^2)$

Finalement, une 0-complètude d'une computation consiste à extraire un diagnostic temporisé de ρ. Dans la section suivante, nous allons montré comment calculer les diagnostics de bornes comme nous l'avons vu avec l'approche utilisant le polyèdre de contraintes.

[3] Si Z est canonique.

7.2.4 Diagnostic temporisé de bornes

Généralement, un automate temporisé utilisant un ensemble d'horloges C possède aussi une horloge universelle (absolue et sans remise à zéro sauf dans l'état initial). Elle est implicite dans l'automate (non contrainte dans une transition). Notons par h cette horloge universelle. Par souci de clarté, nous supposons que h est explicite dans C, quitte à l'ajouter.

Considérons maintenant le chemin symbolique $S_-^+(\rho)$ obtenu par une analyse en avant et en arrière :

$$S_-^+(\rho) \;:\; H_0 = (s_0, g_0) \xrightarrow{a_1} H_1 = (s_1, g_1) \cdots H_{n-1} = (s_{n-1}, g_{n-1}) \xrightarrow{a_n} H_n = (s_n, g_n)$$

Supposons que les $(g_k)_{k \in [0,n]}$ sont bornés et canoniques et considérons $k \in [0, n]$.

Soient $\nu^{M0}(g_k)$ et $\nu^{m0}(g_k)$ les valuations de bornes maximales et minimales, respectivement, associées à g_k données par le théorème 2 de la section 6.2.3.[4] Alors, pour toute valuation $\nu \in g_k$, pour toute $x \in C$, $\nu^{m0}(g_k)(x) \leq \nu(x) \leq \nu^{M0}(g_k)(x)$. Vu que le chemin $S_-^+(\rho)$ est post/pred-stable et selon le corollaire 9, $\nu^{M0}(g_k)$ (resp. $\nu^{m0}(g_k)$) admet une k-complètude computation de ρ.
Considérons :

1. $r^{(M,k)}$: $(\overline{s}, \overline{\nu^{(M,k)}})$ la computation sur la séquence temporisée $\sigma^{(M,k)}$ qui complète $(s_k, \nu^{M0}(g_k))$ telle que :
   - Pour tout $j \in [0, n-1]$, si $(\nu_j^{(M,k)} + d_j)[r_{j+1} := 0] = \nu_{j+1}^{(M,k)}$, alors pour tout $d < d_j$, $(\nu_j^{(M,k)} + d)[r_{j+1} := 0] \notin H_{j+1}$.
2. $r^{(m,k)}$: $(\overline{s}, \overline{\nu^{(m,k)}})$ la computation sur la séquence temporisée $\sigma^{(m,k)}$ qui complète $(s_k, \nu^{m0}(g_k))$ telle que :
   - Pour tout $j \in [0, n-1]$, si $(\nu_j^{(m,k)} + d_j)[r_{j+1} := 0] = \nu_{j+1}^{(m,k)}$, alors pour tout $d > d_j$, $(\nu_j^{(m,k)} + d)[r_{j+1} := 0] \notin H_{j+1}$.

Théorème 3 *$\sigma^{Mk} = \sigma^{(m,k)}$ et $\sigma^{mk} = \sigma^{(M,k)}$ telles que σ^{Mk} et σ^{mk} sont les diagnostics temporisés de bornes de la définition 31, section 7.1.* □

Ainsi $\sigma^{(M,k)}$ et $\sigma^{(m,k)}$ sont exactement les séquences temporisées σ^{Mk} et σ^{mk} identifiées dans la section 7.1.

Preuve. Voir Annexe A. □

[4]Dans le théorème, cas valuations sont notées par $\nu_0^M(g_k)$ et $\nu_0^m(g_k)$, respectivement.

7.2.5 Inclusion des traces

Nous venons d'identifier les diagnostics temporisés de bornes maximales et minimales, en se basant, cette fois-ci, sur une analyse en avant et en arrière. Dans ce cas, pour montrer que $TTrace(A_\rho) \subseteq TTrace(B_{\rho'})$, il suffit de montrer que les diagnostics de bornes associés à ρ sont des traces de $B_{\rho'}$.

7.3 Comparaison avec des travaux similaires

Dans le reste de cette section p est le nombre d'horloges et n la taille du chemin ρ.

7.3.1 Travaux d'Alur et al. [6]

Dans l'article [6], les auteurs se sont intéressés à la génération d'un seul diagnostic temporisé. L'approche utilisée se base aussi sur le polyèdre de contraintes et en particulier sur le graphe de contraintes G associé à un chemin. La méthode utilisée consiste non plus à calculer le graphe minimal associé $b2m(G)$ (et donc la forme canonique du polyèdre de contraintes), mais seulement les plus courts chemins entre le noeud v_0 (qui vaut 0) et tout autre noeud du graphe, tout en ignorant les arcs étiquetés par $+\infty$. L'algorithme se déroule en plusieurs étapes $i \in [1, n]$: à l'étape i,

- G_i est le sous graphe de G contenant les noeuds d'indice compris entre 0 et i.
- $b2m(G_i)$ est alors calculé (graphe minimal).
- On associe un autre graphe réduit G'_i à G_i, contenant seulement les noeuds v_0, v_i et les noeuds ayant un arc sortant vers un noeud d'indice supérieur à i. Le poids d'un arc dans G'_i est égal à son poids dans $b2m(G_i)$.
- On ajoute le noeud v_{i+1} à G'_i pour obtenir le nouveau graphe G_{i+1}.

La complexité de l'algorithme est $O(n \times p^2)$.

7.3.2 Travaux de Tripakis [145, 146]

L'approche de Tripakis présentée dans [145, 146] est très proche de notre approche basée sur l'analyse symbolique. Il a étudié la k-complètude et l'extraction des diagnostics temporisés. La solution proposée pour la k-complètude consiste à choisir arbitrairement un prolongement qui vérifie les contraintes de la zone. Pour la génération du diagnostic temporisé, elle est basée sur l'automate de simulation (détaillé dans section 10.4.2 du chapitre 10). La complexité est aussi en $O(n \times p^2)$.

7.3.3 Autres travaux

Dans [94], les auteurs montrent l'existence d'un diagnostic temporisé associé à un chemin symbolique, mais aucune méthode n'est proposée pour l'extraire. Dans [70], les auteurs proposent d'utiliser l'outil Uppaal, basé sur une analyse d'accessibilité, pour générer un diagnostic temporisé optimal correspondant à un chemin. Dans [114], les auteurs donnent plusieurs algorithmes, basés sur l'automate de simulation, pour générer un diagnostic minimal qui atteigne un état donné.

7.3.4 Nos contributions

Dans la partie III, nous nous sommes intéressés aux deux problématiques suivantes :

1. La génération des diagnostics temporisés pour un chemin ρ, i.e. l'extraction de quelques éléments de $TTrace(A_\rho, n)$.
2. La vérification de $TTrace(A_\rho, n) \subseteq TTrace(B_{\rho'}, n)$ pour deux chemins ρ et ρ' sachant que A_ρ est connu et $B_{\rho'}$ n'est pas connu.

Pour y parvenir, nous avons proposé deux méthodes. La première méthode était basée sur le polyèdre de contraintes associé à un chemin. La deuxième méthode utilise une analyse symbolique d'accessibilité en avant et en arrière. Le tableau suivant résume les différents résultats, ainsi que les hypothèses formulées pour les deux approches.

	Polyèdre de contraintes	Analyse symbolique
Hypothèses	canonique (borné)	canonique (borné)
Diagnostics de bornes	$O(n)$	$O(n \times p^2)$
k-complètude	$O(n^2)$	$O(n \times p^2)$
Inclusion des traces	$O(n^2)$	$O((n \times p)^2)$

FIG. 37: Complexités.

Notons que la complexité de l'inclusion des traces est égale à la complexité d'un diagnostic de bornes ($O(n \times p^2)$) multipliée par $2 \times (n + 1)$ (le nombre des diagnostics de bornes), ce qui explique les résultats du tableau.

Dans les deux approches présentées, nous avons supposé que les polyèdres manipulés sont canoniques et bornés. L'hypothèse d'une borne assure l'existence des diagnostics temporisés maximales. Cette hypothèse n'est pas une limitation, spécialement dans le cas du test qui est une expérience finie et donc bornée. Nous n'avons pas abordé la question du calcul de la forme canonique. Cependant, il existe des méthodes pour calculer cette forme. La méthode la plus simple et la plus utilisée est celle donnée dans [58]. Sa complexité est en temps $O(p^3)$. Dans le cas de l'approche basée sur le polyèdre de contraintes, la complexité de la mise en forme canonique sera alors $O(n^3)$. Cependant, dans l'analyse symbolique, les

polyèdres manipulés sont dans $\Phi(C)$ et donc le calcul de la forme canonique est en temps $O(p^3)$. D'une façon générale, pour un chemin de longueur n, au plus n horloges seront non redondantes et donc $n \geq p$. De plus, n est généralement grand devant p.

Comme conclusion, au niveau complexité, l'analyse symbolique est dominante. Au niveau simplicité, nous dirons que l'approche polyèdre de contraintes est plus simple par rapport à la définition d'une analyse en avant et en arrière.

A notre connaissance, l'identification des diagnostics temporisés de bornes maximales et minimales et l'étude de l'inclusion des traces sont des résultats nouveaux. De plus, la complexité de notre approche est la même que dans [6] ou [145] pour l'identification des diagnostics.

Quatrième partie

Test des Systèmes Communicants

Introduction

Un protocole constitue un ensemble de règles de fonctionnement définies afin de fournir un service de communication donné. La phase de vérification est nécessaire pour s'assurer que le protocole a été bien conçu et fournit le service de communication attendu. Le protocole supposé être vérifié fait généralement l'objet d'une implantation par différents constructeurs pour devenir un produit commercialisé.

Pourquoi tester ? On pourra citer les deux raisons majeures suivantes :

1. *Améliorer la qualité d'un système.* Une machine à laver qui tache les habits n'a pas d'avenir commercial. Une machine à laver qui consomme une importante quantité d'électricité n'est pas rentable pour un utilisateur. En général, un système doit fournir au minimum un degré de qualité acceptable.
2. *Maintenir la qualité d'un système.* Si Windows XP tombe en panne plus souvent que Windows 2000[5], Linux deviendra le premier système d'exploitation au monde [6]. En règle générale, une évolution du système doit fournir aux utilisateurs au moins la qualité du système initial.

Pourquoi développer des méthodes formelles pour le test ? Les protocoles définissent des systèmes relativement complexes. Les comportements possibles sont nombreux, voire infinis. La production manuelle des tests est :

- *un travail de longue haleine.* Ceci est dû à l'aspect volumineux et complexe des spécifications. Lorsque celles-ci sont écrites textuellement, il est impossible d'en faire un traitement automatique, d'où l'intérêt des méthodes formelles.
- *un travail d'expert.* L'écriture des scénaris de test nécessite une certaine expertise du protocole.
- *un travail répétitif.* Lire la spécification, la comprendre, l'interpréter puis écrire des tests, c'est le travail éternel d'un ingénieur de test. Sur la base d'un formalisme donné pour la spécification, il est possible de définir des stratégies de génération réutilisables quelque soit le protocole spécifié dans le formalisme choisi.

Pourquoi différents types de test ? Le type de test est défini suivant les objectifs qu'il vise et les spécificités des besoins. On distingue les types de test suivants :

- *Test de conformité.* L'objectif est de tester la *conformité* d'un système donné par rapport à son modèle (sa spécification). Comme nous le verrons par la suite, ce type de test a été défini en détail par les organismes de normalisation ISO (International Standardization Organization) et ITU (International Telecommunication Union).

[5]reste à vérifier !!!
[6]Ce jour là viendra sans doute.

- *Test d'interopérabilité.* L'objectif est de tester l'aptitude de deux ou plusieurs composantes à échanger l'information et utiliser mutuellement l'information échangée. Il vient palier la non exhaustivité du test de conformité et l'ambiguïté des spécifications.
- *Test de robustesse.* L'objectif est ici de déterminer le comportement dans un environnement "hostile", i.e dans des conditions anormales de fonctionnement, conditions non prévues par la spécification. L'ajout des aléas dans la spécification augmente considérablement sa taille.
- *Test de performance.* Ce test vise à mesurer certains paramètres de performance du système, comme le débit maximal, les délais de transmission, le nombre de connexions supportées en parallèle,...

Seul le test de conformité a été normalisé. Cette classification de test est basée essentiellement sur l'objectif du test. La classification ci-dessous est basée sur le degré d'accessibilité et d'observabilité du système à tester.

- Test *boîte noire* : aucune connaissance du comportement interne du système n'est formulée ; seuls les évènements observables échangés entre le système et son environnement sont pris en compte.
- Test *boîte blanche* : le comportement interne du système est connu dans sa totalité.
- Test *boîte grise* : dans cette catégorie intermédiaire, on accède à certaines communications entre les différentes composantes du système.

Motivations

Nous nous intéressons dans cette partie au test des systèmes communicants. Durant la dernière décennie, plusieurs recherches ont été menées dans le domaine du test et ont donné différentes méthodes basées sur différents modèles formels et différentes applications possibles. Toutes ces méthodes génèrent avec succès des cas de test, mais la plupart d'entre elles ne reposent pas sur des cadres formels bien définis et souffrent de l'explosion du nombre des cas de test générés. En conséquence, la motivation de développer des nouvelles méthodes de génération est encore d'actualité. Ainsi, l'une des motivations de cette partie est la définition de cadres théoriques comprenant des définitions formelles et des méthodes de génération pour les tests de conformité et d'interopérabilité.

Au delà des motivations relatives au développement de cadres formels pour la conformité et l'interopérabilité, un autre point qui motive cette recherche est l'étude théorique de la possibilité d'une modélisation uniforme des différents éléments intervenant dans le test (architectures de test, approches et types de test, séquences de test,...). En effet, une analyse approfondie des différents cadres formels, proposés pour les différents types de test, peut révéler une grande ressemblance entre ces cadres formels. Par exemple, la plupart des méthodes de génération de cas de test de conformité peuvent être adaptées à la génération de cas de test d'interopérabilité. Dès lors, développer un cadre formel pouvant supporter les différents éléments intervenant dans le test aura deux intérêts majeurs. Le premier intérêt est la possibilité de réalisation d'outils de génération de test supportant différents types

et approches du test. Ainsi, avec le même outil, il est possible 1) de modéliser différentes architectures de test, 2) de générer des cas de test de conformité, d'interopérabilité, etc, et 3) de vérifier l'appartenance d'une trace (de l'implantation) à la spécification (approche passive de test). Le deuxième intérêt est la possibilité de la normalisation de l'activité du test. En effet, à l'exception du test de conformité, aucun type de test n'a été normalisé par un organisme (national ou international).

Contributions

Trois contributions majeures peuvent être citées.

1. *Proposition d'un cadre formel pour le test d'interopérabilité des systèmes sans contraintes temporelles.* D'une part, nous proposons une formalisation de la notion d'interopérabilité par le biais d'une relation d'interopérabilité. La relation d'interopérabilité proposée repose sur une comparaison entre les sorties produites par les implantations et les sorties autorisées par les spécifications. Elle tient aussi en compte les interfaces accessibles des implantations. D'autre part, nous proposons une technique de génération de cas de test d'interopérabilité à partir d'outils de génération de cas de test de conformité. Notre approche ne souffre pas de l'explosion du nombre de cas de test générés vue qu'elle ne construit pas le système global représentant le système communicant.
2. *Proposition d'un cadre formel pour le test de conformité des systèmes temporisés.* Tout d'abord, rappelons que l'explosion combinatoire du nombre de cas de test, due à la nature dense du temps, est plus accrue. D'une part, nous proposons une extension temporisée de la relation de conformité *ioco* de Tretmans [144] qui permet de définir la notion de conformité des systèmes réels. L'extension proposée, comparée à d'autres approches, ne sépare pas entre le passage du temps dans un état et la réalisation d'une action. Pour la méthode de génération, elle repose sur l'utilisation de l'automate de simulation introduit par Tripakis [145] (pour générer des chemins faisables) et l'application des résultats de la partie *Analyse des Systèmes Temporisés* de ce document pour générer des cas de test temporisés. En effet, dans la partie III, nous avons étudié le problème d'inclusion de traces entre deux chemins et nous avons montré que ce problème peut être résolu par l'utilisation des diagnostics temporisés de bornes associés à un chemin[7]. Ainsi, nous ne générons que des cas de test relatifs à ces diagnostics. Par conséquent, notre approche minimise le nombre de cas de test considérés tout en couvrant l'espace temporel des états accessibles.
3. *Introduction du test ouvert.* Comme nous allons le voir, nous nous orientons vers une approche dite *ouverte* du test, dans le sens qu'elle ne tient pas compte des différents éléments intervenant dans le test. Cette approche est basée sur le modèle CS introduit dans la partie II. Elle établit une méthodologie pour l'activité du test.

[7]Les diagnostics temporisés de bornes offrent une représentation finie de l'espace des traces associées au chemin.

> La méthodologie choisie est celle des algorithmes génériques de génération (gga). Elle est basée sur la définition d'un critère de couverture sous forme de coloriage. Nous montrons que le modèle CS est un modèle générique pour le test dans le sens qu'il permet (i) de modéliser différents types de communication et d'architectures du test et (ii) d'appliquer le même algorithme générique de génération (gga) pour les différents types et approches du test. Ce résultat nous motive à introduire le concept du test ouvert expliqué dans la section 11.2.6 du chapitre 11.

À notre connaissance, 1) l'utilisation des outils de conformité pour générer des cas de test d'interopérabilité, 2) la minimisation des nombre de cas de test temporisés et 3) l'approche ouverte du test sont des résultats originaux dans le domaine du test.

Organisation

Cette partie se compose de quatre chapitres. Dans le chapitre 8, nous exposons les différents concepts relatifs au test et en particulier aux tests de conformité et d'interopérabilité. Les chapitres 9 et 10 introduisent respectivement deux cadres formels pour les tests d'interopérabilité et de conformité. Ces cadres comportent des définitions et des méthodes de génération de test. Finalement, dans le chapitre 11, nous proposons une modélisation uniforme du test. Cette modélisation est basée sur 1) le modèle CS, 2) la définition d'un critère de couverture sous forme de coloriage de graphe et 3) la méthodologie des algorithmes génériques de génération.

Chapitre 8

Introduction au test

Sommaire

8.1 Norme ISO 9646 [81] 123
8.2 Relations de conformité 125
8.2.1 Concept de la relation de conformité 125
8.2.2 Relation $\leq_{tr}$ 126
8.2.3 Relation ioconf 127
8.2.4 Relation ioco 127
8.3 Objectif de test 129
8.4 Verdict, cas de test et exécution 131
8.4.1 Verdict 131
8.4.2 Cas de test 132
8.4.3 Exécution 132
8.4.4 Propriétés des tests 133
8.5 Interopérabilité 134
8.5.1 Définitions de l'interopérabilité 135
8.5.2 Activités et types du test d'interopérabilité 136
8.5.3 Architectures de test 137
8.5.4 Méthodologie pour l'interopérabilité 139

Le but de ce chapitre est d'introduire les notions de base relatives au test ainsi que les définitions utilisées lors de cette partie.

8.1 Norme ISO 9646 [81]

Le test de conformité consiste à réaliser des expérimentations sur une implantation d'un système afin d'en déduire la correction vis à vis d'une spécification de référence. C'est

un test boîte noire car le code de l'implantation n'est pas connu. Le testeur, qui simule l'environnement, interagit avec l'implantation sous test (IUT Implementation Under Test) en exécutant des tests élémentaires, générés auparavant, appelés *cas de test*. Il observe le comportement de cette dernière à travers des interfaces appelées *Points de Contrôle et d'Observation (PCO)*. Un verdict est alors formulé en analysant les réactions de l'IUT : si pour chaque cas de test, les sorties de l'IUT coïncident avec celles attendues (i.e. celles dérivées de la spécification) alors l'IUT est dite *conforme* à sa spécification ; sinon l'IUT est dit *erronée* et un processus de diagnostic est entamé pour localiser et fixer l'erreur.

Nous décrivons ci-dessous quelques termes définis dans la norme ISO 9646 [81].

Spécification, Implantations, Modèles. Une *spécification* est la prescription d'un comportement à l'aide d'un moyen formel de description. Une *implantation* est une entité matérielle et/ou logicielle possédant des interfaces à travers lesquelles elle peut communiquer avec son environnement. La spécification est un objet formel, tandis que l'implantation est physique. Afin de pouvoir lier de tels objets (de nature différente) par la conformité, on suppose (*hypothèse dite de test*) que l'implantation peut être modélisée dans un formalisme donné, en général, le formalisme de la spécification. Cette hypothèse suppose l'existence de la modélisation de l'implantation, mais a priori la description formelle de la modélisation est inconnue.

Conformité. La *conformité* est une propriété observable sur le comportement d'une implantation. Elle est relative au fonctionnement spécifié. La conformité peut être définie au moyen d'une relation binaire. Une implantation I est dite *conforme* à une spécification S selon la relation binaire R, si l'on a $I\,R\,S$. La relation R est un moyen d'exprimer la validité entre S et I.

Testeur. Le *testeur* est un programme exécutant des tests sur IUT et observant son comportement.

Architecture de test. C'est la description abstraite de la situation du testeur vis à vis de l'IUT, i.e. quelles sont les interfaces contrôlables ou observables par le testeur et comment s'effectue la communication entre le testeur et l'IUT. L'écriture de test dépend étroitement de l'architecture choisie. Il existe plusieurs architectures normalisées : locale, distante, etc.

Cas / suite de tests. Une *suite de tests* est un ensemble de comportements à exécuter sur l'IUT. Chaque élément de cet ensemble est appelé *cas de test*. TTCN (Tree and Tabular Combined Notation) est un langage de description de tests issu de ISO 9646.

Verdicts. L'exécution d'un cas de test conduit à un *verdict* {Pass, Fail, Inc} qui est une affectation de résultats à l'expérience de test.

Objectif de test. L'*objectif* du test décrit de manière abstraite le but d'un cas de test, i.e. la propriété qu'il est supposé tester sur l'IUT. En général, un objectif découle d'une exigence de conformité décrivant une propriété que le système (la spécification et son implantation) est censé vérifier.

8.2 Relations de conformité

Les *relations de conformité* sont un moyen de comparer deux systèmes, et comme nous le verrons par la suite, elles permettent d'exprimer ce qu'est la conformité d'une implantation par rapport à une spécification. En conséquence, une implantation est dite *conforme* si elle est en relation (de conformité) avec une spécification donnée.

8.2.1 Concept de la relation de conformité

“Qu'est ce que l'on entend par une implantation valide ?” La réponse à une telle question peut s'obtenir à l'aide du concept de validité par équivalence usuellement rencontré dans les techniques basées sur les algèbres de processus.

Étant données une spécification M et une relation d'équivalence R, on peut caractériser l'ensemble des implantations valides de M par l'ensemble des systèmes qui lui sont équivalents par R, i.e. l'ensemble :

$$\{ I \mid I\,R\,M \}$$

Diverses relations d'équivalence ont été proposées [115, 67]. Bien que de telles relations soient potentiellement intéressantes, il est reconnu qu'elle ne sont pas indispensables pour exprimer le lien entre une implantation et une spécification [97]. En particulier, le caractère symétrique de la relation n'est pas indispensable ; un système I peut implémenter un système M sans que les deux soient équivalents pour autant. Par exemple, il est généralement admis qu'une implantation réelle soit plus déterministe que sa spécification [97] qui est en général plus abstraite. Ainsi, les relations de conformité sont souvent (mais pas toujours) des préordres, i.e. des relations réflexives et transitives (pas nécessairement symétriques).

Les relations de conformité que nous présentons ici ont été souvent utilisées [40, 123, 143] car elles s'accordaient assez bien au contexte de conformité et de test de l'ISO.

8.2.2 Relation $\leq_{tr}$

Parmi les notations standards des LTSs (section 3.3.3), nous avons défini $Traces(M)$ comme étant l'ensemble des séquences de M de l'état initial. La première relation de conformité que nous présentons est l'inclusion des traces. Cette relation postule que toute trace de l'implantation est une trace de la spécification. Formellement,

Définition 34 *Soient* $M, I \in \mathcal{IOLTS}(\Sigma)$ *:*

$I \leq_{\mathbf{tr}} M =_{\Delta} Traces(I) \subseteq Traces(M)$. □

Il est facile de voir que $\leq_{tr}$ est réflexive et transitive, donc c'est un préordre.

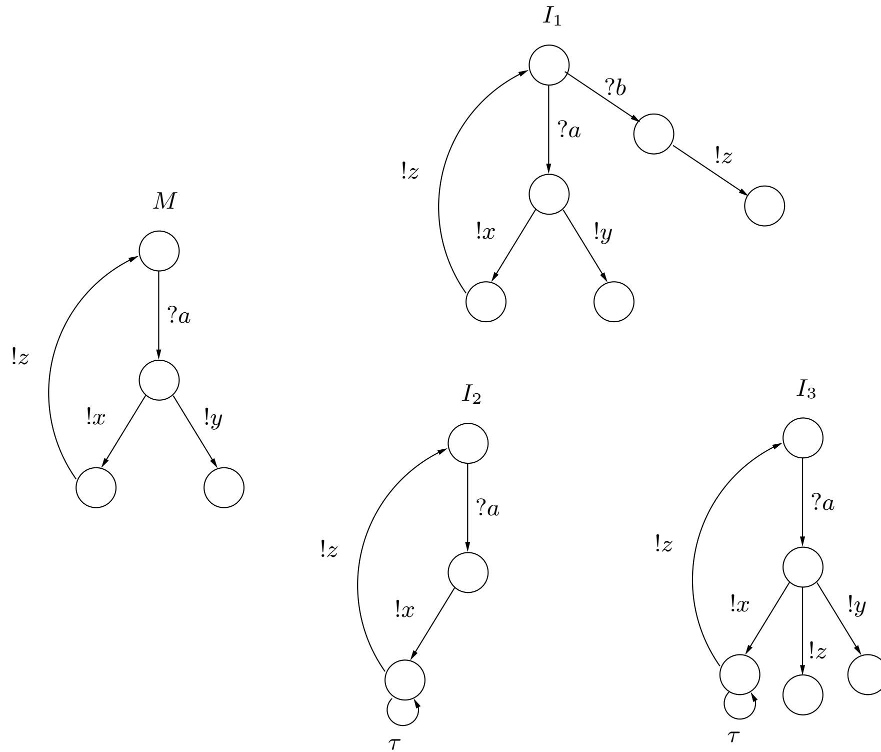


FIG. 38: Relation $\leq_{tr}$.

Exemple 8.1 *Considérons les IOLTSs de la* FIG.*38.* $\neg(I_1 \leq_{tr} M)$ *vu que la trace* $?b.!z$ *de* I_1 *n'est pas une trace de* M*. Cependant* $I_2 \leq_{tr} M$*. Finalement,* $\neg(I_3 \leq_{tr} M)$ *vu que* $?a.!z$ *est une trace* I_3 *mais n'est pas une trace de* M*.* □

8.2.3 Relation ioconf

La relation $\leq_{tr}$ est basée sur les traces et ne fait pas de distinction entre les entrées qui sont contrôlables par l'environnement du système et les sorties qui sont contrôlables par le système lui même. Tretmans [144] a introduit une relation *ioconf* qui fait une telle distinction. Cette relation établit qu'une implantation I est conforme à une spécification M si, après une trace de M, les sorties produites par I sont prévues par S. Rappelons que $Out(M, \sigma)$ est l'ensemble des événements produits par M après l'application de la trace σ (section 3.4). Formellement,

Définition 35 *Soient* $M, I \in \mathcal{IOLTS}(\Sigma)$ *:*

I **ioconf** $M =_{\Delta} \forall \sigma \in Traces(M) \Rightarrow Out(I, \sigma) \subseteq Out(M, \sigma)$. □

La relation *ioconf* permet la spécification partielle et par conséquent, l'implantation peut ajouter un traitement additionnel lors d'une entrée non prévue par la spécification.

Exemple 8.2 *Considérons encore les IOLTSs de la* FIG.*38.* $I_1\, ioconf\, M$ *malgré le fait que* I_1 *a ajouté la branche dont la trace est* $?b.!z$*. On effet,* $?b.!z$ *n'est pas dans* $Trace(M)$ *mais* $Out(M, \epsilon) = Out(I_1, \epsilon) = \emptyset$*. De même,* $I_2\, ioconf\, M$*, malgré que* I_2 *n'a pas implémenté l'émission de* y *après la réception de* a *(*$Out(I_2, ?a) = \{!x\} \subseteq Out(M, ?a) = \{!x, !y\}$*). Finalement,* $\neg(I_3\, ioconf\, M)$*, car* $Out(I_2, ?a) = \{!x, !y, !z\} \not\subseteq Out(M, ?a) = \{!x, !y\}$*).* □

8.2.4 Relation ioco

Le test permet d'observer le blocage d'un système par l'utilisation de temporisateurs. Le blocage se produit lorsque le système s'arrête d'évoluer.

Le blocage ou l'absence d'action dans un système peut être dû à plusieurs causes. On distingue :

- *Blocage de sortie* : se produit lorsque le système est en attente dans un état d'une entrée venant de son environnement. Concrètement, ce type de blocage se produit dans un état q lorsque $out(q) \subseteq \Sigma_i$.
- *Blocage vivant (livelock)* : se produit lorsque le système diverge par une suite infinie d'actions internes.
- *deadlock* : se produit lorsque le système ne peut plus évoluer. Ce cas correspond à un état q tel que $out(q) = \emptyset$.

La théorie *ioco* de Tretmans considère les sorties et les blocages possibles d'une spécification. Afin de considérer l'observation des blocages dans le test, nous avons besoin de modéliser ceux-ci dans la spécification. En effet, l'observation d'un blocage de l'implantation ne doit pas forcément produire un verdict $Fail$, qui correspond au rejet de l'implantation, car il se peut que ce blocage soit prévu dans la spécification.

À partir de l'IOLTS M de la spécification, nous avons besoin de considérer un IOLTS $\delta(M)$, appelé automate *suspendu*, obtenu par l'ajout des informations de blocage. Dans $\delta(M)$, un blocage est modélisé par un événement de sortie $!\delta$ visible par l'environnement et ne faisant pas partie des événements de la spécification. $\delta(M)$ est construit en ajoutant des boucles $q \xrightarrow{!\delta} q$ pour chaque état de blocage q. Formellement,

Définition 36 *Soit $M = (Q, q_0, \Sigma, \rightarrow)$ un IOLTS. L'automate suspendu de M est l'IOLTS $\delta(M) = (\delta(Q), q_0, \delta(\Sigma), \rightarrow_\delta)$ tel que :*

- *$\delta(\Sigma) = \delta(\Sigma_o) \cup \delta(\Sigma_i)$ avec $\delta(\Sigma_i) = \Sigma_i$, $\delta(\Sigma_o) = \Sigma_o \cup \{!\delta\}$ et $!\delta \notin \Sigma_o$.*
- *$\rightarrow_\delta$ est obtenue à partir de $\rightarrow$ par ajout de $q \xrightarrow{!\delta}_\delta q$ pour tous les états de blocage (i.e. deadlock, livelock et blocage de sortie).* □

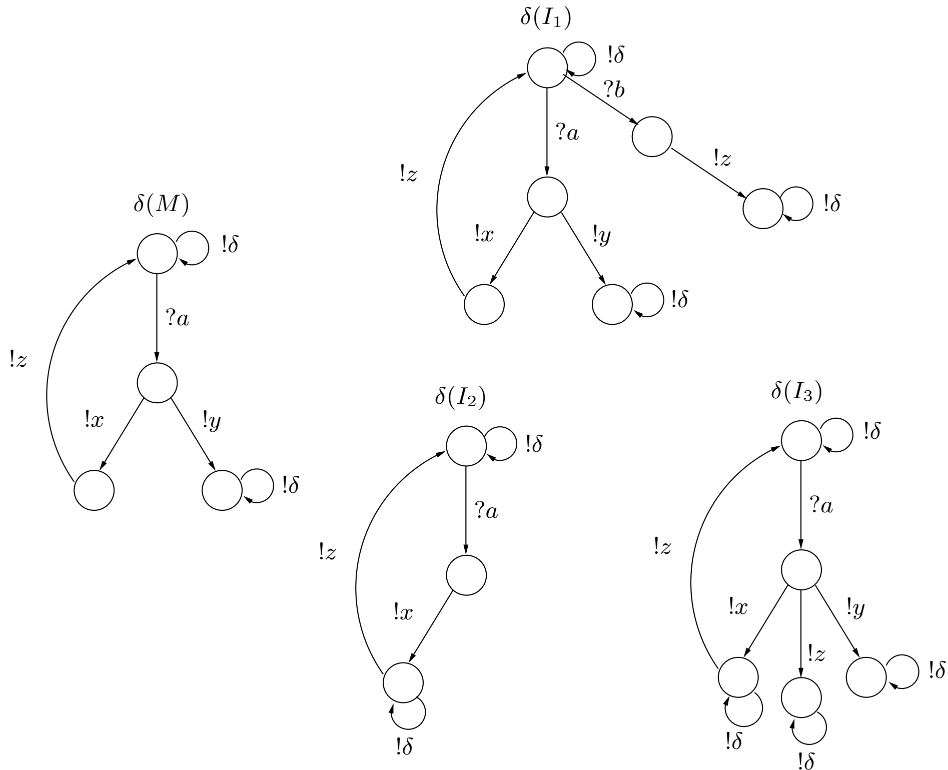


Fig. 39: Relation ioco.

Exemple 8.3 *Considérons l'IOLTS M de la* Fig.*38. L'automate suspendu de M est donné dans la* Fig.*39. Dans l'état initial, M ne peut changer d'état qu'après avoir reçu $?a$. Donc, dans cet état, il y aura un blocage de sortie qui est modélisé dans $\delta(M)$ (*Fig.*39) par la boucle $!\delta$ sur cet état. De même, après l'émission de $!y$ dans M, ce dernier atteint un état sans successeur. Dans ce cas, il y aura un deadlock modélisé par $!\delta$ dans $\delta(M)$.* □

Maintenant, la relation de conformité entre I et M est exprimée par la relation *ioco* définie par :

Définition 37 *Soient* $M, I \in \mathcal{IOLTS}(\Sigma)$ *:*

I **ioco** $M =_{\Delta} \forall \sigma \in Traces(\delta(M)) \Rightarrow Out(\delta(I), \sigma) \subseteq Out(\delta(M), \sigma).$ □

ioco étend $ioconf$ en considérant non seulement les traces de M mais aussi les traces avec blocages de M, i.e. les traces de l'automate $\delta(M)$. La relation *ioco* est expliquée dans la FIG.39. La première implantation est conforme à la spécification tandis que les deux suivantes ne le sont pas :

- $I_1\, ioco\, M$: on peut vérifier qu'en tout état, les sorties de I_1 sont incluses dans celles de M. I_1 permet une entrée supplémentaire par rapport à M mais ceci est autorisé : seules les sorties comptent. On peut donc faire des spécifications partielles.
- $\neg(I_2\, ioco\, M)$: le blocage de I_2 après la séquence $?a.!x$ n'est pas autorisé dans la spécification.
- $\neg(I_3\, ioco\, M)$: la sortie $!z$ après l'entrée $?a$ n'est pas autorisée dans la spécification.

8.3 Objectif de test

Un objectif de test est une description abstraite des comportements à tester, en général une séquence d'actions. C'est un formalisme simple ayant un comportement fini. Pour la synthèse du test, l'objectif de test est utilisé comme un critère de sélection de test. Pour permettre une sélection fine, on considérera deux ensembles distincts d'états marqués servant soit à accepter soit à rejeter des séquences d'actions de la spécification. Lorsque le système testé est décrit dans un formalisme F, l'objectif de test est aussi donné dans le même formalisme. Nous considérons par la suite, le cas des formalismes IOLTS et TIOA.

Spécification IOLTS

Définition 38 (Objectif de test) *Soit* $M = (Q, q_0, \Sigma, \rightarrow)$ *une spécification. Un objectif de test (TP) pour la spécification* M *est un IOLTS déterministe, observable et acyclique* $TP = (Q^{TP}, q_0^{TP}, \Sigma^{TP}, \rightarrow_{TP})$ *de même alphabet que* M ($\Sigma^{TP} \subseteq \Sigma$) *et équipé de deux ensembles d'états finaux* $Accept^{TP} \subseteq Q^{TP}$ *et* $Reject^{TP} \subseteq Q^{TP}$*. De plus* TP *vérifie :*

- *Pour tout état* $q \in Q^{TP}$*, si* $q \notin Reject^{TP} \cup Accept^{TP}$*, alors il existe un chemin de source* q *qui atteint un état de* $Accept^{TP}$*.* □

Les états $Accept^{TP}$ représentent les comportements cibles tandis que les états $Reject^{TP}$ sont utilisés pour stopper l'exploration de la spécification lorsque des actions indésirables sont rencontrées. Nous assumons de plus qu'un état qui n'est ni de $Accept^{TP}$ ni de $Reject^{TP}$

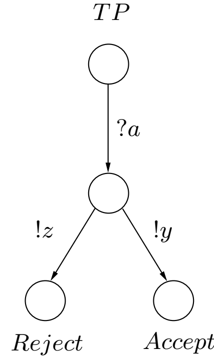


FIG. 40: Exemple d'objectif de test.

puisse atteindre un état de $Accept^{TP}$. Ceci permet aussi de stopper l'exploration de la spécification dès qu'un comportement non désiré est rencontré. Finalement, un état de $Accept^{TP}$ (resp. $Reject^{TP}$) sera dit simplement un état *Accept* (resp. *Reject*).

Exemple 8.4 *La* FIG.*40 donne un exemple d'un TP pour l'IOLTS M de la* FIG.*38. Dans TP, on veut tester le comportement suivant : la réception de a puis l'émission de y tel que entre ses deux actions z n'est pas émis. Dans chaque état de TP, on autorise tout comportement sauf celui qui mène vers un état Reject.* □

Définition 39 (Objectif de test complet) *Soient $M = (Q, q_0, \Sigma, \rightarrow)$ une spécification et un objectif $TP = (Q^{TP}, q_0^{TP}, \Sigma^{TP}, \rightarrow_{TP})$ de M. TP est dit complet si*

1. $\Sigma^{TP} = \Sigma$.
2. *L'IOLTS de TP est complet : $\forall q \in Q^{TP}$, $\forall a \in \Sigma^{TP}$, $q \xrightarrow{a}_{TP}$.* □

Un TP complet signifie que chaque état autorise toutes les actions. Considérer un objectif de test complet va à l'encontre de l'utilisation d'un TP. Cependant, pour satisfaire la complétude, nous utiliserons l'étiquette "*" dans une transition $q \xrightarrow{\star}_{TP} q$ comme abréviation de toutes les transitions de source et de destination q tels que : si $q \xrightarrow{a}_{TP} q$ alors il n'existe pas de transition $q \xrightarrow{a}_{TP} q'$ avec q' est un état *Reject* ($q' \notin Reject^{TP}$). En d'autres mots, $q \xrightarrow{\star}_{TP} q$ représente toute transition sur un événement de $a \in \Sigma^{TP}$ ne menant pas vers un état *Reject* depuis q.

Les transitions $q \xrightarrow{\star}_{TP} q'$ sont implicites dans un objectif de test mais explicites dans un objectif de test complet. Il est évident qu'un objectif de test peut être transformé en un objectif de test complet. D'une façon générale, nous utiliserons des objectifs de test qui ne sont pas complets sauf dans une partie de la section 11.2.4.

Spécification TIOA

Dans le cas d'une spécification exprimée par le TIOA $A = (S^A, s_0^A, \Sigma^A, C^A, \rightarrow_A)$, l'objectif de test $TP = (S^{TP}, s_0^{TP}, \Sigma^{TP}, C^{TP}, \rightarrow_{TP})$ est un TIOA déterministe, observable et acyclique, avec les mêmes hypothèses de la définition 38. Comme le test est une expérience finie, l'objectif doit tester un comportement fini. Ainsi, nous assumons que le temps ne diverge pas dans TP (TP est dit *borné* dans ce cas), i.e. il existe $d \in \mathbb{R}^{\geq 0}$ tel que pour $\sigma \in TTrace(TP) : delay(\sigma) \leq d$.

Définition 40 (Produit synchrone) *Soient A un TIOA et TP un objectif de test de A. Le produit synchrone de A et TP est le TIOA SP défini par :*
- $s_0^{SP} = (s_0^A, s_0^{TP})$.
- $S^{SP} = \{s_1 \| s_2 \mid s_1 \in S^A,\ s_2 \in S^{TP}\}$.
- $C^{SP} = C^A \cup C^{TP}$.
- $\rightarrow_{SP}$ *et* Σ^{TP} *sont obtenues par les règles suivantes :*

  1. $s_1 \xrightarrow{Z,a,r}_A s_1',\ a \notin \Sigma^{TP} \quad \Rightarrow \quad s_1 \| s_2 \xrightarrow{Z,a,r}_{SP} s_1' \| s_2$.
  2. $s_1 \xrightarrow{Z,a,r}_A s_1',\ s_2 \xrightarrow{Z',a,r'}_{TP} s_2',\ a \neq \tau \quad \Rightarrow \quad s_1 \| s_2 \xrightarrow{Z \wedge Z',a,r \wedge r'}_{SP} s_1' \| s_2'$.

Remarquons que la synchronisation sur les événements n'implique pas la compatibilité des contraintes temporelles. Finalement, nous considérons par la suite que le graphe associé à un TP est un simple arbre.

8.4 Verdict, cas de test et exécution

8.4.1 Verdict

Un cas de test est une expérience réalisée par le testeur sur l'implantation. Nous modélisons un cas de test par un IOLTS dont le graphe associé est un arbre. Les branches d'un cas de test décrivent les séquences d'interactions entre le testeur et l'IUT. Le rôle d'un cas de test est de détecter si l'IUT est conforme à sa spécification (la conformité étant donnée par une relation de conformité : ioconf, ioco,...).

Pour pouvoir détecter la non conformité d'une implantation par rapport à une spécification, les cas de test sont décorés par des verdicts. La signification des verdicts est la suivante :

Fail signifie que l'IUT n'est pas conforme à la spécification. Selon la relation de conformité R, un verdict $Fail$ est assigné à chaque entrée du testeur ne correspondant à aucune sortie de la spécification. Cela est réalisé par l'ajout implicite d'une transition étiquetée par "Other" et dont la destination est un état étiqueté par $Fail$, dans chaque état du cas de test. Notons aussi que le verdict $Fail$ est associé à l'expiration du temporisateur modélisant le silence.

Pass signifie que l'IUT a atteint, après une séquence d'interactions, un état valide de la spécification.

Inc est utilisé lorsqu'une entrée du testeur correspond à une sortie de la spécification qui mène à un comportement non considéré par le cas de test, du fait que le test n'est pas exhaustif. Ce genre de verdict est souvent utilisé dans les techniques de génération orientées objectif de test.

8.4.2 Cas de test

Définition 41 *Un cas de test TC pour un système modélisé par un IOLTS $M = (Q, q_0, \Sigma, \rightarrow)$ est un IOLTS $TC = (Q^{TC}, q_0^{TC}, \Sigma^{TC}, \rightarrow_{TC})$ dont le graphe associé est un arbre. TC est équipé de trois ensembles d'états $Fail^{TC} \subseteq Q^{TC}$, $Inc^{TC} \subseteq Q^{TC}$ et $Pass^{TC} \subseteq Q^{TC}$ caractérisant le verdict tels que : $\Sigma^{TC} = \Sigma_i^{TC} \cup \Sigma_o^{TC}$, $\Sigma_i^{TC} \subseteq \Sigma_o$, $\Sigma_o^{TC} \subseteq \Sigma_i$*[1]*. De plus, nous formulons les hypothèses suivantes sur TC :*

- *Les états possédant un verdict ne sont atteignables que par des entrées. Formellement, pour tout $(q, a, q') \in \rightarrow_{TC}$, $q' \in Pass^{TC} \cup Fail^{TC} \cup Inc^{TC}$ si et seulement si $a \in \Sigma_i$.*
- *TC est observable : aucune transition n'est étiquetée par un événement interne : pour tout (q, a, q'), si $(q, a, q') \in \rightarrow_{TC}$ alors $a \neq \tau$.*
- *TC est contrôlable : il n'existe pas un choix entre une sortie et une autre action. Formellement, pour tout $q \in Q^{TC}$, pour tout $a \in \Sigma_o^{TC}$, $q \xrightarrow{a}_{TC}$ implique que pour tout $b \neq a$, $q \not\xrightarrow{b}_{TC}$.*
- *TC est input-complet.* □

Lors de la représentation graphique d'un TC, une transition étiquetée par $?other$ dénotera la réception de tout autre événement non prévu dans l'état source de cette transition. Finalement, la FIG.41 illustre un exemple d'un cas de test pour la spécification M donnée dans la FIG.38.

8.4.3 Exécution

Un cas de test est exécuté par le testeur sur l'IUT. L'exécution est une composition de l'implantation et du cas de test sur les événements de synchronisation :

$$\frac{q_1 \xrightarrow{a}_I q_2, q_3 \xrightarrow{a}_{TC} q_4}{(q_1, q_3) \xrightarrow{a}_{I\|TC} (q_2, q_4)} \qquad \frac{q_1 \xrightarrow{\tau}_I q_2}{(q_1, q_3) \xrightarrow{\tau}_{I\|TC} (q_2, q_3)}$$

Ainsi, $TC\|I$ ne peut se bloquer que dans un état ayant un verdict (un état feuille). Le

[1]Remarquons que dans le cas de la relation *ioco*, on a $\Sigma_i^{TC} \subseteq \Sigma_o \cup \{?\delta\}$.

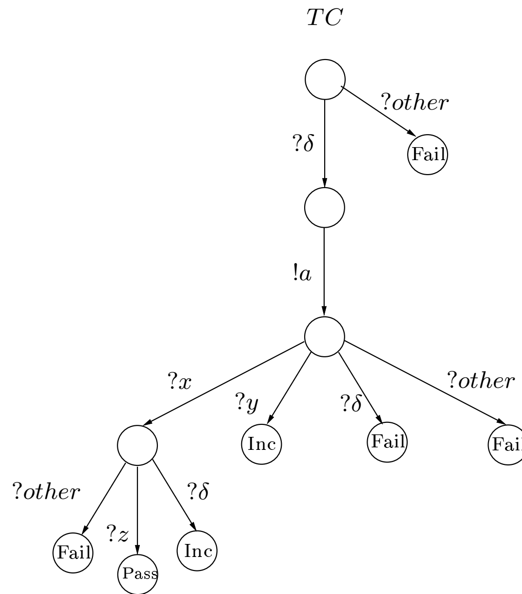


FIG. 41: Exemple de cas de test.

verdict associé à une exécution d'un cas de test TC sur I dépend entièrement de l'état final (feuille) atteint.

8.4.4 Propriétés des tests

Étant donné une spécification M et une suite de tests T, T peut être qualifiée d'*exhaustive* ou/et de *correcte* (voir définitions ci-dessous). Pour une suite de test T idéale, une implantation est conforme si et seulement si elle ne révèle pas d'erreur lorsqu'on lui applique T. Cependant, la nature infinie des comportements considérés entraîne qu'une telle suite de tests n'est pas toujours applicable en pratique.

Exhaustivité. Un test est exhaustif si une implantation qui ne révèle pas d'erreur pendant ce test est effectivement une implantation conforme.

Correction. Un test est correct si une implantation qui révèle une erreur pendant ce test est nécessairement non conforme.

Complétude. Un test est complet s'il est exhaustif et correct.

Une propriété minimale requise pour une suite de tests est la correction : une suite de tests ne doit pas rejeter des implantations conformes. Cette propriété est possible en général, mais ne garantit pas la conformité étant donné qu'une suite acceptant toute implantation est correcte. La propriété d'exhaustivité garantit la détection de toute implantation

non conforme. Cependant, elle nécessite une suite infinie de cas de test, dû par exemple à la présence de boucles dans la spécification. Une alternative à cette propriété est la propriété de l'exhaustivité de la méthode de génération des cas de test.

8.5 Interopérabilité

La réalisation des logiciels communicants repose sur l'implantation de protocoles fournissant la fonctionnalité souhaitée. Du fait que les différents équipements sont fournis par différents constructeurs et pour augmenter la confiance dans l'implantation de ces protocoles selon les standards internationaux, plusieurs méthodologies de test de protocoles ont été introduites. Ainsi, l'approche du test de conformité a été standardisé par ISO [76] et ITU-T [79]. La théorie sous-jacente au test de conformité est que toutes les implantations conformes doivent interopérer, bien qu'en pratique ce n'est pas le cas. Une autre approche de test est le test d'interopérabilité qui consiste en la confrontation d'une ou plusieurs implantations. Les standards sont utilisés alors comme 1) une référence pour ajuster les problèmes et les incompatibilités et comme 2) un guide des fonctionnalités à tester et des comportements attendus.

Le test de conformité inspire une confiance limitée dans la communauté des utilisateurs. En revanche, il fournit un moyen utile aux développeurs, durant les premières phases du développement d'un logiciel, comme une simple vérification. Parmi les raisons de l'échec du test de conformité à garantir l'interopérabilité et le bon fonctionnement des systèmes communicants, on peut citer :

- Les standards des protocoles restent ambigus malgré l'effort de formalisation : ils permettent souvent des options et des choix qui peuvent se contredire. Les implantations basées sur différents choix peuvent ne pas interopérer.
- L'effort de vérification n'est pas complet : seulement des aspects particuliers sont vérifiés.
- Les suites de tests de conformité ne sont pas exhaustives : elle ne permettent pas de garantir la conformité aux standards. Les considérations de coût limitent la taille des suites de tests, ainsi la couverture du test n'est pas complète.

Il est clair qu'en présence de l'exhaustivité des tests pour une spécification formelle prouvée, le test d'interopérabilité perd sa motivation. En pratique, l'exhaustivité est en général impossible (ressources limitées). Du fait que le test d'interopérabilité n'a pas été normalisé, on trouve une multitude de définitions relatives à la définition de l'interopérabilité, de l'activité du test d'interopérabilité et des architectures de test.

8.5.1 Définitions de l'interopérabilité

Une définition précise de l'interopérabilité est quelque chose d'élusive, cependant la signification fonctionnelle est claire : des composantes communiquent entre elles et fournissent les services attendus. Plusieurs définitions ont été proposées pour définir l'interopérabilité mais aucune standardisation d'ISO et d'ETSI n'a été proposée. En revanche des définitions ambiguës émergent des deux organismes :

1. ISO DTR-10000 [78]. L'interopérabilité est l'aptitude de deux ou plusieurs composantes à échanger l'information et à utiliser mutuellement l'information échangée.
2. ETSI ETR [55]. L'interopérabilité est l'aptitude d'un système distribué à échanger mutuellement les PDUs à travers la plate-forme de communication.

Ces deux définitions s'accordent sur le fait que l'interopérabilité est la capacité ou le degré de communication et d'échange d'informations entre entités. ETSI ETR considère que les entités sont des systèmes distribués contrairement à ISO DTR-10000 qui considère aussi bien les systèmes distribués que locaux. Par conséquent, 1) un comportement isolé d'une seule entité du système n'est pas un comportement d'interopérabilité et 2) aucune référence aux standards des spécifications n'est explicite, contrairement à la conformité. Selon ces définitions, l'interopérabilité est un comportement distinct qui complète la conformité.

Du fait de cette ambiguïté sur l'interopérabilité, d'autres définitions ont été introduites. On trouve :

3. Kang et al.[85]. Un ensemble d'objets est conforme en interopérabilité à sa spécification si sa réaction (observable) aux événements de son environnement est la même que celle prévue par la spécification résultante de la composition des différentes spécifications des objets du système (équivalence des traces observables).

Les auteurs définissent alors une relation d'implantation qui consiste en l'équivalence des traces du système global. Du fait que le système est considéré comme une seule entité, l'interopérabilité implique la conformité de chaque entité par rapport aux événements de l'environnement.

4. Koné et al. [90]. Deux implantations I_1 et I_2 sont conformes en interopérabilité à deux spécifications S_1 et S_2 ssi I_1 (resp. I_2) est conforme à S_1 (resp. I_2) et $I_1 \| I_2$ est conforme à $S_1 \| S_2$.
5. Barbin et al. [12]. Deux implantations interopèrent si elles vérifient une relation d'interopérabilité.

Pour [90], la relation d'interopérabilité est vue comme une relation de conformité. Elle est définie en terme d'inclusion de la partie observable des traces de l'implantation dans la partie observable des traces de la spécification. Ceci pour chaque entité, ainsi que l'entité

obtenue par composition. L'interopérabilité n'est donc qu'une conséquence de la conformité de plusieurs entités. Selon [12], elle est définie en terme de satisfaction d'une relation d'interopérabilité. Les auteurs proposent 9 relations qui dépendent de l'architecture de test utilisée.

8.5.2 Activités et types du test d'interopérabilité

L'aptitude d'un ensemble d'équipements d'un système à interopérer est une condition nécessaire pour la bonne coopération du système ou pour l'interfonctionnement des différents processus d'applications supportées par les équipements. Cette aptitude à interopérer comporte :

- Une aptitude à interopérer au niveau spécifications. Les différentes spécifications des composantes peuvent interopérer.
- Une aptitude à interopérer au niveau implantations. Les différentes spécifications des composantes sont supposées interopérer, ainsi que leurs implantations.

On distingue alors, le test d'interopérabilité dont l'objectif est l'interopérabilité des implantations, de la vérification d'interopérabilité dont l'objectif est l'interopérabilité des spécifications.

Selon [135], l'activité de test d'interopérabilité doit vérifier seulement les messages extérieurs qui mènent à une communication inter-composantes. Ce qui n'est pas le cas de [12] qui considère que l'activité doit vérifier une relation d'interopérabilité.

Avant de clôturer cette discussion sur l'activité de test d'interopérabilité, il est utile de citer d'autres définitions qui nous semblent intéressantes. ASD (Advanced Studies Department) du COS (Corporation for Open systems) définit quatre types d'interopérabilité par rapport à deux critères [59] :

1. Le moyen utilisé pour montrer l'interopérabilité : par preuve (contexte théorique), ou par test (contexte pratique). Ce critère distingue la vérification d'une spécification et le test d'une implantation.
2. Le nombre de couches considérées dans le système : une ou plusieurs couches.

Ces deux critères combinés définissent quatre types d'interopérabilité :

1. Interopérabilité théorique en couche.
2. Interopérabilité théorique en système.
3. Interopérabilité testable en couche.
4. Interopérabilité testable en système.

Les notions d'interopérabilité en couche et en système sont les suivantes : "en couche" est en relation avec les protocoles du même système (cas des protocoles d'OS (Open System)).

Cette activité tente à tester/vérifier l'interopérabilité de deux composantes (chacune ayant été testée/vérifiée) à travers leur fonctionnalité au niveau du service partagé ; "en système" est en relation avec des composantes de deux ou plusieurs systèmes. Le but est de tester/vérifier l'interopérabilité des protocoles d'un système avec un autre. Notons que [75] définit le test horizontal pour interopérabilité testable en système, le test vertical pour interopérabilité testable en couche et le test en diagonal pour l'appel de procédure dans le cas des protocoles.

8.5.3 Architectures de test

Elles décrivent l'emplacement du testeur par rapport à l'IUT ainsi que les points d'accès aux interfaces de communications de l'IUT avec son environnement. Le degré d'observabilité, de contrôlabilité et la configuration du système communicant ont une grande influence sur l'architecture à adopter. Plusieurs architectures ont été proposées reprenant une ou plusieurs de ces dépendances. Ces architectures peuvent être classées comme suivant :

Architectures de test pour la conformité

- Architectures Standards OSI [77]. OSI définit 1) un système sous test (SUT) comme un système qui utilise les protocoles standards OSI depuis la couche physique jusqu'à la couche application et 2) une implantation sous test (IUT), une implantation d'un protocole OSI. L'interface entre une IUT et la couche adjacente dans le SUT est appelée point de contrôle et d'observation (PCO). Une IUT possède un (ou deux) PCO situé "en bas ou en haut de" l'IUT (dans un accès à distance à l'IUT, ce point est situé en bas). Le système de test (TS) qui communique avec l'IUT grâce aux PCOs, se compose d'une ou deux composantes : un testeur supérieur (UT) et/ou un testeur inférieur (LT). La communication entre le UT et le LT est réalisée à travers une procédure de coordination de test (TCP). Le canal de communication entre l'IUT et le TS peut être local ou distant et supposé fiable. OSI distingue quatre types d'architectures de test pour le test de conformité : la méthode centralisée, la méthode distante, la méthode répartie et la méthode coordonnée, qui sont définies essentiellement en terme de coordination entre les testeurs dans le processus de test. Dans la méthode centralisée, deux PCOs sont définis avec le UT et le LT, mais peuvent être vus comme un port unique vu que l'IUT et le TS se trouvent sur le même site. Dans l'architecture répartie, un UT et un LT sont définis. Le LT est local au système de test et le UT est accédé en distance à travers le service de communication adjacent grâce à la procédure de coordination TCP. Le testeur ne possède alors qu'un PCO (celui du LT). L'architecture coordonnée est semblable à celle répartie, mais utilise un protocole de coordination (TMP), semblable au TCP, entre l'UT et l'LT. L'architecture distance correspond à celle distribuée, mais seul le testeur inférieur est utilisé et donc un seul PCO.

- Architecture de type Ferry [157, 158]. Cette architecture est un support d'implantation des architectures OSI. Elle définit un accès à un protocole grâce à des fonctionnalités supposées être implémentées dans SUT. Dans le cas d'un accès aux bornes supérieure et inférieure de l'IUT dans la pile SUT, l'architecture est appelée *ferry clip*. Le ferry clip offre une médiation entre l'IUT et SUT. Cette architecture a été expérimentée dans [152]. Ce dernier élargit les fonctionnalités exigées sur le SUT et le ST : rajout de fonctionnalité de transfert des informations de contrôle et addition d'un protocole de gestion des suites de test au TS.

- Architecture Multi-port [150, 103]. Cette architecture généralise les architectures OSI, dans le contexte d'une IUT communicant avec plusieurs entités ISO simultanément. Dans ce contexte, plusieurs UTs et LTs sont associés aux différentes interfaces de l'IUT. Le système de test [150] possède une fonctionnalité de contrôle qui permet de lancer l'ensemble des LTs et de coordonner les différents testeurs dans le cas local (les PCOs sont sur le même site que le SUT). Dans le cas d'interfaces distribuées, [103] introduit une méthode qui généralise la synchronisation entre deux testeurs à plusieurs testeurs.

Architectures de test pour l'interopérabilité

- Architectures passives et actives. Dans le test passif, le testeur est un agent qui observe les interactions de l'IUT sans interagir avec elle, tandis que dans le test actif, le testeur contrôle et interagit avec l'IUT [59, 73]. Le Forum ATM [42] définit une architecture qui consiste à connecter des testeurs aux différentes IUTs et à placer des POs entre deux IUTs et des PCOs entre un testeur et une IUT. Dans le cas où seuls les POs sont utilisés et placés entre l'IUT et le testeur [112], ce genre de test est appelé interopérabilité pure. L'intérêt de ce genre d'architecture est la facilité de mise en oeuvre. Dans certaines situations, cette architecture se présente comme l'unique solution [100].

- Architectures OSI Distribuées [44, 16, 43]. Les architectures d'OSI traitent seulement le cas d'un protocole isolé dans le SUT. Dans le cas des systèmes distribués, Rafiq et al. [44, 16, 43] définissent des architectures équivalentes à celles d'OSI pour le test de conformité. Dans la méthode centralisée, le système de test est réduit à un seul testeur composé de plusieurs testeurs attachés aux différents ports du système réparti. Dans les autres méthodes, le système de test consiste en plusieurs testeurs. S'il n'existe pas de coordination entre les testeurs et que chacun d'eux interagit avec le PCO qui lui correspond, indépendamment des autres, on parle de méthode distantse. Si la coordination entre les testeurs est prise en charge par le processus de test, la méthode est dite répartie. Si cette coordination doit suivre un processus normalisé, la méthode est dite coordonnée. Cette dernière correspond à un cas particulier de la méthode répartie. La difficulté que pose le portage des architectures OSI vers les systèmes distribués est en terme de coordination entre les différentes entités du TS qui peuvent être physiquement réparties. L'étude de ces

méthodes dans [16, 43] établit que la méthode centralisée est la plus puissante des quatre en termes de contrôle et observation et que la méthode à distance est la plus faible. Par ailleurs, les auteurs établissent dans [44] les contraintes temporelles qui garantissent une équivalence entre la méthode centralisée et la méthode répartie dans le cas du test d'applications n'imposant pas de contraintes temporelles de fonctionnement à leur environnement. Notons enfin que ces architectures sont conçues pour un test de type boîte noire, vu que le système distribué est considéré comme une seule entité.

Architecture de test pour la conformité et l'interopérabilité

- Architecture à cheval [124](Astride responder). Elle correspond à la première architecture de test d'interopérabilité proposée par Omar Rafiq et Richard Castanet. Elle consiste en un testeur centrale pour le test d'un système réparti. Le contrôle et l'observation des SUTs sont assurés par des PCOs entre TS et SUT. L'architecture définit aussi des POs implantés entre les SUTs.

- Architecture Générique [150]. Une architecture générique pour la conformité, l'interopérabilité et la qualité de service est présentée dans [150]. Cette architecture propose une boîte-à-outils permettant d'exprimer les propriétés spécifiques d'une application ou d'un système réparti à tester. Elle se compose de plusieurs instances pour différents types de composantes :

1. Implantation sous test (IUT), i.e. une partie du système à tester.
2. Interface de composante (IC) permet d'interfacer les différentes IUTs. Les ICs sont différentes des PCOs, du fait qu'ils n'accèdent pas directement aux IUTs (cas de composantes imbriquées).
3. Composante de test (TC) associée à une IUT, permet de formuler des verdicts après une coordination avec les autres TCs.
4. Composante de contrôle (CC) permet d'établir l'environnement spécifique pour l'exécution des tests.

Les différentes composantes et propriétés d'un système peuvent alors être exprimées grâce à cette boîte à outils.

8.5.4 Méthodologie pour l'interopérabilité

Contrairement au test de conformité, aucune méthodologie standard du test d'interopérabilité n'a vu le jour, du fait de la confusion autour du test d'interopérabilité et de la difficulté à définir les types de test suffisamment objectifs pour être certifiés. En revanche, deux approches majeures ont été développées et appliquées [126]. Une approche développée par SPAG [140] (the Standard Promotion and Application Group) appelée PSI (Process

for System Interoperability). Dans cette approche, le test d'interopérabilité est vu comme une étape supplémentaire qui suit le test de conformité basé sur un SIS (System Interoperability Statement). L'autre approche utilisée par EuroSInet [127] consiste à soumettre les implantations des différents constructeurs à un organisme dont le rôle est de vérifier d'abord la conformité de chaque implantation. Les différents vendeurs de ces implantations se mettent d'accord et définissent un ensemble de scénarios à tester. L'organisme de test établit une matrice de résultats contenant pour chaque composante, les différentes composantes avec lesquelles elle a été testée et elle va être testée.

Chapitre 9

Cadre formel pour le test d'interopérabilité

Sommaire

9.1 Préliminaires **141**
9.2 Définitions de l'interopérabilité **142**
9.3 Dérivation de cas de test d'interopérabilité **145**
9.3.1 Hypothèses de test 146
9.3.2 Algorithme de génération 146
9.4 Génération de cas de test d'interopérabilité à partir d'outils de conformité **147**
9.5 Comparaison **148**

Dans ce chapitre, nous proposerons un cadre théorique pour le test d'interopérabilité des systèmes sans contraintes temporelles. Ce cadre englobe des définitions de l'interopérabilité, ainsi qu'une méthode de génération de cas de test d'interopérabilité.

9.1 Préliminaires

Étant donné un système communicant S composé d'un ensemble de composantes (M_i), chaque composante M_i définit un ensemble d'actions A^{M_i} et un ensemble de points d'interactions (ports ou interfaces) P^{M_i} à travers lesquels M_i communique avec les autres composantes, ainsi que son environnement.

Spécifications. Nous supposons que chaque composante M_i peut être modélisée par un IOLTS $(Q^{M_i}, q_0^{M_i}, \Sigma^{M_i}, \rightarrow_{M_i})$ tel que chaque événement de Σ^{M_i} correspond à une action de A^{M_i} sur une interface de P^{M_i} :

- $\Sigma^{M_i} \subseteq P^{M_i} \times A^{M_i}$ avec P_i^M un ensemble fini d'interfaces (ports) de communication et A^{M_i} l'alphabet des actions échangées par M_i sur P^{M_i}. Σ^{M_i} est partitionné en deux ensembles : $\Sigma^{M_i} = \Sigma_i^{M_i} \cup \Sigma_o^{M_i}$

Pour tout $(p, a) \in \Sigma^{M_i}$, $(p, a) \in \Sigma_i^{M_i}$ si a est une action d'entrée et $(p, a) \in \Sigma_o^{M_i}$ si a est une action de sortie. Par la suite, $p?a$ dénotera $(p, a) \in \Sigma_i^{M_i}$, $p!a$ dénotera $(p, a) \in \Sigma_o^{M_i}$, $\bar{\mu} = p!a$ si $\mu = p?a$ et $\bar{\mu} = p?a$ si $\mu = p!a$.

Implantations. Nous supposons que chaque implantation I_i de M_i peut être modélisée par un IOLTS $(Q^{I_i}, q_0^{I_i}, \Sigma^{I_i}, \rightarrow_{I_i})$ tel que chaque événement de Σ^{I_i} correspond à une action de A^{I_i} sur une interface $P^{I_i} \subseteq P^{M_i}$.

Notations. Rappelons que $\sigma.\sigma'$ est la concaténation de deux traces σ et σ', $\sigma_{|L}$ est la projection de σ sur l'alphabet L et que $M_1 \| M_2$ est la composition synchrone de deux IOLTSs. Pour définir formellement la notion d'interopérabilité, nous aurons besoin des notations additionnelles suivantes : soient $X \subseteq P^{M_i}$, $L \subseteq \Sigma^{M_i}$, et $\sigma \in Traces(M_i)$.

- $Out_L(M_i, \sigma) = Out(M_i, \sigma) \cap L$.
- M_i **visible** X est l'IOLTS $(Q^{M_i}, q_0^{M_i}, \Sigma^{M_i \, \mathbf{visible} \, X}, \rightarrow_{M_i \, \mathbf{visible} \, X})$ tel que
  1. $\Sigma^{M_i \, \mathbf{visible} \, X} = \{(p, a) \in \Sigma^{M_i} | p \in X\}$
  2. $\rightarrow_{M_i \, \mathbf{visible} \, X}$ est la plus petite relation définie par $(\mu \in \Sigma_\tau^{M_i})$:
     (a) $q \xrightarrow{\mu}_{M_i} q'$, $\mu \notin \Sigma^{M_i \, \mathbf{visible} \, X}$ $\Rightarrow$ $q \xrightarrow{\tau}_{M_i \, \mathbf{visible} \, X} q'$
     (b) $q \xrightarrow{\mu}_{M_i} q'$, $\mu \in \Sigma^{M_i \, \mathbf{visible} \, X}$ $\Rightarrow$ $q \xrightarrow{\mu}_{M_i \, \mathbf{visible} \, X} q'$
- $M_i|X = (Q^{M_i|X}, q_0^{M_i|X}, \Sigma^{M_i|X}, \rightarrow_{M_i|X})$ est l'IOLTS M_i **visible** X après déterminisation :
  1. $Q^{M_i|X} = 2^{Q^{M_i}}$
  2. $\Sigma^{M_i|X} = \Sigma^{M_i \, \mathbf{visible} \, X}$
  3. $q_0^{M_i|X} = q_0^{M_i} \, after \, \epsilon$.
  4. $\rightarrow_{M_i|X}$ est définie par :
     $(q, a, q') \in \rightarrow_{M_i|X}$ si $q' = q \, after \, a$ avec $q, q' \in 2^{Q^{M_i}}$ et $a \in \Sigma^{M_i|X}$.
- $P_{M_i}(\sigma, X) = \{\sigma' \in Traces(M_i) \mid \sigma_{|\Sigma^{M_i|X}} = \sigma'_{|\Sigma^{M_i|X}}\}$ l'ensemble des traces de M_i qui coïncident avec σ sur X.

Par la suite, $\mathbf{M_1 \|_X M_2}$ dénotera $(\mathbf{M_1 \| M_2})|\mathbf{X}$.

9.2 Définitions de l'interopérabilité

Pour décider de l'exactitude d'une implantation, un critère clair est nécessaire. Dans le contexte du test de conformité, nous avons vu plusieurs relations de conformité qui

définissent clairement la notion d'exactitude. Malheureusement, l'interopérabilité n'a pas connu une telle formalisation.

Le point d'entrée du test d'interopérabilité est la donnée des spécifications, des implantations de ces spécifications définissant quelques interfaces accessibles et un critère qui doit être vérifié par ces implantations. Ainsi, deux implantations interopèrent si elles vérifient le critère d'interopérabilité. Le cadre présenté ici est basé sur la comparaison entre les sorties du système à tester et les sorties prévues par les spécifications modulo des projections sur les interfaces accessibles.

Soient (I_i) des implantations, (M_i) des spécifications, $i \in \{1, 2\}$ et $X \subseteq P^{M_1} \cup P^{M_2}$ un ensemble d'interfaces.

Définition 42 $\mathbf{interop_X}(I_1, I_2) =_\Delta \forall \sigma \in Trace(M_1 \|_X M_2), \Rightarrow$
$Out_{\Sigma^{I_1}}(I_1 \|_X I_2, \sigma) \subseteq \bigcup_{\sigma' \in P_{M_1 \| M_2}(\sigma, X)} Out_{\Sigma^{M_1|X}}(M_1, \sigma'_{|\Sigma^{M_1}})$ □

La première relation définie est la relation $\mathbf{interop_X}$. Elle établit que, durant l'interaction entre I_1 et I_2 lorsqu'une trace σ de $M_1 \|_X M_2$ est injectée au système sous test, les comportements observables du système projetés sur les interfaces accessibles de I_1 sont inclus dans l'union des comportements observables (projetés sur les interfaces accessibles) de M_1 lors de l'application de la projection sur X de toute trace $\sigma' \in Traces(M_1 \| M_2)$ (σ' coïncide avec σ sur X).

La relation $\mathbf{interop_X}$ est paramétrée par les interfaces accessibles de l'implantation. Un cas intéressant est lorsque toutes ces interfaces sont accessibles (test boîte grise) : $X = P^{M_1} \cup P^{M_2}$. Dans ce cas, pour $\sigma \in M_1 \| M_2$, nous avons $P_{M_1 \| M_2}(\sigma, P^{M_1} \cup P^{M_2}) = \{\sigma\}$. La relation $\mathbf{interop_X}$ s'écrit donc comme suivant (l'indice $_X$ est omis) :

$$\mathbf{interop}(I_1, I_2) =_\Delta \forall \sigma \in Trace(M_1 \| M_2) \Rightarrow Out_{\Sigma^{I_1}}(I_1 \| I_2, \sigma) \subseteq Out(M_1, \sigma_{|\Sigma^{M_1}}).$$

Exemple 9.1 *Considérons les deux IOLTS M_1 et M_2 de la* FIG.*42. I_1 et I_2 deux implantations de M_1 et M_2 respectivement. Supposons que $X = \{p1, p2, p3\}$. On remarque que interop(I_1, I_2). En effet,*

- $Traces(M_1 \| M_2) = \{p1?a, p1?a.p2!x, p1?a.p2!x.p3!b\}$
- *Pour* $\sigma = p1?a$, *on a :* $Out_{\Sigma^{I_1}}(I_1 \| I_2, \sigma) = \emptyset \subseteq Out(M_1, \sigma_{|\Sigma^{M_1}}) = \{p2!x\}$
- *Pour* $\sigma = p1?a.p2!x$, *on a* $Out_{\Sigma^{I_1}}(I_1 \| I_2, \sigma) = \emptyset \subseteq Out(M_1, \sigma_{|\Sigma^{M_1}}) = \emptyset$
- *Pour* $\sigma = p1?a.p2!x.p3!b$, *on a* $Out_{\Sigma^{I_1}}(I_1 \| I_2, \sigma) = \emptyset \subseteq Out(M_1, \sigma_{|\Sigma^{M_1}}) = \emptyset$

Ainsi I_1 interopère avec I_2. Cependant $\neg$ interop(I_2, I_1). En effet,

- *Pour* $\sigma = p1?a$, *on a :* $Out_{\Sigma^{I_2}}(I_1 \| I_2, \sigma) = \{p2!y\} \not\subseteq Out(M_2, \sigma_{|\Sigma^{M_2}}) = \emptyset$.

En conclusion, I_1 interopère avec I_2 mais I_2 n'interopère pas avec I_1. □

Maintenant, soit $Y = P^{M_1} \cap P^{M_2}$ l'ensemble des interfaces communes à M_1 et M_2.

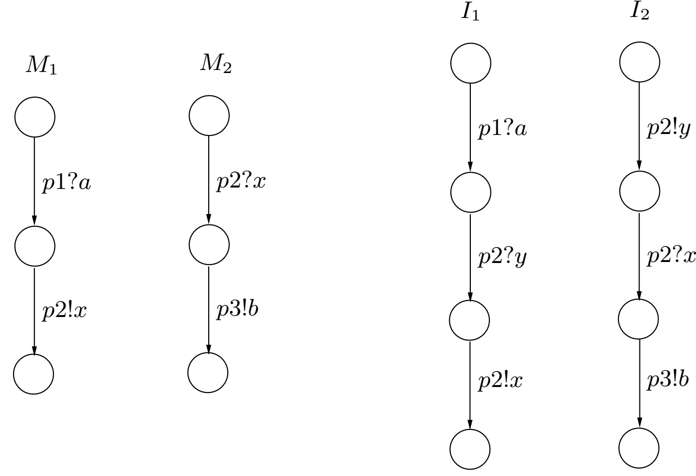


FIG. 42: Relation interop.

Définition 43 $\mathbf{intercom_X}(I_1, I_2) =_\Delta \forall \sigma \in Trace(M_1\|_X M_2),\ \forall \sigma' \in P_{M_1\|M_2}(\sigma, X),\ \sigma'_{|Y} \neq \epsilon \Rightarrow$

$$Out_{\Sigma^{I_1}}(I_1\|_X I_2, \sigma) \subseteq \cup_{\sigma' \in P_{M_1\|M_2}(\sigma,X)} Out_{\Sigma^{M_1|X}}(M_1, \sigma'_{|\Sigma^{M_1}})$$ □

$\mathbf{intercom_X}$ est similaire à $\mathbf{interop_X}$, mais ne fait intervenir que les communications inter-composantes. Généralement, les composantes sont testées à la conformité et ainsi $\mathbf{intercom_X}$ évite de re-tester la conformité des composantes. Lorsque toutes les interfaces sont accessibles (test boîte grise), $\mathbf{intercom_X}$ s'écrit comme suivant (l'indice $_X$ est omis) :

$$\mathbf{intercom}(I_1, I_2) =_\Delta \forall \sigma \in trace(M_1\|M_2)\ \sigma_{|Y} \neq \epsilon$$
$$\Rightarrow Out_{\Sigma^{I_1}}(I_1\|I_2, \sigma) \subseteq Out(M_1, \sigma_{|\Sigma^{M_1}}).$$

Finalement, remarquons que les deux relations définies ne dépendent pas de l'architecture de test. Comme résultat, le lemme suivant montre l'équivalence entre $\mathbf{interop_X}$ et **ioconf**.

Lemme 11 *Supposons que M_1, M_2, I_1 et I_2 sont input-complets et que $\Sigma_o^{M_1} \cap \Sigma_o^{M_2} = \emptyset$, $\Sigma_o^{M_1} \cap \Sigma_o^{I_2} = \emptyset$, $\Sigma_o^{I_1} \cap \Sigma_o^{M_2} = \emptyset$, $\Sigma_o^{I_1} \cap \Sigma_o^{I_2} = \emptyset$. Alors :*

$$\mathbf{interop_X}(I_1, I_2) \wedge \mathbf{interop_X}(I_2, I_1) \ \equiv\ I_1\|_X I_2 \ \mathbf{ioconf}\ M_1\|_X M_2$$

Preuve. Voir Annexe B. □

9.3 Dérivation de cas de test d'interopérabilité

Dans cette section, nous allons montrer comment générer des cas de test pour vérifier la relation $interop_X$ dans le cas d'un système composé de deux entités M_1 et M_2 définissant un ensemble d'interfaces accessibles X. L'approche présentée utilise un objectif de test TP pour le système $M_1\|_X M_2$. L'idée est alors de décomposer TP en deux objectifs de test TP_1 et TP_2 pour M_1 et M_2 respectivement et de générer deux cas de test TC_1 et TC_2 pour TP_1 et TP_2 respectivement. Cependant, la composition de TC_1 et TC_2 ne vérifie pas en général TP. Pour cette raison, nous définissons des compositions particulières des objectifs de test et des cas de test (pc et tc respectivement).

Définition 44 *Soient $TP_1 = (Q^{TP_1}, q_0^{TP_1}, \Sigma^{TP_1}, \rightarrow_{TP_1})$ et $TP_2 = (Q^{TP_2}, q_0^{TP_2}, \Sigma^{TP_2}, \rightarrow_{TP_2})$ deux objectifs de test pour une spécification M. La composition synchrone de TP_1 et TP_2 selon pc, notée $TP = TP_1 \; pc \; TP_2$, est l'objectif de test $TP = (Q^{TP}, q_0^{TP}, \Sigma^{TP}, \rightarrow_{TP})$ défini par :*

- $q_0^{TP} = (q_0^{TP_1}, q_0^{TP_2})$
- $Q^{TP} = \{ (q_1, q_2) \,|\, q_1 \in Q^{TP_1},\ q_2 \in Q^{TP_2} \}$
- $\Sigma^{TP} \subseteq \Sigma^{TP_1} \cup \Sigma^{TP_2}$ *et* $\rightarrow_{TP}$ *sont obtenus comme suivant* ($\mu \in \Sigma_\tau^{TP}$) :
  1. $q_1 \xrightarrow{\mu}_{TP_1} q_1',\ \mu \notin \Sigma^{TP_2},\ q_2 \notin Reject^{TP_2} \quad \Rightarrow \quad q_1\|q_2 \xrightarrow{\mu}_{TP} q_1'\|q_2$
  2. $q_2 \xrightarrow{\mu}_{TP_2} q_2',\ \mu \notin \Sigma^{TP_1},\ q_1 \notin Reject^{TP_1} \quad \Rightarrow \quad q_1\|q_2 \xrightarrow{\mu}_{TP} q_1\|q_2'$
  3. $q_1 \xrightarrow{\mu}_{TP_1} q_1',\ q_2 \xrightarrow{\mu}_{TP_2} q_2' \quad \Rightarrow \quad q_1\|q_2 \xrightarrow{\mu}_{TP} q_1'\|q_2'$. □

Les états $Reject^{TP}$ et $Accept^{TP}$ sont définis par :

- $Reject^{TP} = \{(q_1, q_2) \in Q^{TP} \,|\, q_1 \in Reject^{TP_1}$ *ou* $q_2 \in Reject^{TP_2}\}$
- $Accept^{TP} = \{(q_1, q_2) \in Q^{TP} \,|\, q_1 \in Accept^{TP_1}$ *et* $q_2 \in Accept^{TP_2}\}$.

Définition 45 *Soient $TC_1 = (Q^{TC_1}, q_0^{TC_1}, \Sigma^{TC_1}, \rightarrow_{TC_1})$ et $TC_2 = (Q^{TC_2}, q_0^{TC_2}, \Sigma^{TC_2}, \rightarrow_{TC_2})$ deux cas de test pour une spécification M. La composition synchrone de TC_1 et TC_2 selon tc, notée $TC = TC_1 \; tc \; TC_2$, est le cas de test $TC = (Q^{TC}, q_0^{TC}, \Sigma^{TC}, \rightarrow_{TC})$ défini par :*

- $q_0^{TC} = (q_0^{TC_1}, q_0^{TC_2})$
- $Q^{TC} = \{ (q_1, q_2) \,|\, q_1 \in Q^{TC_1},\ q_2 \in Q^{TC_2} \}$
- $\Sigma^{TC} \subseteq \Sigma^{TC_1} \cup \Sigma^{TC_2}$ *et* $\rightarrow_{TC}$ *sont obtenus comme suivant* ($\mu \in \Sigma_\tau^{TC}$) :
  1. $q_1 \xrightarrow{\mu}_{TC_1} q_1',\ \mu \notin \Sigma^{TC_2},\ q_2 \notin Fail^{TC_2} \cup Inc^{TC_2} \quad \Rightarrow \quad q_1\|q_2 \xrightarrow{\mu}_{TC} q_1'\|q_2$
  2. $q_2 \xrightarrow{\mu}_{TC_2} q_2',\ \mu \notin \Sigma^{TC_1},\ q_1 \notin Fail^{TC_1} \cup Inc^{TC_1} \quad \Rightarrow \quad q_1\|q_2 \xrightarrow{\mu}_{TC} q_1\|q_2'$
  3. $q_1 \xrightarrow{\mu}_{TC_1} q_1',\ q_2 \xrightarrow{\mu}_{TC_2} q_2' \quad \Rightarrow \quad q_1\|q_2 \xrightarrow{\mu}_{TC} q_1'\|q_2'$. □

Les ensembles $Fail^{TC}$, Inc^{TC} et $Pass^{TC}$ sont définis par :

- $Fail^{TC} = \{(q_1, q_2) \in Q^{TC} \,|\, q_1 \in Fail^{TC_1}$ *ou* $q_2 \in Fail^{TC_2}\}$

- $Inc^{TC} = \{(q_1, q_2) \in Q^{TC} \mid q_1 \in Inc^{TC_1} \text{ ou } q_2 \in Inc^{TC_2}\}$
- $Pass^{TC} = \{(q_1, q_2) \in Q^{TC} \mid q_1 \in Pass^{TC_1} \text{ et } q_2 \in Pass^{TC_2}\}$.

9.3.1 Hypothèses de test

Nous supposons que TP est un objectif de test de $M_1 \| M_2$[1] modélisé par un IOLTS. Nous supposons aussi que les deux spécifications M_1 et M_2 interopèrent, i.e. pour toute $\sigma \in Traces(M_1)$ (resp. $\sigma \in Traces(M_2)$), il existe $\sigma' \in Trace(M_1 \| M_2)$ telle que $\sigma = \sigma'_{|\Sigma^{M_1}}$ (resp. $\sigma = \sigma'_{|\Sigma^{M_2}}$).

9.3.2 Algorithme de génération

Soit $Y = P^{M_1} \cap P^{M_2}$ les interfaces de communication communes à M_1 et M_2. L'algorithme de génération des cas de test pour le système $M_1 \|_X M_2$ est présenté dans la figure 43.

Input : Trois IOLTS M_1, M_2 et TP ; des interfaces accessibles $X \subseteq P^{M_1} \cup P^{M_2}$
Output : Cas de test de $M_1 \|_X M_2$.

Début

1. Calcul des projections : $TP_1 = TP|P^{M_1}$ et $TP_2 = TP|P^{M_2}$.
2. Calcul d'un cas de test vérifiant TP_1 dans $M_1 \Rightarrow TC_1$.
3. Suppression des branches de TC_1 qui mènent vers les verdicts $Fail$, ou Inc et remplacement des étiquettes $Pass$ par *Accept*. $\Rightarrow T_1$.
4. Calcul de la projection de T_1 sur Y : $T_2 = T_1|Y$.
5. Réalisation de la composition d'objectifs de test de TP_2 et T_2 : $TP'_2 = TP_2 \, pc \, T_2$
6. Addition à TP'_2, dans chaque état, des synchronisations non envisagées avec M_1. Ces synchronisations mèneront à des états *Reject*.
7. Calcul d'un de test vérifiant TP'_2 dans $M_2 \Rightarrow TC_2$.
8. Réalisation de la composition de cas de test de TC_1 et TC_2 : $TC = TC_1 \, tc \, TC_2$ en respectant l'ordre d'apparition des événements dans TP.
9. Calcul de la projection $TC|X$.
10. Sélection des cas de test.

Fin

FIG. 43: Algorithme de génération de cas de test d'interopérabilité.

[1] TP peut contenir des événements de M_1 ou de M_2 ou des deux.

À partir de l'objectif de test TP, on construit un objectif de test TP_1 (resp. TP_2) pour la spécification M_1 (resp. M_2) en calculant la projection de TP sur les interfaces P^{M_1} (resp. P^{M_2}) (ligne 1). TP_1 est alors utilisé pour générer un cas de test TC_1 pour M_1 (ligne 2). Les branches de TC_1 qui mènent à un verdict $Fail$ ou Inc sont alors supprimées ; les états $Pass$ deviennent des états $Accept$ de T_1 (ligne 3) (T_1 devient un objectif de test). TC_1 peut amener des synchronisations avec M_2 non explicites dans TP et par la suite ne figurent pas dans TP_2. Ainsi, la projection de T_1 (ligne 4) sur les interfaces communes à M_1 et M_2 contient donc toutes les synchronisations entre ces derniers. Le calcul de la composition synchrone des objectifs de test TP_2 et T_2 (ligne 5) permet de tenir compte des synchronisations invoquées dans TC_1 et TP. Dans TP_2', on interdit (ligne 6) toute synchronisation avec M_1 non explicite dans TP_2' pour éviter, lors du calcul d'un cas de test respectant TP_2' (ligne 7), l'apparition de synchronisations non prévues dans TC_1. À ce stade là, TC_1 et TC_2 sont deux cas de test pour M_1 et M_2 qui respectent l'objectif de test TP. Les lignes 8, 9 et 10 calculent un cas de test global pour $M_1\|_X M_2$.

Remarque 11 *Dans le cas où TP ne contient que des événements d'une seule entité, par exemple M_1, TP_2 peut être vide i.e. $TP_2 = \emptyset$. Dans ce cas, l'algorithme transforme un cas de test de conformité en un cas de test d'interopérabilité.* □

Complexité. La complexité de l'algorithme dépend de la taille (nombre de transitions) de M_1, M_2 et TP : $O(taille(TP) * (taille(M_1) + taille(M_2)) + taille(TP)^2)$.

9.4 Génération de cas de test d'interopérabilité à partir d'outils de conformité

Dans le cas de la conformité, il existe plusieurs outils de génération de cas de test (TGV [57], TorX [15],...). Pour générer des cas de test d'interopérabilité de M_1 et M_2 avec ses outils, une première solution consiste à construire le système global par composition de M_1 et M_2. L'inconvénient de cette solution est que la complexité de génération est en $O(taille(M_1) \times taille(M_2) \times taille(TP))$. L'autre solution que nous proposons est d'utiliser l'algorithme ci-dessus en ajoutant un module externe qui réalise les deux compositions tc et pc.

Exemple d'application. Considérons les deux IOLTS M_1 et M_2 de la FIG.44. M_1 et M_2 partagent les interfaces $p2$ et $p3$. Supposons que $X = \{p1, p2, p3, p4\}$. TP est un objectif de test de $M_1\|M_2$ qui consiste à tester l'émission de y par M_2 suivie de l'émission de b par M_1 puis d'émission de d par ce dernier. Les différentes étapes de génération d'un cas de test qui vérifie TP sont illustrées dans la FIG.45. Les deux projections de TP sur l'alphabet de M_1 et M_2 sont réalisées. Notons que la projection de $p3?b$ sur M_1 donne $p3!b$. TC_1 et M_1 sont

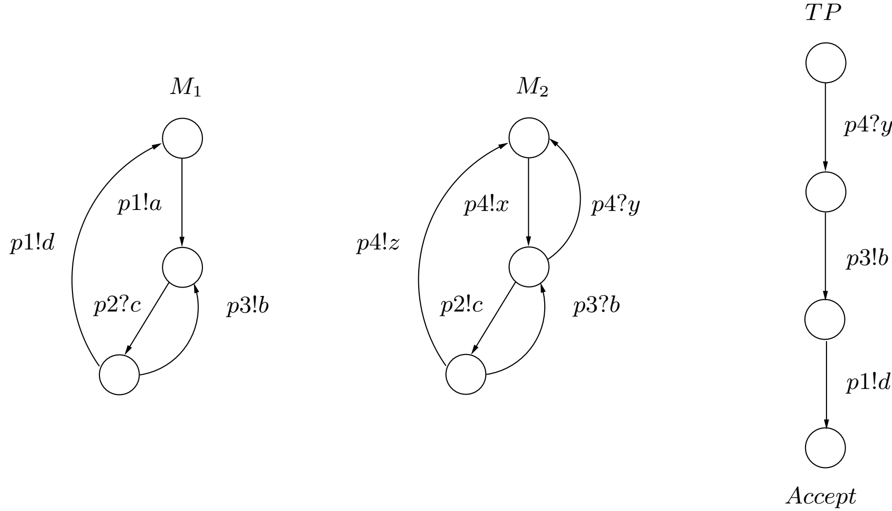


FIG. 44: Exemple d'application.

alors utilisés pour générer un cas de test qui vérifie TC_1. TC_1 amène la synchronisation $p2?c$ non explicite dans TP. Ainsi, la composition de T_2 et TP_2 permet de tenir compte de cette nouvelle synchronisation dans TP_2'. Ce dernier, par l'ajout des états *Reject* interdit toute autre synchronisation avec M_2 non prévue dans T_2 et TP_2. TP_2' est alors utilisé pour générer un cas de test de M_2. Remarquons que TC_1 et TC_2 vérifient TP.

9.5 Comparaison

Cette comparaison porte essentiellement sur travaux relatifs à la définition de l'interopérabilité, et la génération de cas de test d'interopérabilité.

Définition de l'interopérabilité. Dans [12], les auteurs proposent 9 relations d'interopérabilité basées sur l'architecture de test utilisée pour les systèmes asynchrones. Ces relations se décomposent en trois classes qui dépendent des interfaces d'implantations considérées. Ces relations peuvent être obtenues à partir de la relation $\mathbf{interop_X}$ en définissant l'ensemble des interfaces considérées X. Par exemple, la relation *interop* est équivalente à la relation unilatérale totale définie dans [12]. Tretmans et al. [31] montrent que, sous certaines hypothèses, la relation **ioco** de conformité est convenable pour le test d'interopérabilité. Dans [90], deux implantations I_1 et I_2 sont conformes en interopérabilité à deux spécifications S_1 et S_2 si I_1 (resp. I_2) est conforme à S_1 (resp. I_2) et $I_1\|I_2$ est conforme à $S_1\|S_2$. La relation de conformité est alors définie en terme d'inclusion de la partie observable des traces de l'implantation dans la partie observable des traces de la spécification. La méthode de génération proposée consiste à calculer les testeurs canoniques T_1 et T_2 de chaque spécification. Le testeur du système $S_1\|S_2$ est alors $T_1\|T_2$.

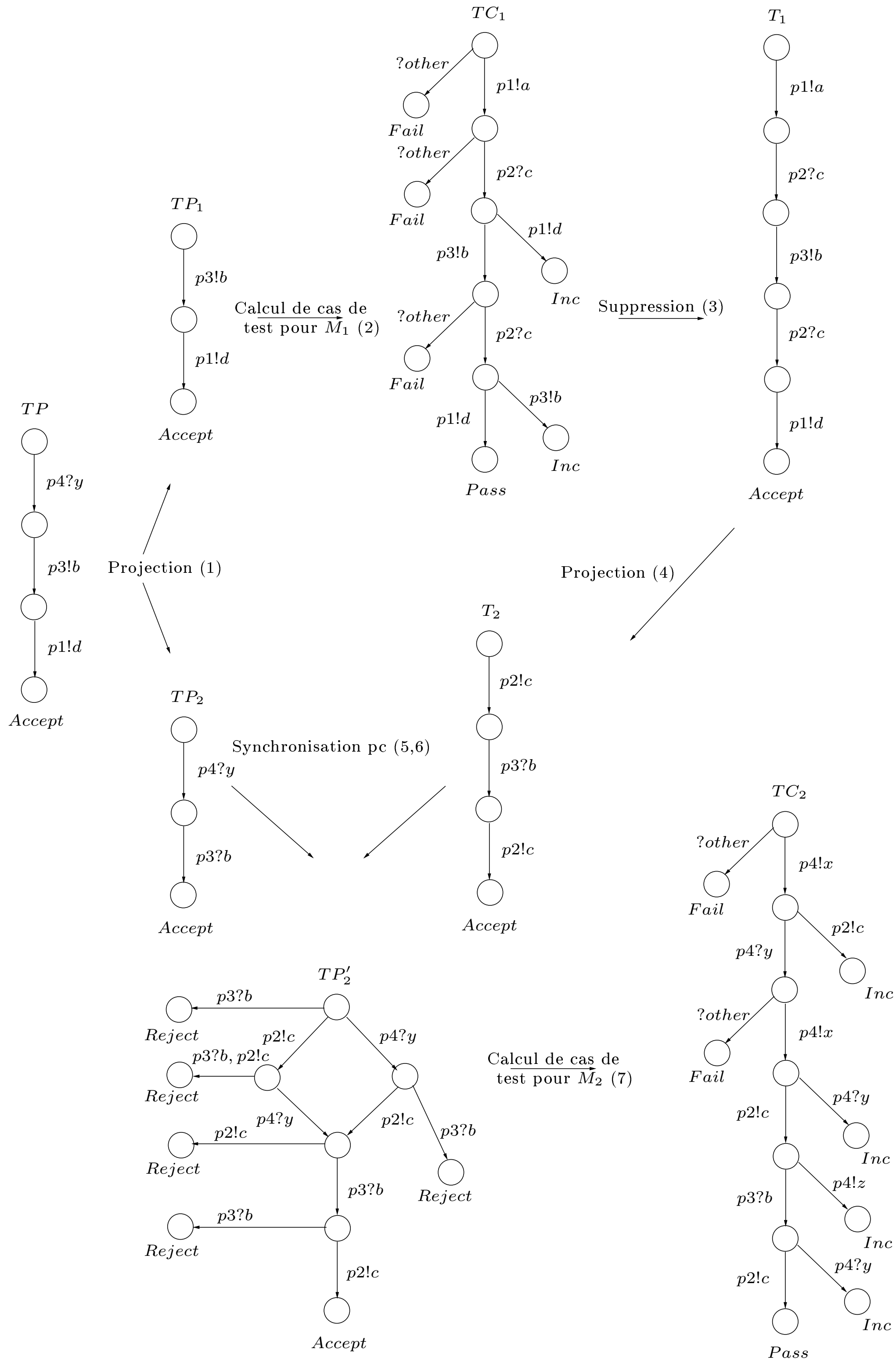


FIG. 45: Exemple de génération.

Méthodes de génération. La majorité des méthodes du test d'interopérabilité proposées sont basées sur l'analyse d'accessibilité. [124] est l'un des premiers articles sur l'interopérabilité. L'approche adoptée se base sur le calcul du graphe d'accessibilité. Dans [8], les différentes entités communicantes sont considérées comme une seule entité (boîte noire). L'idée des auteurs est d'interfacer un outil de test de conformité (TVEDA [123]) et un outil de test d'interopérabilité (SDE/TCGEN [7]), dans le but de générer des tests de conformité et d'interopérabilité. La dérivation des tests de conformité (TVEDA) et d'interopérabilité (TCGEN) se fait séparément puis, à la main, les auteurs les combinent pour réduire le nombre de test. L'inconvénient est que la méthode est non automatique et ne permet pas de déterminer la cause de la non interopérabilité (boîte noire). [147] présente une expérience de test d'interopérabilité avec le protocole FTAM. La méthode présentée dans [119] identifie les états et les transitions du système sous test. Elle consiste à calculer un graphe du comportement global du système suivant des règles de synchronisation définies. Pour générer les cas de test, pour chaque transition à identifier dans le système global, le préambule est calculé à partir du système global tandis que le posteambule est la concaténation de deux séquences d'identification, des deux états locaux constituant l'état global, calculées séparément dans chaque spécification. L'avantage est la détection des erreurs de transfert, mais la méthode se confronte à l'explosion combinatoire des nombres d'état du graphe du comportement global. Kang, Kim et al. ont proposé un ensemble de travaux relatifs à l'interopérabilité [85, 86, 136, 133, 87, 130, 134, 142, 135]. Dans [85, 86, 87], ils proposent des techniques de génération pour les protocoles symétriques basées sur le calcul du comportement global. [135] traite la dérivation de cas de test d'interopérabilité pour la partie contrôle et données des protocoles. Une suite de tests squelette pour la partie contrôle est d'abord générée. Chaque cas de test est alors paramétré. La suite est alors complétée en affectant des valeurs aux paramètres. L'approche adoptée dans [65] est une approche "un contre N" dans laquelle une seule entité interopère avec le reste du système communicant. L'approche est alors appliquée au système VoIP. Pour éviter l'explosion combinatoire des nombres de cas de test, [91] propose un approche basée sur l'utilisation d'un objectif de test. Le principe consiste à définir des règles de synchronisation pour la composition parallèle : pour deux spécifications et un objectif de test, les auteurs construisent à la volée un chemin qui respecte les trois spécifications. Ceci en respectant les règles de synchronisation et en faisant une recherche en profondeur.

Comparé aux autres travaux, le cadre introduit dans ce chapitre considère aussi bien la définition de l'interopérabilité que la dérivation des cas de test. Notre approche consiste utiliser des outils de génération des tests de conformité pour générer des test d'interopérabilité. De plus, la complexité de génération est inférieure à la complexité des approches construisant l'automate de la spécification globale.

Chapitre 10

Cadre formel pour le test de conformité

Sommaire

10.1 Introduction **152**
10.2 Définitions de la conformité **152**
10.2.1 Rappels 152
10.2.2 Relation $\leq_{ttr}$ 153
10.2.3 Relation $ioconf_t$ 154
10.2.4 Relation $ioco_t$ 155
10.3 Cas de test et testeurs **156**
10.3.1 Test adaptatif et statique 157
10.3.2 Formalisme de cas de test 157
10.4 Dérivation de cas de test abstraits et squelettes **159**
10.4.1 Hypothèses de test 159
10.4.2 Automate de simulation 160
10.4.3 Cas de test abstraits et squelettes 161
10.5 Dérivation de tests digitaux **162**
10.6 Dérivation de tests analogiques **165**
10.7 Comparaison **168**
10.7.1 Définition de la conformité 168
10.7.2 Dérivation de tests 168

Dans ce chapitre, nous proposons un cadre théorique pour le test de conformité des systèmes temporisés. Ce cadre comportent des définitions de la conformité ainsi que des méthodes de génération de cas de test temporisés. Ce chapitre utilise quelques notions introduites dans le chapitre 4 et la section 7.2.

10.1 Introduction

Les systèmes temps-réel (RTS) trouvent des applications dans différents domaines. Nous les trouvons aussi bien dans la vie de chaque jour, comme les systèmes téléphoniques et les systèmes vidéo, que dans les hôpitaux pour le guidage des patients et dans les aéroports pour le contrôle du trafic aérien. Tous ses systèmes sont sensibles au temps. Ainsi, leur comportement ne dépende pas que du résultat logique des calculs mais aussi des instants où les entrées sont réalisées et des instants où les résultats sont produits. Il est connu dans la communauté de recherche que les dysfonctionnements des RTSs sont généralement dûs à la violation des contraintes temporelles qui gouvernent le comportement du système. De tels dysfonctionnements peuvent avoir des conséquences catastrophiques sur la vie humaine et l'environnement. Par conséquent, il est important de s'assurer que les implantations des RTSs soient sans erreur avant leur déploiement.

Nous nous intéressons dans ce chapitre au test de conformité boîte noire. Durant la dernière décennie, plusieurs recherches ont été menées dans le test des RTSs et ont donné différents méthodes basées sur différents modèles formels temporisés et différentes applications possibles. Toutes ces méthodes génèrent avec succès des cas de test temporisés mais la plupart d'entre elles souffrent de l'explosion du nombre des cas de test générés. En conséquence, la motivation de développer des nouvelles méthodes de génération est encore d'actualité.

Spécifications. Nous supposons que la spécification du RTS à tester est donnée sous forme d'un TIOA A observable, non-bloquant et événementiellement déterministe.

Implantations. Nous supposons que l'implantation peut être modélisée par un TIOA I input-complet. Notons que nous n'exigeant pas que I soit connue, mais simplement qu'elle existe. La condition d'input-complétude est requise, ainsi l'implantation peut accepter les entrées du testeur dans chaque état (elle peut ignorer les entrées illégales ou se déplacer vers un état d'erreur.)

10.2 Définitions de la conformité

Afin de définir formellement la notion de conformité entre I et A, nous proposons dans cette partie une extension temporisée des relations de conformité $\leq_{tr}$, $ioconf$ et $ioco$.

10.2.1 Rappels

Soient $A = (S, s_0, \Sigma, C, \rightarrow)$ un TIOA, $Q^A = (Q, q_0, \Gamma, \rightarrow_A)$ la sémantique de A, $a \in \Sigma$, $q, q' \in Q$, $d \in \mathbb{R}^{\geq 0}$ et $\sigma = (a_1, d_1)...(a_n, d_n)$ une séquence temporisée ($\sigma \in TW(\Sigma)$). Dans la section 4.3.4, nous avons défini :

1. $Run(A)$: l'ensemble des séquences temporisées admettant une computation de A défini par :

$$Run(A) = \{\sigma \mid A \textit{ admet une computation sur } \sigma \in TW(\Sigma_\tau)\}.$$

2. $TTrace(A)$: l'ensemble des traces temporisées de A défini par :

$$TTrace(A) = \{\sigma \mid \exists \sigma' \in Run(A),\, \sigma = \sigma'_{|\Sigma}\}.$$

Dans la section 4.2, nous avons défini :

1. $q \overset{a}{\Rightarrow} q' \quad \triangleq \quad$ il existe $q'' \in Q$ tel que $q \xrightarrow{\tau}_A^* q'' \xrightarrow{a}_A q'$.
2. $q \overset{\epsilon(d)}{\Rightarrow} q' \quad \triangleq \quad$ il existe une exécution d'attente $q = q_0 \xrightarrow{\alpha_1}_A q_1 \xrightarrow{\alpha_2}_A q_2 \cdots \xrightarrow{\alpha_n}_A q_n = q'$ telle que $d = \sum\{d_i \mid \alpha_i = \epsilon(d_i)\}$.
3. $q \overset{\sigma}{\Rightarrow} q' \quad \triangleq \quad$ il existe une exécution $q = q_0 \overset{\epsilon(d'_1)}{\Rightarrow} q'_1 \overset{a_1}{\Rightarrow} q_1 \cdots \overset{\epsilon(d'_n)}{\Rightarrow} q'_n \overset{a_n}{\Rightarrow} q_n = q'$ telle que $d_i = \sum_{1 \le j \le i} d'_j$, pour tout $i \in [1, n]$.

où la relation $\xrightarrow{\tau}_A^*$ représente la clôture réflexive de $\xrightarrow{\tau}_A$. Remarquons que la relation $q \overset{a}{\Rightarrow} q$ abstrait uniquement les actions silencieuses qu'il est possible de faire avant l'action a et que $\xrightarrow{\tau}^*$ correspond à $\overset{\epsilon(0)}{\Rightarrow}$.

10.2.2 Relation $\le_{ttr}$

Nous proposons de définir une extension temporisée de la relation de conformité $\le_{tr}$ définie dans 8.2.2.

Définition 46 *Soient $A, I \in \mathcal{TIOA}(\Sigma)$, alors $I \le_{ttr} A =_\Delta TTrace(I) \subseteq Trace(A)$.* □

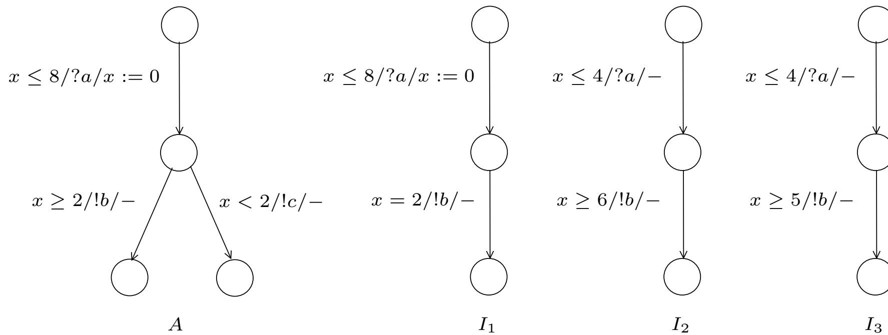


FIG. 46: Relations $\le_{ttr}$ et $ioconf_t$.

Exemple 10.1 *Considérons la* FIG. *46 et supposons que* I_1, I_2 *et* I_3 *soient des implantations de* A. *On peut remarquer que les traces temporisées de* A *sont* $TTrace(A) = \{\epsilon, (?a, d_1), (?a, d_1).(!b.d_1 + d_2), (?a, d_1).(!c.d_1 + d_3) \mid d_1 \in [0, 8], d_2 \in [2, +\infty[, d_3 \in [0, 2[\}$ *et que :*

1. $I_1 \leq_{ttr} A$. *En effet,* $TTrace(I_1) = \{\epsilon, (?a, d_1), (?a, d_1).(!b.d_1 + 2) \mid d_1 \in [0, 8]\} \subseteq TTrace(A)$.
2. $I_2 \leq_{ttr} A$. *En effet,* $TTrace(I_2) = \{\epsilon, (?a, d_1), (?a, d_1).(!b.d_2) \mid d_1 \in [0, 4], d_2 \in [6, +\infty[\} \subseteq TTrace(A)$.
3. $\neg(I_3 \leq_{ttr} A)$. *En effet,* $TTrace(I_3) = \{\epsilon, (?a, d_1), (?a, d_1).(!b.d_2) \mid d_1 \in [0, 4], d_2 \in [5, +\infty[\} \not\subseteq TTrace(A)$ *car* $(?a, 4).(!b, 5) \in TTrace(I_3)$ *et* $(?a, 4).(!b, 5) \notin TTrace(A)$.
□

10.2.3 Relation $ioconf_t$

Nous proposons de définir une extension temporisée de la relation de conformité $ioconf$ définie dans 8.2.3. Pour ce faire, nous aurons besoin des notations supplémentaires suivantes. Soit $\sigma \in \mathcal{TW}(\Sigma)$ une séquence temporisée, $q \in Q$ et $P \subseteq Q$.

- $q\, after\, \sigma \quad \triangleq \quad \{q' \in Q \mid q \stackrel{\sigma}{\Rightarrow} q'\}$.
- $out_e(q) \quad \triangleq \quad \{(a, d) \in \Sigma_o \times \mathbb{R}^{\geq 0} \mid q \stackrel{\epsilon(d)}{\Rightarrow} q'_1 \stackrel{a}{\Rightarrow} q_1\}$.
- $out_e(P) \quad \triangleq \quad \bigcup_{q \in P} out_e(q)$.
- $Out_e(A, \sigma) \quad \triangleq \quad \{(a, d) \in \Sigma_o \times \mathbb{R}^{\geq 0} \mid (a, d - delay(\sigma)) \in out_e(A\, after\, \sigma)\}$.

$Out_e(A, \sigma)$ contient les événements temporisés observables produits lors de l'application de la séquence temporisée σ. Remarquons que si $(a, d) \in Out_e(A, \sigma)$ alors l'instant d d'émission de l'événement a est par rapport à l'état initial et non par rapport à l'état d'émission, ce qui implique que $\sigma.(a, d) \in TW(\Sigma)$. Maintenant, nous proposons la relation $ioconf_t$ comme extension temporisée de $ioconf$ pour les RTSs.

Définition 47 *Soit* $A, I \in \mathcal{TIOA}(\Sigma_i, \Sigma_o)$. *Alors*

$$I \; ioconf_t \; A \; =_{\Delta} \; \forall \sigma \in TTrace(A) \Longrightarrow \; Out_e(I, \sigma) \; \subseteq \; Out_e(A, \sigma).$$ □

$ioconf_t$ permet, comme $ioconf$, (i) une spécification partielle des comportements et (ii) l'ajout de l'implantation de la sous spécification.

Exemple 10.2 *Considérons la* FIG. *46 et supposons que* I_1, I_2 *et* I_3 *sont des implantations de* A. *Soit* $\sigma = (?a, d)$ *telle que* $d \in [0, 8]$. *Alors* $Out_e(A, \sigma) = \{(!c, d + d_1), (!b, d + d_2) \| \; d_1 \in [0, 2[, d_2 \in [2, \infty[\}$. *On peut remarquer que :*

1. $I_1 \; ioconf_t \; A$. *En effet,* $Out_e(I_1, \sigma) = \{(!b, d + 2)\}$ *ce qui implique que* $Out_e(I_1, \sigma) \subseteq Out_e(A, \sigma)$.

2. *I_2 ioconf$_t$ A. Si $d \geq 4$, alors $Out_e(I_2, \sigma) = \emptyset \subseteq Out_e(A, \sigma)$. Sinon, $Out_e(I_2, \sigma) = \{(!b, d_2) \mid d_2 \in [6, +\infty[\} \subseteq Out_e(A, \sigma)$. En effet, si I_2 émet b à l'instant d_2, alors $d_2 - d \geq 2$, ce qui implique qu'entre la réception de a et l'émission de b, I_2 attend au moins deux unités de temps.*
3. *$\neg(I_3$ ioconf$_t$ $A)$. En effet, il suffit de voir que pour $d = 4$, $(!b, 5) \in Out_e(I_3, (?a, 4))$ et $(!b, 5) \notin Out_e(A, (?a, 4))$.* □

10.2.4 Relation $ioco_t$

De même, nous proposons de définir la relation $ioco_t$ comme extension temporisée de *ioco* définie dans la section 8.2.4. Pour ce faire, définissons :

- $out_\epsilon(q) \quad \triangleq \quad \{(a, d) \in \{\epsilon\} \times \mathbb{R}^{\geq 0} \mid q \overset{\epsilon(d)}{\Rightarrow} q_1\}$.
- $Out_\epsilon(A, \sigma) \quad \triangleq \quad \{(a, d) \in \{\epsilon\} \times \mathbb{R}^{\geq 0} \mid (\epsilon, d - delay(\sigma)) \in out_\epsilon(A\, after\, \sigma)\}$.

L'événement temporisé (ϵ, d) correspond à l'absence d'actions observables du système durant d unité de temps. La définition de $Out_\epsilon(A, \sigma)$ est la même que celle de $Out_e(A, \sigma)$ sauf que $Out_\epsilon(A, \sigma)$ capture les événements non observables qui sont dûs au passage du temps dans un état, soit la réalisation d'actions internes. $Out(A, \sigma)$ est dans ce cas l'union de $Out_\epsilon(A, \sigma)$ et $Out_e(A, \sigma)$: $Out(A, \sigma) = Out_e(A, \sigma) \cup Out_\epsilon(A, \sigma)$.

Définition 48 *Soit $A, I \in \mathcal{TIOA}(\Sigma_i, \Sigma_o)$. Alors*

I $ioco_t$ $A =_\Delta \forall \sigma \in TTrace(A) \Longrightarrow Out(I, \sigma) \subseteq Out(A, \sigma)$. □

Remarque 12 *Pour les automates à entrées/sorties sans urgence ou invariants, les relations ioconf$_t$ et ioco$_t$ sont identiques. En effet, la sémantique des TIOAs autorise le passage du temps dans chaque état et ainsi $Out_\epsilon(A, \sigma) = \{\epsilon\} \times \mathbb{R}^{\geq 0}$.*

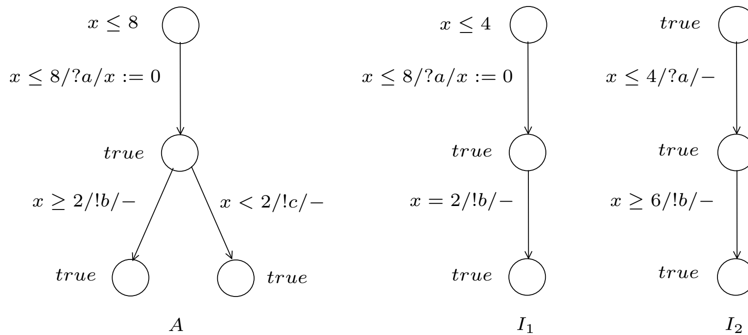


FIG. 47: Relation $ioco_t$.

Exemple 10.3 *Pour illustrer la relation $ioco_t$, considérons la* FIG. *47, une version de la* FIG. *46 avec invariants. On peut remarquer que I_1 $ioconf_t$ A et I_2 $ioconf_t$ A. Maintenant,*

1. *I_1 $ioco_t$ A. En effet, $Out(I_1, \epsilon) = \{(\epsilon, d)\,|d \in [0, 4]\} \subseteq Out(A, \epsilon) = \{(\epsilon, d)\,|d \in [0, 8]\}$.*
2. *$\neg(I_2$ $ioco_t$ $A)$. En effet, $Out(I_2, \epsilon) = \{(\epsilon, d)\,|d \in [0, +\infty[\} \subseteq Out(A, \epsilon) = \{(\epsilon, d)\,|d \in [0, 8]\}$.* □

10.3 Cas de test et testeurs

Comme nous avons dit auparavant, un cas de test (ou simplement un test) est une expérience réalisée par le testeur sur l'implantation. Dans le cas des RTS, il existe différents types de cas de test qui dépendent de la capacité du testeur à observer et réagir aux événements. Nous considérons ici deux types de cas de test (la terminologie est empruntée à [68, 92]).

1. Les tests utilisant des horloges *analogiques*, appelés des *tests analogiques*. Ces derniers peuvent mesurer précisément le délai entre deux actions observées et émettre une sortie à n'importe quel instant.
2. Les tests utilisant des horloges *digitales*, appelés des *tests digitaux*. Ces derniers peuvent seulement compter le nombre de *"ticks"* produits entre deux actions d'une horloge digitale et émettre une sortie immédiatement après avoir observé une action ou un tick.

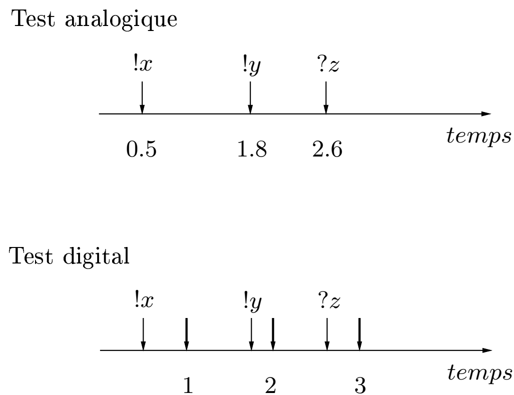


FIG. 48: Test analogique et digital.

Nous dirons que :

1. Un testeur est
   – *analogique*, s'il utilise des horloges analogiques pour mesurer le temps.

   - *digital*, s'il utilise des horloges digitales pour mesurer le temps.
2. Une IUT est
   - *analogique*, si elle implémente des horloges analogiques.
   - *digitale*, si elle implémente des horloges digitales pour mesurer le temps.

L'utilisation d'un testeur digital n'implique pas que l'IUT soit digitale, mais seulement la capacité d'observation et de réactions du testeur est discrète comme nous allons le voir par la suite. Pour simplifier nous supposons que le testeur peut contrôler le lancement de l'IUT et ainsi il démarre en même temps que l'IUT.

La FIG.48 illustre le comportement du testeur analogique et du testeur digital pour le même test. Dans le premier cas, le testeur analogique reçoit x à l'instant 0.5, y à 1.8 et émet z à 2.6. Pour le testeur digital, la réception de x n'est observée qu'au premier tick, y au deuxième tick et il émet z au troisième tick.

Ainsi, les testeurs analogiques peuvent mesurer précisément le temps. Ils trouvent une application intéressante dans le cas des IUT digitales dont le temps entre deux ticks est inconnu. Cependant, en général, il est difficile (sinon impossible) d'implémenter ces testeurs pour des IUTs analogiques. Les testeurs digitaux en revanche ne mesurent que les ticks d'une horloge périodique (un compteur), mais ils sont faciles à implémenter pour toute IUT. Au niveau verdict, ils peuvent annoncer un verdict $Pass$ à la place de $Fail$ (on ne distingue pas la réception de a à 2.3 de la même réception de a à 2.8 [1]).

10.3.1 Test adaptatif et statique

Un test analogique ou digital peut être qualifié de *adaptatif* ou *statique*. Dans un test adaptatif, la réponse du testeur dépend des observations réalisées dans le passé. Il peut être vu comme un algorithme dont l'entrée est l'historique observable. Par contre, dans le cas du test statique, la réponse du testeur est la même et elle est connue à l'avance.

L'utilisation de test statique pour un testeur digital est suffisante du fait que les ticks peuvent être vus comme des événements de sortie et ainsi un test statique peut être décrit par un arbre fini. Cependant, dans le cas d'un testeur analogique, il existe une infinité d'instants pour réaliser une action et en conséquence, un test ne peut être modélisé par un arbre fini et on ne peut qu'utiliser des cas de test adaptatifs.

10.3.2 Formalisme de cas de test

La distinction entre un test analogique et digital d'une part et entre un test statique et adaptatif d'autre part, nous conduit à définir un cas de test temporisé (TTC), squelette (STC) et abstrait (ATC).

[1] Pour le testeur digital, ces deux événements se produisent au troisième tick.

Définition 49 *Un cas de test temporisé TTC, pour un système modélisé par un TIOA* $A = (S, s_0, \Sigma, C, \rightarrow)$, *est un TIOA* $TTC = (S^{TTC}, s_0^{TTC}, \Sigma^{TTC}, C^{TTC}, \rightarrow_{TTC})$, *dont le graphe associé est un arbre. TTC est équipé de trois ensembles d'états* $Fail^{TTC}$, Inc^{TTC} *et* $Pass^{TTC}$. C^{TTC} *est restreint à une horloge universelle h bornée et sans remise à zéro dans toutes les transitions.* □

Les hypothèses formulées dans la cas de test non temporisé (section 8.4.2) restent valables dans le cas temporisé. Nous supposons que tous les cas de test temporisés vérifient ces hypothèses. Un TTC peut être utilisé par un testeur digital.

Définition 50 *Un cas de test squelette STC, pour un système modélisé par un TIOA* $A = (S, s_0, \Sigma, C, \rightarrow)$, *est un TIOA* $STC = (S^{STC}, s_0^{STC}, \Sigma^{STC}, C^{STC}, \rightarrow_{STC})$, *dont le graphe associé est un arbre. STC est équipé de trois ensembles d'états* $Fail^{STC}$, Inc^{STC} *et* $Pass^{STC}$. □

La différence entre un STC et un TTC est que STC peut définir plusieurs horloges avec remises à zéro tandis que TTC définit une seule horloge sans remise à zéro.

Définition 51 *Un cas de test abstrait ATC, pour un système modélisé par un TIOA* $A = (S, s_0, \Sigma, C, \rightarrow)$, *est un automate* $ATC = (Q^{ATC}, q_0^{ATC}, \Sigma^{ATC}, \rightarrow_{ATC})$, *dont le graphe associé est un arbre. ATC est équipé de trois ensembles d'états* $Fail^{ATC}$, Inc^{ATC} *et* $Pass^{ATC}$. *Les états* Q^{ATC} *sont des états symboliques* (s, Z) *tels que* $s \in S$ *et* $Z \in \Phi(C)$. □

Approche de génération

L'approche de génération de tests digitaux et analogiques que nous introduisons par la suite se décompose en deux phases. La première phase, présentée dans la section 10.4, consiste à dériver des test abstraits et squelettes. Cette dérivation est basée sur l'utilisation de l'automate de simulation. Les tests ainsi générés ne sont pas directement exploitables par le testeur. La deuxième phase consiste, à extraire à partir des tests squelettes, des cas tests temporisés exploitables par le testeur. Pour le testeur digital, cette phase est présentée dans la section 10.5 et elle repose sur l'utilisation des diagnostics temporisés de bornes. Pour le testeur analogique, cette phase est présentée dans la section 10.6.

10.4 Dérivation de cas de test abstraits et squelettes

10.4.1 Hypothèses de test

Par la suite et dans le cas d'un testeur digital, nous supposons que les bornes des contraintes sur la spécification sont exprimées en nombre de ticks. Dans le cas contraire, si une unité de temps correspond à n ticks, alors toutes les bornes des contraintes sont multipliées par n. De plus, *nous supposons que les cas de test (temporisés, squelettes et abstraits) sont de simples arbres tels qu'il existe un seul état étiqueté par Pass et tout état n'ayant pas de verdict se trouve sur le chemin qui mène vers l'état Pass*. Dans le cas où il existe plusieurs états *Pass*, le test est décomposé en plusieurs cas de test comme le montre la FIG.49.

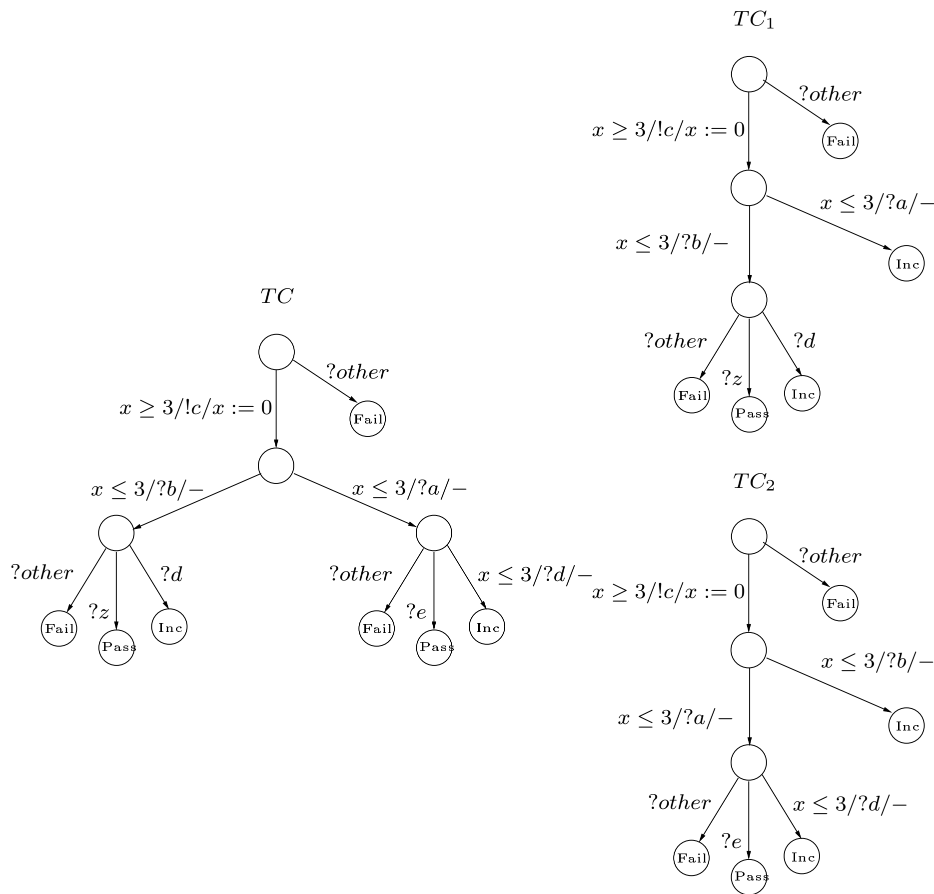


FIG. 49: Décomposition de test.

10.4.2 Automate de simulation

Nous rappelons ici l'automate de simulation (graphe de simulation dans [145]) introduit par Tripakis [145] en vue de son utilisation pour la génération de test.

Considérons un TA $A = (S, s_0, \Sigma, C, \rightarrow)$. Rappelons qu'une zone de A est un ensemble d'états $H = \{(s, \nu) \| \ \nu \in Z\}$ avec $s \in S$ un état de A, $Z \in \Phi(C)$ un polyèdre et $\nu \in \mathcal{V}(C)$ une valuation (section 7.2.1). H est noté simplement (s, Z).

Soient $H = (s, Z)$ une zone, $t = (s, a, Z', r, s')$ une transition de A et $c \geq c_{max}(A)$ une constante naturelle. La clôture du successeur temporel de H est définie par :

$$\mathbf{post}_c(H, t) = \{(s', close(((Z \cap Z_1)[r := 0])^{\uparrow}, c))\}$$

Intuitivement, $post_c()$ contient tous les états et leurs états c-équivalents atteignables à partir des états de H, en faisant une transition discrète, puis en laissant le temps s'écouler. Remarquons que $post_c(H, t)$ est aussi une zone.

Maintenant, l'automate de simulation $SA(A, c)$ associé à A de clôture $c \geq c_{max}(A)$ est l'automate défini par : les états de $SA(A, c)$ sont des zones. Son état initial est $H_0 = (s_0, zero^{\uparrow})$. Les transitions de $SA(A, c)$ sont générées en utilisant une recherche en profondeur partant de H_0. Pour un état $H_i = (s_i, Z_i)$ de $SA(A, c)$ déjà généré et l'état successeur $H_{i+1} = (s_{i+1}, Z_{i+1}) = post_c(H_i, t_i) \neq \emptyset$ avec $t_i = (s_i, a_i, Z_i, r_i, s_{i+1})$ une transition de A :

- S'il existe un état déjà généré H_j tel que $H_{i+1} \subseteq H_j$ alors $H_i \xrightarrow{a_i} H_j$ est une transition de $SA(A, c)$.
- Sinon $H_i \xrightarrow{a_i} H_{i+1}$ est une transition de $SA(A, c)$.

Soient σ une séquence temporisée de $TW(\Sigma_\tau)$ et $\pi = (s_0, Z_0) \xrightarrow{a_1} (s_1, Z_1) \cdots \xrightarrow{a_n} (s_n, Z_n)$ un chemin de $SA(A, c)$. σ est dite *inscrite* dans π s'il existe une computation $r = (\overline{s}, \overline{\nu})$ de A sur σ telle que pour tout $i \in [0, n]$, $\nu_i \in Z_i$.

Lemme 12 *Soient A un TIOA et $SA(A, c)$ son automate de simulation. Alors :*

1. *L'ensemble des états et des transitions de $SA(A, c)$ est fini.*
2. *A admet une computation sur σ telle que $event(\sigma_{|\sigma|}) = s \in S$ ssi il existe un noeud de $SA(A, c)$ de la forme (s, Z) (l'accessibilité d'un état s est préservée dans $SA(A, c)$).*
3. *Pour tout chemin π de $SA(A, c)$, il existe une séquence σ inscrite dans π.*
4. *Pour toute σ admettant une computation de A, il existe un unique chemin π tel que σ est inscrite dans π.*

En pratique, $SA(A, c)$ est de taille raisonnable et peut être exploité dans des outils de vérification et de test [145].

Preuve. La preuve est donnée dans [145].

Le corollaire suivant découle de la définition de l'automate de simulation.

Corollaire 10 *Si A est événementiellement déterministe alors $SA(A, c)$ est déterministe.* □

10.4.3 Cas de test abstraits et squelettes

Considérons un TIOA A. Selon l'approche adoptée par le testeur, il est possible de définir une stratégie de génération exploitant directement la spécification A[2]. Dans ce cas, les tests générés portent sur l'ensemble des comportements de A. Il est aussi possible de définir une stratégie de génération guidée par un objectif de test TP. Dans ce cas, les tests portent uniquement sur les comportements de $A \| TP$. Par la suite, S dénotera soit la spécification A, dans le cas d'une stratégie de génération directe, soit le produit synchrone de A et TP, dans le cas d'une stratégie de génération orientée objectif de test.

Pour pouvoir générer des cas de test pour S, on doit disposer d'une méthode pour décider de l'accessibilité des états de S. Or, un chemin de S n'admet pas forcement une computation et ainsi ses états ne sont pas toujours accessibles. Nous proposons d'utiliser l'automate de simulation $SA(S, c)$ associé à S. D'après le lemme 12, $SA(S, c)$ donne une représentation finie de l'espace des états accessibles de S. De plus, tout chemin de $SA(S, c)$ admet une computation de S et toute computation de S est inscrite dans un chemin de S. Comme $SA(S, c)$ est un automate non temporisé, il est possible d'utiliser des méthodes de génération classiques (TT, W, Wp, UIO,...) pour générer des cas de test. Les cas de test ainsi générés sont abstraits. Notons par $ATC_M(S)$ l'ensemble des cas de test générés par la méthode M, M étant l'une des méthodes citées auparavant. Puisque S est événementiellement déterministe alors, selon le corollaire 10 et pour tout $ATC \in ATC_M(S)$, il existe un unique test squelette STC tel que si $(s_1, Z_1) \xrightarrow{a}_{ATC} (s_2, Z_2)$ alors $s_1 \xrightarrow{Z,a,r}_{STC} s_2$. $STC_M(S)$ dénotera l'ensemble des cas de test squelettes ainsi construits à partir des cas de test abstraits de $ATC_M(S)$.

Les tests abstraits ou squelettes ne peuvent pas être utilisés directement comme des tests pour S, du fait qu'ils ne déterminent pas les instants de réalisation des actions. Dans la section suivante, nous utiliserons les tests squelettes pour générer des tests directement exploitables par le testeur.

Notons finalement qu'un élément de $STC_M(S)$ peut être calculé à la volée sans construire l'automate de simulation et qu'il est possible d'appliquer des critères de couverture pour $SA(S, c)$ pour sélectionner l'ensemble $STC_M(S)$. Nous renvoyons le lecteur à la section 11.1 pour plus de détails sur les critères de couverture.

[2]Le testeur peut, par exemple, opter pour une couverture des transitions ou des états de A.

10.5 Dérivation de tests digitaux

Dans la section précédente, nous avons montré comment générer des cas de test squelettes pour S à partir de l'automate de simulation. Maintenant, nous montrons comment générer des test pour un testeur digital à partir d'un test squelette $STC \in STC_M(S)$. Comme nous l'avons dit auparavant, un cas de test temporisé (statique) est suffisant pour un testeur digital. Rappelons que STC est un simple arbre qui ne contient qu'un seul état étiqueté par $Pass$. Nous assumons que STC est borné. Ceci du fait que le test est une expérience bornée. Dans le cas où un STC n'est pas borné, on ajoute une contrainte supplémentaire qui borne le temps global sur la transition dont l'état de destination est $Pass$

L'algorithme de génération des test digitaux (des cas de test temporisés) à partir d'un cas de test squelette est donné dans FIG.51. L'idée de l'algorithme est la suivante : à partir de STC, on considère le chemin $\rho = t_1 ... t_n$ qui mène vers l'état $Pass$ tel que $t_i = (s_{i-1}, Z_i, a_i, r_i, s_i)$, $i \in [1, n]$. Les hypothèses formulées au début de la section 10.4 impliquent que les états sans verdict de STC sont les états $(s_i)_{i \in [0,n-1]}$ de ρ. Pour le chemin ρ, on génère les diagnostics temporisés de bornes (σ^k) associés à ρ définis dans le chapitre 7. Pour chaque σ^k, on décore les différentes branches de STC par les verdicts qui découlent de cette dernière.

L'entrée de l'algorithme est un STC. En sortie, l'algorithme génère entre 1 et $2(n+1)$ cas de test temporisés où n est la longueur du chemin ρ de STC qui mène vers un verdict $Pass$. Nous commentons l'algorithme pour la génération de $TTC^k = (S^k, s_0^k, \Sigma^k, C^k, \rightarrow_k)$, avec $k \in [1, 2(n+1)]$.

TTC^k est initialisé aux états, alphabet de STC. Il utilise une horloge globale h. Au lancement de l'algorithme, l'ensemble de ses transitions $\rightarrow^k$ est vide. Les états verdicts de TTC^k ($Pass^k$, Inc^k et $Fail^k$) correspondent à ceux de STC (ligne 2). L'algorithme calcule des diagnostics temporisés de bornes $(\sigma^k)_{k \in [1,2(n+1)]}$, ainsi que les computations associées $((\overline{s}, \overline{\nu^k}))_{k \in [1,2(n+1)]}$ du chemin ρ (ligne 3). Les transitions $t_i = (s_{1-1}, Z_i, a_i, r_i, s_i)$ du chemin ρ qui mènent vers le verdict $Pass$ sont ajoutées à TTC^k, en remplaçant la contrainte Z_i et la mise à jour r_i par $Z_i := (h = time(\sigma_i^k))$ et $r_i = \emptyset$, pour $i \in [1, n]$ (lignes 5 et 6). Ceci correspond aux instants d'occurrences des événements qui mènent vers le verdict $Pass$.

Les étapes suivantes de l'algorithme correspondent à la décoration de TTC^k par les verdicts $Fail$ et Inc. Dans un état courant s_i de TTC^k, chaque transition sortante de s_i dans STC est examinée pour déterminer le verdict. Pour ce faire, on commence à affecter (s_i, ν_i^k) à la zone H_i (ligne 9). ν_i^k est alors confondue avec le polyèdre qui ne contient que ν_i^k. Pour toute transition $t = (s_i, Z, a, r, s) \in \rightarrow_{STC}$ sortante de s_i, on réalise les opérations suivantes (ligne 10) :

1. On calcule le successeur, noté H, de H_i par t dans le but de déterminer tous les états atteignables à partir de s_i par la transition t (ligne 11).
2. Si l'état destination s de la transition t n'est pas un état qui mène vers le verdict

Entrée : Un cas de test squelette $STC = (S^{STC}, s_0^{STC}, \Sigma^{STC}, C^{STC}, \rightarrow_{STC})$.
Sortie : Cas de test temporisés $(TTC^k = (S^k, s_0^k, \Sigma^k, C^k, \rightarrow_k))_{k\in[1,2(n+1)]}$.
Données : L'unique chemin $\rho = t_1...t_n$ de STC qui mène vers le verdict $Pass$ tel que $t_i = (s_{i-1}, Z_i, a_i, r_i, s_i)$, $i \in [1, n]$. $(H_i)_{i\in[0,n-1]}$ et H des zones.
/* $s_0 = s_0^{STC}$ */
Début

/* Initialisation des TTC^k */

1. **Pour** k de 1 à $2(n+1)$ **Faire**
2. $S^k = S^{STC}$, $s_0^k = s_0^{STC}$, $\Sigma^k = \Sigma^{STC}$, $C^k = \{h\}$, $\rightarrow_k = \emptyset$, $Inc^k = Inc$, $Pass^k = Pass$ et $Fail^k = Fail$.

/* Calcul de diagnostics temporisés de bornes associés à ρ */

3. Calculer les $2(n+1)$ diagnostics temporisés de bornes $(\sigma^k)_{k\in[1,2(n+1)]}$ associés à ρ, ainsi que leurs computations relatives $((\overline{s}, \overline{\nu^k}))_{k\in[1,2(n+1)]}$.

/* Par la suite ν_i^k sera confondue avec le polyèdre $\bigwedge_{x\in C^{STC}}(x = \nu_i^k(x))$, pour tout $i \in [0, n]$, $k \in [1, 2(n+1)]$ */

4. **Pour** k de 1 à $2(n+1)$ **Faire**
5. **Pour** i de 1 à n **Faire**
6. Ajouter $(s_{i-1}, h = time(\sigma_i^k), a_i, \emptyset, s_i)$ à $\rightarrow_k$

7. **Pour** k de 1 à $2(n+1)$ **Faire**
8. **Pour** i de 0 à $n-1$ **Faire**
9. $H_i = (s_i, \nu_i^k)$
10. **Pour** $t = (s_i, Z, a, r, s) \in \rightarrow_{STC}$ **Faire**
11. Calculer $H = (s, Z') = post(H_i, t)$.
12. **Si** $s \neq s_{i+1}$ **Alors**
13. Ajouter $(s_i, min(Z', h) \leq h \leq max(Z', h), a, \emptyset, s)$ à $\rightarrow_k$.
14. **Sinon Si** a est une réception **Alors**
15. Ajouter $(s_i, min(Z', h) \leq h < \nu_{i+1}^k(h), a, \emptyset, s^1)$ à $\rightarrow_k$.
16. Ajouter $(s_i, \nu_{i+1}^k(h) < h \leq max(Z', h), a, \emptyset, s^2)$ à $\rightarrow_k$.
17. Ajouter s^1 et s^2 à Inc^k.

Fin

FIG. 50: Algorithme de génération des tests digitaux.

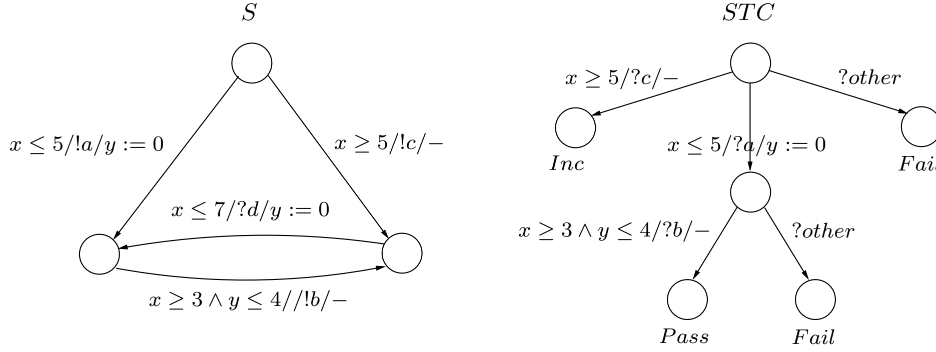


FIG. 51: Spécification et cas de test squelette.

$Pass$ (les états qui mènent vers le verdict $Pass$ sont les s_i, $i \in [0, n-1]$), alors s a un verdict Inc ou $Fail$ (ligne 12). Dans ce cas, on calcule le temps minimal et maximal à partir de H_i pour atteindre s. Ainsi $min(Z', h) = d$ (resp. $max(Z', h) = d'$) si $d' \leq h \leq d' \wedge \bigwedge_{h \neq x, y \in C^{STC}} (x - y \leq c_{xy})$ est la forme canonique de Z et on ajoute la transition $(s_i, min(Z', h) \leq h \leq max(Z', h), a, \emptyset, s)$ à $\rightarrow_k$ (ligne 13). s garde son verdict initial.

3. Dans le cas contraire ($s = s_{i+1}$), soit l'événement a est une réception ou une émission. Rappelons que les hypothèses formulées sur le cas de test interdisent le choix entre une sortie et une entrée et celui-ci est input-complet dans un état où une réception est possible. Si c'est une émission, dans ce cas c'est le testeur qui la contrôle et on n'a pas besoin d'ajouter un verdict Inc pour les autres états de H. Si c'est une réception, les instants d'occurrences selon h compris entre $[min(Z', h), max(Z', h)]$ vérifient les contraintes de t, mais ne peuvent pas tous conduire à l'état $Pass$. Dans ce cas, on ajoute les deux états s^1 et s^2 à S^k et les deux transitions $(s_i, min(Z', h) \leq h < \nu_{i+1}^k(h), a, \emptyset, s^1)$ et $(s_i, \nu_{i+1}^k(h) < h \leq max(Z', h), a, \emptyset, s^2)$ à $\rightarrow_k$. s^1 et s^2 seront donc dans Inc^k (lignes 14-17).

Notons finalement que pour montrer que les traces d'un STC sont incluses dans l'implantation, il suffit que l'exécution des $2(n+1)$ TTC^k associés STC donne toujours un verdict $Pass$.

Exemple d'application. Considérons la spécification S de la FIG.51. Un cas de test squelette généré à partir de l'automate de simulation associé à S est donné dans FIG.52.

L'un des diagnostics temporisés de bornes du chemin menant à $Pass$ est $\sigma = (?a, 5).(?b, 9)$. Le cas de test temporisé associé à cette trace est donné dans FIG.52. L'événement $?other$ spécifie toutes les réceptions autre que celles renseignées ou des réceptions à des instants en dehors des intervalles des réceptions renseignées.

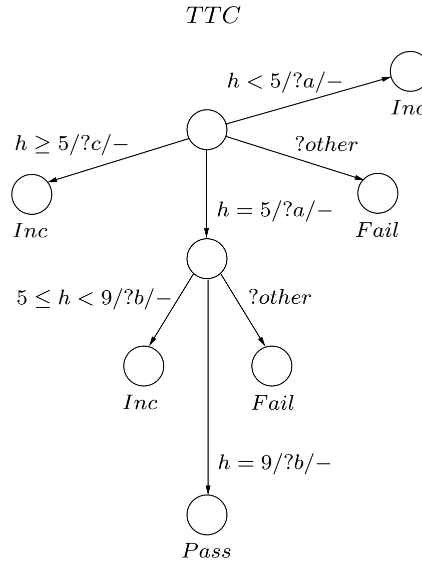


FIG. 52: Cas de test temporisé.

10.6 Dérivation de tests analogiques

Dans le but de réduire l'explosion combinatoire des nombres de tests, nous proposons dans cette section une méthode pour générer des tests analogiques (adaptatifs) à partir d'un $STC \in STC_M(S)$. Ce dernier sera utilisé comme un guide des fonctionnalités que le testeur doit réaliser. L'exécution d'un test analogique est vu comme un algorithme.

L'idée de la méthode de génération est la suivante. Étant un STC qui peut être qualifié de statique, le testeur interagit avec l'IUT selon les comportements décrits dans le STC et retourne un verdict associé à l'exécution du STC. Le verdict annoncé dépend des interactions et de l'état courant du testeur dans le STC.

La FIG.53 illustre l'algorithme de génération. L'entrée de l'algorithme est un test squelette. La sortie est un verdict ($Fail$, $Pass$, Inc). L'algorithme considère l'unique chemin ρ qui mène vers le verdict $Pass$, un chemin symbolique $S(\rho)$, ainsi que des zones H_i, $i \in [0, n-1]$ et H. La première phase de l'algorithme initialise les données utilisées. H_0 est initialisée à l'état initial de STC et à la valuation nulle ν_0 (ligne 1), ce qui correspond à l'instant de démarrage du testeur. $S(\rho)$ est initialisé au chemin symbolique post/pred stable associé à ρ (voir section 7.2.2).

Dans l'itération i, $i \in [0, n-1]$, la zone courante est $H_i = (s_i, \nu_i)$. D'après les hypothèses formulées sur les cas de test, dans un état donné, soient toutes les transitions sortantes sont sur des événements de réceptions (s_i est un état d'attente), soit une seule transition sortante sur un événement d'émission. Dans ce dernier cas, l'état destination ne possède pas de verdict.

Si s_i est un état d'attente alors le testeur reçoit un événement b après d unités de

Entrée : Un cas de test squelette $STC = (S^{STC}, s_0^{STC}, \Sigma^{STC}, C^{STC}, \rightarrow_{STC})$.
Sortie : Un verdict
Données : L'unique chemin temporisé de STC qui mène vers le verdict $PASS$
$\rho = t_1...t_n$, $t_i = (s_{i-1}, Z_i, a_i, r_i, s_i)$, $i \in [1, n]$, un chemin symbolique
$S(\rho) = t'_1...t'_n$,$t'_i = (H'_{i-1} = (s_{i-1}, Z'_{i-1}), a_i, H'_i = (s_i, Z'_i))$, H et
$(H_i = (s_i, \nu_i))_{i \in [0,n-1]}$ des zones /* $s_0 = s_0^{STC}$ */
Début

/* Initialisation des TTC_k */
1. $H_0 = (s_0, \nu_0)$
2. Initialiser $S(\rho)$ au chemin symbolique post/pred stable associé à ρ.

3. **Pour** i de 0 à $n-1$ **Faire**

/* Les transitions sortantes de s_i sont étiquetées par une réception */
4. **Si** s_i est un état d'attente **Alors**
5. Recevoir l'événement b, après d unités de temps, correspondant à la transition $t = (s_i, Z, a, r, s) \in \rightarrow_{STC}$

6. $H_{i+1} = (s, (\nu_i + d)[r := 0])$ /* Nouvel état atteint par la réception de b à d */
7. $H = post(H_i, t)$ /* calcul de tous les états atteignables par t dans STC */

8. **Si** $H \cap H_{i+1} = \emptyset$ **Alors** /* b n'est pas un état atteignable par t */
9. Retourner $Fail$
10. **Sinon** /* b est un état atteignable par t */
11. **Si** s possède un verdict **Alors** /* s ne mène pas à $Pass$ */
12. Retourner verdict s
13. **Sinon** /* Dans ce cas, $s = s_{i+1}$ */
14. **Si** $H_{i+1} \cap H'_{i+1} = \emptyset$ **Alors**
15. Retourner Inc

/* La transition sortante de s_i est étiquetée par une émission */
16. **Sinon**
17. Calculer les diagnostics temporisés de bornes $(\sigma^k)_{k \in [0,2(n+1)]}$ associés au symbolique $S(\rho)$
18. Choisir d parmi $\{\sigma^k_{i+1}(h) - \nu_i(h) \mid k \in [0, 2(n+1)]\}$
19. Émettre a correspondant à $t = (s_i, Z, a, r, s) \in \rightarrow_{STC}$ après d unités de temps.
20. $H_{i+1} = (s_{i+1}, (\nu_i + d)[r := 0])$

/* Mise à jour du chemin symbolique */
21. $H'_{i+1} = H_{i+1}$
22. **Pour** j de $i+2$ à n **Faire**
23. $H'_j = post(H'_{j-1}, t_j) \cap H'_j$

Fin

FIG. 53: Algorithme de génération des tests adaptatifs.

temps après la dernière action (ligne 5). Comme la spécification est événementiellement déterministe, alors la transition t correspondante à l'événement b est définie d'une manière unique. Après cette réception, le testeur change d'état en mettant à jour H_{i+1} (ligne 6). Il calcule tous les états atteignables à partir de H_i par t (ligne 7). Deux cas se présentent :

1. $H \cap H_{i+1} = \emptyset$. Dans ce cas, la réception de b après d unités de temps n'est pas atteignable à partir de la zone courante H_i. Le verdict dans ce cas est $Fail$ (lignes 8,9)

2. H_{i+1} est atteignable à partir de H_i. Si l'état s possède un verdict, ce dernier est retourné (lignes 11,12). Sinon, s n'a pas de verdict, ce qui implique que s fait partie du chemin qui mène vers $Pass$ et par la suite $s = s_{i+1}$. Vu que le testeur essaie d'atteindre l'état $Pass$ et que tous les états de H ne mènent pas forcement à cela, on vérifie que $H_{i+1} \cap H'_{i+1}$ n'est pas vide. En effet, $S(\rho)$ est post-pred stable, donc tout élément de H'_i admet un successeur. Donc, si $H_{i+1} \cap H'_{i+1}$ est vide, on déduit que la réception de b à d est permise, mais elle ne permet pas d'atteindre l'état $Pass$. Le verdict dans ce cas est Inc (lignes 13-15).

Dans le cas où s est un état émission, le testeur commence par calculer les diagnostics temporisés de bornes associés au chemin $S(\rho)$ (ligne 17). Notons qu'entre temps $S(\rho)$ peut se retrouver changé par rapport à sa première valeur (à cause des mises à jour des lignes 21-23). L'émission de l'événement par le testeur ne se fait pas aléatoirement, mais selon un délai qui coïncide avec l'un des diagnostics temporisés de bornes déjà calculés (lignes 18,19). Ainsi, le testeur, au lieu de choisir n'importe quel instant possible d'émission de a, choisit un instant qui coïncide avec un diagnostic temporisé de bornes. H_{i+1} est alors mise à jour (ligne 20). Finalement, le chemin symbolique $S(\rho)$ est mis à jour par le calcul des nouvelles zones post-pred stables qui mènent vers l'état $Pass$ (lignes 21-23).

L'algorithme est exécuté plusieurs fois pour le même STC. Une condition suffisante pour l'arrêt de l'exécution est la couverture des diagnostics temporisés de bornes de ρ. Dans ce cas STC est couvert par l'implantation.

Heuristique

Dans le cas d'une émission par le testeur, l'algorithme de la Fig.53 calcule, à chaque étape, les diagnostics temporisés de bornes du chemin symbolique $S(\rho)$ actualisé, puis choisit un instant qui coïncide avec un diagnostic de bornes. Une autre alternative à ce calcul répétitif, est le choix d'un d correspondant au délai minimal ou le maximal pour atteindre H'_{i+1} à partir de H_i. Certes, la complexité de l'algorithme diminue mais en revanche, elle ne garantit pas la couverture des diagnostics de bornes et par la suite l'inclusion des traces de STC dans celles de l'implantation.

10.7 Comparaison

On distingue les travaux portant sur la définition de la conformité de ceux qui proposent des méthodes de génération.

10.7.1 Définition de la conformité

Lorsque la spécification est supposée input-complète, la relation $ioco_t$ est équivalente à l'inclusion des traces. Dans [38], une relation $tioco_M$ est introduite pour définir la notion de conformité des systèmes de transitions étiquetés temporisés (TLTS). Les TLTS sont une classe réduite des TAs qui considère une seule horloge remise à zéro dans chaque transition. La relation la plus proche à $ioco_t$ est la relation *tioco* définie dans [92]. La relation *tioco* permet de tester le silence des implantations. Il est clair qu'une implantation $ioco_t$ est *tioco* et inversement. Cependant, la définition de *tioco* est basée sur une séparation entre le passage du temps et l'occurrence des événements. C'est un choix pragmatique. Cependant, nous pensons que l'association entre l'instant d'occurrence et la réalisation d'un événement est plus naturelle et appropriée au test des systèmes temporisés vu que le testeur ne sera amené qu'à appliquer une action à un instant absolu et observer une action à un instant donné. De plus, comme nous allons le voir par la suite, cette séparation a une conséquence sur la taille des cas de test temporisés générés.

10.7.2 Dérivation de tests

L'approche présentée dans [92] est la plus proche de la notre. Krichen et al. [92] proposent une méthode de dérivation de tests digitaux et analogiques, basée sur une analyse symbolique, pour une spécification modélisée par un automate temporisé avec urgence, non-déterministe et non-observable. Certes, le modèle adopté est plus riche, cependant la méthode proposée souffre des problèmes suivants :

1. La méthode de génération des tests digitaux considère les "ticks" de l'horloge comme des événements observables. Une conséquence directe de ce choix est la présence de chaînes de "ticks" ce qui augmente considérablement la taille des tests ainsi générés comme établi dans [92] : "Digital-clock test can sometimes grow large because they contain a number of "chains" of ticks". Pour remédier à ce constat, les auteurs proposent une heuristique pour compacter les chaînes de ticks ce qui ne donne pas toujours un test minimal pour un test transformé : "Reducing the size of test representations is a non-trivial problem in general, related to compression and algorithmic complexity theory".
2. Le choix par le testeur des instants d'émissions est aléatoire aussi bien pour les tests digitaux qu'analogiques. En conséquence, le nombre de tests générés est très important comme on peut le constater dans l'exemple d'application donné dans [92] :

> "We have obtained 68, 180, 591, and 2243 tests for depth levels 5, 6 , 7 and 8, respectively"

Ces problèmes n'apparaissent pas avec notre approche du fait de l'association entre l'occurrence d'un événement et son instant d'occurrence et l'utilisation des diagnostiques temporisés de bornes comme une base pour le choix des instants d'émissions. Comme nous l'avons montré auparavant, l'inclusion des traces de l'implantation dans celle de la spécification peut être montrée par les traces associées aux diagnostiques de bornes.

Une extension de la théorie du test pour les machines de Mealy dans le cas des systèmes temps réel a été proposée par Springintveld et al. [141]. Les auteurs proposent une discrétisation de l'automate des régions. La discrétisation proposée tient compte du nombre d'horloges utilisées ainsi que des contraintes temporelles. A partir du modèle généré, ils génèrent des cas de test temporisés. Cette extension conduit à un ensemble fini et complet de tests mais la méthode est fortement exponentielle et non utilisable en pratique.

Une autre méthode pour générer des tests pour un automate temporisé déterministe discret est donnée par En-Nouaary et al. dans [54]. Les auteurs construisent un automate grille à partir de l'automate des régions et utilisent la méthode Wp pour la génération en assurant une bonne couverture de la spécification initiale. Cependant, le nombre de tests générés est important.

Dans [89], les auteurs considèrent le modèle des automates temporisés discrets. Ce modèle est transformé en un automate non-temporisé mais ayant deux événements spéciaux sur les horloges : "set" et "expire". L'avantage de la méthode est la limitation de l'explosion combinatoire. Cependant, cette approche peut générer des tests non-exécutables avec les événements "set" et "expire".

Dans [45], une horloge implicite est utilisée. Le modèle de base est l'automate temporisé discret. Cette approche a trois limitations : le temps est discret, ce qui restreint les systèmes considérés, le test concerne seulement les domaines du temps (et non pas un événement se produisant à un instant précis) et finalement, le nombre de tests générés peut être important.

Dans [49], la spécification du système est basée sur le graphe de contraintes. À partir d'un modèle de fautes, les auteurs définissent un critère de test et génèrent des tests selon ce critère. L'utilisation du graphe de contraintes ne permet d'exprimer que les délais entre deux événements successifs.

Dans [105], la génération se fait à partir d'une logique temporelle avec une arithmétique basée un temps discret.

[71] propose une méthode de génération basée sur "must/may" traçabilité. Les auteurs proposent d'abord de tester la correction des états et des transitions de l'implantation. Pour cela, la spécification est transformée en FSM et la méthode UIOv est utilisée. Pour la correction des contraintes des transitions, les auteurs adoptent un modèle de fautes et utilisent la "must/may" traçabilité méthode pour générer des tests.

[117] considère des automates à enregistrement d'événements, une sous-classe des automates temporisés et utilise un "must/may" préordre.

Dans [70], les auteurs proposent d'utiliser l'outil Uppaal pour générer l'exécution de cumul temporel minimal pour atteindre un état donné.

Un problème majeur de ces méthodes est le nombre important de tests générés même dans le cas d'un modèle discret.

Chapitre 11

Test ouvert

Sommaire

11.1 Couverture structurelle 172
11.1.1 Critères de couverture 172
11.1.2 Terminologies et définitions 174
11.1.3 Coloriage comme critère de couverture 175
11.1.4 Coloriage ordonné comme critère de couverture 176
11.1.5 Comment définir un coloriage et un coloriage ordonné 178
11.1.6 Conclusion 179
11.2 Modélisation uniforme du test 179
11.2.1 Méthodologie des algorithmes génériques de génération 180
11.2.2 Modélisation de l'architecture de test 181
11.2.3 Modélisation de l'approche passive du test 182
11.2.4 Modélisation de l'approche active du test 186
11.2.5 Comparaison 188
11.2.6 Test ouvert 189

La *couverture* des tests exécutés par le testeur est un concept de mesure de la qualité du test effectué. Dans le contexte des systèmes non-temporisés, plusieurs *critères* de couverture ont été définis. Une des conséquences du choix d'un critère de couverture est la définition d'une méthode de génération propre au critère considéré (états, transitions, variables,...). En conséquence, le changement du critère implique le changement de la méthode de génération[1]. Nous expliquons ce constat par le fait que la couverture de structure ne repose pas sur un formalisme bien défini. Dans le cas où un critère de couverture est défini dans un formalisme donné, il est alors possible de définir une méthode de génération basée que ce formalisme et d'appliquer la même méthode de génération, indépendamment

[1]Par exemple, une méthode de couverture de variables ne peut pas être appliquée à un critère de couverture de transitions ou d'états.

du critère que le testeur veut considérer pour une spécification donnée. Dans ce chapitre, nous introduisons :

1. Une formalisation de la notion de couverture structurelle basée sur le *coloriage de couverture*. Un coloriage de couverture est une fonction qui associe à chaque élément (état ou transition) d'un graphe, (i) une couleur qui détermine le groupe d'appartenance de l'élément et (ii) un ensemble $friend()$ qui détermine les couleurs amies à un élément donné. Nous définissons alors un critère de couverture structurelle comme étant une famille (un ensemble) de coloriage de couverture. Comme résultat, nous montrons que la plupart des critères de couverture définis dans la littérature peuvent être représentés par une famille de coloriages de couverture.
2. Une modélisation uniforme des différents types et approches de test. Cette modélisation est basée sur le modèle CS introduit dans la partie II. La méthodologie retenue est celle des algorithmes génériques de génération (gga). Elle est basée sur la définition d'un critère de couverture sous forme d'une famille de coloriage. Comme résultat final, nous introduisons la notion du *test ouvert*, ainsi que l'activité du test ouvert.

11.1 Couverture structurelle

11.1.1 Critères de couverture

Étant données des implantations (I_i) des spécifications (S_i) et une relation de satisfaction R qui caractérise l'ensemble des implantations valides par rapport à R, vérifier que "(I_i) sont R valides (S_i)" consiste à vérifier que (I_i) possèdent un ensemble de comportements définis par R. Or, cet ensemble est généralement infini, en particulier lorsque (S_i) contiennent des cycles/boucles. Sachant que le test est une expérience finie, il est alors impossible de générer une suite finie et complète de test pour une relation donnée. Cependant, une implantation peut être R valide à un certain degré. Le testeur opte, en général, pour une stratégie de génération qui permet de couvrir la spécification d'une certaine façon. Ceci permet d'avoir un certain degré de confiance dans l'activité du testeur.

Dans le contexte des systèmes non-temporisés, plusieurs critères de couverture ont été définis comme les couvertures de transitions, d'états, de variables et de chemins. Nous appellerons ces couvertures, des *couvertures structurelles*. Dans le contexte des systèmes temporisés, l'espace d'états est infini. Un critère de couverture essaie, dans ce cas, de couvrir une abstraction de cet espace d'états que ce soit le graphe des régions [141] ou le graphe de partitions [117]. Nous appellerons cette couverture, une *couverture comportementale*. Généralement, pour les systèmes temporisés, on combine les deux types de couvertures : structurelle et comportementale.

Nous rappelons ici quelques critères de couverture structurelle utilisés pour le test des protocoles [32, 70, 117, 141, 148]. Pour les travaux relatifs au test logiciel se référer à [14, 159].

Soient $A = (S, s_0, L, C, P, V, V_0, Pred, Ass, \rightarrow)$ un ETIOA modélisant un protocole et TS une ensemble de chemins de A de l'état initial.

Définition 52 (Couverture d'états) *TS satisfait le critère de couverture des états de A si pour tout élément de S, il existe au moins un chemin de TS qui le traverse.* □

Définition 53 (Couverture de transitions) *TS satisfait le critère de couverture des transitions de A si pour tout élément de $\rightarrow$, il existe au moins un chemin de TS qui le couvre.* □

Définition 54 (Couverture de chemins) *TS satisfait le critère de couverture des chemins de A si tout chemin p de A est couvert par au moins un chemin de TS.* □

Définition 55 (Couverture de variables (Définition-utilisation)) *Étant donnée une variable $v \in C \cup V$ (horloge ou variable discrète), TS satisfait le critère de couverture de la variable v si TScouvre tous les chemins dans lesquels v est définie (v apparaît dans la partie gauche d'une affectation) et plus tard utilisée (v apparaît dans la définition d'une garde ou dans la partie droite d'une affectation).* □

Intuitivement, si v est définie dans la transition t_d et utilisée dans la transition t_u, alors le triplet (v, t_d, t_u) est valide s'il existe un chemin de t_d à t_u dans lequel v n'est pas réaffectée. TS couvre v si elle couvre tous les triplets valides de v.

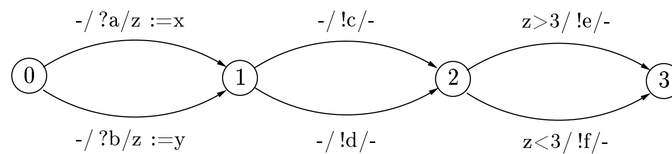


FIG. 54: Couverture.

Quelle est la différence entre ces différents critères ? La couverture d'états considère les chemins qui passent par chaque état. Ainsi, dans la FIG.54, il suffit d'un seul chemin pour satisfaire ce critère. De même, la couverture de transitions considère les chemins qui passent par chaque transition. Ainsi, dans la FIG.54, il suffit de deux chemins pour satisfaire ce critère. La couverture de chemins considère tous les chemins à partir de l'état initial. Ainsi, dans la FIG.54, une suite de chemins qui satisfait ce critère contient huit chemins (comparer avec les deux chemins de la couverture de transitions). Finalement, la couverture de variables considère les chemins où une variable est définie puis utilisée et non redéfinie entre une définition et une utilisation. Par exemple, dans la FIG.54, la variable z

est définie pour la première fois dans les transitions qui partent de l'état initial 0. Elle est utilisée dans les gardes des transitions qui partent de l'état 2. Cette couverture nécessite alors quatre chemins. En effet, pour chaque définition de z, il y a deux utilisations de z (les transitions qui partent de l'état 2).

Finalement, signalons que la définition de la couverture de chemins est une définition informelle qui ne précise pas comment appliquer cette couverture pour un protocole ayant un comportement infini. De plus, considérer des chemins infinis va à l'encontre de l'objectif de la couverture qui tente de sélectionner une représentation finie de l'espace des traces.

11.1.2 Terminologies et définitions

Soient $G = (N, E)$ un graphe orienté, $color = \{b, r, w, \cdots\}$ un ensemble de symboles appelés couleurs. Un coloriage de G est une fonction de $N \cup E$ vers *color* qui assigne à chaque élément de $N \cup E$ une *couleur*. Par la suite, nous utiliserons le terme *élément* pour référencer un noeud ou un arc de G.

Définition 56 (coloriage) *Un coloriage de G est une fonction col de* $NE(G) = N \cup E \mapsto color$. □

Pour simplifier, nous supposons que les coloriages considérés couvrent toutes les couleurs de l'ensemble *color* : $\forall c \in color$, il existe $e \in NE$ tel que $col(e) = c$.

Définition 57 (coloriage ordonné) *Un coloriage ordonné de G est une fonction col de* $NE(G) = N \cup E \mapsto color \times \mathbb{N}$. □

Un coloriage ordonné de G est une fonction de $N \cup E$ vers $color \times \mathbb{N}$ qui assigne à chaque élément de $N \cup E$ une *couleur*, un *rang*. Tout coloriage ordonné *col* définit un coloriage associé, noté col_{color}, défini par : $col_{color}(e) = c$ si et seulement si $col(e) = (c, r)$.

Définition 58 *Une fonction (amie)* $friend()$ *est une fonction de* $color \mapsto 2^{color}$ *qui assigne à une couleur c un ensemble de couleurs amies de c.* □

Une couleur de $friend(c)$ sera dite *amie* de la couleur c. Finalement, nous étendons la définition de coloriage et coloriage ordonné à un ensemble de graphes.

Définition 59 (coloriage distribué) *Soient* $(G_i = (N_i, E_i))_{i \in I}$ *un ensemble de graphes.*

1. *Un coloriage distribué est une fonction col :* $\bigcup_{k \in I}(N_k \cup E_k) \mapsto color$.
2. *Un coloriage distribué ordonné est une fonction col :* $\bigcup_{k \in I}(N_k \cup E_k) \mapsto color \times \mathbb{N}$. □

11.1.3 Coloriage comme critère de couverture

Avoir un modèle pour la couverture permet de présenter différents critères d'une façon uniforme et d'adopter la même méthodologie de génération pour ces derniers. Nous proposons de définir un critère de couverture comme un coloriage. Soit A un ETIOA. A sera confondu, par la suite, avec son graphe associé.

Considérons un coloriage *col* de A et une fonction amie $friend()$ associée à A. Le principe d'un coloriage utilisé comme un critère de couverture est de séparer les éléments de A en groupes de couleurs différentes puis de couvrir chaque groupe de couleur c par une suite $TS(c)$ de chemins de A. En effet, l'ensemble des éléments de A (transitions ou états) de même couleur c représente un comportement à couvrir. La manière dont ces éléments doivent être couverts dépend de l'ensemble $friend(c)$. Pour une couleur c, l'ensemble $friend(c)$ détermine les couleurs des éléments permises dans les chemins qui couvrent c. En d'autres mots, aucun chemin qui couvre la couleur c ne doit contenir un élément (transition ou état) colorié par une couleur de $color \backslash (friend(c) \cup \{c\})$. Formellement,

Définition 60 *Soient c une couleur, col un coloriage de A, $friend()$ une fonction amie associée à A et $TS(c)$ un ensemble de chemins de A de l'état initial. $TS(c)$ satisfait le critère de couverture col pour la couleur c si et seulement si :*

1. *Tous les chemins de $TS(c)$ sont exécutables.*
2. *Les couleurs des éléments qui apparaissent dans un chemin $p \in TS(c)$ sont dans $\{c\} \cup friend(c)$.*
3. *Pour chaque élément de A colorié par c, il existe un chemin $p \in TS(c)$ qui le couvre.*

Intuitivement, $TS(c)$ satisfait le critère *col* pour c si les chemins de $TS(c)$ sont exécutables, i.e. les différentes contraintes sont compatibles. Par exemple, dans le cas d'un TIOA, cela correspond à ce que tout chemin de $TS(c)$ admet des diagnostics temporisés. Remarquons aussi que dans le cas où A est un simple automate, tous les chemins sont exécutables. La deuxième condition sur $p \in TS(c)$ est que les couleurs des éléments formant p soient dans $\{c\} \cup friend(c)$. Finalement, tous les éléments de couleur c sont couverts.

Définition 61 *Soit $TS(c)$ un ensemble de chemins de A qui satisfait le critère de couverture col pour la couleur c. La suite $TS(c)$ est dit exhaustive si $TS(c)$ contient tous les chemins qui couvrent c.* □

En règle général, si $TS(c)$ est exhaustive alors elle est infinie (dû à la présence des boucles).

Exemple 11.1 *Prenons le cas simple où A est un automate de la* FIG.*55. Le coloriage col associé à A est le suivant (l'état initial est n_1) :*

– *n_1, e_6, e_7 et e_8 ont une couleur rouge.*

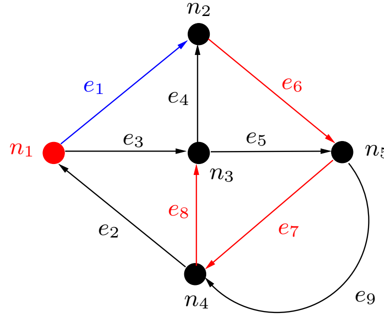


FIG. 55: Coloriage de A.

- *e_1 a une couleur bleue.*
- *Les autres éléments ont une couleur noire.*

Supposons que $friend(rouge) = \{noir\}$. Considérons les trois suites de chemins suivantes :

1. $TS_1(rouge) = \{e_3.e_4.e_6.e_7.e_8\}$
2. $TS_2(rouge) = \{e_3.e_5.e_7.e_8,\ e_3.e_4.e_6\}$
3. $TS_3(rouge) = \{e_1.e_6.e_7.e_8\}$

On remarque que $TS_1(rouge)$ et $TS_2(rouge)$ satisfont le critère de couverture col pour la couleur rouge. Cependant, $TS_3(rouge)$ ne satisfait pas ce critère car la couleur de e_1 n'est pas dans $friend(rouge)$. $TS_1(rouge)$ et $TS_2(rouge)$ ne sont pas exhaustives. Pour cette exemple, il n'existe pas de $TS(rouge)$ exhaustive et finie. □

Corollaire 11 *Les critères de couverture transitions, états et chemins peuvent être représentés par un coloriage.* □

Preuve. Pour les transitions, considérons le coloriage col_t qui assigne le rouge à toutes les transitions et le noir à tous les états tel que $friend(rouge) = \{noir\}$. Une suite $TS(rouge)$ de chemins qui satisfait le critère col_t correspond à la couverture de transitions. Pour le critère d'états, col_s inverse les couleurs (le rouge est assigné aux états et le noir aux transitions). Pour les chemins, considérons col_p qui affecte la couleur rouge aux transitions et aux états. Une suite $TS(rouge)$ exhaustive qui satisfait le critère *col* pour *rouge* satisfait la couverture des chemins. □

11.1.4 Coloriage ordonné comme critère de couverture

Nous proposons de définir un critère de couverture comme un coloriage ordonné. Soit *col* un coloriage ordonné de A. Rappelons que col_{color} est le coloriage associé à *col* défini

par : $col_{color}(e) = c$ si et seulement si $col(e) = (c, r)$. Le principe d'un coloriage ordonné utilisé comme un critère de couverture est de séparer les éléments de A en groupes de couleurs différentes puis de couvrir chaque groupe de couleur c par une suite $TS(c)$ de chemins de A. De plus, cette suite doit vérifier que pour tout chemin $p \in TS(c)$, la suite des rangs de la restriction de p aux éléments de couleur c est croissante. Formellement,

Définition 62 *Soient c une couleur, col un coloriage ordonné de A et $TS(c)$ un ensemble de chemins de A. $TS(c)$ satisfait le critère de couverture col pour la couleur c si et seulement si :*

1. *$TS(c)$ satisfait le critère de couverture col_{color} pour la couleur c.*
2. *Pour tout $p \in TS(c)$, pour tous les éléments e_i et e_j du chemin p (état ou transition), tels que $col(e_i) = (c, r_i)$ et $col(e_j) = (c, r_j)$, si e_i apparaît avant e_j dans p alors $r_i \leq r_j$.*

Intuitivement, en plus de la satisfaction par $TS(c)$ du critère de couverture col_{color}, les rangs des éléments de couleur c forment une suite croissante dans chaque chemin de $TS(c)$.

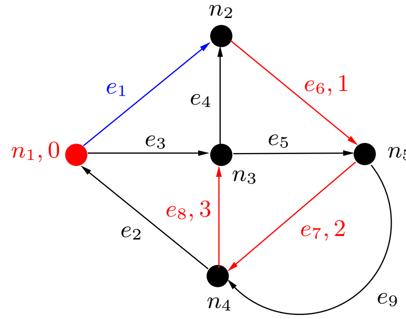


Fig. 56: Coloriage ordonné.

Exemple 11.2 *La* Fig.*56 illustre un exemple de coloriage ordonné col défini par (l'état initial est n_1) :*

- *n_1, e_6, e_7 et e_8 ont une couleur rouge.*
- *le rang de n_1 est 0, de e_6 est 1, de e_7 est 2 et de e_8 est 3.*
- *e_1 a une couleur bleue et un rang 0 (non représenté dans* Fig.*56).*
- *Les autres éléments ont une couleur noire et un rang 0 (non représenté dans* Fig.*56).*

Supposons que $friend(rouge) = \{noir\}$. Considérons les deux suites de chemins suivantes :

1. *$TS_1(rouge) = \{e_3.e_4.e_6.e_7.e_8\}$*
2. *$TS_2(rouge) = \{e_3.e_5.e_7.e_8.e_4.e_6\}$*

On remarque que $TS_1(rouge)$ et $TS_2(rouge)$ satisfont le critère de couverture col_{color} pour la couleur rouge (le même coloriage que dans la Fig.*55). Cependant $TS_2(rouge)$ ne satisfait pas le critère col du fait que e_7, dont le rang est 2, apparaît avant e_6 de rang 1.* □

Corollaire 12 *Les critères de couverture transitions, états, chemins et variables peuvent être représentés par un coloriage ordonné.* □

Preuve. Pour les trois premiers critères, on choisit col tel que le rang de tout élément soit égal à 0 et on construit col_{color} comme cela est indiqué dans la preuve du corollaire 11.
Considérons maintenant le critère de couverture de variables et soit v une variable de A. Notons par D_v l'ensemble des transitions où v est définie et U_v l'ensemble des transitions où v est utilisée. Pour tout élément e de D_v, nous définissons le coloriage ordonné col_e donné par :

1. $col_e(e) = (rouge, 1)$
2. $col_e(e_1) = (rouge, 2)$ pour tout élément e_1 de U_v.
3. $col_e(e_2) = (bleu, 0)$ pour tout élément e_2 de D_v tel que $e \neq e_2$.
4. $col_e(e_3) = (noir, 0)$ pour tout autre élément.
5. $friend_e(rouge) = \{noir\}$.

Soit $TS_e(rouge)$ un ensemble de chemins qui satisfait le critère col_e pour la couleur rouge. La suite $(TS_e(rouge))_{e \in D_v}$ satisfait le critère de couverture de l'ensemble $(col_e)_{e \in D_v}$. Elle satisfait aussi le critère de couverture des variables. □

11.1.5 Comment définir un coloriage et un coloriage ordonné

La définition d'un coloriage col et d'un coloriage ordonné col' est générale. Lors de l'utilisation de col et col' comme critères de couverture, certaines règles de définition doivent être respectées :

1. Dans le cas où on veut couvrir une transition e de couleur $c \in color$, col (resp. col') doit affecter une couleur à l'état destination de e ($dst(e)$) dans $friend(c) \cup \{c\}$. La même remarque pour l'état source de e ($src(e)$).
2. Dans le cas où on veut couvrir un état e de couleur $c \in color$, il doit exister au moins une transition e' telle que la couleur de e' et de $src(e')$ sont dans $friend(c) \cup \{c\}$.
3. Dans le cas de col', le rang de toute transition e doit être inférieur ou égal au rang de $dst(e)$ et supérieur ou égal au rang de $src(e)$.

En général, pour définir un critère de couverture comme un coloriage, on choisit trois couleurs : rouge, bleu et noir. La couleur rouge définit les comportements à couvrir. La couleur noire définit les comportements autorisés ($friend(rouge) = \{noir\}$). Enfin, la couleur bleue définit les comportements non autorisés [2]. En pratique et comme nous allons le voir dans la section 11.2, les comportements à couvrir portent sur les états.

[2]Dans certains cas, plus d'une couleur est à couvrir et par conséquent, une famille (ensemble) de coloriages peut être utilisée.

Finalement, signalons que la couverture d'un élément d'une couleur donnée, avec les conditions citées dans les définitions 60 et 62, n'est pas toujours possible du fait du "mauvais choix" du coloriage.

11.1.6 Conclusion

Dans cette section, nous avons introduit le coloriage et le coloriage ordonné. Un coloriage est une fonction qui assigne à chaque élément (transition ou état) d'un graphe G une couleur. L'ensemble des éléments de G de même couleur c représente un comportement à couvrir. La manière dont ces éléments doivent être couverts dépend de l'ensemble des couleurs $friend(c)$ amies à c. L'ensemble $friend(c)$ définit les couleurs des éléments permises dans les chemins qui couvrent c. En d'autres mots, aucun chemin qui couvre la couleur c ne doit contenir un élément (transition ou état) colorié par une couleur dans $color \backslash friend(c)$. Le coloriage ordonné étend le coloriage en assignant un rang aux éléments. Ce rang détermine l'ordre relatif d'apparition de l'élément. Nous avons aussi défini un critère de couverture comme un coloriage (resp. coloriage ordonné). Nous avons montré que les coloriages ordonnés peuvent exprimer les critères de couverture de transitions, états, chemins et variables.

Cette façon simple de décrire un critère de couverture comme un coloriage a l'avantage de définir un formalisme pour décrire les comportements à couvrir, indépendant de la signification de ces comportements. Par conséquent, la méthode de couverture est aussi indépendante de ces comportements.

Dans la section suivante, nous introduisons une méthodologie de génération de cas de test basée sur les coloriages et le modèle CS, en vu d'introduire le concept du test ouvert.

11.2 Modélisation uniforme du test

Comme nous l'avons dit auparavant, l'utilisation des spécifications formelles fournit un support pour l'automatisation de la génération des tests. Plusieurs modèles de base (LTS, IOLTS,...) et langages de description (LOTOS, IF, UML,...) ont été proposés pour décrire les protocoles et les comportements attendus par ces derniers d'une manière formelle. Dû à la nature des protocoles (ou fonctionnalités) testés, le contexte actuel du test offre une multitude de types de test : test de conformité, test d'interopérabilité, test dans le contexte, dans le cas d'une entité communicante à travers son environnement, (test de robustesse,...). Les façons pour tester les systèmes communicants peuvent être classées en deux groupes de base. L'approche la plus naturelle, dite le *test actif*, dérive les tests à partir de la spécification. L'autre approche, dite le *test passif* [1], contrairement à l'approche actif, ne génère pas des tests mais plutôt vérifie la validité d'une trace d'exécution d'une implantation (la trace est une exécution valide de la spécification).

Cette diversité dans les types, les modèles et les approches traduit la spécificité des

besoins. En effet, les différents types de test sont une conséquence de la composition des systèmes à tester et des architectures de test : la conformité considère seulement une entité, tandis que l'interopérabilité et le test dans le contexte considèrent plusieurs entités communicantes interagissant selon une architecture de test donnée. La multitude des modèles de base se justifie par des besoins différents de spécification : les systèmes ont des comportements événementiels, peuvent manipuler des données et être soumis à des contraintes temporelles. Que l'on soit dans le cadre du test actif ou bien dans le cadre du test passif, nous sommes confrontés au même problème. Il s'agit d'un problème d'accessibilité d'un état ou d'une transition de la spécification.

Dans cette section, nous traitons le test des protocoles dans l'optique d'une approche permettant d'appréhender les différents types et approches de test au sein du même modèle et avec la même méthodologie. Le modèle retenu est celui du modèle des systèmes communicants CS. La méthodologie choisit est celle des algorithmes génériques de génération (gga). Elle est basée sur la définition d'un critère de couverture sous forme d'un coloriage. Nous montrons que le modèle CS est un modèle générique pour le test dans le sens qu'il permet (i) de modéliser différents types de communication et d'architectures du test et (ii) d'appliquer le même algorithme générique de génération (gga) pour les différents types et approches du test. Ce résultat nous motive à introduire le concept du test ouvert expliqué dans la fin de cette section.

11.2.1 Méthodologie des algorithmes génériques de génération

La majorité des algorithmes de génération de test est basé sur une recherche en profondeur d'un état ou d'une transition cible dans le graphe d'accessibilité. Il est alors possible de définir un algorithme générique de génération pour les différents types de test. Dans cette partie, nous montrons comment définir un tel algorithme. Par souci de simplification, nous ne traitons pas le cas des coloriages ordonnés, mais l'approche présentée par la suite reste valable dans ce cas.

Définition 63 *Un système communicant sous test (CSUT) est un couple* (S, col) *tel que* $S = (SP, SV, SV_0, (M_i)_{1 \leq i \leq n}, Top)$ *est un CS et col est un coloriage de la sémantique de* S $(\zeta(S))$*. L'ensemble de tous les CSUTs est noté* $\mathcal{CSUT}$*.* □

Définition 64 *Un algorithme générique de génération (gga) pour* $\mathcal{CSUT}$ *est un algorithme qui calcule, pour tout* $(S, col) \in \mathcal{CSUT}$*, pour toute couleur* c*, un ensemble de chemins* $TS(c)$ *de* $\zeta(S)$ *tel que* $TS(c)$ *vérifie exhaustivement le critère de couverture col pour* c*.* □

Un algorithme est *gga* si pour une couleur c, *gga* retourne une suite de chemins $TS(c)$ de $\zeta(S)$ qui satisfait *col* pour la couleur c tel que $TS(c)$ est exhaustive. Un algorithme *gga* ne dépend ni du CSUT considéré ni de la couleur choisie. Nous supposons seulement

l'exhaustivité de l'algorithme et non pas du résultat.

Dans la définition de CSUT, le coloriage *col* est associé à $\zeta(S)$, ce qui implique sa construction. Or, en pratique, *col* est défini comme un coloriage distribué de $Comp(S) = Top \cup (M_i)_{i \in [1,n]}$. En conséquence, la construction de la sémantique du CS n'est plus nécessaire. Dans le cas d'utilisation d'un coloriage distribué, la définition de *gga* subit une légère modification.

Définition 65 *Un algorithme générique de génération distribué est un algorithme qui calcule, pour tout système S, pour tout coloriage distribué col et pour toute couleur c, un ensemble de chemins $TS(c)$ de la sémantique $\zeta(S)$ de S tel que : pour tout $A \in Comp(S)$, $TS_A(c)$ satisfait le critère de couverture col_A pour la couleur c, avec :*

1. *$TS_A(c)$ est la projection des chemins de $TS(c)$ sur l'alphabet de A.*
2. *col_A est le coloriage restriction de col aux éléments de A.* □

Par la suite, nous utiliserons le terme coloriage conjointement pour référencer un coloriage ou un coloriage distribué et le terme *gga* pour référencer un algorithme générique de génération ou un algorithme générique de génération distribué.

11.2.2 Modélisation de l'architecture de test

Considérons un système S composé des entités M_i, $i \in [1, n]$, $n \in \mathbb{N}$. Comme nous l'avons vu dans le section 5.2, S peut être décrit d'une façon naturelle dans le modèle CS. En effet, lorsque S est composé d'une seule entité, la description de S en CS considère une topologie statique (i.e. un seul état) et unitaire (i.e. un événement se synchronise avec lui même). Dans le cas où S est composé de plusieurs entités, la modélisation en CS dépend des communications autorisées entre les entités (M_i) (broadcast, multicast, unicast). Ces communications doivent être données explicitement pour pouvoir construire la modélisation en CS. Lors d'une communication binaire entres les entités (M_i), i.e. une entité échange un message avec une seule entité (le cas le plus simple), la construction de la description CS est immédiate. En effet, il suffit de considérer une topologie statique et binaire dont les vecteurs de synchronisations expriment les synchronisations possibles entre les entités (M_i). Par la suite, $CS(S)$ dénotera la description de S dans le modèle CS.

Modélisation de l'architecture de test

Une architecture de test pour S définit des points d'observation POs (resp. et de contrôle PCOs). Les POs et PCOs déterminent les canaux de communication auxquels le testeur peut accéder et par conséquent, les synchronisations vues par ce dernier. Une synchronisation sur un canal inaccessible au testeur ne sera pas visible.

La prise en compte de l'architecture de test dans la description de S (et par conséquent dans la génération de test) est relativement simple. En effet, l'alphabet du traducteur de $CS(S)$ détermine les actions globales observables des synchronisations. Si une synchronisation se fait sur un canal inaccessible alors elle n'est plus observable. En conséquence, à partir de $CS(S)$, il est possible de définir un $CS_\tau(S)$ tel que l'action globale des synchronisations inobservables est une action interne (τ). Ceci pour toute architecture de test.

Ainsi, du point de vue test, le modèle CS n'est pas seulement un modèle de description des composantes communicantes, mais aussi un modèle capable d'incorporer une architecture de test donnée.

Exemple de modélisation

Supposons que S est composé des entités $A = (S^A, s_0^A, L^A, C^A, P^A, V^A, V_0^A, Pred^A, Ass^A, \to_A)$ et $B = (S^B, s_0^B, L^B, C^B, P^B, V^B, V_0^B, Pred^B, Ass^B, \to_B)$ partageant les paramètres SP et les variables SV. SV_0 correspond aux valeurs initiales des variables. Notons par L^{AB} (resp. L^{BA}) l'ensemble des événements de L^A (resp. L^B) qui se synchronisent avec un événement de L^B (resp. L^A). Par exemple, si $L^A = \{?begin, ?end, !busy\}$ et $L^B = \{!end, ?busy, !CD\}$ alors $L^{AB} = \{?end, !busy\}$ et $L^{BA} = \{!end, ?busy\}$. Pour simplifier, nous supposons que pour tout $a \in L^{AB}$ il existe un unique $b \in L^{BA}$ qui se synchronise avec a.

Une modélisation en CS de S est donnée par $CS(S) = CS_1 = (SP, SV, SV_0, (A, B), TopS)$, avec $TopS$ l'automate de la FIG.57 (a). $TopS$ est une topologie statique. Le vecteur $< G, L^{AB}, L^{BA} >$ dénote l'ensemble des vecteurs de synchronisation de la forme $< g_{ab}, a, b >$ tels que $a \in L^{AB}$ synchronise avec $b \in L^{BA}$ et leur synchronisation donne lieu à une action globale observable g_{ab}. De même, le vecteur $< G^A, L_\tau^A \backslash L^{AB}, \bullet >$ dénote les vecteurs de la forme $< g_a, a, \bullet >$ tel que $a \in L_\tau^A \backslash L^{AB}$ [3].

Maintenant, supposons que la synchronisation des événements de L^{AB} avec les événements de L^{BA} soit inobservable comme c'est le cas pour l'architecture de test boîte noire. Alors, la modélisation de S en CS est $CS_\tau(S) = CS_2 = (SP, SV, SV_0, (A, B), TopS')$, avec $TopS'$ l'automate de la FIG.57 (b). Le vecteur $< \tau, a, b >$ de $< \tau, L^{AB}, L^{BA} >$ considère que la synchronisation de a avec b donne lieu à une action interne τ. D'une façon générale, on peut modéliser la synchronisation en actions internes d'une partie des événements de synchronisations.

11.2.3 Modélisation de l'approche passive du test

L'approche passive du test considère des traces d'exécution (observations) d'une implantation et vérifie leur validité par rapport à la spécification. Une trace peut contenir des valeurs pour les variables et les horloges. Elle ne commence pas forcement de l'état initial

[3] Rappelons que pour deux ensembles L_1 et L_2, $L_1 \backslash L_2$ est l'ensemble $\{a \,|\, a \in L_1 \wedge a \notin L_2\}$.

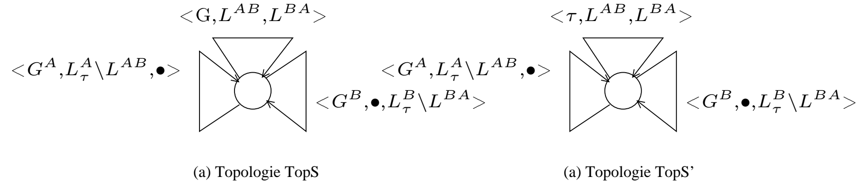


FIG. 57: Architecture de test.

de l'implantation, mais des observations réalisées lors du lancement du testeur qui peut ne pas coïncider avec le lancement de l'implantation.

Trace passive

Une trace d'exécution peut être vue comme un automate étendu linéaire, i.e dont le graphe associé est linéaire (un chemin). Nous proposons de modéliser une trace par un ETIOA $PTrace$ vérifiant certaines conditions. Tout d'abord, $PTrace$ est observable, i.e. aucune transition n'est pas étiquetée par τ. $PTrace$ ne possède pas de paramètres vu qu'il correspond à une exécution du système et donc les paramètres ont été déjà instanciés. L'ensemble des horloges de $PTrace$ est restreint à une seule horloge. Cette horloge peut être :

- globale (universelle) et sans remise à zero notée par h. Dans ce cas, nous supposons que l'horloge h est démarrée en même temps que le démarrage de l'implantation. Ainsi, $PTrace$ sera décrit avec h si le testeur connaît l'instant du démarrage de l'implantation.
- avec remises à zéro (non globale), notée h_l. Dans ce cas, nous supposons que h_l n'est pas utilisée dans une transition (n'apparaît pas dans la garde d'une transition) avant sa définition (sa remise à zéro) auparavant dans une transition qui précède la transition courante. Ainsi, $PTrace$ sera décrit avec h_l si le testeur ne connaît pas l'instant du démarrage de l'implantation. h_l permettra seulement de décrire les délais entre les observations des événements.

Définition 66 *Soit* $CS_1 = (SP, SV, SV_0, (M_i)_{i\ in[1,n]}, (\Sigma, Tr = (S^{tr}, s_0^{tr}, L^{tr}, \rightarrow_{tr})))$ *un système communicant. Une trace passive de* CS_1 *est un ETIOA linéaire et observable* $PTrace = (S, s_0, L, C, P, V, V_0, Pred, Ass, \rightarrow)$ *tel que :*

1. $L \subseteq L^{tr}$,
2. $C = \{h\}$ *ou* $C = \{h_l\}$,
3. $P = \emptyset$,
4. $V \subseteq SV$,
5. $V_0 = \emptyset$,
6. $Ass = \emptyset$ *si* $C = \{h\}$ *et* $Ass = \{h_l := 0\}$ *si* $C = \{h_l\}$. □

Une trace passive d'un système modélisé par une CS CS_1 est un ETIOA linéaire, observable, sans paramètre et ayant une seule horloge. L'alphabet de $PTrace$ est inclus dans l'alphabet du traducteur de CS_1, étant donné que ce dernier correspond aux actions observables du système. Les variables utilisées dans une trace passive sont des variables visibles du système. Nous imposons que ses variables figurent dans les variables partagées du système, vu que ces dernières sont visibles par les autres composantes (et par l'environnement). Remarquons que, dans le cas d'un système communicant composé d'une seule entité A, A peut être modélisée par un CS dans lequel les variables de A figurant dans la trace passive sont des variables partagées du CS.

Modélisation du test passif

Nous proposons de modéliser le test passif de $CS_1 = (SP, SV, SV_0, (M_i)_{i\ in[1,n]}, (\Sigma, Tr = (S^{tr}, s_0^{tr}, L^{tr}, \rightarrow_{tr})))$ par la trace passive $PTrace = (S, s_0, L, C, P, V, V_0, Pred, Ass, \rightarrow)$.

Définition 67 *La modélisation passive de* CS_1 *par la trace passive* $PTrace = (S, s_0, L, C, P, V, V_0, Pred, Ass, \rightarrow)$ *est le CS* $CS'_1 = (SP', SV', SV'_0, ((M_1)_{i\in[1,n]} \cup PTrace), PTop')$ *défini par :*

1. $SP' = SP$,
2. $SV' = SV$,
3. $SV'_0 = SV_0$,
4. $PTop' = (\Sigma', Tr' = (S^{tr'}, s_0^{tr'}, L^{tr'}, \rightarrow_{tr'}))$ *tel que :*

   (a) *Le traducteur* Tr' *est défini par :*
   - $S^{tr'} = \{s_1 \| s_2 \,|s_1 \in S, s_2 \in S^{tr}\}$
   - $s_0^{tr'} = s_0 \| s_0^{tr}$
   - $L^{tr'} \subseteq L$
   - $\rightarrow_{tr'}$ *est définie par :*

     i. $s_1 \xrightarrow{pred,a,ass} s'_1,\ s_2 \xrightarrow{a}_{tr} s'_2,\ a \neq \tau \quad \Rightarrow \quad s_1 \| s_2 \xrightarrow{a}_{tr'} s'_1 \| s'_2$

     ii. $s_2 \xrightarrow{\tau}_{tr} s'_2 \quad \Rightarrow \quad s_1 \| s_2 \xrightarrow{\tau}_{tr'} s_1 \| s'_2$

   (b) Σ' *et la fonction de composition* f' *sont définies par (*f *est la fonction de composition de* CS_1*) :*
   - $\Sigma' = \Sigma \times (L \cup \{\bullet\})$
   - *Pour tout* $a \in L_\tau^{tr'}$ *tel que* $a = f(a_1, ..., a_n)$ *:*

     i. $a = f'(a_1, ..., a_n, a)$ *si* $a \neq \tau$.

     ii. $a = f'(a_1, ..., a_n, \bullet)$ *si* $a = \tau$. □

Le traducteur associé à CS' a le même comportement observable que $PTrace$. En effet, la fonction de composition f' synchronise entre chaque événement a de $PTrace$ et le même

a de Tr. Elle autorise aussi le traducteur de CS_1 à réaliser des actions internes, du fait que ces dernières ne sont pas observables et par conséquent ne figurent pas dans $PTrace$.

Maintenant, considérons le coloriage distribué col de $Comp(CS'_1)$ qui affecte à l'état terminal (le dernier état du chemin) de $PTrace$ la couleur rouge et aux autres éléments des composantes de $Comp(CS'_1)$ la couleur noire telle que $friend(rouge) = \{noir\}$. Soit gga un algorithme générique de génération pour le système sous test (CS'_1, col).

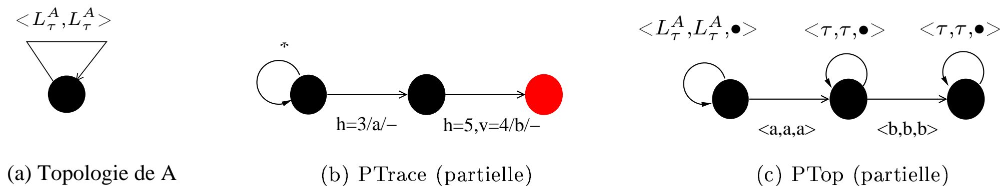
Corollaire 13 *Si gga retourne un ensemble non vide de chemins pour la couleur rouge alors $PTrace$ est une trace valide de CS_1, sinon elle ne l'est pas.* □

Exemples de modélisation

Nous allons donner deux exemples du test passif dans le cas d'un système composé d'une seule entité. Ces exemples sont facilement exportables dans le cas de plusieurs entités.

Trace partielle. Soit une spécification S exprimée sous forme d'un ETIOA A. La modélisation $CS(S)$ en CS de S considère la topologie statique donnée dans la FIG.58 (a). Supposons que la trace modélisée par l'ETIOA $PTrace$ de la FIG.58 (b) soit une trace d'une implantation de S. Dans cette trace, l'implantation a effectuée des actions (le symbole '*') puis l'action $a \in L^A$ à l'instant 3, puis l'action $b \in L^A$ à l'instant 5 selon l'horloge globale h telle que la valeur de la variable partagée v vaut 4.

La modélisation passive de $CS(S)$ et $PTrace$ est le CS défini par $CS'(S) = (\emptyset, \{v\}, \emptyset, (A, PTrace), PTop)$, avec $PTrace$ l'ETIOA de la FIG.58 (b) et $PTop$ la topologie de la FIG.58 (c). Nous avons colorié le dernier état de $PTrace$ avec le rouge et les autres éléments de A et $PTop$ avec le noir. Ainsi, $CS'(S)$ et ce coloriage forment un CSUT. Dès lors, gga permet de décider si $PTrace$ est une trace valide de A : si gga retourne un ensemble vide ($TS(rouge) = \emptyset$) alors $PTrace$ n'est pas valide, sinon c'est une trace valide.

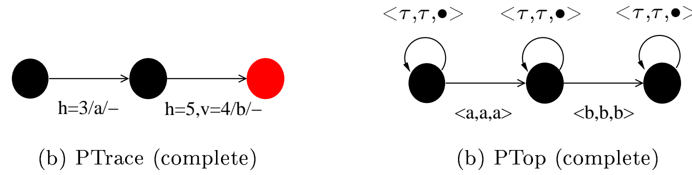


(a) Topologie de A (b) PTrace (partielle) (c) PTop (partielle)

FIG. 58: Test passif (1)

Trace complète. La FIG.59 (b) illustre la même trace passive que l'exemple précédent mais dans une observation complète, i.e. le testeur a démarré l'observation de l'implantation au

démarrage de cette dernière. Le seule différence sur $PTop$ est qu'on n'autorise que la synchronisation sur a et b ou une action interne de A. C'est une topologie partielle.



(b) PTrace (complete) (b) PTop (complete)

FIG. 59: Test passif (2)

Remarque 13 *La trace passive $PTrace$ est considérée comme une entité de CS' à part entière sans aucune distinction par rapport aux autres entités. Ceci permet d'étendre la forme des traces que le test passif considère à des traces sous forme d'ETIOAs définissant des états finaux. Finalement, l'exemple du test passif de la spécification S est un test à une composante. Dans le cas de plusieurs composantes, il se peut que le testeur ne réalise que des observations séparées de chaque composante. La difficulté dans ce cas est de réordonner les différentes traces pour construire une trace globale. Nous pensons que les techniques d'estampillage et spécialement la technique présentée dans [83], peuvent être utilisées. Ce sujet dépasse le cadre de ce document et mérite plus d'investigation.* □

11.2.4 Modélisation de l'approche active du test

Dans le test actif, la dérivation se fait à partir de la spécification. Cette dérivation peut ne concerner qu'une partie de la spécification dans le but de limiter l'explosion combinatoire des états, comme c'est le cas pour la technique de génération orientée objectif de test. Contrairement au test passif, les travaux relatifs au test actif considèrent une ou plusieurs entités communicantes [46, 57, 71, 92, 144].

Dérivation à partir du comportement global

Pour un CSUT S, la suite de chemins $TS()$ générés par un algorithme gga peut être utilisée pour dériver des cas de test de S. gga peut être alors une adaptation des méthodes UIO, Wp,... Nous nous intéressons par la suite à la technique de dérivation orientée objectif de test.

Modélisation de test actif orienté TP

L'utilisation de l'objectif de test comme critère de sélection permet de restreindre le comportement testé à une partie de la spécification. Soit CS =

$(SP, SV, SV_0, (M_i)_{i\ in[1,n]}, (\Sigma, Tr = (S^{tr}, s_0^{tr}, L^{tr}, \rightarrow_{tr})))$ un CS. Rappelons qu'un objectif de test (TP) pour CS est un ETIOA acyclique $TP = (S, s_0, L, C, P, V, V_0, Pred, Ass, \rightarrow)$ équipé de deux ensembles $Accept \subseteq S$ et $Reject \subseteq S$ tels que : $L \subseteq L^{tr}$, $C \backslash \{h\} \cap C^i \backslash \{h\} = \emptyset$ pour tout $i \in [1, n]$, $P \subseteq SP$, $V \subseteq SV$, $V_0 = \emptyset$.

Nous proposons de modéliser le test actif de $CS_2 = (SP, SV, SV_0, (M_i)_{i\ in[1,n]}, (\Sigma, Tr = (S^{tr}, s_0^{tr}, L^{tr}, \rightarrow_{tr})))$ et $TP = (S, s_0, L, C, P, V, V_0, Pred, Ass, \rightarrow)$ en CS. Pour ce faire, nous supposons que TP est complet (voir la définition 39 de la section 8.3) ce qui implique que $L = L^{tr}$.

Définition 68 *La modélisation active de CS_2 et TP est le CS $CS_2' = (SP', SV', SV_0', ((M_1)_{i\in[1,n]} \cup TP), TPTop')$ défini par :*

1. $SP' = SP$,
2. $SV' = SV$,
3. $SV_0' = SV_0$,
4. $TPTop' = (\Sigma', Tr = (S^{tr}, s_0^{tr}, L^{tr}, \rightarrow_{tr}))$
5. Σ' *et la fonction de composition f' sont définis par (f est la fonction de composition de CS_2) :*
   - $\Sigma' = \Sigma \times (L^{tr} \cup \{\bullet\})$
   - *Pour tout $a \in L_\tau^{tr'}$ tel que $a = f(a_1, ..., a_n)$:*
     (a) $a = f'(a_1, ..., a_n, a)$ *si* $a \neq \tau$.
     (b) $a = f'(a_1, ..., a_n, \bullet)$ *si* $a = \tau$. □

TP est considéré comme une composante de CS_2'. Le traducteur de CS_2' est exactement le traducteur associé à CS_2. La fonction de composition f' est une extension de f sur les événements de TP.

Maintenant, considérons le coloriage distribué *col* de $Comp(CS_2')$ qui affecte aux états *Accept* de TP la couleur rouge, aux états *Reject* de TP la couleur bleue et aux autres éléments des composantes de $Comp(CS_2')$ la couleur noire telle que $friend(rouge) = \{noire\}$. Soit *gga* un algorithme générique de génération pour le système sous test (CS_2', col).

Corollaire 14 *Les chemins $TS(rouge)$ retournés par gga pour la couleur rouge et le CSUT (CS_2', col) vérifient TP.* □

Remarque 14 *La modélisation active de la définition 68 n'est pas unique pour le test actif. En effet, une autre modélisation possible est donnée par le même CS de la définition 68 tel que son traducteur Tr' est défini par :*

1. $s_2 \xrightarrow{\tau}_{tr} s_2' \quad \Rightarrow \quad s_1 \| s_2 \xrightarrow{\tau}_{tr'} s_1 \| s_2'$
2. $s_1 \xrightarrow{pred,a,ass} s_1',\ s_2 \xrightarrow{a}_{tr} s_2',\ a \neq \tau \quad \Rightarrow \quad s_1 \| s_2 \xrightarrow{a}_{tr'} s_1' \| s_2'$

Exemple d'application

La FIG.60 (a) illustre un objectif de test pour une spécification S ($CS(S)$ a une topologie statique). TP teste que l'implantation peut effectuer l'action a puis l'action b dans l'intervalle $[2, 3]$ selon l'horloge globale h. De plus, TP interdit l'apparition de l'événement c dans le comportement testé. L'objectif de test complet associé à TP est donné dans la FIG.60 (b). Rappelons qu'une transition $q \xrightarrow{\star} q$ dénote les transitions $q \xrightarrow{a} q$ telles qu'il n'existe pas de transition $q \xrightarrow{a} q'$, avec q' est un état *Reject*. La topologie $TPTop$ est définie dans FIG.60 (c). Le vecteur $< L^S, L^S, L^S >$ dénote les vecteurs $< a, a, a >$ $(a \in L^S)$. Nous avons colorié les états *Accept* du TP complet en rouge, les états *Reject* en bleu et les autres états de CS_2' en noir. Ainsi, CS_2' devient un CSUT. Dès lors, une application directe de l'algorithme *gga* permet de générer les comportements vérifiant TP.

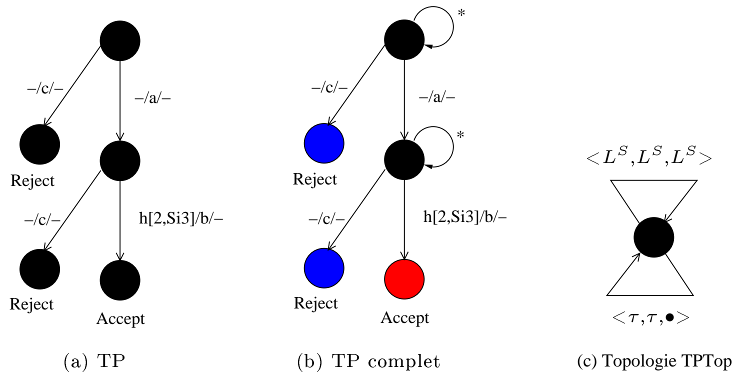


FIG. 60: Test actif d'une composante

En conclusion, le test actif orienté objectif de test consiste en une modélisation en CS, le choix d'un coloriage et l'application de la méthodologie *gga*.

11.2.5 Comparaison

Dans cette section, nous avons montré que les différents types, approches et architectures peuvent être traités d'une manière uniforme, basée sur le modèle CS et la méthodologie *gga*, reposant sur la définition d'un critère de couverture comme un coloriage (resp. coloriage ordonné).

À notre connaissance, ce sujet ainsi que les résultats obtenus sont nouveaux dans le domaine du test. Un grand intérêt de ces résultats est la possibilité de construction d'outils de test pouvant supporter différents types, approches et architectures de test.

Finalement, concernant le test passif, la majorité des travaux ont été menés par Ana Cavalli, David Lee et al. L'approche adoptée dans [1] [4] consiste en une analyse en arrière

[4] Il existe un grand nombre d'articles sur le test passif, nous avons choisi d'en citer que [1], vu qu'il considère aussi la partie données dans une trace (mais ne considère pas la partie temporelle).

de la spécification contenant une seule composante. Notre approche considère une analyse en avant, réalisée par *gga*, de la modélisation en CS du test passif.

11.2.6 Test ouvert

Nous pensons qu'il n'existe pas d'arguments forts pour différencier les différents critères relatifs au test des protocoles. Ceci nous suggère d'introduire le concept du **test ouvert**.

Définition 69 (Test Ouvert) *Étant donné un ensemble d'implantations* (I_i)*, des spécifications* (S_i) *et une propriété* P*, le test ouvert de* (I_i) *par rapport à* (S_i) *consiste à vérifier que les* (I_i) *satisfont les* (S_i) *selon la propriété* P *:*

Test ouvert : (I_i) **satisfont** (S_i) **selon** P

La propriété P peut être l'appartenance d'une trace de (I_i) à (S_i) (test passif), la conformité de (I_i) par rapport à (S_i), l'interopérabilité des (I_i) par rapport à (S_i),...

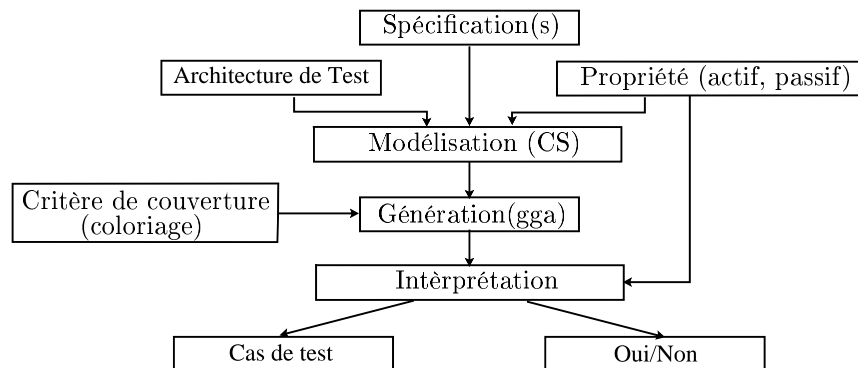


FIG. 61: Activités du test ouvert.

Les différentes activités du test ouvert sont exprimées dans la FIG.61. Les entrées du test ouvert sont une spécification, une architecture de test, et une propriété (test passif, test actif, conformité,...) du test. La première étape consiste à décrire ces entrées dans un modèle. La modèle de description peut être le modèle CS comme tout autre modèle pouvant supporter les différents critères de test. Dans la deuxième phase, un critère de couverture est choisi ainsi qu'une méthodologie associée. Ici, nous avons représenté le critère "coloriage" et la méthodologie "*gga*". Les résultats de la méthodologie sont alors interprétés dans la troisième phase selon la propriété d'entrée (passive ou active). Dans le cas d'une propriété passive, l'interprétation correspond à un verdict "Oui" ou "Non". Dans le cas contraire, on obtient des cas de test.

Cinquième partie

Implémentation et Outils

Chapitre 12

Implémentation

Sommaire

12.1 Représentation symbolique . . . 193
12.1.1 Préliminaires . . . 193
12.1.2 Représentation symbolique des polyèdres : DBMs . . . 194
12.1.3 Implantation des opérations sur les polyèdres . . . 195
12.2 Trace symbolique . . . 198
12.2.1 Préliminaires . . . 198
12.2.2 Trace symbolique . . . 199

12.1 Représentation symbolique

La manipulation des polyèdres requiert une structure de données pour représenter les polyèdres. Cette structure de données doit permettre de tester l'inclusion de deux polyèdres et de calculer aisément les différentes opérations définies sur les polyèdres, i.e. l'intersection, le successeur et le prédécesseur, l'image par une remise à zéro,... Dans cette section, nous allons présenter la structure de données DBM (Difference Bound Matrix), ainsi que l'implantation des différentes opérations définies sur les polyèdres.

12.1.1 Préliminaires

Une borne est un couple $(c, \prec)$ tel que $c \in \mathbb{Z}$ et $\prec \in \{\leq, <\}$. Un ordre total est alors défini sur les bornes de la façon suivante : si $m = (c, \prec)$ et $m' = (c', \prec')$ alors $m < m'$ ssi $c < c'$ ou $c = c'$ et $\prec$ est plus stricte que $\prec'$; $m \leq m'$ si $m < m'$ ou m et m' sont identiques. Nous définissons aussi l'opération d'addition de m et m' par $m + m' = (c + c', \prec \wedge \prec')$.

Le complément de $m = (c, \prec)$, noté $-m$, est la borne $(-c, \prec)$. Finalement, $min(m, m')$ est égale à m si $m \leq m'$ et à m' sinon.

Dans le reste de cette partie, $V = \{v_1, ..., v_n\}$ dénotera un ensemble de variables à valeurs dans $\mathbb{R}^{\geq 0}$ et $V_0 = V \cup \{v_0\}$ l'ensemble des variables V muni d'une variable fictive v_0 qui vaut 0 tout le temps.

12.1.2 Représentation symbolique des polyèdres : DBMs

Il existe plusieurs structures de données symboliques qui permettent de représenter les polyèdres. Ici nous nous limiterons à la description des DBMs [52] (Difference Bound Matrix) ou "matrice des bornes", la structure de données introduite par David Dill. Pour plus d'informations sur les autres structures, se référer à [52, 35].

Cette représentation est particulièrement intéressante car elle donne lieu à des algorithmes simples et performants pour le calcul de la forme canonique. Par exemple, l'algorithme de Floyd-Warshall de recherche des plus courts chemins dans un graphe valué [58], présenté par la suite, est de complexité $O(n^3)$. Cet algorithme permet à la fois de calculer la forme canonique d'un polyèdre et de tester s'il est vide (en testant la présence des boucles négatives, i.e. de nombres négatifs sur la diagonale).

Définition 70 *Une matrice de bornes (DBM) de dimension n est une matrice carrée $(n+1) \times (n+1)$, dont les éléments sont des bornes. Si $M = (m_{ij})$ est une DBM, alors m_{ij} est l'élément de la ligne i et de la colonne j.*

L'idée derrière la représentation d'un polyèdre par une DBM $M = (m_{ij})$ est la suivante : un élément m_{ij} représentera la borne supérieure de la différence des variables $v_i - v_j$. Par exemple, la contrainte $v_i - v_j \leq 5$ sera codée par $m_{ij} := (5, \leq)$, tandis que la contrainte $v_i - v_j \geq 2$ sera codée par $m_{ji} := (-2, \leq)$. La ligne et la colonne d'indice 0 sont utilisées pour coder les contraintes sur une seule variable, par exemple $x_i \leq 3$ sera codée par $m_{i0} := (3, \leq)$ et $x \geq 4$ sera codée par $m_{0i} := (-4, \leq)$.

Formellement, considérons un polyèdre Z et $G(Z)$ son graphe de contraintes. Z est représenté par la DBM $M = (m_{ij})$ de dimension n, telle que :

$$\begin{cases} m_{ij} := (l_{ij}, \leq) & ssi\ v_j \xrightarrow{l_{ij}} v_i \text{ est un arc de } G, i \neq j \\ m_{ii} := (0, \leq) \end{cases}$$

Par exemple, si $n = 2$, le polyèdre $v_2 \leq 2 \wedge v_1 \geq 2 \wedge v_1 - v_2 \leq 3$ peut être représenté par la DBM M :

$$M = \begin{array}{cccc} & v_0 & v_1 & v_2 \\ v_0 & (0, \leq) & (-2, \leq) & (\infty, \leq) \\ v_1 & (\infty, \leq) & (0, \leq) & (3, \leq) \\ v_2 & (2, \leq) & (\infty, \leq) & (0, \leq) \end{array}$$

Inversement, toute DBM M de dimension n représente un polyèdre Z sur V tel que :

$$Z = \bigwedge_{v_i, v_j \in V_0, i \neq j} (v_i - v_j \leq m_{ij})$$

Par exemple, la DBM

$$M = \begin{array}{cccc} & v_0 & v_1 & v_2 \\ v_0 & (0, \leq) & (\infty, \leq) & (-3, \leq) \\ v_1 & (2, \leq) & (0, \leq) & (-1, \leq) \\ v_2 & (\infty, \leq) & (\infty, \leq) & (0, \leq) \end{array}$$

représente le polyèdre $Z = v_1 \leq 2 \wedge v_2 \geq 3 \wedge v_1 - v_2 \leq -1$ (ou encore $v_1 \leq 2 \wedge v_2 \geq 3$).

Coder un polyèdre par une DBM demande alors $O(n^2)$ d'espace mémoire. Plusieurs algorithmes ont été proposés pour réduire l'espace mémoire nécessaire pour représenter un polyèdre [155, 95]. Ces réductions sont basées, en général, sur des transformations du graphe de contraintes.

12.1.3 Implantation des opérations sur les polyèdres

Nous allons à présent décrire l'implantation des opérations sur les polyèdres en utilisant leur représentation sous forme de DBMs.

```
Forme_Canonique(matrice M = (m_ij))
{
  Pour k = 0 à n Faire
    Pour i = 0 à n Faire
      Pour j = 0 à n Faire
          m_ij := min{m_ij, m_ik + m_kj} ;
      Si(m_ii < (0, ≤)) Alors retourner M_∅ ;
  retourner M ;
}
```

Fig. 62: Mise en forme canonique : algorithme de Floyd-Warshall.

Forme canonique

Il faut noter que deux DBMs différentes peuvent représenter le même polyèdre. La mise sous forme canonique permet d'affirmer que deux DBMs représentent le même polyèdre si elles sont identiques. Cette forme réduite est obtenue en appliquant un algorithme des plus courts chemins. L'algorithme est dû à Floyd-Warshall (Fig.62) permet ce calcul. Si

un élément de la diagonale est négatif, alors le polyèdre représenté par cette matrice est vide et l'algorithme retourne la matrice $M_\emptyset$ dont chaque élément est égal à $(0, <)$. Par convention, $M_\emptyset$ représentera la forme canonique du polyèdre vide.

Par la suite, la forme canonique d'une DBM M sera notée $cf(M)$. Si $M = cf(M)$, alors M est dite *canonique*. Soient $M = (m_{ij})$ et $M' = (m'_{ij})$ les DBMs canoniques de dimension n représentant respectivement les polyèdres Z et Z'.

Test d'inclusion : $Z \subseteq Z'$

$$Z \subseteq Z' \ ssi \ \forall 0 \leq i, j \leq n, \ m_{ij} \leq m'_{ij}$$

Ainsi, tester l'inclusion de deux polyèdres se fait en temps linéaire (taille de la matrice).

Intersection : $Z \cap Z'$

$Z \cap Z'$ est représenté par la DBM $M'' = (m''_{ij})$ telle que :

$$m'' = min(m_{ij}, m'_{ij})$$

Notons qu'il peut arriver que M'' ne soit pas canonique.

Successeur : $Z^\uparrow$

$Z^\uparrow$ est représenté par la DBM canonique $M' = (m'_{ij})$ telle que :

$$m'_{ij} = \begin{cases} (\infty, \leq) & si \ j = 0 \\ m_{ij} & sinon \end{cases}$$

Prédécesseur : $Z^\downarrow$

$Z^\uparrow$ est représenté par la DBM (pas forcement canonique) $M' = (m'_{ij})$ telle que :

$$m'_{ij} = \begin{cases} (0, \leq) & si \ i = 0 \\ m_{ij} & sinon \end{cases}$$

Successeur après ré-initialisation : $Z[X := 0]$

Soit $X \subseteq V$ un ensemble de variables à remettre à zéro. Soit la fonction totale $\lambda : V_0 \mapsto V_0$ définie par : pour toute $v_i \in V_0$,

$$\lambda(v_i) = \begin{cases} v_i & si\ v_i \notin X \\ v_0 & sinon \end{cases}$$

Notons d'abord que la remise à zéro d'une horloge $v_i \in X$ est équivalent à remplacer v_i par $v_0 = \lambda(v_i)$. Maintenant, lorsqu'on remplace v_i par v_j, alors v_i et v_j deviennent égales et toutes les contraintes sur v_j deviennent des contraintes sur v_i. Ainsi $Z[X := 0]$ est représenté par la DBM (pas forcement canonique) $M' = (m'_{ij})$ telle que pour toutes $v_i, v_j \in V_0$,

$$si\ \lambda(v_i) = v_j\ alors\ ligne_i(M') = ligne_j(M)\ et\ colonne_i(M') = colonne_j(M)$$

où $ligne_i(M)$ (resp. $colonne_i(M)$) représente la ligne (resp. colonne) de M d'indice i. Ainsi, M' est obtenue par des remplacements de lignes et de colonnes.

Prédécesseur après ré-initialisation : $[X := 0]Z$

Rappelons qu'une variable v_i de $[X := 0]Z$ est remplacée dans Z par $\lambda(v_i)$. Maintenant, supposons qu'on a deux contraintes $v_k - v_l \leq l_{kl}$ et $v_r - v_s \leq v_{rs}$ et qu'on remplace (i) v_k et v_r par v_i et (ii) v_l et v_s par v_j. On obtient alors les contraintes $v_i - v_j \leq v_{kl}$ et $v_i - v_j \leq v_{rs}$ et ainsi $v_i - v_j \leq min(v_{kl}, v_{rs})$. Ainsi, $[X := 0]Z$ est représenté par la DBM $M' = (m'_{ij})$ telle que pour toute $v_i \in V_0$:

$$m'_{ij} = min\{m_{kl} \,|\, \lambda(v_k) = v_i \wedge \lambda(v_l) = v_j\}$$

c-clôture : $close(Z, c)$

$close(Z, c)$ est représenté par la DBM $M' = (m'_{ij})$ telle que pour $0 \leq i \neq j \leq n$:

$$m'_{ij} = \begin{cases} (0, \leq) & si\ m_{ij} > (c, \leq) \\ (-c, \leq) & si\ m_{ij} + (c, \leq) < (0, \leq) \\ m_{ij} & sinon \end{cases}$$

Ainsi, une borne supérieure telle que $v_i \leq c'$, avec $c' > c$, est remplacée par $v_i \leq \infty$ et une borne inférieure telle que $v_i \geq c'$, avec $c' > c$, est remplacée par $v_i \geq c$. Les autres bornes restent inchangées.

Extraction de valuations de bornes

Le calcul des valuations de bornes (minimales et maximales) est direct. Pour tout $k \in [0, n]$:

1. La valuation $\nu_k^M(Z)$ est définie par :
   - Si $k = 0$ alors $\nu_k^M(Z)(v_i) = m_{i0}$.
   - Sinon $\nu_k^M(Z)(v_i) = -m_{0k} + m_{ik}$, et $\nu_k^M(Z)(v_k) = -m_{0k}$

   pour tout $i \in [1, n]$, $i \neq k$.
2. La valuation $\nu_k^m(Z)$ est définie par :
   - Si $k = 0$ alors $\nu_k^M(Z)(v_i) = -m_{0i}$.
   - Sinon $\nu_k^m(Z)(v_i) = m_{k0} - m_{ki}$, et $\nu_k^m(Z)(v_k) = m_{k0}$

   pour tout $i \in [1, n]$, $i \neq k$.

12.2 Trace symbolique

Dans cette section, nous étudierons l'exécutablité d'un chemin dans un ETIOA à travers la définition de la trace symbolique.

12.2.1 Préliminaires

Considérons un ETIOA $M = (S, s_0, L, C, P, V, V_0, Pred, Ass, \rightarrow)$ et un chemin $\rho = t_1 ... t_n$ de M de l'état initial tel que $t_i = (s_{i-1}, pred_i, a_i, ass_i, s_i)$ pour tout $i \in [1, n]$.

Définition 71 *ρ est dit exécutable (ou faisable) s'il existe des valeurs pour les paramètres P et les variables V et des instants de tir des transitions t_i tels que les prédicats $pred_i$ soient vérifiés, pour tout $i \in [1, n]$.* □

Exemple 12.1 *Considérons le chemin ρ défini par :*

$$\rho \, : \, s_0 \xrightarrow{x \leq 2/?a/y:=0} s_1 \xrightarrow{x \geq p \wedge y \leq 2/!b/v=p+1} s_2 \xrightarrow{v \geq 5/?c} s_3$$

avec p un paramètre et v une variable. On peut remarquer que :

- *Le prédicat $v \geq 5$ de la transition de source s_2 et de destination s_3 est vérifié si et seulement si $p \geq 4$. En effet, la dernière valeur de v avant cette transition correspond à l'affectation $v = p + 1$.*
- *Le prédicat $x \geq p \wedge y \leq 2$ de la transition de source s_2 et de destination s_3 est vérifié si et seulement si $p \leq 4$. En effet, le temps d'attente maximale dans l'état s_1 correspond à la borne de $y \leq 2$ du fait que y est remise à zéro dans la transition entrante à s_1, ce qui implique que la valeur de x est inférieure à 4.*

En conclusion, la seule valeur de p pour que le chemin ρ soit exécutable est 4. Une trace temporelle dans ce cas est $(?a, 2).(!b, 4).(?c, 5)$. □

Comme nous venons de le voir dans cet exemple, pour décider de l'exécutabilité d'un chemin, on est amené à résoudre un système de contraintes associé à ρ. Dans la section qui suit, nous allons décrire comme construire le système de contraintes associé à ρ à travers le calcul de la trace symbolique.

12.2.2 Trace symbolique

Tout d'abord, rappelons que pour un ETIOA $M = (S, s_0, \Sigma, C, P, V, V_0, Pred, Ass, \rightarrow)$, l'ensemble des prédicats $Pred$ et l'ensemble des mises à jour Ass sont définis par [1] :

- $Pred = \Phi(C, P, V) \cup \tilde{P}[P, V]$ tel que :
  1. $\tilde{P}[P, V]$ est un ensemble d'inégalités linéaires sur V et P.
  2. $\Phi(C, P, V)$ est défini par :

$$\phi := \phi_1 \, |\phi_2| \, \phi_1 \wedge \phi_2, \;\; \phi_1 := x \leq f(P, V), \;\; \phi_2 := f(P, V) \leq x$$

  avec x une horloge de C et $f(P, V)$ une expression linéaire de P et V.
- $Ass = \{x := 0 \,|\, x \in C\} \cup \{v := f(P, V) \,|\, v \in V\}$

Par la suite, nous supposons que pour l'ETIOA $M = (S, s_0, \Sigma, C, P, V, V_0, Pred, Ass, \rightarrow)$ et le chemin $\rho = t_1 ... t_n$ de M partant de l'état initial, les ensembles C, V et V_0 sont définis par : $C = \{c_1, ..., c_k\}$, $V = \{v_1, ..., v_m\}$ et $V_0 = \{v_{01}, ..., v_{0m}\}$.

Pour pouvoir décider de l'exécutabilité de ρ, la première étape consiste à remplacer les occurrences des variables par leurs valeurs qui peuvent changer d'une transition à l'autre. Dans la deuxième étape, les différentes horloges sont remplacées, dans chaque transition, par leurs instants de tir. Le chemin ne contenant que des paramètres et des instants de tir de chaque transition ainsi obtenu est appelé *la trace symbolique de ρ*. La procédure **SymbolicTrace** de la FIG.63 calcule la trace symbolique associée à un chemin ρ.

L'entrée de l'algorithme est un chemin ρ de M. La sortie est un chemin ρ' dont le prédicat, de chaque transition, est en fonction des paramètres P, des valeurs V_0 et de l'ensemble $\{h_1, ..., h_n\}$ correspondant aux instants de franchissement des transitions selon une horloge globale h. La procédure utilise deux vecteurs $V1$ et $V2$. $V1$ contient les valeurs courantes des variables V dans chaque étape de la procédure. Ces valeurs peuvent être en fonction des paramètres P. $V2$ est un vecteur d'entiers : $V2[q]$ stocke l'indice de la dernière transition où

[1] Pour plus de détails, voir la section 4.7

Procédure SymbolicTrace :

Entrée : Un chemin initial $\rho = t_1...t_n$, $t_i = (s_{i-1}, a, pred_i, ass_i, s_i)$, d'un ETIOA $M = (S, s_0, \Sigma, C, P, V, V_0, Pred, Ass, \rightarrow)$.

Sortie : Une trace symbolique $\rho' = t'_1...t'_n$, $t'_i = (s_{i-1}, a, pred'_i, \emptyset, s_i)$.

Variables Temporaires : Deux vecteurs de contexte $V1$ de taille m et $V2$ de taille k.

Début

/*Initialisation des vecteurs */

1. **Pour** $i := 1$ **à** m **Faire**
2. $V1[i] := v_{0i}$

3. **Pour** $i := 1$ **à** k **Faire**
4. $V2[i] := 0$

/*Mise à jour des vecteurs */

5. **Pour** $i := 1$ **à** n **Faire**
6. $pred'_i := UpdatePredicates(pred_i, V1, V2, i)$;
7. $UpdateContext(ass_i, V1, V2, i)$;

Fin

FIG. 63: Procédure SymbolicTrace.

l'horloge $c_q \in C$ a été réinitialisée. $SymbolicTrace$ se compose d'une phase d'initialisation des vecteurs (lignes 1-4) et une phase de mise à jour des vecteurs (lignes 5-7).

Dans la phase d'initialisation, la valeur courante de chaque variable v_i correspond à sa valeur initiale v_{0i} dans V_0 : $V1[i] := v_{0i}$. De plus, toutes les horloges sont initialisées dans l'état initial : $V2[i] := 0$.

Dans la phase de mise à jour, le prédicat $pred_i$ de la transition courante t_i (ligne 6) est utilisé pour calculer le nouveau prédicat $pred'_i$ de t'_i. $Pred_i$ est obtenu en remplaçant les variables et les horloges de $pred_i$ par leurs valeurs courantes dans $V1$ et $V2$. Ce remplacement est réalisé par l'appel à la procédure **UpdatePredicates**. L'affectation ass_i de t_i est alors utilisée pour calculer les nouvelles valeurs des variables de $V1$ ainsi que les nouvelles remises à jour des horloges dans $V2$, après le franchissement de t_i. Cette mise à jour est réalisée à travers l'appel à la procédure **UpdateContext** (ligne 7).

La procédure **UpdatePredicates** est illustrée dans la FIG.64. Rappelons qu'un prédicat s'écrit comme une conjonction d'une contrainte d'horloges $p_1(c_1, ..., c_k, P, v_1, ..., v_m)$ et une contrainte de variables $p_2(P, v_1, ..., v_m)$. Dans la contrainte d'horloges, toute horloge c_p réinitialisée pour la dernière fois dans la transition $t_{V2[p]}$ est remplacée par $h_i - h_{V2[p]}$, où i est l'indice de l'étape courante (ligne 1, partie $p_1(h_i - h_{V2[1]}, ..., h_i - h_{V2[k]}, P, ...)$). En effet, $h_i - h_{V2[p]}$ correspond au temps écoulé depuis la dernière ré-initialisation de c_p. Dans la

Procédure UpdatePredicates :
Entrée : Un prédicat $pred = p_1(c_1, ..., c_k, P, v_1, ..., v_m) \wedge p_2(P, v_1, ..., v_m)$, un indice i, $V1$ et $V2$ deux vecteurs.
Sortie : Un prédicat $predUpdated$.
Début

1. $predUpdate := p_1(h_i - h_{V2[1]}, ..., h_i - h_{V2[k]}, P, V1[1], ..., V[m])$
$\wedge p_2(P, V1[1], ..., V[m])$

Fin

FIG. 64: Procédure UpdatePredicates.

contrainte de variable, toute variable v_j est remplacée par sa valeur courante dans l'étape i. Cette valeur vaut $V1[j]$ (ligne 1, partie $p_1(..., P, V1[1], ..., V[m])$). Cette démarche est aussi appliquée à $p_2(P, v_1, ..., v_m)$.

Procédure UpdateContext :
Entrée : Une affectation ass, un indice i et deux vecteurs $V1$ et $V2$.
Sortie : Deux vecteurs $V1$ et $V2$.
Début

1. **Pour** $j := 1$ **à** m **Faire**
2. **Si** $v_j := f(v_1, .., v_m, P) \in ass$ **Alors** $V1[j] := f(V[1], .., V[m], P)$

3. **Pour** $j := 1$ **à** m **Faire**
4. **Si** $c_j := 0 \in ass$ **Alors** $V2[j] := i$

Fin

FIG. 65: Procédure UpdateContext.

La procédure **UpdateContext** est illustrée dans la FIG.65. Elle met à jour les valeurs courantes des variables dans $V1$ en tenant compte des nouvelle affectations (lignes 1-2) et les ré-initialisations des horloges en positionnant les champs de $V2$ à l'indice de l'étape courante (lignes 3-4).

Corollaire 15 *Supposons que $\rho' = t'_1 ... t'_n$, $t'_i = (s_{i-1}, a, pred'_i, \emptyset, s_i)$ pour $i \in [1, n]$, est la trace symbolique associée au chemin ρ. Alors ρ est exécutable si et seulement si le prédicat $pred' = \bigwedge_{i \in [1,n]} pred'_i$ est vrai pour certaines valeurs des paramètres P et des instants de tirs $(h_i)_{i \in [1,n]}$.* □

Chapitre 13

L'outil TGSE

Sommaire

13.1 Architecture du générateur de test de TGSE 202
13.2 Algorithme de génération gga 208
13.3 TGSE et le projet Calife 210
13.4 Étude de Cas : CSMA/CD 211

TGSE pour génération de test, simulation et émulation (Test génération, simulation and emulation) est un ensemble de logiciels regroupant différentes activités du test que nous avons développé au LaBRI. Il comporte un générateur de séquences de test, pour les systèmes temporisés et étendus, basé sur 1) le modèle CS, 2) le critère de couverture exprimé sous forme de coloriage et 3) la méthodologie *gga*. Il comporte aussi un simulateur, à travers la plate-forme Calife, permettant l'exécution graphique d'une séquence générée par TGSE. Finalement, l'émulateur temps réel de TGSE permet l'exécution réelle des différents systèmes. L'émulateur joue aussi le rôle d'un générateur de code.

Par la suite, nous allons décrire les principes et les algorithmes mis en oeuvre dans le générateur de test de TGSE.

13.1 Architecture du générateur de test de TGSE

L'architecture du générateur de test de TGSE est illustrée dans la Fig.66. Elle comporte plusieurs fonctionnalités réalisées par différents modules.

Module Compilation

L'entrée de TGSE est une description du système sous une syntaxe simple permettant de définir les différentes entités du système. Dans un souci de réutilisation des composantes,

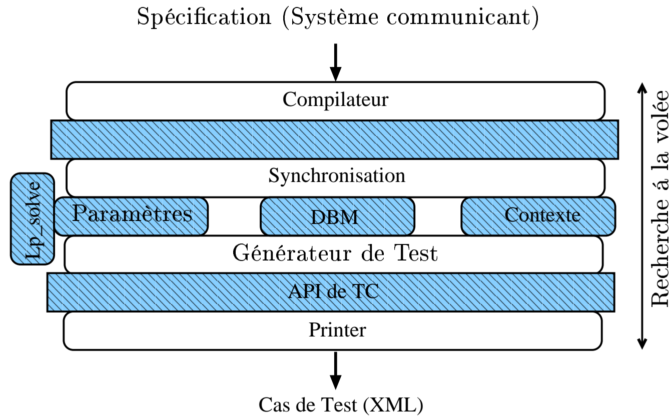


FIG. 66: Architecture logicielle

chaque entité est définie dans un fichier séparé. Un fichier système décrit les chemins d'accès aux entités ainsi que les vecteurs de synchronisation. Dans la version actuelle, TGSE ne supporte qu'une topologie statique. La syntaxe de description des entités est la suivante. Lorsqu'une entité définit un état à couvrir, la variable final_states est positionnée à l'indice de cet état. Ceci est équivalent à colorier cet état par la couleur *rouge*. Une transition est décrite par six champs :

1. (numéro_état,étiquette),
2. événement (*nop* pour une action interne),
3. prédicat_horloges (# pour un prédicat vrai),
4. prédicat_variables (# pour un prédicat vrai),
5. mise_à_jour_horloges (de la forme $x := 0$ et # pour l'absence de reset)
6. mise_à_jour_variables (de la forme $v := v + p$ et # pour l'absence d'affectation).

Finalement, le module de compilation traduit le système sous test en structures de données *C* utilisées par le module de synchronisation. Ci-dessous, des fichiers d'entrées du protocole CSMA/CD composé d'un bus, deux émetteurs et un objectif de test :

```
*********************************************
P_AUTO Sender
{
nb_states = 4
initial_state = 10
clocks = x


(10,Wait), (10,Wait), ?CD, #, #, x:=0, #
(10,Wait), (11,Transmit), !begin, #, #, x:=0, #
```

```
(10,Wait), (12,Retry), ?busy, #, #, x:=0, #
(10,Wait), (12,Retry), ?CD, #, #, x:=0, #
(11,Transmit), (12,Retry), ?CD, x[0,Sig[, #, x:=0,#
(11,Transmit), (11,Transmit), ?busy, #, #, #, #
(11,Transmit), (13,Finish), !end, #, x[lambda,lambda], #, #
(12,Retry), (11,Transmit), !begin, x[0,2*4], #, x:=0,#
(12,Retry), (12,Retry), ?CD, x[0,2*Sig], #, x:=0,#
(12,Retry), (12,Retry), ?busy, x[0,2*Sig], #, x:=0,#
(13,Finish), (10,Wait), nop, #, #, x:=0, #
}
*****************Sender.aut******************


**********************************************
TESTER TP
{
nb_states = 2
initial_state = 10
final_states = 11


(10,toto), (11,titi), ?busy, #, #, #, #
}
*********************Tp.aut******************


**********************************************
P_AUTO Bus
{
nb_states = 3
initial_state = 10
clocks = y


(10,Idle), (11,Active), ?begin, #, #,y:=0, #
(11,Active), (10,Idle), ?end, #, #, y:=0, #
(11,Active), (12,Collision), ?begin, y[0,Sig[, #, y:=0,#
(11,Active), (11,Idle), !busy, y[Sig,+inf], #, #,#
(12,Collision), (10,Idle), !CD, #, #, #, #
}
*****************Bus.aut*********************
```

```
*********************************************
SYSTEM CSMA_CD_2
{
nb_automatons:3
nb_vectors: 8
clocks : S
parameters = Sig Lambda
automaton_files:[../Example/Sender.aut ../Example/Bus.aut
                 ../Example/Sender.aut ]
tester_file: Tp.aut


VECTORS

<?busy ,!busy ,?busy ,?busy>
<* ,?end ,!end ,*>
<* ,?begin ,!begin ,!begin>
<!end ,?end ,* ,*>
<!begin ,?begin ,* ,!begin>
<* ,?begin ,!begin ,*>
<!begin ,?begin ,* ,*>
<?CD ,!CD ,?CD ,*>
}
*****************CSMA-CD.sys*****************
```

Module Synchronisation

Ce module implémente les fonctionnalités relatives au calcul de la sémantique du système communicant. Ce calcul est réalisé à la volée. Il implémente l'API $SynchronizationOnVectors()$ qui permet de choisir une synchronisation possible dans l'état courant du système, en se basant sur les vecteurs de synchronisation. Cette API retourne une structure de donnée "Element" qui contient les états d'arrivée, les transitions choisies, ainsi que le contexte relatif à cette synchronisation. Le choix des transitions, des vecteurs de synchronisation ainsi que des automates qui les réalisent est paramétré par des variables pour chaque donnée. Les valeurs possibles de ces variables sont RANDOM, pour un choix aléatoire et FIFO pour un respect de l'ordre d'apparition dans la définition du système. La construction de la sémantique est paramétrée par le nombre maximal d'apparitions de la même transition dans un chemin. Ainsi, à chaque transition, on associe une variable *Lock* qui contient le nombre maximal d'apparitions de cette dernière dans un cas de test généré. Ce module implémente aussi les APIs *getInitialStates* qui retourne l'état initial de $\zeta(S)$ et *getSuccessors* qui retourne les successeurs d'un état courant de $\zeta(S)$.

Sous-module Contexte. Il implémente des fonctionnalités relatives à la mise à jour des variables, des prédicats, de la trace symbolique (les APIs *UpdateContext*, *UpdatePredicates*, *SymbolicTrace*,...).

Sous-module DBM. C'est une librairie qui implémente les opérations sur polyèdres et le calcul des diagnostics de bornes. Ses principales APIs utilisables par le module de génération sont *post*(), *pred*(), *TimedDiagnostics*().

Sous-module Paramètres. Pour une trace symbolique, ce sous-module implémente les APIs *checkSymbolicTrace* et *getParameterValues*. *checkSymbolicTrace* construit le système de contraintes associé à la trace symbolique, interagit avec l'outil de programmation linéaire *lp_solve* et détermine si le système admet une solution. *getParameterValues* instancie les valeurs des paramètres dans le cas où le système de contraintes admet une solution.

Module Générateur de Test

C'est le coeur de l'outil. Il implémente un algorithme *gga* de recherche en profondeur à la volée de l'automate sémantique du système. L'algorithme *gga* calcule d'une manière aléatoire et uniforme (qui dépend des paramètres d'entrée) un chemin de l'état initial qui se termine dans un état à couvrir. Le chemin ainsi généré est décoré par les différents verdicts. La section suivante présente en détail *gga*.

Sous-module API pour TC. Ce sous-module implémente spécialement l'API *TC*() qui décore un chemin obtenu par *gga* par les différents verdicts.

Module Printer

La génération de cas de test se termine par l'API *writeTrace*() du module Printer. *writeTrace*() transforme les structures de données du cas de test généré en format XML. Un exemple de la sortie de TGSE est ci-dessus.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Trace SYSTEM "trace.dtd">
<Trace Label="CSMA_CD_2">
<StateSync>

<State Index="1" Component="Sender">
<Loc Label="Wait"/>
<Bounds/>
</State>
```

```
<State Index="2" Component="Bus">
<Loc Label="Idle"/>
<Bounds/>
</State>
<State Index="3" Component="Sender">
<Loc Label="Wait"/>
<Bounds/>
</State>
<State Index="4" Component="TP">
<Loc Label="init"/>
<Bounds/>
</State>
<Diag/>
<Glob/>

</StateSync>
<Action Label="Action">
<Comp Label="epsilon" Index="0"/>
<Comp Label="begin" Type="Wait" Index="1"/>
<Comp Label="begin" Type="Send" Index="2"/>
<Comp Label="epsilon" Index="3"/>
</Action>
<StateSync>

<State Index="1" Component="Sender">
<Loc Label="Wait"/>
<Bounds/>
</State>
<State Index="2" Component="Bus">
<Loc Label="Active"/>
<Bounds/>
</State>
<State Index="3" Component="Sender">
<Loc Label="Transmit"/>
<Bounds/>
</State>
<State Index="4" Component="TP">
<Loc Label="init"/>
<Bounds/>
</State>
<Diag/>
<Glob/>
```

```
</StateSync>
<Action Label="Action">
<Comp Label="busy" Type="Wait" Index="0"/>
<Comp Label="busy" Type="Send" Index="1"/>
<Comp Label="busy" Type="Wait" Index="2"/>
<Comp Label="busy" Type="Wait" Index="3"/>
</Action>
<StateSync>

<State Index="1" Component="Sender">
<Loc Label="Transmit"/>
<Bounds/>
</State>
<State Index="2" Component="Bus">
<Loc Label="Collision"/>
<Bounds/>
</State>
<State Index="3" Component="Sender">
<Loc Label="Transmit"/>
<Bounds/>
</State>
<State Index="4" Component="TP">
<Loc Label="Final"/>
<Bounds/>
</State>
<Diag/>
<Glob/>

</StateSync>
</Trace>
```


Signalons finalement que dans le cas où aucun état à couvrir n'est accessible dans le parcours courant, l'algorithme *gga* est relancé automatiquement pour une nouvelle tentative (le lancement est paramétrable). De plus, il est possible de générer un cas de test minimal en nombre de transitions pour un nombre de tentatives donné.

13.2 Algorithme de génération gga

Dans sa version actuelle, TGSE ne supporte pas tout le cadre formel présenté dans ce document et qu'il est en cours d'extension. Nous ne présentons ici que la partie supportée par TGSE.

```
Fonction gga() :
  Begin
      States := getInitialStates(), SymbolicState = ∅, Element := NULL, Path := ∅,
      SymbolicTrace := ∅, SymbolicPath := ∅

      Do

         Element := SynchronizationOnVectors(States) ;
         push(Element,Path)

         If (Element ≠ NULL) Then

            If type = Extended Then
               push(SymbolicTrace, SymbolicTrace(Element))
               If checkSymbolicTrace(SymbolicTrace) = false Then
                  pop(SymbolicTrace)
                  pop(Path)

            Else
               SymbolicState = post(SymbolicTrace,Element)
               If SymbolicState ≠ ∅ Then
                  push(SymbolicTrace,SymbolicState)
               Else
                  pop(Path)

            States := getSuccessors(Element) ;

         Else
            pop(Path)
            States := getSuccessors(top(Path))

         If AcceptStates(States) = true Then
            exit(EXIT_SUCCESS)

      While Path ≠ ∅

      If Path ≠ ∅
         If type = Extended Then
            getParameterValues(SymbolicTrace)
         Else
            TimedDiagnostics(SymbolicPath)
         TC(Path) ;

  End
```

FIG. 67: Algorithme de génération *gga*.

Hypothèses. L'algorithme *gga* implémenté considère un système communicant sous test (S, col) tel que :

1. S est une topologie statique,
2. *col* est un coloriage distribué à trois couleurs : rouge, noir et bleu tel que $friend(rouge) = \{noir\}$. Le rouge définit les états à couvrir (*Accept*) et le bleu les états à éviter (*Reject*).

Ainsi, un état d'un automate est modélisé par un couple $(s, couleur)$.

Structures de données. Nous utilisons les structures de données suivantes :
- *States* : un (n+1)-uplet $(s_{tr}, s_1, ..., s_n)$.
- *Context* : une structure contenant deux champs *Variable* et *Reset* :

  1. *Variable* : un vecteur contenant les valeurs des variables discrètes.
  2. *Reset* : un vecteur d'entiers contenant, à chaque étape i, les indices de transitions où les horloges ont été remises à zéro pour la dernière fois.

- *Transitions* : un tableau de $n + 1$ champs. $Transition[i]$ correspond à un pointeur sur la transition courante de l'automate M_i de $Comp(S)$.
- *Element* : une structure de donnée contenant trois champs : un champ de type *States*, un champ de type *Context* et un champ de type *Transitions*.
- *Path* : une pile d'éléments. Elle est gérée par les opérations "push", "top" et "pop".

Description de gga. L'algorithme de génération *gga*, appliqué à un CS S, réalise une recherche en profondeur de $\zeta(S)$. Durant la traversée de $\zeta(S)$, *gga* calcule un chemin symbolique (SymbolicPath) dans le cas d'une spécification temporisée, et une trace symbolique (SymbolicTrace) dans le cas d'une spécification étendue (type =Extended). Lorsqu'un état à couvrir est rencontré, la traversée s'achève par l'appel à *SymbolicPath* ou à *SymbolicTrace*. Dans ce cas, l'appel à l'API $TC()$ permet de décorer le chemin ainsi choisi par les différents verdicts. La FIG.67 illustre le pseudo-code en langage C de *gga*.

13.3 TGSE et le projet Calife

Le projet RNRT Calife et son successeur Averroès sont des projets académiques et industriels regroupant un ensemble de partenaires (France Telecom R&D, CRIL Technologie, LaBRI, LSV, Loria, LRI). Le but de Calife est de définir une plate-forme générique (Open Source) permettant d'interfacer des outils de vérification et de génération de test. Dans le cadre de ce projet, la participation du LaBRI consiste, en outre, à intégrer TGSE dans cette plate-forme.

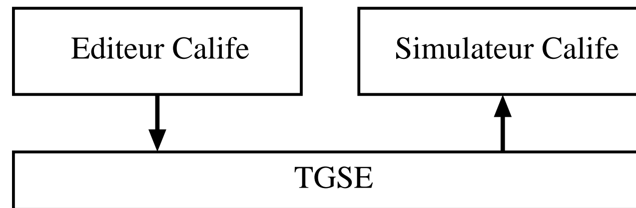


Fig. 68: Interfaces de TGSE avec Calife.

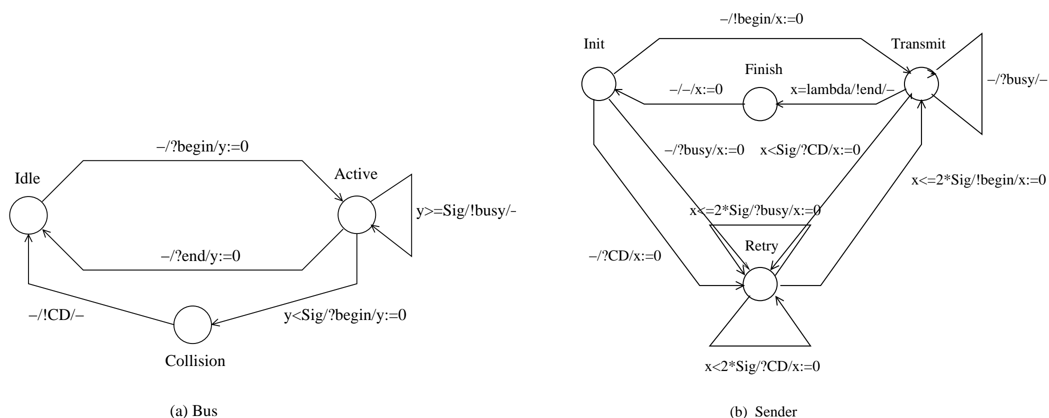
Plate-forme Calife. La plat-forme Calife comporte un éditeur et un simulateur. L'éditeur fournit une interface agréable pour manipuler les différents types d'automates (temporisés, hybrides, étendus,...). Son simulateur permet de simuler graphiquement l'exécution d'un ensemble d'automates. Cette plate-forme supporte deux types d'utilisation : un utilisateur normal qui modélise sa spécification et utilise les outils que la plate-forme fournit pour vérifier ou générer des cas de test, et le mode expert qui permet d'intégrer un outil à la plate-forme.

TGSE sous Calife. Nous avons intégré une partie de TGSE dans calife en définissant les différentes transformations requises. TGSE peut être utilisé en mode graphique à travers Calife. Dans ce cas, la saisie des spécifications se fait à travers l'éditeur Calife qui permet, signalons le, la génération automatique des vecteurs de synchronisation. Il offre le choix d'une synchronisation par rendez-vous, broadcast, la synchronisation binaire d'Uppaal, ou par labels identiques. Ceci pour une topologie statique. L'appel à TGSE se fait à travers l'interface graphique. Calife génère dans ce cas les fichiers d'entrée de TGSE. TGSE produit un cas de test, en format XML selon une DTD Calife, qui pourra être simulé sous Calife. Le schéma de la Fig.68 représente les communications entre Calife et TGSE.

13.4 Étude de Cas : CSMA/CD

Nous avons expérimenté TGSE avec le protocole CSMA/CD à un bus, plusieurs émetteurs et un objectif de test consistant à détecter une collision $!CD$ à un instant inférieur à 5 unités de temps. Ceci sous un portable DELL INSPIRON 5100, de processeur Intel P4 (2,4GHZ) et à 256Mo de RAM sous Mandrake 10.0 (TGSE a été aussi testé sous WindowsNT et RedHat). Dans le tableau de la Fig.70, la colonne *Lock* représente le nombre de fois qu'une transition peut figurer dans un cas de test, Taille du TC représente la taille moyenne d'un cas de test généré, Nb Sender correspond au nombre des émetteurs considérés et Temps CPU est le temps moyen de génération.

On peut remarquer que la génération avec $Lock = 1$ prend plus de temps. Ceci s'explique par le fait que l'algorithme *gga* atteint des états dont les transitions sont saturées et ainsi il est obligé de dépiler plusieurs fois.



(a) Bus (b) Sender

FIG. 69: Modélisation CSMA/CD.

Finalement, bien que le protocole CSMA/CD soit de taille réduite, l'utilisation de plusieurs émetteurs augmente sa complexité. Les résultats obtenus sont très encourageants et des améliorations sont en cours.

Lock	Nb Sender	Taille du TC	Temps CPUs (s)
1	2	3	0.077
1	5	3	0.303
1	10	3	0.621
1	20	3	0.914
10^3	2	18	0.027
10^3	5	55	0.098
10^3	10	79	0.234
10^3	20	130	0.793

FIG. 70: Expérimentation.

Sixième partie

Conclusions et Perspectives

Chapitre 14

Conclusion

Au cours de ce document, trois thèmes ont été abordés : la modélisation des systèmes communicants, l'analyse des systèmes temporisés et le test des systèmes communicants.

Nous avons introduit le modèle CS comme une description des systèmes communicants. L'approche du modèle CS consiste à séparer la modélisation des entités, la modélisation du flux de contrôle partagé et la modélisation des ressources communes. Les entités sont modélisées par des automates temporisés étendus, le flux de contrôle partagé est modélisé par une topologie de communication (automate) et les ressources communes par des variables et des paramètres. L'utilisation du modèle CS a été illustrée par trois exemples : l'exemple des p producteurs et q consommateurs, l'exclusion mutuelle et la norme CSMA/CD.

Pour l'analyse des systèmes temporisés, nous avons abordé le problème d'extraction des diagnostics temporisés et d'inclusion des traces temporisées entre deux chemins. L'extraction des diagnostics trouve une application intéressante lors de la vérification d'une propriété du système du fait qu'elle permet une bonne compréhension des raisons derrière la satisfaction ou la non satisfaction de la propriété. Le problème d'inclusion des traces s'avère important dans le domaine du test pour montrer l'inclusion des traces de l'implantation dans celles de la spécification. Pour ces deux problèmes, la solution proposée repose sur l'identification des diagnostics des bornes (maximales et minimales) qui correspondent à des exécutions limites du système. Nous avons montré comment calculer ces diagnostics, soit par la résolution de contraintes à travers le polyèdre de contraintes, soit par une analyse symbolique en avant et en arrière.

Pour le test que nous qualifions de *test classique*, nous nous sommes intéressés au test d'interopérabilité des systèmes sans contraintes temporelles et au test de conformité des systèmes temporisés. Pour les deux tests, nous avons introduit un cadre formel comportant des définitions formelles des tests, ainsi que des méthodes de génération. Dans la définition des méthodes de génération, nous nous sommes focalisés sur l'applicabilité des méthodes et de leurs résultats. Ainsi, la méthode de génération des tests temporisés, basée sur les diagnostics de bornes, que nous avons proposée ne souffre pas de l'explosion du nombre de cas de test générés du fait que les cas de test considérés correspondent aux diagnostics

temporisés de bornes. Ces derniers procurent une représentation finie de l'espace des traces relatives à un chemin.

Notre proposition de l'approche ouverte du test se veut un aboutissement de l'étude menée sur le test classique d'une part et la modélisation des systèmes communicants d'autre part. En effet, nous avons montré qu'il est possible d'appréhender les différents types, approches et architectures de test au sein d'un même modèle et avec la même méthodologie. Nous avons proposé une modélisation uniforme de ces différents aspects de test. Cette modélisation était basée d'une part, sur le modèle des systèmes communicants (CS) et d'autre part sur la méthodologie des algorithmes génériques de génération (GGA) qui considère les critères de couverture structurels sous la forme de coloriage ou de coloriage ordonné de graphes. Le concept du test ouvert est relativement simple : des implantations satisfont des spécifications selon une propriété P. P peut être l'appartenance d'une trace d'implantation à la spécification (test passif), la conformité des implantations par rapport aux spécifications, l'interopérabilité des implantations,...

Concrètement, les résultats de cette thèse ont été implémentés dans notre outil TGSE (Génération de Test, Simulation et Émulation). Dans sa version actuelle, TGSE supporte la génération automatique des cas de test (test actif) et la vérification de l'appartenance d'une trace à une spécification donnée. TGSE a été incorporé partiellement dans la plate-forme Calife réalisée dans le cadre des projets RNRT Calife et Averroès.

Finalement, le protocole CSMA/CD a été pris comme une étude de cas dans ce document. D'une part, la modélisation de CSMA/CD fait intervenir des paramètres et des horloges, aspects traités dans cette thèse et d'autre part, il présente un bon exemple pour tester la capacité et les performances de l'outil TGSE.

Perspectives

Au cours de ce document nous avons traité certains aspects relatifs au test. Certains d'autres n'ont pas été abordés. Comme perspectives de ce travail, nous citons les cinq points suivants :

1. **Normalisation du test.** À l'exception du test de conformité, il n'existe aucune norme définie pour les autres types de test. À travers une étude plus approfondie et détaillée du concept du test ouvert introduit dans ce document, nous pensons qu'il est possible de définir une norme unique pour le test.

2. **Relâchement des contraintes sur le modèle temporel.** Le cadre formel présenté dans ce document repose sur le modèle des automates temporisés événementiels déterministes et observables. Or, les domaines d'application de ce modèle restent relativement limités. Dans le cas des automates temporisés non-déterministes et non-observables, des techniques de déterminisation à la volée peuvent être appliquées. Cependant, ils souffrent de l'explosion combinatoire du nombre de cas de test générés.

3. **Étude de la couverture.** Due à la nature non-exhaustive du test, la mesure de la couverture d'une suite de test permet d'avoir un certain degré de confiance dans l'activité du testeur. La plupart des travaux relatifs à la mesure de couverture datent des années 90s et ils n'ont pas eu un réel succès dans le domaine industriel. Nous pensons que la formalisation des critères de couverture et des hypothèses du test est un sujet à approfondir qui permettrait de définir des métriques significatives et pratiques. Une première étape est déjà réalisée par la définition du coloriage. Nous envisageons de définir une métrique qui mesure le nombre d'éléments couverts par une suite de test pour une couleur donnée, ceci sous l'hypothèse d'uniformité.
4. **Test de code.** La programmation orientée objet et les approches basées sur les composantes sont largement utilisées dans le développement des logiciels. Par conséquence, le test de ces logiciels nécessite des techniques basées sur des modèles à états. Nous envisageons d'étudier l'applicabilité de certaines méthodes de génération de test de protocoles dans le domaine logiciel.
5. **Extension de l'outil.** Seule une partie des résultats présentés dans ce document ont été implémentés dans l'outil TGSE. Nous envisageons d'étendre TGSE, de l'interfacer avec d'autres outils, ainsi que de réaliser l'étude de cas du protocole CSMA/CA prévue dans le cadre du projet Calife.

Bibliographie

[1] Batiste Alcalde, Ana cavalli, Dongluo Chen, Davy Khuu and David Lee. Network protocol system passive testing for fault management : a backward checking approach. *Proceeding of FORTE/PSTV'2004*, Madrid, Spain. LNCS 3235, September 2004.

[2] R. Alur and D. Dill. Automata for modeling real-time systems. *In Proc. of the 17yh ICALP'90*, vol 443 of LNCS, pp. 322-334, Springer-Verlag, 1990.

[3] R. Alur, T.A. Henzinger and M. Vardi. Parametric real-time reasoning. *In Proc. 25th ACM Symposium on Theory of Computing*, pp. 592-601, ACM, 1993.

[4] R. Alur and D. Dill. A theory of timed automata, *Theoretical Computer Science*, 126 :183-235, 1994.

[5] R. Alur, C Courcoubetis, N. Halbwachs, T.A. Henzunger, P.H. Ho, X. Nicollin, A. Olivero, J. Sifakis and S. Yovine. The algorithmic analysis of hybrid systems.*Theoretical Computer Science*, 138 :3-34, 1995.

[6] R. Alur, R. Kurshan and M. Viswanathan. Membership problems for timed and hybrid automata. 19th IEEE Real-Time Systems Symposium, 1998.

[7] N. Arakawa and T. Soneoka. A test case generation method for concurrent programs. *Protocol Test Systems, IV, Elsevier Science Publishers*, 1992.

[8] N. Arakawa, M. Phalippou, N. Risser and T. Soneoka. Combination of conformance and interoperability testing. *Formal Description Techniques*, V(C-10) M. Diaz and R. Gruz (Eds.) Elsevier Science Publishers, 1993.

[9] André Arnold et M. Nivat. Comportements de processus. *In Colloque AFCET "Les mathématiques de l'Informatique"*, pp. 35-68, 1982.

[10] André Arnold. Systèmes de transitions finis et sémantique des processus communicants. *Masson* 1992.

[11] André Arnold. Finite transition systems. Semantics of communicating systems. *Prentice-Hall*, 1994.

[12] S. Barbin, L. Tanguy and C. Viho. Towards a formal framework for interoperability testing. *Proceedings FORTE01*, Korea, August 28-31, 2001.

[13] Tomás Barros, Rabéa Boulifa and Eric Madelaine. Parameterized models for distributed java objects. *FORTE*, Madrid, Spain. LNCS 3235, September 2004.

[14] B. Beizer. Software testing techniques. *2nd ed. International Thomoson Computer Press.* The United States of America, 2001.

[15] A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, L. Heerink. Formal test automation, a simple experiment. Proceedings of 12th Int. Workshop on Testing of Communicating Systems, pp. 179-196, 1999.

[16] M. Benattou, L. Cacciari, R. Pasini and O.Rafiq. Principe and tools for testing open distributed systems. *Pro. of IWTCS'99*, Kluwer, pp. 77-92, 1999.

[17] B. Bérard, P. Gastin and A. Petit. On the power of non observable actions in timed automata. *Proceedings of the 13th Annual Symposium on Theoretical Aspects of Computer Science (STACS'96)* no. 1046 in Lecture Notes in Computer Science, pp. 257-268, Springer Verlag, 1996.

[18] B. Bérard, P. Gastin and A. Petit. Removing ϵ-transitions in timed automata. *in Proceedings of the 14th Annual Symposium on Theoretical Aspects of Computer Science (STACS'97)*, Springer Verlag, 1997.

[19] B. Bérard and C. Dufourd. Timed automata and additive clock constraints. *Information Processing Letters*, vol. 75(1-2) :1-7, 2000.

[20] B. Bérard, P. Castéran, E. Fleury, L. Fribourg, J.-F. Monin, C. Paulin, A. Petit, and D. Rouillard. Automates temporisés calife. *Fourniture F1.1*, Calife, 2000.

[21] J. A. Bergstra and J. W. Klop. Process Algebra for Synchronous Communication. *Information and Computation*, 60 :109-137, 1984.

[22] I. Berrada, R. Castanet and P. Félix. Techniques de génération de séquences de test temporisées : Manuel, intégration et études de cas. Projet RNRT Calife, Fourniture 3.6&7&8, December, 2002.

[23] I. Berrada, R. Castanet and P. Félix. Techniques de test d'interopérabilité pour les systèmes temporisés. Projet RNTL Averroès, Fourniture 3.2.1, November 2003.

[24] I. Berrada, R. Castanet and P. Félix. A formal approach for real time test generation. Workshop on testing real-time and embedded systems (satellite event of FM), Pisa (Italy), September 2003.

[25] I. Berrada, R. Castanet and P. Félix. From the feasibility analysis to real-time test generation. *Studia Informatica Universalis*, Vol3, Number 2, pp 141-168, 2004.

[26] I. Berrada, R. Castanet, P. Félix. Generating interoperability test cases from conformance test case generation tools. FORTE 2004, Madrid, Spain, September 2004 (Short Paper).

[27] I. Berrada et P. Félix. TGSE : Un outil générique pour le test. CFIP'05, Bordeaux (France), Ed Hermès, Mars 2005.

[28] I. Berrada, R. Castanet and P. Félix. Testing communicating systems : a model, a methodology and a tool. TESTCOM 2005, LNCS, Montreal (Canada), June 2005.

[29] I. Berrada, R. Castanet and P. Félix. Outil TGSE : Modélisation et génération de test d'interopérabilité pour les systèmes temporisés. Projet RNTL Averroès, Fourniture 3.3.1, June 2005.

[30] I. Berrada, R. Castanet and P. Félix. Interoperability testing for communicating systems. *Tarot Summer School 2005*, Paris, France, 27 June - 1 July 1, 2005.

[31] Machiel van der Bijl, Arend Rensink, Jan Tretmans, Compositional testing with ioco. *Proceedings of the 3rd International Workshop on Formal Approaches to Testing of Software, FATES2003*, Montreal, Canada, 2003.

[32] G. V. Bochmann, A. Das, R. Dssouli, M. Duluc, A. Ghedami and G. Luo. Fault models in testing. *In Kroon J. Heijink R.J and Brinksma E. editors. Protocol Test System IV, volume C-3 of IFIP Transactions*, pp. 17-30. North-Holand, 1992.

[33] S. Bornot, J. Sifakis and S. Tripakis. Modeling urgency in timed systems. *In Compositionality*, volume 1936 of LNCS. Springer, 1998.

[34] Patricia Bouyer. Modèles et algorithmes pour la vérification des systèmes temporisés. Thèse de doctorat, Laboratoire Spécification et Vérification, ENS Cachan, April 2002.

[35] P. Bouyer. Untameable timed automata!. *In Proc. 20th Ann. Symp. STACS'2003*, vol 2607 of LNCS, 620-631, Springer. Feb 2004, Berlin, Germany, 2003.

[36] Marc Boyer. Contribution à la modélisation des systèmes à temps contraint et application multimédia. *PhD thesis*, Université de Toulouse III, July, 2001.

[37] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm and Laurent Mounier. IF : A validation environment for timed asynchronous systems. *In Computer Aided Verification*, pp. 543-547, 2000.

[38] Laura Brandán and Ed Brinksma. A test generation framework for quiescent real-time systems. *Proceedings of the 4rd International Workshop on Formal Approaches to Testing of Software, FATES2004*, Linz, Austria September 21, 2004.

[39] E. Brinksma. On the design of extended LOTOS - A specification language for open distributed system. *PhD thesis, Department of Computer Science, University of Twente*, 1988.

[40] Ed. Brinksma. A theory for the derivation of test. S. Aggarwal and Sabnani, eds. *In Protocol Specification, Testing and Verification VIII*, pp. 63-74, North-Holland, 1988.

[41] Ed Brinksma and Jan Tretmans. Testing transition systems : an annotated Bibliography. *In the proceeding of Modelling and Verification of Parallel Processes (MOVEP2k)*, vol 2067 of LNCS, pp. 187-195. Springer-Verlag, 2001.

[42] W. Buehler. Introduction to ATM forum test specifications, Version 1.0. *ATM Forum Technical Committee* , Testing Subworking Group, af-test-0022.000.

[43] L. Cacciari and O. Rafiq. Controllability and Observability in distributed testing. *Information and Software Technology*, 41 :767-780, 1999.

[44] L. Cacciari and O. Rafiq. Contraintes temporelles dans le test réparti d'applications non temps-réel. *In CFIP'2000, Ingenierie des protocoles, Hermes*, October 2000.

[45] Rachel Cardell-Oliver. Conformance testing of real-time systems with timed automata specifications. *Formal Aspects of Computing*, 12(5) :350-371, 2000.

[46] Ana Cavalli, David Lee, Christian Rinderknecht and Fatiha Zaïdi. Hit-or-Jump : an algorithm for embedded testing with applications to IN services. *Proceeding of FORTE/PSTV'99*, Beijing, China, October 1999.

[47] T.S. Chow. Testing software design by finite state machine. *IEEE Transactions on Software Engineering*, SE-4(3), May, 1978.

[48] C. Choffrut and M. Goldwurm. Timed automata with periodic clock constraints. *Journal of Automata, Languages and Combinatorics*, vol. 5(4) :371-404, 2000.

[49] Duncan Clarke and Insup Lee. Automatic test generation for the analysis of a real-time system : case study. In *3rd IEEE Real-Time Technology and Applications Symposium*, 1997.

[50] E. Clarke, O. Grumberg and D. Peled. Model-Checking. *The MIT Press, Cambridge*, Massachusetts, 1999.

[51] J. Crow and B. Di Vito. Formalizing space shuttle software requirements : Four case studies. *ACM Transactions on Software Engineering and Methodology*, 7(3) : 296-332, 1998.

[52] D. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Proceeding of first CAV*, number 407 in LNCS, Springer-Verlag, France, 1989.

[53] S. Eastbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo and D. Hamilton. Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering*, 24(1) :4-14, 1997.

[54] A. En-Nouaary, R. Dssouli, F. Khenedek and A. Elqortobi. Timed test cases generation based on state characterization technique. *In 19th IEEE Real Time Systems Symposium (RTSS'98)*, Madrid, Spain, 1998.

[55] ETSI ETR (ETSI Technical Report) 130. Methods for Testing and Specification (MTS) ; Interoperability and Conformance Testing / A Classification Scheme. April 1994.

[56] J. Fernandez, H. Gravel, L. Mounier, A. Rasse, C. Rodriguez and J. Sifakis. A tool Box for verification of lotos programs. *In 14th ICSE*, Melbourne, Australia, May 1992.

[57] J. Fernandez, C. Jard, T. Jéron, C Viho, An experiment in automatic generation of test suites for protocols with verification technology. *Sci. Comput. Program.* 29(1-2) : pp. 123-146, 1997.

[58] Robert W. Floyd. Algorithm 97 (shortest path). *Communications of the ACM*,18(3) :165-172, 1964.

[59] J. Gadre, C. Rohrer, C. Summers and S. Stmington. A COS study of OSI interoperability. *Computer Standards and Interfaces*, 9 :217-237, 1990.

[60] L. Gallon and G. Juanole. Critical time distributed systems : Qualitative and quantitative analysis based on stochastic Petri nets. *FORTE'95*, Montreal, Canada, October 1995.

[61] Hubert Garavel. Défense et illustration des algèbres de processus. *École d'été temps réel*, September, 2003.

[62] A. Gill. Introduction to the theory of finite-state machines. *Mc Graw-Hill*, New York - USA, 1962.

[63] Jens Chr. Godskesen, Brian Nielsen and Arne Skou. Connectivity testing through model-checking. *In Proc of FORTE'04*, pp. 167-183, Madrid, Spain, Sept 2004,

[64] G. Gonenc. A method for the design of fault detection experiment. *IEEE transactions on Computers*, C-19 : 551-558, 1970.

[65] N. Griffeth, R. Hao, D. Lee, R.K. Sinha, Integrated system interoperability testing with applications to VoIP. *Proceedings of FORTE XIII/PSTV XX*, pp. 69-84, October 10-13, 2000.

[66] N. Halbwachs. Synchronous programming of reactive systems. *Kluwer Academic Publishers*, 1993.

[67] M. Hennessy. Algebraic theory of processes. *MIT Press*, Cambridge, MA, 1988.

[68] T. Henzinger, Z. Manna and A. Pnueli. What good are digital clocks ?. *ICALP'92*, LNCS 623, 1992.

[69] Thomas A. Henziger, Oei-Hsin Ho and Howard Wong-Toi. A user guide to hytech. *In Tools and Algorithms for Construction and Analysis of Systems.* pp. 41-71, 1995.

[70] Anders Hessel, Kim G. Larsen, Brian Nielson, Paul Pettersson and Arne Skou. Time-optimal real-time test case generation using Uppaal. *In FATES2003*, Montreal, Quebec, Canada, October, LNCS 2931, pp. 118-135, Springer.

[71] T. Higashino, A. Nakata, K. Taniguchi and A. Cavalli. Generating test cases for a timed i/o automaton model. *TESTCOM99*, Budapest, Hungary, September 1999.

[72] C. A. R. Hoare. Communicating Sequential Processes. *Premtice-Hall*, 1985.

[73] D. Hogrefe. Conformance testing based on formal methods. *In Proc of Formal Description Techniques*, ed. J. Quemada, A. Fernandez, 1990.

[74] G. Huet, G. Kahn and C. Paulin-Mohring. The Coq proof Assistant - A tutorial, version 6.1. *Rapport technique 204*, INRIA, 1997.

[75] Melania Ionescu. Une strategie de test de telecommunications. *Ph.D Thesis*, University Every Val d'Essonne, 21 June 2001.

[76] ISO/IEC/9646. OSI Conformance Testing Methodology and Framework - Parts 1-7, 1994.

[77] ISO/TC97/SC21. OSI Conformance Testing Methodology and Framework - Parts 1-5, ISO, 1991.

[78] ISO/IEC JTC1 DTR-10000. Information Technology - Framework and Classification of International Standard Protocol. 1994.

[79] ITU-T X.290 Series, Conformance Testing Methodology and Framework, 1994.

[80] ISO/IEC. ESTELLE : A formal description technique based on an extended state transition model. *International Standard no 9074, International Organization for Standardization , Information Processing Systems ,Open Systems Interconnection*, Genève, September 1988.

[81] ITU-T SG 10 :Q.8 ISO :IEC JTCI/SC21 WG7. Information Retrieval, Transfer and Management of OSI ; Framework : Formal Methods in Conformance Testing. Committee Draft CD 13245-1, ITU-T proposed recommendation Z 500. ISO-ITU-T, 1996.

[82] ITU-T. Specification and Description Language (SDL). *ITU-T Recommendation no Z.100, International Telecommunication Union*, November 1999, Genève.

[83] Claude Jard, Thierry Jéron, Lénaïck Tanguy and César Viho. Remote testing can be as powerful as local testing. *Proceeding of FORTE/PSTV'99*, Beijing, China. October 1999.

[84] Thierry Jéron and P. Morel. Test generation derived from model-checking. *CAV'99*, Trento, Italy, volume 633 of LNCS, pp. 108-122. Springer-Verlag, July 1999.

[85] S. Kang and M. Kim. Test sequence generation for adaptive interoperability testing. *In Proceedings of Protocol Test Systems VIII*, pp. 187 - 200, 1995.

[86] S. Kang, M. Kim, Interoperability test suite derivation for symmetric communication protocols. *Proceedings of FORTE/PSTV'97*, pp. 57-72, November 18-21, 1997.

[87] S. Kang, J. Shin and M. Kim. Interoperability test suite derivation for communication protocols. *Computer Network* pp 347 - 364, vol 32, 2000.

[88] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala and Frits Vaandrager. Timed I/O automata : a mathematical framework for modeling and analyzing real-time systems. *RTSS 2003 : The 24th IEEE International Real-Time Systems Symposium*, Cancun, Mexico, pp. 166-177, December 2003.

[89] A. Koumsi, M. Akalay, R. Dssouli, A. En-Nouaary, L. Granger. An approach for testing real time protocols, *TESTCOM*, Ottawa, Canada, 2000.

[90] O. Koné and R. Castanet. Deriving coordinated testers for interoperability. *Protocol Test System VI*, Pau, France 28 - 30 September 1993.

[91] O. Koné and R. Castanet. Test generation for interworking systems. *Computer Communications*, pp. 642 - 652, 23, 2000.

[92] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. *In SPIN 2004*, Spring-Verlag Heidelberg, pp. 109-126, 2004.

[93] Francois Laroussinie and Kim G. Larsen. CMC : a tool for compositional model-checking of real-time systems. *In Proc of FORTE-PSTV'98*, Ed by Stan Budkowski, Ana Cavalli, Elie Najm, pp. 439-456. Kluwer Academic, Paris, France, 1998.

[94] Kim G. Larsen, Paul Pettersson and Wang Yi. Diagnostic model-checking for real-time systems. *In Proc. of WVCHS III*, number 1066 in LNCS, pp. 575-586. Springer-Verlag, October 1995.

[95] K. G. Larsen, F. Larsson, P. Pettersson and Wang Yi. Efficient verification of real-time systems : compact data structure and state-space reduction. *In Proc 18 th IEEE Real-Time Systems Symposium, RTSS'97*, IEEE Computer Society Press, San Francisco, California, USA, December 1997.

[96] K. G. Larsen, P. Pettersson and W. Yi. Uppaal in a Nutshell. *Journal of Software Tools for Technology Transfer*, 1(1-2) :134-152, Oct. 1997.

[97] Guy Leduc. A framework based on implementation relations for implementing LOTOS specifications. *In Computer Networks ans ISDN Systems* 25, pp. 23-41. North Holand 1992.

[98] David Lee and M. Yannakakis. Testing finite state machines : state identification and verification. *IEEE Trans. Computer*, 43(3) :306-320, 1994.

[99] David Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. *Proceedings of the IEEE*, 84(8) :1090-1123, August, 1996.

[100] D. Lee et al. Passive testing and applications to network management. *ICNP'97 International Conference on Network Protocols*, Atlanta, Georgia, October 28-31, 1997.

[101] M. Lindal, P. Pettersson and W. Yi. Formal design and analysis of a gear controller. *In the proceeding of the fourth workshop on Tools and Algorithms for the Construction and Analysis of Systems*, Lisbon, Portugal, volume 1384 of LNCS, pp. 281-297. Springer-Verlag, 1998.

[102] The formal description technique LOTOS. *Edited by P.H. van Eijk, C.A. Vissers and M. Diaz*, North Holland, 1989.

[103] Gang Luo, Rachida Dssouli, Gregor V. Bochmann, Pallapa Venkataram and Abderrazak Ghedamsi. Generating synchronizable test sequences based on finite state machine with distributed ports. *IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test Systems*, Pau, France 28-30 September 1993.

[104] N. A. Lynch. I/O automata : a model for discrete event systems. *In Proc. of 22nd Conf. on Information Sciences and Systems*, pp. 29-38, Princeton, NJ, USA, March 1988.

[105] Dino Mandrioli, Sandro Morasca and Angelo Morzenti. Generating test cases for real-time systems from logic specifications. *ACM Transactions on Computer Systems*, 13(4) :365-398, 1995.

[106] P. Merlin. A study of the recoverability of computer system. *PhD thesis*, Dep. Comput. Sci, Univ. California, 1974.

[107] P. Merlin and D.J. Faber. Recoverability of communicating protocols. *IEEE Trans Comm*, 1976.

[108] R. Milner. A calculus of communicating systems. *Lecture Notes Computer Science 92*, 1980.

[109] R. Milner. Four Combinators for Concurrency. *ACM*, 1982.

[110] R. Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science 25*, 267-310, 1983.

[111] R. Milner. Communication and Concurrency. *Prentice-Hall*, 1989.

[112] K. Myungchul, K. Gyuhyeong, D.C. Yoon. Interoperability testing methodology and guidelines. *Digital Audio-Visual Council, System Integration TC*, DA-VIC/TC/SYS/96/06/006, 1996.

[113] S. Naito and M. Tsunoyama. Fault detection for sequential machines by transition tours. *In Proc. IEE Fault Tolerant Comput. Symp., IEEE Computer Press*,pp 238-243, 1981.

[114] P. Niebert, S. Tripakis and S. Yovine. Minimum-time reachability for timed automata. *In Mediterranean Conference on Control and Automation*, 2000.

[115] R. De Nicola and M.C.B Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34, pp 83-133, 1984

[116] R. De Nicola, R. Segala. A process algebraic view of I/O automata. *In Theoretical Computer Science, 138 :391-423*, March 1995.

[117] B. Neilson ans A. Skou. Automated test generation for timed automata. *TACAS'01*, LNCS 2031, Springer 2001.

[118] Maurice Nivat. Sur la synchronisation des processus.
Revue Technique Thomson-CSF, no.11, pp. 899-919, 1979.

[119] N. Okazaki, M. R. Park, K, Takahashi and N. Shiratori. A new test sequence generation method for interoperability testing. *Protocol Test System VII*, Tokyo, Japan, 8 - 10 November 1994.

[120] Carl Adam Petri. Bonn : Institut für Instrumentelle Mathematik, *Schriften des IIM Nr. 2*, 1962. Second Edition : New York, riffiss Air Force Base, *Technical Report RADC-TR-65–377*, vol.1, pp. : Suppl. 1, English translation, 1966.

[121] Carl Adam Petri. Fundamentals of a theory of asynchronous information Flow. *Proc. of IFIP Congress 62*, pp. 386-390 ,Amsterdam, North Holland Publ. Comp., 1963.

[122] Paul Pettersson. Modelling and verification or real-time systems using timed automata : theory and practice. *PhD thesis*, Uppsala University, Uppssala, Sweden, February 1999.

[123] M. Phalippou. Raltions d'implanatations et hypothèses de test sur les automates à entrées et sorties. *PhD thesis, Université de Bordeaux 1*, 1994.

[124] O. Rafiq and R. Castanet. From conformance testing to interoperability testing. *Pro. 3rd IWPTS*, October-November, Washington D.C, 1990.

[125] C. Ramchandani. Analysis of asynchronous concurrent systems by timed Petri nets. *PhD thesis*, MIT, 1974.

[126] O. Rayner. Future directions for protocol testing, learning the lessons from the past. *IFIP TC6, 10th International Workshop on Testing of Communicating Systems (IWTCS'97)*, Cheju Island, Korea, September 8 - 10 , 1997.

[127] C. Read. EurOSInet : raising the standards, IT Perspectives Conference. *The future of Information Technology,* pp. 7 - 73, 1988.

[128] J. M. Rushby. Theorem proving for verification. *In the proceeding of Modelling and Verification of Parallel Processes (MOVEP2k)*, vol 2067 of LNCS, pp. 187-195. Springer-Verlag, 2001.

[129] R. Ruys and R. Langerak. Validation of the Bosch'mobile communication network architecture with SPIN. *In participant Proceedings of the third SPIN, Workshop SPIN'97*, Enschede, The Netherlands, 1997.

[130] J. Ryu, M. Kim, S. Kang, S. Seol. Interoperability test suite generation for the TCP data part using experimental design techniques. *Proceedings of TESTCOM'00*, pp. 127-142, August 29-September 01, 2000.

[131] K. Sabnani and A. Dahbura. A protocol test test generation procedure. *Computer Networks and ISDN Systems*, 15 :285-297, 1988.

[132] K. V. Sam. Formalisation de l'interfonctionnement dans les réseaux de télécommunication et définition d'une théorie de test pour systèmes concurrents. Université Paris 7, June 2001.

[133] S. Seol, M. Kim, S. Kang. Interoperability test suite derivation for the TCP protocol. *In FORTE/PSTV'99*, 1999.

[134] S. Seol, M. Kim and S. T. Chanson. Interoperability test generation for communication protocols based on multiple stimuli principle. *Proceedings of TestCom'22)*, Berlin, Germany, March 19 - 22 , 2002.

[135] S. Seol, M. Kim, S. Kang, J. Ryu. Fully automated interoperability test suite derivation for communication protocols. *Computer Networks* Volume 43, pp. 735 - 759, December 2003.

[136] J. Shin and S. kang. Interoperability test suite derivation for the ATM/B-ISDN signaling protocol. *Proceedings of IWTCS'98*, Tomsk, Russia, August31-September 2, 1998.

[137] Joseph Sifakis. Use of Petri nets for performance evolution. *In Proc. of Measuring Modeling and Evaluating Computer Systems*, pp. 75-93, North-Holland, 1977.

[138] Robert de Simone. Higher-level synchronising devices in MEIJE-SCCS. *Theoretical Computer Science*, 37 :245-267, 1985.

[139] P. Schnoebelen, B. Bérard, M. Bidoit, F. Laroussinie and A. Petit. Systems and software verification - Model-Checking techniques and tools. *Springer-Verlag*, 2001.

[140] SPAG. Process to support interoperability. *Participants Handbook*, 1993.

[141] Jan Springintveld, Frits Vaandrager and Pedro R. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 252(1-2) :225-257, March 2001.

[142] V. Trenkaev, M. Kim, S. Seol. Interoperability testing based on a fault model for a system of communicating FSMs. *Proceedings of TestCom'03*, Sophia Antipolis, France, May 26-28, 2003.

[143] Jan Tretmans. A formal approach to conformance testing. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[144] Jan Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software - Concepts and Tools* 17(3) : pp. 103-120, 1996.

[145] Stavros Tripakis. The formal analysis of timed systems in practice. PhD thesis, Université Joseph Fourier de Grenoble, 1998.

[146] Stavros Tripakis. Timed diagnostics for reachability properties. *In Tools and Algorithms for the Construction and Analysis of Systems, TACAS'99*, Amsterdam, Holland, 1999.

[147] G. S. Vermmeer and H. Blik. Interoperability testing : basis of the accepting of communicating systems. *Protocol Test System VI*, Elsevier Science Publishers 1994.

[148] S. T. Vuong and J. Alilociv-Curgus. On test coverage metrics for communication protocols. *In Kroon J. Heijink R.J and Brinksma E. editors. Protocol Test System IV, volume C-3 of IFIP Transactions*, North-Holand, 1992.

[149] N. Walter. Timed nets for modeling and analysing protocols with time. *In Proc of the IFIP Conference on Protocol Specification Testing and Verification*, North-Holland, 1983.

[150] T. Walter, I. Schieferdecker and J. Grabowski. Test architecture for distributed systems - State of art and beyond. *IFIP TC6, 11th International Workshop on Testing of Communication Systems (IWTCS'98)*, Tomsk, Russia, August 31 - September 2, 1998.

[151] Glynn Winskel and Mogens Nielson. Models of Concurrency. *Vol. 3 of Handbook of Logic in Computer Science*, Oxford University Press, 1994.

[152] Marc F. Witteman and Ronald C. van Wijtswinkel. ATM broadband network testing using the Ferry principle. *IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test Systems*, Pau, 28-30 France, September 1993.

[153] Nina Yevtushenko.Conformance methods and architectures. *Tarot Summer School*, June 28, Paris, France, 2005.

[154] W. Yi. A calculus of real-time systems. *PhD thesis*, Chalmers University of Technology, Sweden, 1991.

[155] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. *In Proc 1996 IEEE Real-Time Systems Symposium, RTSS'96*, IEEE Computer Society Press, Washington DC, USA, December 1996.

[156] Sergio Yovine. Kronos : A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1-2) :123-133, December 1997.

[157] H.X. Zeng and D. Rayner. The impact of Ferry concept on protocol testing. *Protocol Spécification, Testing and Verification*, North-Holland publishers, pp. 519-531, 1986.

[158] H.X. Zeng, S.T. Chanson and B.R. Smith. On Ferry clip approaches in protocol testing, *Computer Networks and ISDN Systems*, Vol. 17(2) :77-88, 1989.

[159] H. Zhu, P. Hall and I. May. Software unit test coverage and adequacy. *ACM Computing Surveys* 29(4), 1997.

Annexe A

Preuves de la partie III

Propriété 1.

Preuve. Il suffit de remarquer que si c est un cycle alors il existe une suite finie $(ec_i)_{i\in[1,n]}$ de e-cycles telle que :

$$w(c) = \sum_{i\in[1,n]} w(ec_i).$$

En conséquence, $\forall i \in [1,n]$, $w(ec_i) \geq 0 \Rightarrow w(c) \geq 0$. □

Propriété 2.

Preuve. G minimal et positif implique que les 2-cycles sont positifs est trivial. Montrons que si les 2-cycles de G sont positifs alors tout cycle de G est positif (et par conséquence G sera positif). Soit ec un e-cycle de G qui traverse les noeuds n_i et n_j. Alors ec peut etre ecris sous la forme de $p_i.p_j$ tel que $p_i \in path(n_i \rightarrow n_j)$ et $p_i \in path(n_j \rightarrow n_i)$. Comme G est minimal, alors

$$w(n_i \rightarrow n_j) \leq w(p_i),\ w(n_j \rightarrow n_i) \leq w(p_j)$$

Par conséquence,

$$w(n_i \rightarrow n_j \rightarrow n_i) = w(n_i \rightarrow n_j) + w(n_j \rightarrow n_i) \leq w(p_i) + w(p_j) = w(ec)$$

Comme $n_i \rightarrow n_j \rightarrow n_i$ est un 2-cycle et il est positif, on déduit alors que ec l'est aussi. □

Lemme 3.

Preuve. Soit c un e-cycle qui traverse les k arcs e_i : $c = e_1 \cdots e_k$.

$\Leftarrow$: D'après la construction de $b2m(G)$:

$$w_{b2m(G)}(\overline{e}_k) \leq w_G(e_1 \cdots e_{k-1}) \; et \; w_{b2m(G)}(e_k) \leq w_G(e_k),$$

Donc,

$$w_{b2m(G)}(e_k.\overline{e}_k) = w_{b2m(G)}(e_k) + w_{b2m(G)}(\overline{e}_k) \leq w_G(e_1 \cdots e_{k-1}) + w_G(e_k) = w_G(c).$$

Comme $b2m(G)$ est positif et minimal alors selon la propriété 2, les 2-cycles sont positifs. $e_k.\overline{e}_k$ est un 2-cycle donc $w_{b2m(G)}(e_k.\overline{e}_k)$ est positif et par conséquence $w_G(c)$ est positif.

$\Rightarrow$: Soit e un arc. Par construction de $b2m(G)$, il existe deux chemins p_1 et p_2 tels que :

$$w_G(p_1) = w_{b2m(G)}(e) \; et \; w_G(p_1) = w_{b2m(G)}(\overline{e})$$

Il suffit alors de remarquer que $e.\overline{e}$ est un 2-cycle, $p_1.p_2$ est un cycle et

$$w_G(p_1.p_2) = w_{b2m(G)}(e.\overline{e})$$

Comme G est positif, alors $e.\overline{e}$ est positif et d'après la propriété 2, $b2m(G)$ est positif. □

Lemme 4.

Preuve. Soit c un e-cycle. Si c ne traverse pas le noeud n_i alors son poids est identique dans G et $R_{i\to *}(G)$. Maintenant, soit $c = p.e.p'$ est un e-cycle tel que $dst(e) = n_i$ ($e \in in(n_i)$), et $p'.p \in path(\overline{e})$. Remarquons que $src(p'.p) = n_i$ et qu'il ne traverse pas n_i. Donc son poids est le même dans G et $R_{i\to *}(G)$.

$$w_{R_{i\to *}(G)}(c) = w_{R_{i\to *}(G)}(p) + w_{R_{i\to *}(G)}(e) + w_{R_{i\to *}(G)}(p').$$

Par construction de $R_{i\to *}(G)$, on déduit que :

$$w_{R_{i\to *}(G)}(c) = w_G(p) - w_G(\overline{e}) + w_G(p')$$

Si $w_{R_{i\to *}(G)}(c) < 0$ alors :

$$w_G(p) + w_G(p') < w_G(\overline{e}).$$

Comme $p'.p \in path(\overline{e})$, alors contradiction avec le fait que G est minimal. □

Lemme 5.

Preuve. D'après le lemme 4, on déduit que $R_{i\to*}(G)$ est positif. Avant de présenter la preuve, remarquons que pour un graphe positif G' et un arc e, s'il existe un chemin $p \in path(e)$ qui traverse un noeud deux fois tel que $w_{G'}(p) \leq w_{G'}(e)$, alors il existe un chemin $p' \in path(e)$ qui ne traverse aucun noeud deux fois tel que $w_{G'}(p') \leq w_{G'}(e)$. Ceci est une conséquence directe de la positivité de G'. Ainsi, les poids, des chemins de $path(e)$, potentielllement inférieurs au poids de e dans G' sont ceux des chemins qui ne traversent aucun noeud deux fois. Par la suite, nous ne considérons que des chemins de $path(e)$ qui ne traverse aucun noeud deux fois.

Soit e un arc et p un chemin de $path(e)$. Par construction de $R_{i\to*}(G)$, seuls les arcs de $in(n_i)$ ont été changés par rapport à G.

Cas 1 : $e \in out(n_i)$. p ne passe pas par un arc de $R_{i\to*}(G)$ dont le poids a été changé par rapport à G. Alors $w_{R_{i\to*}(G)}(p) = w_G(p) \geq w_G(e) = w_{R_{i\to*}(G)}(e)$ (G est minimal). Par conséquence, $w_{G_1}(e) = w_G(e)$.

Cas 2 : $e \in in(n_i)$. Supposons que $w_{R_{i\to*}(G)}(p) < w_{R_{i\to*}(G)}(e) = -w_{R_{i\to*}(G)}(\overline{e})$. Sachant que $p \in path(e)$, donc $c = p.\overline{e}$ est e-cycle dont le poids est négatif. Contradiction avec le fait que $R_{i\to*}(G)$ est positif. Donc, $w_{R_{i\to*}(G)}(p) \geq w_{R_{i\to*}(G)}(e) = -w_G(\overline{e})$. Par conséquence, $w_{G_1}(e) = -w_G(\overline{e})$.

Cas 3 : $e \notin out(n_i) \cup in(n_i)$. Dans ce cas $w_{R_{i\to*}(G)}(e) = w_G(e)$.

1. Si p ne traverse pas le noeud n_i alors $w_{R_{i\to*}(G)}(p) = w_G(p) \geq w_G(e) = w_{R_{i\to*}(G)}(e)$ (G est minimal).
2. Si p passe par n_i. Soient $e_1 \in in(n_i)$ et $e_2 \in out(n_i)$ tels que $src(e_1) = src(e)$ et $dst(e_2) = dst(e)$. D'après les deux cas précédents, pour tout $p_1 \in path(e_1)$, pour tout $p_2 \in path(e_2)$, $w_{R_{i\to*}(G)}(p_1) \geq w_{R_{i\to*}(G)}(e_1)$ et $w_{R_{i\to*}(G)}(p_2) \geq w_{R_{i\to*}(G)}(e_2)$. Comme p peut être décomposé en $p_1 \in path(e_1)$ et $p_2 \in path(e_2)$: $p = p_1.p_2$, alors $w_{R_{i\to*}(G)}(p) \geq w_{R_{i\to*}(G)}(e_1) + w_{R_{i\to*}(G)}(e_2)$. Il suffit de voir que $w_{G_1}(e) = min\{w_{R_{i\to*}(G)}(e), w_{R_{i\to*}(G)}(e_1) + w_{R_{i\to*}(G)}(e_2)\}$, et qu' à partir des deux points suivants, $w_{G_1}(e) = -w_G(\overline{e}_1) + w_G(e_2)$.
   (a) $e_2 \in out(n_i)$ implique $w_G(e_2) \leq w_G(\overline{e}_1) + w_G(e)$, i.e. $-w_G(\overline{e}_1) + w_G(e_2) \leq w_G(e)$ (G est minimal).
   (b) $w_{R_{i\to*}(G)}(e) = w_G(e)$, $w_{R_{i\to*}(G)}(e_1) = -w_G(\overline{e}_1)$ et $w_{R_{i\to*}(G)}(e_2) = w_G(e_2)$. □

Lemme 8.

Preuve. Supposons que Z peut être écrit sous la forme

$$Z = \bigwedge_{v_i \in V_k} (v_i - v_0 \leq l_{i0} \wedge v_0 - v_i \leq l_{0i}) \wedge Z'$$

avec $Z' \in \Phi(V)$, et considérons

$$Z_k = \bigwedge_{v_i \in V_k} (v_i - v_0 \leq \nu(v_i) \wedge v_0 - v_i \leq -\nu(v_i)) \wedge Z'$$

S'il existe $i \in [1, k]$ tel que $\nu(v_i) > l_{i0}$ ou $\nu(v_i) < -l_{0i}$, alors ν ne peut pas être complété dans Z car on aura $Z_k \not\subseteq Z$. Maintenant nous proposons de calculer $cf(Z_k)$. Soit $G(Z)$ (resp. $G(Z_k)$) le graphe de contraintes associé à Z (resp. Z_k). $G(Z)$ est minimal car Z est canonique. Considérons maintenant le graphe $GM(Z_k) = (V_0, \overline{\mathbb{Z}}, E)$ construit par les étapes suivantes :

Pour tout $i \in [1, k]$,

$$w_{GM(Z_k)}(v_i \rightarrow v_0) = w_{G(Z_k)}(v_i \rightarrow v_0),\ w_{GM(Z_k)}(v_0 \rightarrow v_i) = w_{G(Z_k)}(v_0 \rightarrow v_i)$$

Pour tout $i \in [k+1, n]$,

$$w_{GM(Z_k)}(v_i \rightarrow v_0)$$

$$=$$

$$min(\{w_{G(Z_k)}(v_i \rightarrow v_j) + w_{G(Z_k)}(v_j \rightarrow v_0) \mid j \in [1, k]\} \cup \{w_{G(Z_k)}(v_i \rightarrow v_0)\}),$$

et

$$w_{GM(Z_k)}(v_0 \rightarrow v_i)$$

$$=$$

$$min(\{w_{G(Z_k)}(v_0 \rightarrow v_j) + w_{G(Z_k)}(v_j \rightarrow v_i) \mid j \in [1, k]\} \cup \{w_{G(Z_k)}(v_0 \rightarrow v_i)\})$$

Pour tout $i \in [1, n]$, pour tout $j \in [1, n]$, $i \neq j \neq 0$,

$$w_{GM(Z_k)}(v_i \rightarrow v_j) = min(w_{GM(Z_k)}(v_i \rightarrow v_0) + w_{GM(Z_k)}(v_0 \rightarrow v_j), w_{G(Z_k)}(v_i \rightarrow v_j))$$

Il suffit de voir alors que $GM(Z_k)$ est minimal et qu'il correspond à $G_m(Z_k) = GM(Z_k)$. D'après la propriété 2, $G_M(Z_k)$ est positif si et seulement si ses 2-cycles sont positif.

En conclusion, la construction de $G_m(Z_k)$ donne la forme canonique $cf(Z_k)$ de Z_k. Cette construction se fait en $O(n^2)$. Vérifier que $cf(Z_k) \not\sim false$ se fait aussi en $O(n^2)$ grâce aux 2-cycles. Si $cf(Z_k) \not\sim false$ alors, l'application directe du théorème 2 à $cf(Z_k)$ (dans le cas ou $Z_k \not\sim false$), nous donne entre 1 et $(n + 1)$ valuation qui complète ν dans Z si Z_k n'est pas borné et entre 1 et $2 \times (n+1)$ valuations de Z dans le cas contraire. □

Théorème 3.

Preuve.

Nous montrons le théorème dans le cas où $k = 0$. Pour $k \neq 0$ la preuve est similaire. Montrons d'abord que pour toute séquence σ admettant une computation de ρ, $time(\sigma_k^{(M,0)}) \leq time(\sigma_k) \leq time(\sigma_k^{(m,0)})$.
L'idée de la preuve est de montrer que les computations $(\overline{s}, \overline{\nu})$ et $(\overline{s}, \overline{\nu'})$) définies respectivement par :

1. $\nu_0(x) = 0$ pour tout $x \in C$, et $\nu_i = (\nu_{i-1} + time(\sigma_i^M) - time(\sigma_{i-1}^M))[r_i := 0]$ pour tout $i \in [1, n]$.
2. $\nu'_0(x) = 0$ pour tout $x \in C$, et $\nu'_i = (\nu'_{i-1} + time(\sigma_i^m) - time(\sigma_{i-1}^m))[r'_i := 0]$ pour tout $i \in [1, n]$.

sont des computation de ρ sur σ^M et σ^m respectivement. Nous donnerons une preuve pour σ^M. La preuve pour σ^m est similaire. La preuve est par récurrence.

Pour $i = 0$ il est clair que $\nu_0 \in H_0$. Supposons que pour tout $j \in [0, i]$, $\nu_j \in H_j$ et montrons que $\nu_{i+1} \in H_{i+1}$. Rappelons que $H_{i+1} = (s_{i+1}, g_{i+1})$ et que g_{i+1} est canonique et borné. Soit $\nu^1 = \nu_0^M(g_{i+1})$ la valuation de bornes maximale associée à g_{i+1} et donnée par le théorème 2. Cette valuation vérifie que pour toute valuation $\nu \in g_{i+1}$, pour toute $x \in C$, $\nu(x) \leq \nu^1(x)$. Vu que le chemin $S_-^+(\rho)$ est post/pred-stable, alors selon le corollaire 8, il existe une computation $r : (\overline{s}, \overline{\nu''})$ sur un mot temporisé σ, tel que : $s = s_{i+1}$ et $\nu''_{i+1} = \nu^1$.
Pour toute horloge $x \in C$, soit p_x l'indice inférieur à $i + 1$ de la transition t_{p_x} où x a été remise à zéro pour la dernière fois.

- Si $\nu^1(x) = 0$ alors $\nu_{i+1}(x) = 0$.
- Sinon, $\nu^1(x) = \nu''_{i+1}(h) - \nu''_{p_x}(h) \geq \nu_{i+1}(h) - \nu_{p_x}(h) = \nu_{i+1}(x)$. Comme v_i admet un successeur dans H_{i+1} (le chemin $S_-^+(\rho)$ est post/pred-stable), alors il existe $\nu_2 \in H_{i+1}$ telle que pour toute $x, y \in C$ non remise à zéro dans t_{i+1}, $\nu_2(x) - \nu_2(y) = \nu_i(x) - \nu_i(y) = \nu_{i+1}(x) - \nu_{i+1}(y) \leq \nu^1(x) - \nu^1(y)$. Ceci est une conséquence de l'évolution en même rythme des horloges.

En conclusion, $\nu_{i+1} \in H_{i+1}$. Par conséquent, $\sigma^{(M,0)}$ (resp. $\sigma^{(m,0)}$) est la séquence σ^{M0} (resp.σ^{m0} =) donnée par la définition 31, section 7.1.

Annexe B

Preuves de la partie IV

Cette partie présente quelques propriétés des IOLTS

Propriété 3 *Soient M_1 et M_2 deux IOLTS, X un ensemble d'interfaces et* $\sigma \in Traces(M_1\|_X M_2)$. *Alors*

$Out(M_1\|_X M_2, \sigma) = \cup_{\sigma' \in P_{M_1\|M_2}(\sigma,X)} Out_{\Sigma^{(M_1\|M_2)|X}}(M_1\|M_2, \sigma')$. □

Preuve. Pour montrer l'égalité de la propriété, nous montrerons l'inclusion des deux ensembles dans les deux sens.

$\Leftarrow$) Soient $\sigma' \in P_{M_1\|M_2}(\sigma, X)$ et $!a \in Out_{\Sigma^{(M_1\|M_2)|X}}(M_1\|M_2, \sigma')$. Alors,

$$!a \in \Sigma^{(M_1\|M_2)|X} \; et \; (q_0^{M_1}, q_0^{M_2}) \overset{\sigma'.!a}{\Rightarrow}_{M_1\|M_2}$$

Vu que $\sigma \in Traces(M_1\|_X M_2)$, alors $\sigma_{|\Sigma^{(M_1|M_2)|X}} = \sigma$ et

$$(\sigma'.!a)_{|\Sigma^{(M_1\|M_2)|X}} = \sigma'_{|\Sigma^{(M_1\|M_2)|X}}.!a = \sigma_{|\Sigma^{(M_1\|M_2)|X}}.!a = \sigma.!a$$

En faisant la projection de $(q_0^{M_1}, q_0^{M_2}) \overset{\sigma'.!a}{\Rightarrow}_{M_1\|M_2}$ sur $\Sigma^{(M_1\|M_2)|X}$, on obtient $(q_0^{M_1}, q_0^{M_2}) \overset{\sigma.!a}{\Rightarrow}_{M_1\|_X M_2}$, et ainsi $!a \in Out(M_1\|_X M_2, \sigma)$.

$\Rightarrow$) Soit Let $!a \in Out(M_1\|_X M_2, \sigma)$. Par construction la seule différence entre $M_1\|_X M_2$ et $M_1\|M_2$ réside dans les transitions sur les événement n'appartenant pas à $\Sigma^{(M1\|M_2)|X}$. Ces dérniers sont remplacés dans $M_1\|_X M_2$ par τ. Ainsi, si $(q_0^{M_1}, q_0^{M_2}) \overset{\sigma.!a}{\Rightarrow}_{M_1\|_X M_2}$, alors il existe $\sigma' \in M_1\|M_2$ telles que $(q_0^{M_1}, q_0^{M_2}) \overset{\sigma'.!a}{\Rightarrow}_{M_1\|M_2}$ et $\sigma'_{/\Sigma^{(M_1\|M_2)|X}} = \sigma_{/\Sigma^{(M_1\|M_2)|X}} = \sigma$. □

Propriété 4 *Supposons que M_1 et M_2 sont input-complets. Alors*

$\forall \sigma \in Traces(M_1\|M_2) \Rightarrow Out(M_1\|M_2, \sigma) = Out(M_1, \sigma_{/\Sigma^{M_1}}) \cup Out(M_2, \sigma_{/\Sigma^{M_2}})$. □

Preuve. Pour montrer l'égalité de la propriété, nous montrerons l'inclusion des deux ensembles dans les deux sens.

$\Rightarrow$) Trivial.

$\Leftarrow$) Soit $!a \in Out(M_1, \sigma_{/\Sigma^{M_1}})$. Deux cas sont alors possibles :
- Si $!a \notin \Sigma^{M_2}$ ($!a$ n'est pas un événement de synchronisation). Dans ce cas, M_1 évolue indenpendamenet de M_2, et ainsi $!a \in Out(M_1 \| M_2, \sigma)$.
- Si $!a \in \Sigma^{M_2}$. Comme M_2 est input-complet, alors la synchronisation entre M_1 et M_2 est possible et ainsi $!a \in Out(M_1 \| M_2, \sigma)$.

Le même raisonnement reste valable pour $!a \in Out(M_2, \sigma_{/\Sigma^{M_2}})$. □

Propriété 5 *Supposons que M_1 et M_2 sont input-complets et que $\Sigma_o^{M_1} \cap \Sigma_o^{M_2} = \emptyset$. Alors :*
$\Sigma_o^{(M_1\|M_2)|X} \cap \Sigma_o^{M_1} = \Sigma_o^{M_1|X}$, *et* $\Sigma_o^{(M_1\|M_2)|X} \cap \Sigma_o^{M_2} = \Sigma_o^{M_2|X}$. □

Preuve. Du fait que M_1 et M_2 sont input-complets, toute émission possible dans un état q de M_1 ou M_2 est aussi possible dans un état gloable de $M_1 \| M_2$ contenant q et réciproquement. Ainsi, $\Sigma_o^{M_1\|M_2} = \Sigma_o^{M_1} \cup \Sigma_o^{M_2}$ et $\Sigma_o^{(M_1\|M_2)|X} = \Sigma_o^{M_1|X} \cup \Sigma_o^{M_2|X}$. Vu que $\Sigma_o^{M_1} \cap \Sigma_o^{M_2} = \emptyset$ alors $\Sigma_o^{M_1|X} \cap \Sigma_o^{M_2|X} = \emptyset$. Ainsi $\Sigma_o^{(M_1\|M_2)|X} \cap \Sigma_o^{M_1} = \Sigma_o^{M_1|X}$. Le même raisonnement pour le deuxième point de la propriété. □

Corollaire 16 *Supposons que M_1 et M_2 sont input-complets et que $\Sigma_o^{M_1} \cap \Sigma_o^{M_2} = \emptyset$. Soit $\sigma \in Traces(M_1 \|_X M_2)$, alors*
- $Out_{\Sigma^{M_1}}(M_1 \|_X M_2, \sigma) = \cup_{\sigma' \in P^{M_1\|M_2}(\sigma, X)} Out_{\Sigma^{M_1|X}}(M_1, \sigma'_{/\Sigma^{M_1}})$.
- $Out_{\Sigma^{M_2}}(M_1 \|_X M_2, \sigma) = \cup_{\sigma' \in P_{M_1\|M_2}(\sigma, X)} Out_{\Sigma^{M_2|X}}(M_2, \sigma'_{/\Sigma^{M_2}})$.

Preuve. Selon la propriété 3

$$Out(M_1 \|_X M_2, \sigma) = \cup_{\sigma' \in P_{M_1\|M_2}(\sigma, X)} Out_{\Sigma^{(M_1\|M_2)|X}}(M_1 \| M_2, \sigma')$$

ce qui implique que

$$Out_{\Sigma^{M_1}}(M_1 \|_X M_2, \sigma) = \cup_{\sigma' \in P_{M_1\|M_2}(\sigma, X)} Out_{\Sigma^{(M_1\|M_2)|X} \cap \Sigma^{M_1}}(M_1 \| M_2, \sigma').$$

Et selon la propriété 5, on obtient :

$$Out_{\Sigma^{M_1}}(M_1 \|_X M_2, \sigma) = \cup_{\sigma' \in P_{M_1\|M_2}(\sigma, X)} Out_{\Sigma^{M_1|X}}(M_1 \| M_2, \sigma')$$

D'apres la propriété 4, on déduit que :
$Out_{\Sigma^{M_1}}(M_1 \|_X M_2, \sigma) = \cup_{\sigma' \in P_{M_1\|M_2}(\sigma, X)} Out_{\Sigma^{M_1|X}}(M_1, \sigma'_{/\Sigma^{M_1}}) \bigcup \cup_{\sigma' \in P_{M_1\|M_2}(\sigma, X)}$

$Out_{\Sigma^{M_1|X}}(M_2, \sigma'_{/\Sigma^{M_2}})$.
Comme $\Sigma_o^{M_1} \cap \Sigma_o^{M_2} = \emptyset$ alors $\Sigma_o^{M_1|X} \cap \Sigma_o^{M_2} = \emptyset$, et ainsi

$$Out_{\Sigma^{M_1}}(M_1 \|_X M_2, \sigma) = \cup_{\sigma' \in P_{M_1 \| M_2}(\sigma, X)} Out_{\Sigma^{M_1|X}}(M_1, \sigma'_{/\Sigma^{M_1}})$$

□

Lemme 11.

Preuve. Soit $\sigma \in Traces(M_1 \|_X M_2)$.

$\Rightarrow$) I_1 et I_2 sont input-complets, alors selon la preuve de la propriété 5,

$$Out(I_1 \|_X I_2, \sigma) = Out_{\Sigma^{I_1}}(I_1 \|_X I_2, \sigma) \bigcup Out_{\Sigma^{I_2}}(I_1 \|_X I_2, \sigma)$$

Comme $interop_X(I_1, I_2)$ implique que

$$Out_{\Sigma^{I_1}}(I_1 \|_X I_2, \sigma) \subset \cup_{\sigma' \in P_{M_1 \| M_2}(\sigma, X)} Out_{\Sigma^{M_1|X}}(M_1, \sigma'_{/\Sigma^{M_1}})$$

ou encore $interop_X(I_2, I_1)$ implique

$$Out_{\Sigma^{I_2}}(I_1 \|_X I_2, \sigma) \subset \cup_{\sigma' \in P_{M_1 \| M_2}(\sigma, X)} Out_{\Sigma^{M_2|X}}(M_1, \sigma'_{/\Sigma^{M_2}})$$

Comme,

$$Out(M_1 \|_X M_2, \sigma) = Out_{\Sigma^{M_1}}(M_1 \|_X M_2, \sigma) \bigcup Out_{\Sigma^{M_2}}(M_1 \|_X M_2, \sigma)$$

par application du corollaire 16, on déduit que

$$Out(I_1 \|_X I_2, \sigma) \subset Out(M_1 \|_X M_2, \sigma)$$

$\Leftarrow$) $I_1 \|_X I_2 \; ioconf \; M_1 \|_X M_2$ implique que

$$Out_{\Sigma^{I_1}}(I_1 \|_X I_2, \sigma) \bigcup Out_{\Sigma^{I_2}}(I_1 \|_X I_2, \sigma) \subset Out_{\Sigma^{M_1}}(M_1 \|_X M_2, \sigma) \bigcup Out_{\Sigma^{M_2}}(M_1 \|_X M_2, \sigma)$$

car M_1, M_2, I_1 et I_2 sont input-complets. Nous avons supposé que $\Sigma_o^{I_1} \cap \Sigma_o^{M_2} = \emptyset$, ce qui implique que

$$Out_{\Sigma^{I_1}}(I_1 \|_X I_2, \sigma) \cap Out_{\Sigma^{M_2}}(M_1 \|_X M_2, \sigma) = \emptyset$$

et ainsi

$$Out_{\Sigma^{I_1}}(I_1 \|_X I_2, \sigma) \subset Out_{\Sigma^{M_1}}(M_1 \|_X M_2, \sigma)$$

Par application du corollaire 16, on déduit que

$$Out_{\Sigma^{M_1}}(M_1 \|_X M_2, \sigma) = \cup_{\sigma' \in P_{\delta(M_1 \| M_2)}(\sigma, X)} Out_{\Sigma^{M_1|X}}(M_1, \sigma'_{/\Sigma^{M_1}})$$

et ainsi $interop_X(I_1, I_2)$. La même preuve pour $interop_X(I_2, I_1)$.

Index

Symbols

- $\bowtie$ 21
- $+\infty$ 18
- $-\infty$ 18
- 2^X 18
- $:=$ 18
- $=$ 18
- $?other$ 132
- A_ρ 81
- $CS(S)$ 181
- $Comp(S)$ 181
- $H = (s, \nu)$ 104
- M *visible* X 142
- $M \mid X$ 142
- $M_1 \mid\mid_X M_2$ 142
- $Out(A, \sigma)$ 155
- $Out_L(M, \sigma)$ 142
- $Out_e(A, \sigma)$ 154
- $Out_\epsilon(A, \sigma)$ 155
- P^{M_i} 141
- $R_{*\to i}()$ 88
- $R_{i\to *}()$ 87
- $Run(A)$ 43
- $SA(A, c)$ 160
- $STC_M(S)$ 161
- $S^+(\rho)$ 107
- $S_-(\rho)$ 108
- $TTrace(A)$ 43, 153
- $TTrace(A, n)$ 43
- $TTrace(A_\rho, n)$ 101
- $TW(\Sigma)$ 37
- $T_{total}(Y, X)$ 74
- V_0 90
- $Z[X := 0]$ 22, 197
- $Z \cap Z'$ 196
- $Z \subseteq Z'$ 196
- $Z^\downarrow$ 22, 196
- $Z^\uparrow$ 22, 196
- $[X := 0]Z$ 22, 197
- $\Phi(C, P, V)$ 48
- $\Phi(V)$ 21
- $\Phi_I(C)$ 44
- $\Rightarrow$ 18
- Σ^* 24
- Σ^+ 24
- $\setminus$ 18
- $\bullet$ 44, 58
- $\cap$ 18
- $\cup$ 18
- δ 128
- $\delta(M)$ 128
- $\emptyset$ 18
- ϵ 24
- $\epsilon(d)$ 37
- $\equiv$ 18
- λ 197
- $\leq_{tr}$ 126
- $\leq_{ttr}$ 153
- $\lfloor\,\rfloor$ 50
- $\mathbb{N}$ 18
- $\mathbb{R}$ 18
- $\mathbb{T}$ 18
- $\mathbb{Z}$ 18
- $\overline{\mathbb{T}}$ 18
- $||$ 28
- $\neg$ 18
- $\nu + d$ 21
- $\nu - d$ 21
- $\nu[X := 0]$ 21
- $\overline{e}$ 87

- $\overrightarrow{G}(N, L)$ 19
- $\overrightarrow{G}_c(N, L)$ 20
- $\overrightarrow{G}_m(N, \mathbb{T})$ 85
- $\overrightarrow{\Sigma^{tr}}$ 58
- $\overrightarrow{v}$ 58
- $\prec$ 193
- σ_i 25
- $\sigma_{|L}$ 25
- $\sim$ 21
- τ 24
- $\vee$ 18
- $\wedge$ 18
- $b2m()$ 86
- $c_{max}(A)$ 41, 160
- $c_{max}(Z)$ 22, 41
- $card(X)$ 18
- $cf(M)$ 195
- $close(Z, c)$ 22, 197
- $d.\nu$ 21
- $delay()$ 37
- $dst()$ 19
- $event()$ 37
- $f(P, V)$ 48
- $fr()$ 50
- $friend()$ 174
- $in()$ 19
- $intercom$ 144
- $intercom_X$ 144
- $interop$ 143
- $interop_X$ 143
- $ioco$ 127
- $ioco_t$ 155
- $ioconf$ 127
- $ioconf_t$ 154
- $max(X)$ 18
- $min(X)$ 18
- $out()$ 19
- $p?a$ 142
- $path_G()$ 20
- $post()$ 83, 105
- $post_c()$ 160
- $pred()$ 83, 106
- $q\,after\,\sigma$ 154
- $src()$ 19
- $time()$ 37
- w_G 85
- $w_G()$ 19
- $\mathcal{TIOA}(\Sigma_i, \Sigma_o)$ 48
- $\mathcal{TA}(\Sigma)$ 41
- $\mathcal{V}(V)$ 20

A

- Accept 129
- Accessibilité 104, 160
- Action 24
  - d'événement 37
  - d'attente 37
  - d'attente (voir •) 59
  - d'entrée 48
  - de sortie 48
  - globale de synchronisation 58
  - interne 24, 28
  - observable 28
  - silencieuse 24
  - temporisée 37
- Activité
  - du test 189
- Activité du test 136
- Algèbre
  - alternative gardée 33
  - CCS 32, 52
  - composition parallèle 33
  - de processus 15, 32, 55
  - LOTOS 32
  - préfixage 32
  - produit 63
  - récursion 33
  - renommage 33, 63
  - restriction 33, 63
  - TCCS 52
- Algorithme de Floyd 194
- Algorithme de Peterson 66
- Algorithme générique de génération 180
  - distribué 181
- Alphabet 24, 37

- Analyse
  - d'accessibilité 82
  - en arrière 82, 106, 107
  - en avant 82, 105, 107
  - formelle 3
  - symbolique 107
- Arc
  - conjugué 19, 87
  - destination 19
  - source 19
- Architecture
  - à cheval 139
  - de test 124, 137
  - active 138
  - d'interopérabilité 138
  - de conformité 137
  - Ferry 137
  - Générique 139
  - OSI 137
  - OSI distribuée 138
  - passive 138
- Architecture de test 181
- ATC 158
- Automate 15, 55
  - de simulation 160
  - des régions 50
  - suspendu 128
  - traducteur 58
- Automate temporisé
  - étendu à entrée/sortie (voir ETIOA) . 48
  - à entrée/sortie (voir TIOA) 48
  - avec invariants (voir TAI) 44
  - avec Urgence (voir TAU) 45
  - d'Alur et Dill (voir TA) 40

B

- Blocage 127
  - de sortie 127
  - vivant (livelock) 127
- Borne 193
- Broadcast 55, 59
- Bus 72

C

- Canonique 195
  - forme 195
- Cas de test 124, 132, 156
  - abstrait 158
  - contrôlable 132
  - dérivation 145, 161
  - formalisme 157
  - interopérabilité 145
  - observable 132
  - squelette 158
  - temporisé 157
- Chemin 20, 41
  - exécutable 198
  - faisable 198
  - longueur 20
  - symbolique 82, 112
  - symbolique par $post()$ 107
- CMC 10
- Collision 71
- Coloriage 174
  - distribué 174
  - distribué ordonné 174
  - ordonné 174
- Complétude 133
- Composante 57, 135
- Composition
  - pc 145
  - tc 145
  - fonction 44, 58
  - selon f 44
  - synchrone des LTSs 28
  - synchrone des TAs 43
- Compositionnalité 16
- Computation
  - complétude 110
  - k-complétude 110
  - k-incompléte 110
- Conceptualisation 2
- Conformité 119, 124, 125
  - définitions 152
- Conjuguée 46
- Consommateur 64

- Contrôle ... 3
- Contrainte
  - atomique ... 21, 91
  - de variables ... 21
  - diagonale ... 21, 91
  - diaphantienne ... 91, 94
  - redondante ... 90
- Correction ... 133
- Couleur ... 174
  - amie ... 174
- Couverture ... 171
  - comportementale ... 172
  - critère ... 171, 172
  - d'états ... 173
  - définition-utilisation ... 173
  - de chemins ... 173
  - de transitions ... 173
  - de variables ... 173
  - structurelle ... 172
- CS ... 60
  - binaire ... 62
  - libre ... 62
  - sémantique ... 61
- CSMA/CD ... 71
- CSUT ... 180
- Cycle ... 20, 85
  - élémentaire ... 20
  - 2-cycle ... 20
  - de développement ... 2
  - e-cycle ... 20, 85
  - racine ... 20

D

- DBM ... 194
- Deadline ... 45
- deadlock ... 127
- Delayable ... 46
- Diagnostic temporisé ... 81
  - de bornes ... 82, 103, 112, 162, 165
  - de bornes maximales ... 103, 112
  - de bornes minimales ... 103, 112
  - extraction ... 81
- Diffusion ... 59
- Discret ... 157
- Données ... 4

E

- Eager ... 46
- Elément ... 174
- Emetteur ... 71
- Entité ... 57, 135
- Equivalence ... 93
- Etat
  - atteignable ... 107
  - prédécesseur ... 106
  - successeur ... 83, 106
  - symbolique ... 83, 104
- ETIOA ... 48
- Evénement (voir Action) ... 24
- Exécutabilité ... 16
- Exécutable ... 175
- Exécution ... 132
- Exhaustive ... 175
- Exhaustivité ... 133
- Expression linéaire ... 48
- Expressivité ... 16
- Extraction
  - valuation de borne ... 198

F

- Fail ... 124, 131
- FIFO ... 55, 75
- File d'attente ... 75
- Fonction
  - amie ... 174
  - partielle ... 44
- Forme canonique ... 194
- FSM ... 25

G

- Génie logiciel ... 2
- gga ... 180
- Graphe ... 19
  - étiqueté ... 19
  - complet ... 20
  - d'accessibilité ... 51
  - de contraintes ... 92

des courts chemins 85
minimal 85
orienté 19
positif 85

H

• Horloge 39
analogique 156
digitale 156
universelle 101, 112, 157
• Hypothèses de test 124
• HyTech 9

I

• IF 10
• Implantation 2, 124
• Inégalité linéaire 48
• Inc 131
• Inclusion de traces
temporisées 153
• Inclusion des traces 81
• Indéterminisme 16
• Infinité positive 91
• Interface 141
• Interopérabilité 134
définitions 135
en couches 136
implantations 136
spécifications 136
testable
en couche 136
en système 136
théorique
en couche 136
en système 136
• IOLTS 29
déterministe 30
input-complet 30
notations 30
observable 30
• IUT 124
analogique 157
digital 157

K

• k-incomplète
computation 110
valuation 98
• Kronos 10

L

• Latence 73
• Lazy 46
• LTS 27
notations standards 29

M

• Méthode
DS 9
H 9
TT 9
UIO 9
W 9
• Méthodes formelles 1, 119
• Méthodologie *gga* 180
• Machine
à états finis 25
de Mealy 25
de Moore 25
• Matrice des bornes 194
• Modélisation 2, 181
active 187
passive 184
uniforme 120
• Modèle 124
• Multicast 55, 59

O

• Objectif de test 125, 129
complet 130

P

• Parallélisme
asynchrone 15
synchrone 15
• Pass 124, 131
• Passage
en arrière 111

en avant . . . 110
en avant et en arrière . . . 111
• PCO . . . 124
• Poids . . . 19, 85
• Points de contrôle et d'observation . . . 124
• Polyèdre . . . 21
borné . . . 21
c-clôture . . . 22, 197
c-close . . . 22
canonique . . . 92
convexe . . . 22
de contraintes . . . 82, 102
false . . . 21
forme canonique . . . 93
inclusion . . . 196
intersection . . . 196
opérations . . . 22, 195
Prédécesseur
après ré-initialisation . . . 197
prédécesseur . . . 196
successeur . . . 196
après ré-initialisation . . . 197
true . . . 21
zero . . . 21
• Port . . . 141
• Positivité . . . 86
• post() . . . 105
• Post-stabilité de pred . . . 106
• Prédécesseur temporel . . . 106
• Préordre . . . 126
• pred() . . . 106
• Pred-stabilité de post . . . 106
• Preuve . . . 1
• Procédure
SymbolicTrace . . . 199
UpdateContext . . . 201
UpdatePredicates . . . 200
• Processus . . . 32, 58
• Producteur . . . 64
• Produit synchrone . . . 131
• Propriété . . . 189
• Protocole . . . 119

R

• Région d'horloges . . . 49
• Réseaux de Petri (voir RdP) . . . 31
• Rang . . . 174
• RdP . . . 31
marquage . . . 32
stochastique temporisé . . . 52
temporel . . . 52
temporisé . . . 52
• Reject . . . 129
• Relation
$\leq_{tr}$ . . . 126
$\leq_{ttr}$ . . . 153
$intercom$ . . . 144
$intercom_X$ . . . 144
$interop$ . . . 143
$interop_X$ . . . 143
$ioco$ . . . 127
$ioco_t$ . . . 155
$ioconf$ . . . 127
$ioconf_t$ . . . 154
d'interopérabilité . . . 142
de conformité . . . 125
• RTS . . . 152

S

• Séquence . . . 25
concaténation . . . 25
inscrite . . . 160
longueur . . . 25
projection . . . 25
temporisée . . . 37
• Silence . . . 127
• Spécification . . . 2, 124
hiérarchique . . . 15, 61
• STC . . . 158
• Successeur
temporel
clôture . . . 160
• Successeur temporel . . . 105
• Suite de tests . . . 124
• Synchronisation . . . 58
• Système

communicant asynchrone 55
communicant synchrone 55
communicant 55
sous test 180
communicant (voir CS) 60
distribué asynchrone 75
distribué synchrone 73
hybride 51
parallèle 15
séquentiel 15
• Système de transitions
étiqueté (voir LTS) 27
à entrée/sortie(voir IOLTS) 29
temporisé (voir TLTS) 37

T

• TA 40
événementiellement déterministe 41
composition synchrone 43
computation 42
déterministe 41
non-bloquant 42
observable 41
sémantique 41
• TAI 44
sémantique 45
• TAU 45
deadline 45
delayable 46
eager 46
lazy 46
sémantique 47
urgence à l'Uppaal 46
urgence à la Kronos 46
• Temps 4
de communication 74
• Test 1, 3, 119
actif 179, 186
adaptatif 157
analogique 156, 165
boîte blanche 120
boîte grise 120
boîte noire 120
d'interopérabilité 120
de conformité 119, 123, 152
de performance 120
de robustesse 120
digital 156, 162
hypothèses 146
ouvert 189
passif 179, 182
statique 157
temporisé 152
• Testeur 124
analogique 156
digital 156
• TGV 10
• Théorème de Floyd 94
• tick 156
• TIOA 48
input-complet 48
• TLTS 37
additivité du temps 37
attente 0 38
déterminisme temporel 37
exécution 38
exécution d'attente 38
notations 38
• Topologie
binaire 59
de communication 58
libre 59
statique 59
• TorX 11
• TP 129
• Trace
Complète 185
d'exécution 182
inclusion 103, 113, 126
partielle 185
passive 183
symbolique 199
temporisée 43
• Traducteur 58
• Transformation
$R_{*\to i}()$ 88

$R_{i\to *}()$ 87
$b2m()$ 86
- Transition
  discrète 41
  garde 40
  post-stable 108
  post/pred stable 109
  pred-stable 107
  temporisée 41
- TTC 157
- Types de test 119

U

- Unicast 55, 59
- Universalité 16
- Uppaal 9
- Urgence 45

V

- Vérifiabilité 16
- Vérification 1, 2, 81
- Valuation 20
  équivalence 50
  0-incomplète 98
  c-équivalence 21
  de bornes 96
  de bornes maximales 96
  de bornes minimales 96
  extraction 90
  k-complétude 98
  k-incompléte 98
  opérations 21
- Variable fictive 90, 193
- Vecteur
  binaire 59
  de synchronisation 58
  unitaire 59
- Verdict 124, 131

Z

- Zenon 42
- Zone 104
  post-stable 106
  pred-stable 106

Modélisation, Analyse et Test des Systèmes Communicants à Contraintes Temporelles : Vers une Approche Ouverte du Test

Résumé : Cette thèse se positionne dans le cadre de la modélisation, l'analyse et le test des systèmes communicants, plus particulièrement les systèmes temps-réel, embarqués et à base de composants.

Dans un premier temps, nous introduisons le modèle CS comme une description des systèmes communicants. Le modèle CS est basé sur la séparation entre la description des entités d'un système communicant et la description des communications et du partage des ressources entre ces dernières. Nous montrons aussi comment traiter le cas des systèmes asynchrones et tenir compte des latences du réseau. Dans une seconde partie, nous abordons l'analyse des systèmes temporisés, en particulier, l'extraction des diagnostics temporisés pour un chemin symbolique. Nous montrons aussi comment extraire des valuations dites de bornes à partir d'un polyèdre représenté par matrice de bornes (DBM). Dans une troisième partie, nous considérons la génération automatique des tests temporisés. Nous introduisons un cadre formel, comportant une définition de la notion de conformité pour les systèmes temporisés et une approche de génération automatique des tests de conformité. Dans une quatrième partie, nous proposons une modélisation uniforme des différents types, approches et architecture de test. Cette modélisation est basée sur les algorithmes génériques de génération et la définition d'un critère de couverture structurelle sous forme de coloriage (ordonné) de graphe. Comme conclusion de cette partie, nous introduisons la notion du test ouvert.

Finalement, nous avons réalisé l'outil TGSE (Test Generation, Simulation and Emulation) développé dans le cadre du projet Calife et l'étude de cas CSMA/CD.

Mots-clés : Systèmes Temps-Réel, Systèmes Communicants, Modélisation, Analyse, Diagnostic Temporisé, Test, Conformité, Interopérabilité, Couverture, Minimisation de Test, Test Ouvert.

Modeling, Analyzing and Testing Real-Time Communicanting Systems : Towards an Open Approach of Testing

Abstract : The topics of this thesis are modeling, analyzing and testing communicating systems, specially real-time systems, embedded systems and component-based systems.

The first part of this thesis is related to the study of models for the description of systems. We introduce the CS model as a description of communicating systems which defines a set of common resources, a set of entities, and a topology of communication. The second part considers the analyze of real-time systems, specially the extraction of timed diagnostics for a symbolic path. We show how to extract bounded valuation from a DBM (Difference Bounded Matrix). The third part of this thesis is devoted to test case generation. We introduce a formal framework for conformance and interoperability testing. The forth part focuses on testing methodologies adapted to protocol testing. We give a formal definition of a generic generation algorithm (GGA). We demonstrate that the CS model with a GGA supports various 1) test architectures, 2) test types : conformance, interoperability, embedded, component testing, and 3) test approaches : passive and active testing.

The last part of this thesis presents the main characteristics of the TGSE tool (Test Generation, Simulation, and Emulation). TGSE is made-up of a test case generator, a graphic simulator of the execution of a sequence generated by TGSE, and a real-time emulator.

Keywords : Modeling, Analyze, Timed Diagnostic, Testing, Conformance, Interoperability, Real-Time Systems, Communicanting Systems, Coverage, Test Cases Minimization, Open Testing.

Thèse en informatique, préparée au LaBRI (Université Bordeaux I, 351 cours de la Libération, 33405 Talence).