HABILITATION À DIRIGER DES RECHERCHES

présentée devant

**L'Université de Rennes 1**
**Institut de Formation Supérieure**
**en Informatique et en Communication**

par

Thierry JÉRON

# Contribution à la génération automatique de tests pour les systèmes réactifs

**soutenue le (Version du 6 février 2004) devant le jury composé de**

| | | |
|---|---|---|
| MM. | Gaudel Marie-Claude | Rapporteurs |
| | Brinksma Ed | |
| | Petrenko Alexandre | |
| MM. | Groz Roland | Examinateurs |
| | Jard Claude | |
| | Marie Raymond | |

# Table des matières


**1 Introduction** **5**

**2 Le test de conformité** **11**
2.1 Un peu de terminologie . . . . . 11
2.2 De la nécessité de formaliser pour automatiser . . . . . 12
2.3 Un survol des techniques de génération de tests de conformité . . . . . 14
2.3.1 Méthodes fondées sur les automates . . . . . 14
Modèle et problème . . . . . 15
Séquences d'identification . . . . . 16
Synthèse de test . . . . . 18
Avantages et inconvénients . . . . . 19
2.3.2 Méthodes fondées sur les systèmes de transition . . . . . 20
2.3.3 Techniques algébriques . . . . . 21

**3 Modèles, théorie et principes de la génération de tests à la volée** **25**
3.1 Une théorie du test de conformité . . . . . 26
3.1.1 Modèles de la conformité . . . . . 26
3.1.2 Relation de conformité . . . . . 37
3.1.3 Cas de tests et suites de tests : modèles, exécution et propriétés . . . 42
3.2 Algorithmes de synthèse de cas de tests . . . . . 48
3.2.1 Un algorithme de synthèse non déterministe . . . . . 48
3.2.2 Algorithme de génération avec objectifs de test sur les traces suspendues . . . . . 50
Une première formalisation des objectifs de test . . . . . 51
Algorithme de génération de tests . . . . . 53
Propriétés des tests générés . . . . . 60
3.2.3 Algorithme de génération de tests avec objectifs de test ayant des actions internes . . . . . 61
Justification . . . . . 61
Formalisation des objectifs de test avec actions internes . . . . . 62
Algorithme de génération de tests . . . . . 63
Propriétés des cas de tests produits . . . . . 71
3.2.4 Complexité des algorithmes . . . . . 72
3.2.5 Alternative . . . . . 72

3.3 Efficacité algorithmique : génération à la volée . . . . . 73
3.3.1 Composantes fortement connexes et problèmes d'accessibilité . . . 74
3.3.2 Utilisation des calculs de CFC . . . . . 75
3.3.3 Enchaînement à la volée . . . . . 77
3.4 L'outil TGV . . . . . 78
3.4.1 Architecture de TGV . . . . . 78
3.4.2 Langages supportés . . . . . 79
3.4.3 Autres caractéristiques de TGV . . . . . 80
3.4.4 Études de cas . . . . . 81
3.5 Comparaison avec d'autres techniques et outils . . . . . 83
3.6 Conclusion sur la génération de tests énumérative à la volée . . . . . 85

**4 Génération de tests symboliques 87**
4.1 Le modèle des $\mathcal{IOSTS}$ . . . . . 88
4.1.1 Syntaxe des $\mathcal{IOSTS}$ . . . . . 88
4.1.2 Sémantique des $\mathcal{IOSTS}$ . . . . . 91
4.1.3 Abstraction d'$\mathcal{IOSTS}$ . . . . . 94
4.2 Théorie du test dans le cadre symbolique . . . . . 96
4.3 Principes de la génération de tests symboliques . . . . . 98
4.3.1 Objectifs de test . . . . . 99
4.3.2 Produit synchrone . . . . . 102
4.3.3 Comportements visibles . . . . . 106
4.3.4 Sélection . . . . . 117
4.4 Conclusions sur le test symbolique . . . . . 122
4.4.1 Travaux connexes . . . . . 124
TVéda . . . . . 125
Test symbolique pour Lotos . . . . . 125
GaTeL . . . . . 126
Agatha . . . . . 126
BZ-Testing tool . . . . . 127
Analyse . . . . . 127

**5 Conclusion générale et perspectives 129**

# Table des figures


2.1 Test de conformité . . . . . 11
2.2 Exemple de machine de Mealy . . . . . 15
2.3 Illustration des séquences d'identification . . . . . 17
2.4 Spécification $M1$ et implémentation $M2$ n'ayant pas les mêmes UIO . . . . 19

3.1 Illustration des notations sur les IOLTS . . . . . 28
3.2 Non déterminisme au sens des automates . . . . . 30
3.3 Non déterminisme observable . . . . . 30
3.4 Déterminisation . . . . . 33
3.5 Perte des blocages par déterminisation . . . . . 34
3.6 Construction de l'automate de suspension . . . . . 34
3.7 Comportements visibles et blocages . . . . . 36
3.8 **ioco** par l'exemple (expliqué sur $\Delta(S)$ et $\Delta(IUT)$) . . . . . 39
3.9 Exécution d'un cas de test et verdicts . . . . . 45
3.10 Algorithme de génération de tests non déterministe : exemple . . . . . 50
3.11 Spécification et objectif de tests . . . . . 52
3.12 Résumé des opérations de l'algorithme de synthèse de test *gen_test_OT1* . . 53
3.13 Produit synchrone $PS^{\text{VIS}} = det(\Delta(S)) \times TP$ . . . . . 55
3.14 Graphe de test complet . . . . . 57
3.15 Conflits de contrôlabilité . . . . . 59
3.16 Cas de tests . . . . . 59
3.17 Test dans un contexte . . . . . 62
3.18 Exemple d'objectif de test . . . . . 63
3.19 Résumé des opérations de synthèse de test . . . . . 63
3.20 Spécification $S$ et objectif de test $TP$ . . . . . 64
3.21 Produit $PS = S \times TP$ pour la spécification et l'objetif de la figure 3.20 . . 66
3.22 Comportements visibles $PS^{\text{VIS}} = det(\Delta(PS))$ . . . . . 68
3.23 Graphe de test complet $CTG$ . . . . . 70
3.24 Cas de test . . . . . 71
3.25 Calcul du graphe de test complet . . . . . 77
3.26 Architecture de TGV . . . . . 79
3.27 Cas de test avec temporisateurs . . . . . 80

4.1 Exemple d'$\mathcal{IOSTS}_{loc}$ . . . . . 91
4.2 $\mathcal{IOSTS}$ et sémantique . . . . . 93

4.3 Objectif de test dans le modèle $\mathcal{IOSTS}_{loc}$ . . . . . 100
4.4 Heuristique de calcul de sous-ensemble . . . . . 115
4.5 Calcul de sous-ensemble symbolique . . . . . 116
4.6 Graphe de tests complet pour l'abstraction sur les localités . . . . . 120
4.7 Graphe de tests complet calculé par NBAC . . . . . 121
4.8 Schéma général de la génération de tests symbolique . . . . . 123

# Chapitre 1

# Introduction

**Note :** *Ce document est une synthèse d'une partie des travaux que j'ai effectués depuis quelques années dans le domaine du test. J'ai volontairement concentré le discours de ce document sur cette activité, laissant de coté d'autres recherches que j'ai menées au début de ma carrière de chercheur. Ces autres travaux sont cependant résumés dans le dossier de synthèse de travaux qui accompagne ce document.*

Le problème de la fiabilité du logiciel revêt une importance de plus en plus importante dans notre société moderne où le logiciel envahit notre quotidien. L'électronique fait de plus en plus place à l'informatique dans des domaines aussi variés que les télécommunications, le commerce (électronique), le transport, l'énergie, l'information, la médecine, etc. Par ailleurs le logiciel est de plus en plus complexe, réparti et réactif. Ces facteurs rendent difficiles et coûteuses sa conception, son développement et sa maintenance. On est donc parfois amené à faire des compromis entre la qualité du logiciel et le coût de son développement. Le logiciel est aussi parfois critique, au sens ou sa défaillance peut avoir des conséquences dramatiques de point de vue humain ou matériel. Dans ce cas, on ne voudra pas sacrifier à la qualité, mais on souhaite diminuer le côut à qualité constante, voire croissante. La fiabilité du logiel au moindre coût est donc une préoccupation importante dans bien des domaines de l'informatique.

Déjà le logiciel séquentiel pose de nombreux problèmes de correction. L'analyse, la conception et la réalisation d'un logiciel séquentiel correct est loin d'être une tâche facile. En effet, on sait depuis longtemps que la plupart des problèmes non triviaux qu'on se pose sur les programmes sont indécidables. Ce n'est pas une raison pour abandonner. On peut appréhender le problème sous l'angle de la conception en améliorant les techniques de conception, les méthodes de programmation. On peut aussi le voir sous l'angle de l'analyse des programmes (ou de modèles de ceux-ci), par l'utilisation de techniques de simulation, de vérification, de preuve de programmes, d'analyse statique, de test, etc.

Le problème de correction est encore plus difficile pour le logiciel réparti. La répartition du code et des données rendent encore plus difficile la perception *a priori* des comportements possibles d'un programme. L'état global du programme ne peut en général pas être

capturé à l'exécution. La communication asynchrone entre différents codes répartis est difficile à appréhender. Écrire directement un programme réparti correct est donc quasiment impossible aussitôt qu'on dépasse la taille de l'exemple jouet. Les méthodes de modélisation et conception, comme les méthodes orientées objet tentent là aussi de faciliter le travail du développeur mais ne résolvent pas tout les problèmes.

Dans les phases de conception de logiciel, il est nécessaire de passer par des étapes de raffinement successives et si possible, de vérifier ou de se convaincre à chaque pas qu'on ne perd pas les propriétés acquises au niveau précédent. Pour plusieurs raisons, comme les problèmes de rupture de modèle dans le cycle de développement, de taille aussi, et bien sûr de décidabilité, on ne peut pas en général garantir la correction du code.

La vérification formelle permet de prouver que certains modèles de programmes sont correct. Mais même dans ce cas, le test reste incontournable. En effet, la vérification s'-effectue généralement sur un modèle du programme, mais ne porte pas sur le programme lui même, en tout cas pas dans son environnement réel d'exécution. Le test est d'ailleurs la technique de validation la plus utilisée. C'est aussi, dans le cycle de vie du logiciel, la phase la plus consommatrice en temps, et donc en argent. Ce facteur économique ne doit pas nous laisser insensibles et constitue un argument fort pour l'automatisation. Mais l'automatisation, en particulier l'automatisation de la génération de tests, doit aussi augmenter la confiance dans les résultats des tests. Pour ce faire, l'utilisation de techniques formelles s'impose. L'automatisation du test est donc, à notre avis, un levier très efficace pour persuader les industriels de l'apport des méthodes formelles. Le test étant déjà présent dans le processus de développement, si on est capable de montrer que l'utilisation de techniques formelles pour le test permet non seulement de gagner en qualité de logiciel, mais, de surcroît, de gagner en temps et en coût, et ceci malgré le sur-coût d'une spécification formelle, on convaincra sans aucun doute n'importe quel responsable industriel. Certaines expériences ont d'ailleurs tenté de le montrer montré [20].

La vérification a un désavantage par rapport à l'automatisation du test pour son introduction dans l'industrie. En effet, cette activité n'est en général pas déjà présente dans le cycle de vie du logiciel et son introduction bouscule le processus de développement habituel. Et il est plus difficile de convaincre les industriels que le sur-coût représenté par le développement d'une spécification formelle dédiée à la vérification sera amorti. Pourtant des expériences le montrent (voir [14]. Par contre, la synthèse de test peut servir à l'introduction d'autres techniques de validation formelles. Si le coût d'une spécification formelle est déjà amorti par le gain réalisé par le test, pourquoi ne pas utiliser cette même spécification pour la simulation ou comme point de départ pour la vérification ?

Le test de logiciel a plusieurs facettes, et les techniques de test diffèrent en fonction de plusieurs caractéristiques. D'abord, les techniques de test varient selon le niveau d'observabilité du code, depuis le test *boîte blanche* (dans lequel le code du système testé est connu) principalement utilisé dans le test unitaire ou d'intégration, jusqu'au test *boîte noire* (dans lequel le code du système testé n'est pas connu et seule l'interface du système est accessible) qui est plus utilisé en test système mais peut l'être en test unitaire et d'intégration, en passant par le test *boîte grise* (ou le code est inconnu mais certaines informations internes sont disponibles). Les méthodes de test varient aussi suivant les aspects ou les propriétés

testés : les fonctionnalités, les performances, la robustesse, la fiabilité, etc. Enfin, on n'emploie pas les mêmes techniques selon que l'on est le développeur, un utilisateur, ou une tierce partie donnant un avis indépendant sur le système testé.

Les différentes techniques de test ont cependant des caractéristiques communes. Tester consiste à *stimuler* (ou *contrôler*) le système sous test en fournissant des entrées, et à *observer* le système en termes de sorties ou d'état. Il faut ensuite comparer les observations avec les observations d'un système idéal de référence, et en déduire la correction ou non du système testé. C'est le problème de l'*oracle*. Le but principal du test est de détecter des erreurs (de conception ou de réalisation, suivant le niveau d'application du test) qui peuvent se caractérisent par des fautes (l'observation de la manifestation d'une erreur). Inversement, à travers le test, on souhaite aussi acquérir une certaine confiance dans la réalisation. C'est le cas lorsque le test ne détecte plus d'erreur. Cette confiance se mesure par la capacité d'un test à détecter des fautes, et est fonction de l'effort de test (en termes de quantité et de qualité). Sachant que le test ne peut pas être exhaustif, et ne fournit donc quasiment jamais de garantie de correction, les contraintes économiques imposent que l'effort de test soit un compromis entre la confiance apportée et son coût. Il faut donc sélectionner un ensemble de tests raisonnable (en termes de coût) mais suffisamment complet pour acquérir une confiance suffisante dans le système quand celui-ci passe correctement les tests.

Dans le cadre du test boîte blanche de code séquentiel, cette sélection est basée sur des *critères de couverture* structurels du code (voir le survey de Zhu [80]) qui identifient des séquences d'exécution sur un modèle abstrait du code. La couverture peut se baser sur la structure de contrôle (couverture d'états, de branches, de chemins, etc) ou sur le flot de données (définition et utilisation des variables, conditions, etc). A partir de cette structure, un critère de couverture produit un enseble de chemins dans le code. Se pose alors le problème de la faisabilité de ces chemins sur le code. Les techniques de résolution de contraintes sont assez populaires dans ce contexte. L'autre problème est celui de l'oracle, puisque le code lui-mème ne peut pas servir à décider de la correction du résultat d'un test.

Dans le cadre du test boîte noire (l'implémentation n'est connue que par son interface), la sélection des tests est basée sur une spécification de référence. Si la spécification est formelle, l'oracle est alors fourni par la spécification elle-même. Puisque le code de l'implémentation est inconnu, l'estimation de la qualité des tests, i.e. sa capacité à détecter des erreurs ne peut se faire qu'en faisant des hypothèses sur les implémentations testées.

Dans le cadre du test de logiciel réactif et réparti, nous nous sommes essentiellement intéressés au test de conformité et en particulier à l'automatisation de la génération de tests de conformité. Le test de conformité consiste à tester si une implémentation d'un programme est conforme à sa spécification. C'est un test fonctionnel de type boîte noire. C'est à dire que le comportement interne de l'implémentation sous test (IUT pour *Implementation Under Test*) n'est *a priori* pas connu. La raison en est que pour des exigences de confidentialité de leur savoir-faire, les développeurs de l'implémentation fournie à un client ou donneur d'ordre ne souhaitent en général pas que leur code soit connu. Seul le comportement externe est donc testé par interaction entre l'environnement (l'équipement de test ou le programme de test) et l'IUT.

La principale difficulté du test de conformité consiste en la génération d'un ensemble de tests raisonnable et ayant de bonnes propriétés. Notre vision de la génération de test de

conformité repose sur la formalisation mathématique du test et l'utilisation de techniques de vérification pour la sélection des tests. La formalisation est basée sur la description de modèles pour représenter les objets du test, i.e. principalement les spécifications, les implémentations, les tests. ELle est aussi basée sur une description précise de la notion de conformité par une relation entre implémentations et spécification, la formalisation des exécutions des tests et de leurs verdits, et des propriétés attendues des tests générés par rapport à la relation de conformité. Pour l'algorithmique de la génération de tests, nous nous sommes basés sur des techniques de vérification et d'anlyse statique qui permettent d'identifier de manière exacte ou approchée des comportements ciblés par le test.

Le domaine d'application que nous visions au début de nos travaux sur le test de conformité était le domaine des télécommunications. Cependant, au fil des expérimentations, nous nous sommes aperçu que les techniques employées, se basant sur des modèles assez généraux, ont un spectre d'applicabilité bien plus large. Nous avons pu ainsi abordé des d'applications aussi variés que des protocoles de télécommunication, des protocoles de cohérence de cache, ds systèmes embarqués dans les trsnports, des applications carte à puces, des systèmes d'information.

Le document est structuré comme suit. Dans le premier chapitre, nous présentons un peu de vocabulaire et la problématique du test de conformité. Nous faisons ensuite un rapide survol des principes de base de la génération automatique de tests de conformité pour différents modèles et théories.

Dans le second chapitre, nous présentons les techniques de génération automatique de tests pour des modèles de comportements de spécifications énumérés sous la forme de systèmes de transitions à entrée sortie (IOLTS), qui sont à la base de notre outil TGV. Nous décrivons les modèles et leurs caractéristiques, la théorie du test de conformité sur laquelle cet outil se base. Nous présentons ensuite les algorithmes de génération de tests et leurs propriétés. Nous décrivons l'implémentation de ces algorithmes dans l'outil TGV et quelques caractéristiques de l'outil. Nous finissons par quelques comparaisons avec des travaux connexes.

Dans le troisième chapitre, nous présentons les techniques symboliques de génération de tests, à la bse de notre outil STG. Nous introduisons un modèle généralisé de systèmes de transitions symboliques, les $\mathcal{IOSTS}$. Ces modèles manipulent des données de trois types différents, les constantes symboliques (non instanciées), les variables et les paramètres de communications. Les transitions sont gardées par des prédicats sur ces données et réalisent des communications et des affectations des variables. Leur sémantique est décrite sous la forme d'IOLTS étendus, ce qui permet de réutiliser la théorie du test des IOLTS. On exlpique ensuite la génération de tests à partir de tels modèles, qui consiste à des transformations symboliques sur les $\mathcal{IOSTS}$. La correction de la génération de tests repose alors sur les propriétés des sémantiques des $\mathcal{IOSTS}$ résultats des transformations. Nous ramenons ces preuves de correction à la preuve d'une commutation entre transformation (symbolique sur les $\mathcal{IOSTS}$ ou sémantique sur les IOLTS) et sémantique. Nous montrons comment des techniques d'interprétation abstraite, peuvent permettre une sélection approchée des cas de tests. Nous terminons le chapitre par la comparaison de notre approche avec d'autres techniques symboliques, essentiellement basées sur de la propagation et de la résolution de contraintes.

Dans le dernier chapitre, nous tirons quelques conclusions des travaux présentés et dressons quelques perspectives de prolongement de ces travaux.

# Chapitre 2

# Le test de conformité

Le test de conformité est une technique de validation qui consiste à réaliser des expérimentations sur une implémentation d'un système réactif afin d'en déduire la correction vis à vis d'une spécification de référence. C'est un test de type fonctionnel car on ne s'intéresse qu'à la correction vis à vis des fonctions réalisées par la spécification. C'est un test boîte noire car le code de l'implémentation n'est pas connu. L'expérimentation consiste à faire interagir l'implémentation sous test (*IUT* pour *Implementation Under Test*) avec son environnement. Cet environnement est simulé par un testeur qui exécute des tests élémentaires appelés *cas de test*. Les comportements de l'IUT sont observés à travers des interfaces appelées *PCO* pour *Points de Contrôle et d'Observation* (voir figure 2.1).

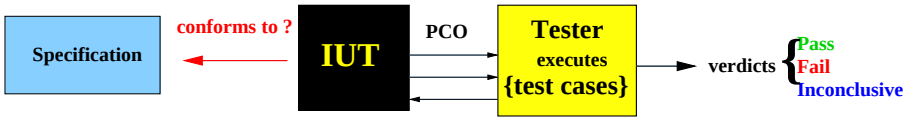



FIG. 2.1 – Test de conformité

Le test de conformité est utilisé dans divers domaines mais a été particulièrement étudié dans le domaine des protocoles de télécommunication. Dans ce cadre, la norme ISO 9646 [42] décrit les ingrédients principaux du test de conformité. Nous décrivons ci-dessous quelques termes définis dans cette norme et utilisés dans la suite du document.

## 2.1 Un peu de terminologie

On appelle *implémentation sous test* (IUT pour *Implementation Under Test*) l'implémentation réelle du système à tester. C'est en général un code exécutable qui n'est pas connu du testeur. Cela peut être aussi du matériel.

Le *testeur* est le programme exécutant les tests sur l'IUT.

L'*architecture de test* est la description abstraite de la situation du testeur vis à vis de l'IUT. Parmi les interfaces de l'IUT, elle distingue celles qui sont *contrôlables* ou *observ-*

*ables*. On utilise le terme *points de contrôle et d'observation (ou PCO)* pour désigner ces interfaces utiles au test. L'architecture de test est fonction de l'architecture réelle de test. La conception des tests dépend donc étroitement de l'architecture choisie. Plusieurs architectures type sont normalisées par [42] : architecture locale, distante, distribuée, etc...

Un *cas de test* est un test élémentaire. Une *suite de tests* est un ensemble de cas de tests. Pour un système réactif, un cas de test peut décrire une suite d'interactions entre le testeur et l'IUT, des opérations sur des temporisateurs de test, des *verdicts*. Il est généralement constitué d'un *préambule*, d'un *corps de test*, d'une *séquence d'identification* et d'un *postambule*. Il fait appel à des procédures qui sont des portions communes à plusieurs cas de test. En particulier, les parties préambule, séquence d'identification et postambule sont décrites par des procédures.

On désigne par *objectif de test* (*test purpose*) le but recherché par chaque cas de test, c'est à dire la propriété qu'il est supposé tester sur l'IUT. En général un objectif découle d'une *exigence de conformité* (*conformance requirement*) décrivant une propriété que le système (la spécification et son implémentation) est censé vérifier.

Le *préambule* est une partie initiale d'un cas de test qui consiste généralement en une séquence d'interactions dont le but est de mener l'IUT dans une configuration particulière permettant d'appliquer le corps du test. Le *corps du test* est la partie du cas de test servant à vérifier l'objectif de test. Après cela, on peut chercher à identifier l'état dans lequel se trouve l'IUT (ou plutôt se trouvait l'IUT avant la phase d'identification) en appliquant une suite d'interaction appelée *séquence d'identification*. Enfin, on peut vouloir revenir dans un état bien identifié, en général l'état initial, afin d'enchainer un autre cas de test, par l'intermédiaire d'un *postambule*. En réalité, il s'agit bien souvent de l'état de contrôle initial et non pas de l'état initial, c'est à dire qu'on ne se préoccupe pas forcément de remettre les variables du protocole à leur valeur initiale. Ceci peut alors poser des problèmes de verdict inconsistant lors de l'exécution du cas de test suivant, si les variables n'ont pas été proprement réinitialisées.

## 2.2 De la nécessité de formaliser pour automatiser

La pratique du test de conformité, comme d'autres types de test, reste encore très souvent artisanale. Les tests sont écrits à la main à partir de spécifications le plus souvent informelles, donc ambigües, par exemple en langage naturel. Le processus de conception des tests peut être systématique mais reste quand même *ad hoc*, pénible. Il est aussi assez fortement découplé de la conception et du développement, les équipes de test étant en général distinctes des équipes de conception et développement. Le métier de testeur étant de plus jugé moins créatif que celui de concepteur ou développeur, il est aussi moins bien considéré, et n'attire pas forcément les meilleurs ingénieurs, ce qui est bien entendu néfaste à la qualité des tests. La production des tests est donc mal maîtrisée, et l'obtention d'une suite de tests de qualité suffisante, est extrêmement coûteuse par rapport au coût global du logiciel (de 30 à 70 % selon les sources). Comme on imagine aisément que la qualité du logiciel est en rapport étroit avec l'effort et la qualité des tests, il est donc impératif de réduire le coût du test. Dans le cas de logiciel critique, bien plus qu'ailleurs, on est intéressé par augmenter la qualité du logiciel, donc la qualité des tests, pour espérer tendre vers le

zéro défaut. Face à la complexité croissante du logiciel, il est donc impératif de réduire le coût du test. Pour celà, l'automatisation est une nécessité.

L'industrie du logiciel, et par conséquent la communauté scientifique s'intéresse donc depuis de longues années à automatiser la production des tests, dans toutes ses phases. Mais l'automatisation nécessite des efforts préalables de formalisation. Dans le cas du test de conformité, la référence est une spécification. Cette spécification sert d'oracle et permet de définir les verdicts.

Le prérequis minimal à l'automatisation est donc la formalisation de la spécification afin que les comportements qu'elle décrit soient suffisamment précis et non ambigus pour que les verdicts aient un sens. On lit parfois que ces spécifications doivent être exécutables pour permettre l'automatisation. Ce n'est pas forcément nécessaire, mais elles doivent être au moins analysables. Elles doivent avoir une sémantique claire dans un modèle mathématique. Il est également nécessaire de formaliser l'interaction entre cas de test et IUT lors de l'exécution des tests, l'observation liée à cette exécution, les verdicts émis par les cas de test. Il faut également formaliser la conformité i.e. la relation entre l'IUT et sa spécification que l'on cherche à établir ou dont on cherche à détecter la violation. Il faut décrire les algorithmes permettant de synthétiser automatiquement les cas de test à partir d'une spécification, des concepts d'interaction, d'observation, de relation de conformité et de verdict, éventuellement d'objectifs de test pour la sélection. Il faut aussi démontrer la pertinence des tests synthétisés par ces algorithmes : permettent-ils de ne rejeter que des implémentations non conformes (correction ou non biais), toutes celles-ci (exhaustivité) ? Une initiative intéressante dans le domaine du télécommunications est la norme *Formal Methods in Conformance Testing* [43] qui répertorie et formalise l'ensemble de ces ingrédients à la synthèse automatique de tests.

En simplifiant, il faut définir les ingrédients suivants :

- la spécification est donnée dans un langage dont la sémantique est définie dans un modèle $\mathcal{SPEC}$. A une spécification $Spec$ on associe donc sa sémantique $S = [[Spec]]$ dans le modèle $\mathcal{SPEC}$. Par exemple pour le langage $SDL$ [79], la sémantique d'une spécification peut être donnée par un système de transitions,
- les implémentations sous test ne sont pas des objets mathématiques, mais, pour raisonner formellement, il faut supposer que toute implémentation est modélisable. Le modèle des implémentations peut être différent de celui des spécifications. On suppose donc que l'implémentation peut être modélisée par un élément $IUT$ du modèle $\mathcal{IUT}$. Il n'est pas toujours nécessaire de prendre pour $\mathcal{IUT}$ le même modèle que $\mathcal{SPEC}$
- la conformité d'une implémentation à sa spécification peut alors se définir comme une relation mathématique $imp \subseteq \mathcal{IUT} \times \mathcal{SPEC}$. Il s'agit en général d'une équivalence ou d'une inclusion de comportements.
- les cas de tests générés doivent appartenir à un modéle $\mathcal{TC}$. On peut utiliser des modèles de séquences, d'arbres, de graphes, et même de programmes.
- la sémantique de l'exécution des tests sur l'implémentation doit être exprimée par une fonction dont le résultat est une observation ou un ensemble d'observations dans un modèle $\mathcal{OBS}$ : $exec : \mathcal{TC} \times \mathcal{IUT} \rightarrow \mathcal{OBS}$.
- le verdict de l'exécution d'un cas de test est une fonction de l'ensemble des observations dans celui des verdicts $verdict : \mathcal{OBS} \rightarrow \mathcal{VERDICTS}$, ce qui permet de

définir quand l'exécution d'un cas de test ou d'une suite de tests rejette ou non une implémentation.
- un algorithme de génération de tests $gen_test$ est une fonction de l'ensemble des spécifications $\mathcal{SPEC}$ dans l'ensemble des cas de tests $\mathcal{TC}$ ou dans l'ensemble des parties de l'ensemble des cas de tests $2^{\mathcal{TC}}$ si l'algorithme génère plusieurs cas de tests.
- tout cela permet finalement de formaliser la cohérence entre la relation de conformité et les verdicts par des propriétés sur les tests obtenus. On souhaite au minimum qu'un cas de test produit par $gen_test$ depuis la sémantique d'une spécification $S$ ne puisse rejetter que des implémentations non conformes à $S$ pour $imp$. On peut également souhaiter que toute implémentation non conforme soit rejettée (ou puisse être rejetée) par un test qu'on peut générer (ou qu'on génère effectivement).

## 2.3 Un survol des techniques de génération de tests de conformité

Les méthodes de génération formelle de test à partir de spécifications formelles se décomposent en plusieurs familles. On ne parlera ici que des méthodes ayant une base formelle bien définie. En effet, il existe des techniques fondées sur des langages dits formels comme SDL, Lotos ou Estelle mais qui n'ont rien de vraiment formel au sens où il n'y a pas de théorie du test sous-jacente.

Les deux familles auxquelles on s'intéresse principalement ont comme base commune qu'elles partent de la donnée (au moins implicitement) des comportements d'une spécification formelle du protocole (on parle souvent de graphe d'états). Elles diffèrent sur le modèle utilisé pour représenter ce graphe d'états et par conséquent sur les techniques mises en œuvre. La première famille, la plus ancienne, est celle des méthodes fondées sur les automates. Elle est issue des méthodes de test de circuit et des problèmes de reconnaissance de machines [61, 29]. La deuxième famille est fondée sur les systèmes de transition et est issue des algèbres de processus et des problèmes d'équivalence de test.

Par la suite, nous allons examiner les deux familles de méthodes citées précédemment. Nous n'allons pas faire une synthèse complète qui nous mènerait beaucoup trop loin, mais plutôt essayer de donner les ingrédients de base de ces méthodes permettant de comprendre leur philosophie. Nous décrivons ensuite la technique de génération à la volée telle qu'elle existe dans l'outil TGV.

En fin de chapitre, on évoquera un autre type de méthodes basé sur les types de données abstraits. Ces techniques se focalisent sur les données et les opérations sur celles-ci, mais se combinent avec les techniques basées sur le contrôle, en particulier les techniques fondées sur les systèmes de transition.

### 2.3.1 Méthodes fondées sur les automates

La première famille de méthodes utilise comme modèle les machines de Mealy. Le lecteur désireux d'en savoir plus lira avec intérêt la synthèse de Lee et Yanakakis [55] ou la bibliographie commenté de Petrenko [66].

### Modèle et problème

Les machines de Mealy sont des automates déterministes dont chaque transition est étiquetée par une entrée et une sortie. Formellement :

**Définition 2.3.1 (Machine de Mealy)**
*Une machine de Mealy $M$ est un quintuplet (S, I, O, $\delta$, $\omega$) tel que :*
- *$S$ est une ensemble fini d'états,*
- *$I$ est l'alphabet fini d'entrée,*
- *$O$ est l'alphabet fini de sortie,*
- *$\delta : S \times I \rightarrow S$ est la fonction de transfert. On l'étend naturellement sur les séquences d'entrée en $\delta^* : S \times I^* \rightarrow S^*$,*
- *$\omega : S \times I \rightarrow O$ est la fonction de sortie. On l'étend également en $\omega^* : S \times I^* \rightarrow O^*$.*

La figure 2.2 décrit une machine de Mealy.

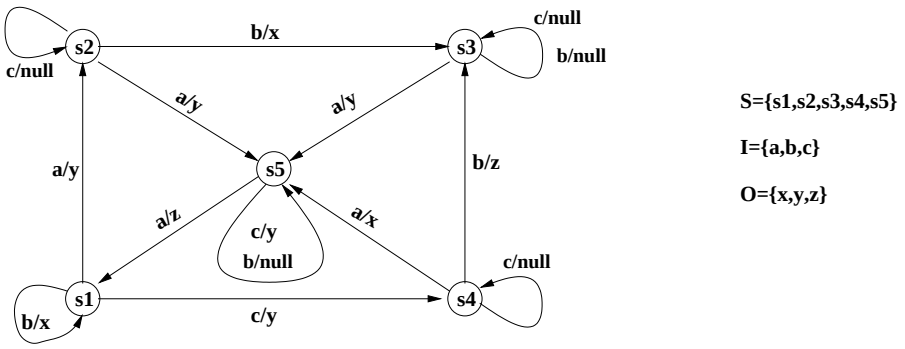



FIG. 2.2 – Exemple de machine de Mealy

La définition prise est directement celle des machines de Mealy déterministes puisque $\delta$ et $\omega$ sont des fonctions. On suppose en général aussi les machines complètes ($\delta$ et $\omega$ sont partout définies). Comme pour les automates d'états finis, il existe une notion d'équivalence entre états : deux états $s_i$ et $s_j$ sont *équivalents* si pour toute suite d'entrée, ils produisent les mêmes suites de sorties : $\forall \sigma \in I^*, \omega^*(s_i, \sigma) = \omega^*(s_j, \sigma)$. On dit qu'une machine est *minimale* si elle n'a pas de paire d'états distincts équivalents. Deux machines $M$ et $M'$ de mêmes alphabets sont dites *équivalentes* si pour tout état de $M$ il existe un état équivalent dans $M'$ et inversement. Pour toute machine $M$, on peut construire une machine minimale équivalente en calculant les classes d'équivalence d'états par raffinement de partition, de façon similaire à la minimisation classique des automates.

Le problème du test de conformité basé sur ce modèle est donc le suivant : étant donnée une machine $M$ connue (la spécification) et une machine $I$ inconnue (l'implémentation sous test) dont on peut seulement observer les entrées/sorties, déterminer par un test (une séquence finie d'entrées/sorties) si $I$ est équivalente à $M$.

Pour que ce problème ait une solution, on suppose en général que la spécification est minimale et fortement connexe, que l'implémentation est aussi minimale et fortement connexe, et qu'elle a le même nombre d'états que la spécification (on peut étendre aisément

à des implémentations ayant un nombre d'états qui n'excède celui de la spécification que par une constante). Moyennant ces hypothèses, pour une spécification donnée $M$, une implémentation $I$ est équivalente à $M$ si elle est isomorphe à $M$. Par ailleurs, l'ensemble des implémentations différentes de $M$ (à isomorphisme près) considérées est fini et borné. L'équivalence entre une implémentation $I$ et la spécification $M$ revient alors à vérifier que $I$ n'a aucune faute des types suivants :

- les fautes de sortie : dans un état $s$, pour une entrée $i$, la sortie produite par $I$ est $o'$ au lieu de la sortie attendue $o$.
- les fautes de transfert : dans un état donné $s$, après la transition étiquetée $i/o$, l'état d'arrivée est $s''$ au lieu de $s'$.

Dans la littérature, on parle souvent de *modèle de fautes* pour désigner l'ensemble de ces implémentations fautives.

La synthèse de test est fondée sur la recherche de la correction et de l'exhaustivité de la suite de test. C'est à dire que l'on cherche une suite de test qui ne rejette que des implémentations non conformes (correction) et qui puisse rejeter toute implémentation $I$ non conforme (exhaustivité) du modèle de faute.

L'exhaustivité implique qu'il faut tester chacune des transitions de $I$ en fonction de celles de $M$. Chaque test élémentaire correspondant à une transition $s \xrightarrow{i/o} s'$ de $M$ a la forme suivante :

- aller en $s$
- appliquer $i$ et vérifier la sortie $o$
- identifier l'état d'arrivée $s'$ par une séquence.

En plus de la correction et de l'exhaustivité de la suite de test, on cherche à optimiser la taille de celle-ci en enchaînant les tests élémentaires en une séquence de taille minimale.

**Séquences d'identification**

Il existe plusieurs méthodes basées sur le même principe et qui diffèrent par la possibilité d'identification de l'état d'arrivée. Cette identification s'effectue par une séquence d'entrées/sorties qui doit permettre d'identifier certainement l'état d'arrivée.

La manière la plus facile d'identifier l'état d'arrivée est de supposer l'existence d'une entrée $status$ à laquelle le système répond par son état courant. C'est l'hypothèse faite pour la méthode dite du tour de transition (TT) [63].

La deuxième méthode appelée DS *Distinguishing Sequence* [32, 36, 41] suppose qu'il existe une séquence d'entrées $DS$ capable de différencier toutes les paires d'états. Formellement, l'existence d'une séquence $DS$ est caractérisée par la formule :

$$\exists DS \in I^*, \forall s_i \in S, \forall s_j \in S, \omega^*(s_i, DS) \neq \omega^*(s_j, DS) \tag{2.1}$$

On peut noter que TT est un cas particulier de DS où la séquence distinguante est $status$.

La troisième méthode est appelée UIO (pour *Unique Input Output*) [54]. Elle suppose que pour chaque état $s_i$, il existe une séquence d'entrées $UIO_i$ qui différencie $s_i$ de tout autre état $s_j$. Formellement

$$\forall s_i \in S, \exists UIO_i \in I^*, \forall s_j \in S, \omega^*(s_i, UIO_i) \neq \omega^*(s_j, UIO_i) \tag{2.2}$$

Enfin la dernière méthode appelée W [11] suppose que toute paire d'états $s_i, s_j$ peut être différenciée par une séquence d'entrée $c_{i,j}$.

Formellement

$$\forall s_i \in S, \forall s_j \in S, \exists c_{ij} \in I^*, \omega^*(s_i, c_{ij}) \neq \omega^*(s_j, c_{ij}) \tag{2.3}$$

Notons que les caractérisations des séquences d'identification DS, $UIO_i$ et $c_{i,j}$ sont proches. Elles diffèrent seulement par la place de la quantification existentielle sur la séquence d'identification. Evidemment, ceci fait toute la différence.

En effet, il est clair que les séquences $c_{ij}$ existent toujours aussitôt que la machine est minimale. En effet, la formule (2.3) est précisément la définition de la minimalité pour les machines de Mealy.

En revanche l'existence des séquences $UIO_i$ pour chaque état $s_i$ n'est pas garantie, même pour des machines minimales. Cependant, il semble que les UIO existent assez souvent en pratique. De plus, bien que leur longueur soit au pire exponentielle dans la taille de la machine, elles sont souvent courtes en pratique.

Bien sûr, les séquences DS existent encore moins souvent que les UIO et en fait rarement. La technique de génération associée est donc en pratique inapplicable.

**Exemple** L'exemple de la machine de la figure 2.3, extrait de [55], permet de mieux se fixer les idées sur la différence entre les différentes séquences d'identification.

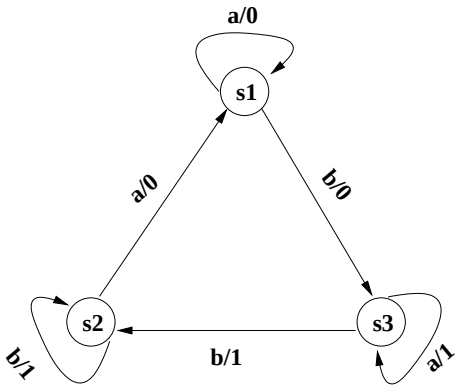



FIG. 2.3 – Illustration des séquences d'identification

$M$ est minimale comme le prouvent les séquences $c_{ij}$ suivantes : $c_{12} = c_{13} = b$ qui distinguent $s_1$ de $s_2$ et $s_3$ et $c_{23} = a$ qui distingue $s_2$ de $s_3$.

On en déduit que $s_1$ possède un UIO, $UIO(s_1) = b$, car $b$ distingue $s_1$ à la fois de $s_2$ et $s_3$. On a aussi $UIO(s_3) = a$ car $\omega(s_3, a) = 1$ et $\omega(s_1, a) = \omega(s_2, a) = 0$. Par contre, on peut facilement montrer que $s_2$ n'a pas d'UIO. En effet :

- $UIO(s_2)$ ne peut débuter par $a$ car $\omega(s_1, a) = \omega(s_2, a) = 0$ et $\delta(s_1, a) = \delta(s_2, a) = s_1$. Donc $s_2$ est indistinguable de $s_1$ après un $a$.

– $UIO(s_2)$ ne peut débuter par $b$ car $\omega(s_2, b) = \omega(s_3, b) = 1$ et $\delta(s_2, b) = \delta(s_3, b) = s_2$. Donc $s_2$ est indistinguable de $s_3$ après un $b$.

Il n'y a donc pas de solution.

Par conséquent cette machine n'a pas non plus de séquence distinguante (sinon celle-ci serait un UIO pour les 3 états).

### Synthèse de test

**Tour des transitions** Expliquons d'abord la méthode la plus simple, celle du tour de transition. Puisque le problème de l'identification est réglé par l'entrée $status$, le test élémentaire d'une transition consiste en l'application de son entrée. Une suite de test correcte et exhaustive consiste donc à parcourir toutes les transitions. Optimiser la suite de tests consiste alors à trouver une séquence de transitions de longueur minimale et passant au moins une fois par chaque transition. C'est un problème classique de type voyageur de commerce appelé Problème du Postier Chinois (voir [31] par exemple). Après transformation de la machine, ce problème se réduit à celui de la recherche d'un circuit Eulérien dans un graphe : un circuit est Eulérien s'il passe une et une seule fois par chaque transition. On utilise le théorème suivant :

**Théorème 2.3.1** *Un graphe dirigé possède un circuit eulérien si et seulement si il est fortement connexe et symétrique.*

Un graphe est symétrique si chaque nœud a un degré entrant (nombre d'arcs entrants) égal à son degré sortant (nombre d'arcs sortants).

Sous les conditions de forte connexité et de symétrie, le calcul d'un circuit Eulérien se fait en temps linéaire par parcours et suppression des transitions empruntées jusqu'à épuisement des possibilités et recollement des circuits intermédiaires construits.

La machine de Mealy de départ est supposée fortement connexe, mais pas nécéssairement symétrique. En général, elle n'a donc pas de circuit Eulérien. Par contre on peut la rendre symétrique en dupliquant certaines transitions.

De plus on a le théorème :

**Théorème 2.3.2** *Un circuit Eulérien d'une augmentation symétrique minimale (obtenue en dupliquant le moins possible de transitions) est un circuit de longueur minimale empruntant chaque transition au moins une fois.*

Le problème se ramène donc au calcul d'une augmentation symétrique minimale de la machine de départ. Il peut se résoudre par programmation linéaire de la façon suivante. On associe une variable $X_{ij}$ à chaque transition de $s_i$ vers $s_j$ représentant le nombre de duplications de cette transition dans l'augmentation symétrique. Pour chaque état $s_i$ on pose alors l'équation $\Sigma_{s_j \to s_i}(X_{ji} + 1) = \Sigma_{s_i \to s_k}(X_{ik} + 1)$ qui caractérise la symétrie en l'état $s_i$. Il s'agit alors de résoudre ce système d'équation en optimisant $\Sigma_{i,j} X_{ij}$.

**Méthode UIO** La méthode UIO est un peu plus complexe et se réduit aussi au calcul de circuit Eulérien. La première étape consiste à calculer l'UIO $UIO_j$ de chaque état $s_j$. Ensuite de calculer la séquence de test en utilisant ces UIO. Ici le test élémentaire de la transition $s_i \xrightarrow{i/o} s_j$ est composé de cette transition suivie de la séquence d'identification

de l'état d'arrivée $s_j$, $UIO_j$. Le problème est de trouver la séquence de longueur minimale contenant au moins une fois chaque test élémentaire. Tout d'abord pour chaque test élémentaire on ajoute au graphe une transition rouge de l'état $s_i$ jusqu'à l'état $s_k$, où $s_k$ est l'état d'arrivée après application de $UIO_j$ dans $s_j$. Il s'agit alors de trouver un circuit de taille minimale passant une fois par chacune de ces transitions rouges et empruntant si nécéssaire les transitions originales (noires). Ceci est à nouveau un problème connu de type voyageur de commerce appelé Problème du Postier Chinois Rural. On se ramène au problème de calcul de circuit Eulérien en construisant un graphe symétrique à partir des transitions rouges et en ajoutant (et dupliquant) le minimum de transitions noires. Le problème d'augmentation "rurale" symétrique est plus complexe mais est à nouveau résolu par programmation linéaire ou calcul de flot. Cependant, pour que la méthode de test soit exhaustive (rejette bien toute implémentation non conforme) il est nécéssaire de vérifier que les UIO calculés sur la spécification sont aussi des UIO de l'implémentation, comme le montre l'exemple de la figure 2.4. La séquence $yx$ est un UIO pour l'état 3 de $M1$, alors qu'elle ne l'est plus pour l'état 3 de l'implémeatation fautive $M2$.

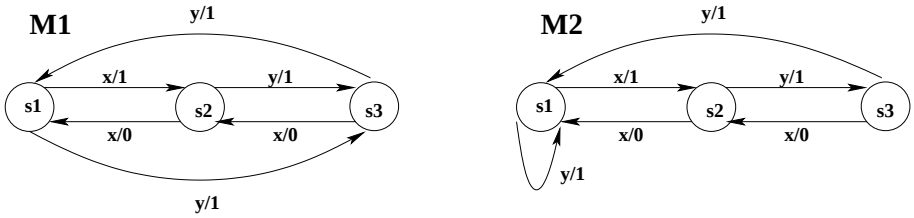



FIG. 2.4 – Spécification $M1$ et implémentation $M2$ n'ayant pas les mêmes UIO

**Avantages et inconvénients**

Le premier avantage de ce type de méthode est l'exhaustivité. Cependant les algorithmes sont relativement coûteux, ce qui cantonne leur applicabilité à des graphes d'états de taille très modeste (de l'ordre de la centaine d'états et de transitions). On voit parfois dans la littérature des applications de ces techniques sur de grosses spécification. En réalité elles ne s'appliquent qu'à un sous-graphe du graphe d'état obtenu en bornant la profondeur d'exploration lors de la construction de ce graphe. L'exhaustivité est donc souvent un leurre.

L'autre avantage est celui de l'identification des états d'arrivée. Cet aspect est intéressant car on constate que les suites de test manuelles possèdent elles aussi très souvent des séquences d'identification. Cependant, ces dernières ne cherchent pas à identifier un état global particulier mais seulement un état de contrôle particulier du système. Il y a donc un léger décalage entre ces méthodes et la pratique. Ce problème est cependant abordé dans les travaux de [65] où les auteurs cherchent à distinguer des ensembles d'états.

L'inconvénient le plus important est celui des hypothèses nécéssaires à l'application de ces méthodes portant d'une part sur la spécification mais surtout sur l'implémentation. On peut concevoir que les spécifications et implémentations soient déterministes quand elles se rapportent à une entité de protocole. C'est beaucoup moins réaliste si l'on considère un

système composé, dans le cas où la communication est asynchrone entre sous-systèmes. On est alors confronté à des situations où plusieurs sorties sont possibles après la même entrée ($\omega$ ne peut plus être une fonction). L'hypothèse sur l'égalité des nombres d'états est encore moins réaliste et impossible à vérifier. Il existe cependant des travaux sur les machines non déterministes et des travaux où le nombre d'états de l'implémentation est borné par celui de la spécification plus une constante.

Un autre inconvénient est dû au modèle des machines de Mealy qui n'est pas le modèle idéal pour représenter la sémantique des langages de spécifications comme SDL. En effet en SDL une transition est souvent constituée d'une entrée suivie de plusieurs sorties ou uniquement de sorties alors que pour les machines de Mealy, une transition comprend une entrée et une sortie.

Enfin, en pratique les tests manuels sont structurés en fonction d'objectifs précis. Ici, un test élémentaire a pour objectif une transition de la machine. D'autre part, l'optimisation de la suite de tests produit un test monolithique (une seule longue séquence), ce qui n'est souvent pas le but cherché en pratique.

Pour toutes ces raisons, nous avons choisi un autre type de technique et de théorie sous-jacente qui nous semble plus appropriée à la génération de test.

### 2.3.2 Méthodes fondées sur les systèmes de transition

Commençons par une remarque. Bien sûr les systèmes de transitions sont ni plus des moins des automates. Il peut donc paraître étrange de distinguer les méthodes basées sur les automates des méthodes fondés sur les systèmes de transitions. La distinction est plus historique d'une part, et dans les restrictions faites sur les modèles. Les méthodes que nous allons voir ici sont directement pensées avec des modèles non-déterministes.

Nous présentons ici très succintement l'historique des théories du test basées sur les systèmes de transition. L'article [7] en présente une bibliographie annotée.

Le modèle des systèmes de transitions (LTS pour *Labelled Tranistion Systems*) [2] est un modèle très simple utilisé depuis les années 80 pour représenter la sémantique opérationelle des processus. Un système de transition est simplement un automate étiqueté par des actions, et n'ayant pas d'état final.

**Définition 2.3.2 (LTS : Système de transition)**
*Un LTS est un quadruplet $M = (Q^M, A^M, \rightarrow_M, q_0^M)$ tel que :*
- *$Q^M$ est un ensemble dénombrable non vide d'états*
- *$q_0^M \in Q^M$ est l'état initial*
- *$A^M$ est l'alphabet des actions observables auquel on adjoint une action $\tau$ représentant une action interne quelconque.*
- *$\rightarrow_M \subseteq Q^M \times (A^M \cup \{\tau\}) \times Q^M$ est la relation de transition*

Nous utilisons $_M$ comme exposant (ou indice) pour identifier les éléments du LTS $M$ car nous aurons souvent besoin de parler de plusieurs LTS à la fois. Ceci sera omis quand il n'y a pas d'ambiguité.

A partir des LTS, on peut définir différentes notions que nous détaillerons plus loin, comme celle de trace (i.e. la projection des séquences d'actions sur les actions observables) et de composition parallèle modélisant l'interaction entre processus (par synchronisation des actions communes).

Les méthodes de génération de tests fondées sur les systèmes de transition ont leur origine dans les travaux sur les équivalences comportementales et préordres, qui cherchent à définir des relations entre LTS suivant les observations que l'ont peut en faire. En particulier le préordre de test $\leq_{te}$ [17, 1] considère les observations qu'un testeur peut effectuer. Grossièrement, on a $I \leq_{te} S$ si pour tout autre LTS observateur $T$ (le testeur), les observations (traces et blocages) que l'on peut faire par composition parallèle de $T$ et $I$ sont incluses dans les observations possibles de $T$ sur $S$.

Plus constructif, Brinksma [8] introduit ensuite la *relation de conformité* **conf**, qui limite l'observation aux traces de la spécification plutôt que de considérer toutes les observations possibles. Il en résulte la notion de *testeur canonique* pour **conf**. Un testeur canonique d'une spécification $S$ est un LTS $T$ qui permet de tester exhaustivement la relation **conf** : une implémentation passe le test si et seulement si elle est conforme à la spécification $S$.

Une relation plus forte que le préordre de test est le *préordre de refus* [69, 53] qui permet au testeur, par l'introduction d'une nouvelle action $\Theta$ représentant l'absence d'action et l'adaptation composition parallèle.

Tous ces travaux, bien que fondateurs, sont restés assez théoriques, car destinés à raisonner sur le test plutôt que de construire réellement des tests. La raison principale est que le modèle de systèmes de transitions avec interaction symétrique n'est pas directement utilisable pour le test : il n'est fait aucune différence entre événements observés (sorties du système sous test) et événements contrôlés (entrées du système). La différence est pourtant fondamentale dans la pratique du test puisque le testeur choisit une entrée et stimule l'IUT par cette entrée et observe, de façon passive, les sorties de l'implémentation en réaction au stimulus.

Cette différence entre entrées et sorties nécessite un modèle plus détaillé. Plusieurs de ces modèles ont été utilisés pour le test dont les Input Output Automata (IOA) de Lynch [59] utilisés par Segala [74], les Input Output State Machines (IOSM) de Phalippou [67], et les Input Output Transition Systems (IOTS) de Tretmans [76]. 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 assez similaires entre eux.

Dans la suite nous nous intéresserons principalement aux travaux de Tretmans sur les IOTS car nous avons pris un modèle similaire au sien, que nous avons appelé Input Output Labelled Transition System (IOLTS). Les systèmes de transitions à entrée/sortie (IOLTS) sont simplement des LTS où, en plus des action sinternes, on distingue deux types d'actions visibles : les entrées et les sorties. Nous nous sommes beaucoup inspiré des travaux de Tretmans et ses collègues de l'Université de Twente au moins sur le plan des modèles et de la théorie du test. Notons simplement que Tretmans considère que la spécification peut être décrite par un LTS alors que nous considérons la spécification aussi comme un IOLTS, mais ceci n'est pas fondamental.

### 2.3.3 Techniques algébriques

Une autre famille de méthodes se fonde sur les spécifications algébriques pour la description de types de données abstraits [4]. Un type de données abstrait est décrits par une signature $\Sigma = (S, F)$ composée d'un ensemble fini $S$ de sortes (des types), et un ensemble

fini $F$ d'opérations sur ces sortes, et un ensemble fini $Ax$ d'axiomes qui sont des équations ou des équations conditionelles construites sur les termes de la signature. Prenons un exemple simplifié de type de donnée chaîne de caractère pour illustrer l'approche :

$\mathbf{type}\ CharList\ \mathbf{is}$
$Char$
**sorts**
$CharList$
**operations**

$$\begin{array}{lrl} empty\ : & \rightarrow CharList & /*\,generator\,*/ \\ cons\ : & Char \times CharList \rightarrow CharList & /*\,generator\,*/ \\ size\ : & CharList \rightarrow Nat & \end{array}$$

**Axiomes**
$size(empty) = 0$
$size(cons(\alpha, L)) = size(L) + 1$

Les sortes sont composées des chaînes de caractère auxquelles ont peut appliquer l'opération *empty* qui construit la chaîne vide, l'opération *cons* qui construit une chaîne de caractère en concaténant un caractère à une chaîne, et l'opération *size* qui détermine la taille de la chaîne. On ne donne ici que des axiomes sur l'opération *size* qui spécifient que la taille d'une chaîne vide est nulle et que la taille d'une chaîne obtenue par concaténation d'une caractère est la taille de la chaîne initiale plus 1.

On imagine que ce type de donnée avec ses axiomes constituent la spécification $Spec$ d'un programme. Tester la conformité d'une implémentation $I$ de $Spec$ consiste à appliquer les opérations et vérifier que les axiomes $Ax$ sont satisfaits par $I$. Autrement dit l'interface du programme est constitué de la signature et des termes que l'on peut construire sur cette signature, et la relation de conformité est la satisfaction des axiomes.

Générer des tests pour tester la conformité à une spécification consiste donc à tester les axiomes en construisant des équations de termes sur la signature. Un test pour le second axiome $size(cons(\alpha, L)) = size(L) + 1$ consiste par exemple à comparer $size_I(cons(A, cons(B, cons(C, empty))))$ et $size_I(cons(B, cons(C, empty))) + 1$ où $size_I$ est l'implémentation dans $I$ de l'opération correspondante $size$ dans $Spec$. Noter que cela implique d'exécuter deux fois l'implémentation avec les termes $cons(A, cons(B, cons(C, empty)))$ et $cons(B, cons(C, empty))$, puis de comparer les résultats de $size_I$.

A partir d'une spécification $Spec$ on peut alors définir l'ensemble de test exhaustif $Exhaust(Spec)$ qui est constitué de l'ensemble des instanciations des axiomes. Moyennant une hypothèse minimale $H_{min}$ de testabilité sur $I$ (incluant le déterminisme) la correction de $I$ est équivalente au succcès de l'ensemble de tests $Exhaust(P)$. Cependant $Exhaust(P)$ est en général infini et ne pourra pas être construit. L'idée est alors d'introduire successivement des hypothèses de test $H$ permettant de réduire l'ensemble des tests à un ensemble fini et de taille raisonnable et qui ait le même pouvoir de test que $Exhaust(P)$. Ceci permet en particulier de formaliser des hypothèses communément utilisées comme l'uniformité et la régularité. L'hypothèse d'uniformité consiste à séparer le domaine d'entrée (l'ensemble des termes) de sorte que les comportements sont équivalents sur ces sous-domaine. Plus précisément, si la suite de test contient un élément par

sous-domaine, passer cette suite de test est équivalent à passer la suite $Exhaust(P)$. L'hypothèse de régularité consiste à choisir une borne sur la taille des termes et à restreindre la suite de tests à des termes de taille inférieure à cette borne, en supposant que l'implémentation passe la suite réduite si et seulement si elle pass $Exhaust(P)$.

Cette théorie du test peut paraître éloignée des précédentes. Cependant, elle se combine avec les théories fondées sur les systèmes de transition. En effet, toute la théorie sur les hypothèses de tests à été interprétée sur des modèles de systèmes de transitions par Phalippou [67]. L'hypothèse de régularité en particulier est similaire à l'hypothèse faite sur les machines de Mealy qui suppose que la taille de l'implémentation n'excède pas la taille de la spécification de plus d'une certaine borne. De nouvelles hypothèses ayant plus de sens dans ce contexte, ont par ailleurs été introduites. C'est le cas par exemple de l'hypothèse d'équité bornée, qui n'a de sens que dans des systèmes non-déterministes, et qui dit que dans tout état, l'implémentation montre ses différents choix si on lui en donne l'occasion un nombre borné de fois.

D'autre part, dans [28], les approches sont combinées pour la génération de tests pour des spécifications Lotos. Cet article se base sur la relation de conformité **conf** sur les systèmes de transition. Les auteurs proposent de combiner l'approche par simulation symbolique de Lotos proposée par Eertink [23] avec l'approche algébrique pour traiter la partie donnée. Nous reviendrons sur cette approche en fin de chapitre 4.

# Chapitre 3

# Modèles, théorie et principes de la génération de tests à la volée

Nous présentons dans ce chapitre les modèles, la théorie et les principes de la génération automatique de tests tels qu'implémentés dans notre outil TGV (*Test Generation with Verification technology*). TGV a été développé conjointement par l'Irisa Rennes et Verimag Grenoble et avec le support du projet Vasy de l'Inria Rhônes-Alpes. J'en suis cependant le principal contributeur, ayant été le principal concepteur des modèles et de l'adaptation de la théorie de Tretmans, mais surtout le principal concepteur de tous les algorithmes nécessaires à la génération, ainsi que l'architecte logiciel du projet.

La théorie du test sous-jacente à TGV est fondée sur un modèle de systèmes de transitions distinguant entrées, sorties et actions internes, le concept de relation de conformité et les notions d'exécution et de verdict. Les principes et algorithmes de la synthèse de tests, fondée sur la sélection à la volée de cas de test en fonction d'objectifs de test, seront ensuite expliqués. Nous décrirons les principales caractéristiques de l'outil TGV et des langages de spécification possibles et survolerons quelques études de cas. En conclusion, nous proposons quelques unes de nos perspectives de recherche.

**Plan du chapitre** Ce chapitre a pour objectif de présenter et éclairer les éléments principaux de la synthèse de tests dans l'outil TGV (acronyme pour *Test Generation with Verification technology*). On s'est bien sûr basé sur un certain nombre de publications. Mais certaines de ces publications ne sont plus très actuelles. Il nous a donc paru important, à un moment où TGV est maintenant diffusé et reconnu internationalement, et que notre recherche s'engage aussi sur d'autres voies, de regrouper ces différents éléments en un seul lieu en faisant le point sur l'état actuel de l'outil. On a profité de l'exercice pour présenter aussi la théorie du test et les résultats d'une manière différente.

Le chapitre est structuré comme suit. La section 3.1 présente la théorie du test sous-jacente à TGV. La section 3.2 présente ensuite les algorithmes de synthèse. L'outil TGV est décrit à la section 3.4. Quelques comparaisons avec d'autres techniques et outils voisins constituent la section 3.5. Enfin, nous concluons sur ce travail dans la section 3.6.

## 3.1 Une théorie du test de conformité

Notre apport à la synthèse de tests se situe principalement sur le plan de l'algorithmique et des outils. Cependant, pour que ces algorithmes et outils aient des fondements solides, nous nous basons sur une théorie du test de conformité développée par le groupe de Jan Tretmans (Université de Twente) [76, 77], avec de petites variations aux conséquences minimes. Notre vision étant plus opérationnelle qu'algébrique, la présentation de cette théorie s'en ressent. D'autres travaux nous ont aussi inspiré, comme ceux de Marc Phalippou [67].

Comme vu précédemment, cette théorie est héritière de travaux précédents du domaine des algèbres de processus sur les équivalences et pré-ordres de test [17, 1], la notion de testeur canonique [8]. La théorie est basée sur la modélisation des comportements des spécifications et des IUT par un raffinement du modèle de systèmes de transitions étiquetés (LTS pour *Labelled Transition System*). La relation de conformité est essentiellement une inclusion partielle de traces et blocages. Nous présentons maintenant cette théorie avec le soucis d'en éclairer les points principaux.

### 3.1.1 Modèles de la conformité

Tel quel, le modèle des LTS n'est pas adapté à la modélisation de systèmes réactifs pour le test. En effet, puisqu'il n'y est fait aucune différence entre entrée et sortie, la composition parallèle de LTS qui modélise en particulier l'interaction entre testeur et IUT, suppose une interaction symétrique entre actions communes. Or dans le cadre du test, la distinction entre entrées et sorties est fondamentale : il faut distinguer les événements du système *contrôlables* par l'environnement (ses entrées) des événements seulement *observables* (ses sorties). En effet, le testeur choisit des entrées du système afin de le stimuler et de le contrôler. Puis il observe, de façon passive, les sorties de l'implémentation en réaction aux stimuli. Et ainsi de suite. On utilise ici le modèle des IOLTS (*Input Output Labelled Transition System*) et quelques variantes, pour modéliser les comportements des objets intervenant dans la génération de tests :

**Définition 3.1.1 (IOLTS : système de transition à entrées/sorties)**
*Un IOLTS est un quadruplet* $M = (Q^M, A^M, \rightarrow_M, q_0^M)$ *où*
- $Q^M$ *est un ensemble dénombrable non vide d'états,*
- $q_0^M \in Q^M$ *est l'état initial.*
- $A^M$ *est un alphabet dénombrable d'actions partitionné en trois ensembles disjoints* $A^M = A_I^M \cup A_O^M \cup I^M$ *où*
  - $A_I^M$ *est l'alphabet d'entrées,*
  - $A_O^M$ *est l'alphabet de sorties,*
  - $I^M$ *est l'alphabet des actions internes,*
- $\rightarrow_M \subseteq Q^M \times A^M \times Q^M$ *est la relation de transition.*

Pour des raisons techniques, on se limitera aux IOLTS dits *non divergents* i.e. n'ayant pas de séquences infinie de transitions étiquetées par une action interne et traversant un ensemble infini d'états. On autorise par contre les boucles finies d'actions internes.

Le modèle d'IOLTS n'est qu'un raffinement du modèle LTS où on distingue entrées, sorties et actions internes. Il est proche du modèle des Input Output Transition Systems

(IOTS) de Tretmans [76]. La différence entre IOLTS et IOTS est minime : un IOTS est supposé complet en entrée (voir plus loin) c'est à dire qu'il ne peut refuser aucune entrée (après d'éventuelles actions internes). Tretmans utilise les IOTS comme modèle d'implémentation alors qu'il utilise le modèle des LTS comme modèle de spécification. Nous préférons utiliser le même modèle pour les spécifications et implémentations, et ne pas supposer ce modèle complet en entrée *a priori*. D'autre part dans les IOTS et LTS, l'action interne $\tau$ est indifférenciée et ne fait pas partie de l'alphabet d'actions, alors que dans les IOLTS, les actions internes font partie de l'alphabet d'actions et nous les différencions, dans le but d'affiner la sélection des tests.

Dans un IOLTS, on dira qu'une action est *visible* si c'est une entrée (contrôlable par son environnement) ou une sortie (observable). On notera

$$A^{\mathrm{M}}_{\mathrm{VIS}} = A^{\mathrm{M}}_{\mathrm{I}} \cup A^{\mathrm{M}}_{\mathrm{O}}$$

l'alphabet des actions visibles.

Pour ne pas surcharger les exemples, on ne précisera pas les alphabets d'entrée et de sortie, et on notera $?a$ pour une entrée $a \in A^{\mathrm{M}}_{\mathrm{I}}$ et $!x$ pour une sortie $x \in A^{\mathrm{M}}_{\mathrm{O}}$, et les actions internes seront simplement préfixées par $\tau$.

**Notations :** nous introduisons maintenant quelques notations pour les IOLTS, dont la plupart sont des notations standard pour les LTS. Ces notations sont illustrées par l'exemple de la figure 3.1.

Soit $M = (Q^{\mathrm{M}}, A^{\mathrm{M}}, \rightarrow_{\mathrm{M}}, q_0^{\mathrm{M}})$ un IOLTS. L'exposant (ou l'indice) $_{\mathrm{M}}$ est omis quand il n'y a pas d'ambiguïté. Soit $\mu_{(i)} \in A^{\mathrm{M}}$ des actions, $a_{(i)} \in A^{\mathrm{M}}_{\mathrm{VIS}}$ des actions visibles, $\tau_{(i)} \in I^{\mathrm{M}}$ des actions internes $\sigma \in {A^{\mathrm{M}}_{\mathrm{VIS}}}^*$ une séquence d'actions visibles, $q, q', q_{(i)} \in Q^{\mathrm{M}}$ des états.

- On note $q \xrightarrow{\mu}_{\mathrm{M}} q'$ pour $(q, \mu, q') \in \rightarrow_{\mathrm{M}}$. De plus $q \xrightarrow{a}_{\mathrm{M}}$ est parfois utilisé à la place de $\exists q' \ : q \xrightarrow{a}_{\mathrm{M}} q'$.
- On note $q \xrightarrow{\mu_1 \cdots \mu_n}_{\mathrm{M}} q' \triangleq \exists q_0, \ldots, q_n \ : q = q_0 \xrightarrow{\mu_1} q_1 \xrightarrow{\mu_2} \cdots \xrightarrow{\mu_n} q_n = q'$.

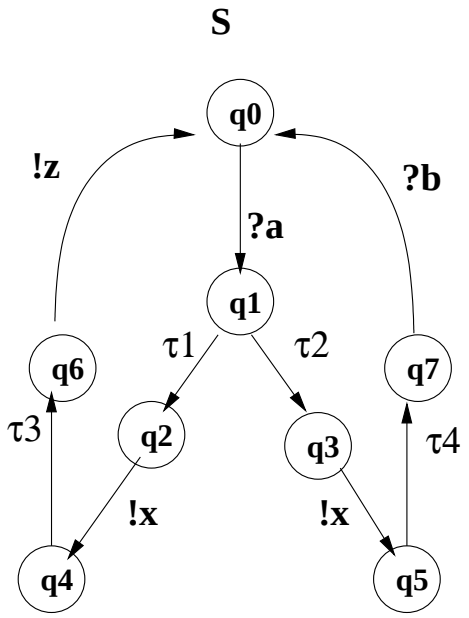
On identifiera parfois un $IOLTS$ à son état initial en notant $M \xrightarrow{\mu}_{\mathrm{M}}$ pour $q_0^{\mathrm{M}} \xrightarrow{\mu}_{\mathrm{M}}$.

On généralise les opérations classiques $pre$ et $post$ et leurs points fixes de la manière suivante. Pour un ensemble d'états $P \subseteq Q^{\mathrm{M}}$ et pour un sous-ensemble $B \subseteq A^{\mathrm{M}}$ de l'alphabet, on note

- $post_B(P) \triangleq \{q \in Q^{\mathrm{M}} \mid \exists b \in B, \exists q' \in P, q' \xrightarrow{b}_{\mathrm{M}} q\}$
  c'est l'ensemble des états de $Q^{\mathrm{M}}$ successeurs d'états de $P$ par une action dans $B$.
- $pre_B(P) \triangleq \{q \in Q^{\mathrm{M}} \mid \exists b \in B, \exists q' \in P, q \xrightarrow{b}_{\mathrm{M}} q'\}$
  c'est l'ensemble des états de $Q^{\mathrm{M}}$ ayant un successeur par $B$ dans $P$

On peut alors définir les opérations classiques suivantes :

- $reach_B(P) \triangleq \mu X.P \cup post_B(X)$
  c'est le plus petit point fixe de la fonction $\lambda X.P \cup post_B(X)$ de $2^{Q^{\mathrm{M}}}$ dans $2^{Q^{\mathrm{M}}}$.
  On a $reach_B(P) = P \cup \bigcup_{i \geq 1} post^i_B(P)$.
  C'est donc l'ensemble des états *accessibles* depuis $P$ par des séquences dans $B^*$, et si $Q^{\mathrm{M}}$ est fini, on peut le calculer de manière effective.
- $coreach_B(P) \triangleq \mu X.P \cup pre_B(X)$
  c'est le plus petit point fixe de la fonction $\lambda X.P \cup pre_B(X)$ de $2^{Q^{\mathrm{M}}}$ dans $2^{Q^{\mathrm{M}}}$.
  On a $coreach_B(P) = P \cup \bigcup_{i \geq 1} pre^i_B(P)$.




$\Gamma_s(q_0) = \{?a\}$,
$Out_s(q_0) = \emptyset$ et
$In_s(q_0) = \{?a\}$
$q_1 \overset{\varepsilon}{\Rightarrow} q_2$ et $q_1 \overset{\varepsilon}{\Rightarrow} q_3$ et
$q_1 \overset{!x}{\Rightarrow} q_4$, $q_1 \overset{!x}{\Rightarrow} q_6$, $q_1 \overset{!x}{\Rightarrow} q_5$, $q_1 \overset{!x}{\Rightarrow} q_7$.
$q_1 \overset{!x.!z}{\Rightarrow} q_0$.
$Traces(S) = Traces(q_0) = \{\varepsilon, ?a, ?a.!x, ?a.!x.!z, ?a.!x.?b, \ldots\}$.
$q_1$ after $\varepsilon = \{q_2, q_3\}$,
$q_0$ after $?a.!x = \{q_4, q_5, q_6, q_7\}$, $q_0$ after $?a.!z = \emptyset$ et
$\{q_2, q_3\}$ after $!x = \{q_4, q_5, q_6, q_7\}$.

FIG. 3.1 – Illustration des notations sur les IOLTS

C'est l'ensemble des états *co-accessibles* de $P$ par des séquences dans $B^*$ et si $Q^{\text{M}}$ est fini, on peut le calculer de manière effective.

On peut aussi généraliser $post$ et $reach$ (on n'en aura pas besoin pour $pre$ et $coreach$) en filtrant les états par un ensemble $R \subseteq Q^{\text{M}}$

– Notons $post_{B|R}(X) \triangleq post_B(X \cap R)$ et définissons
$reach_{B|R}(P) \triangleq \mu X.P \cup post_{B|R}(X) = \mu X.P \cup post_B(X \cap R)$

Si $Q^{\text{M}}$ est fini, $reach_{B|R}(P) = P \cup \bigcup_{i \geq 1} post^i_{B|R}(P)$.
C'est l'ensemble des états accessibles depuis $P$ par des actions dans $B$, mais en ne passant que par des états de $R$. Comme on n'a pas forcément $post_B(P' \cap R) \subseteq R$, à chaque itération du $post$, on garde les états hors de $R$, mais on s'y arrête car l'itération suivante les filtre. Noter qu'on ne filtre pas $P$ par $R$ initialement.

Pour $pre$, $post$, $reach$ et $coreach$, on omettra donc l'indice $B$ si $B = A^{\text{M}}$ et/ou $R$ si $R = Q^{\text{M}}$, et on notera par exemple $reach_B(P)$, $reach_{|R}(P)$, $reach(P)$. Dans le cas où $B = A^{\text{M}}$ et $R = Q^{\text{M}}$, on retrouve d'ailleurs les définitions habituelles de $post$, $reach$, $pre$ et $coreach$. Ces opérations $post$, $pre$ et les points fixes $reach$ et $coreach$ sont couramment utilisées en model-checking.

– $\Gamma_{\text{M}}(q) \triangleq \{\mu \in A^{\text{M}} \mid q \xrightarrow{\mu}_{\text{M}}\}$ dénote l'ensemble des actions tirables en $q$, parmi lesquelles :
– $Out_{\text{M}}(q) \triangleq \Gamma(q) \cap A^{\text{M}}_{\text{O}}$ est le sous-ensemble des sorties et
– $In_{\text{M}}(q) \triangleq \Gamma(q) \cap A^{\text{M}}_{\text{I}}$ le sous-ensemble des entrées.

On étend ces notions aux ensembles d'états. Pour $P \subseteq Q^{\text{M}}$,
– $\Gamma_{\text{M}}(P) \triangleq \cup_{q \in P} \Gamma_{\text{M}}(q)$,
– $Out_{\text{M}}(P) \triangleq \cup_{q \in P} Out_{\text{M}}(q) = \Gamma_{\text{M}}(P) \cap A^{\text{M}}_{\text{O}}$ et
– $In_{\text{M}}(P) \triangleq \cup_{q \in P} In_{\text{M}}(q) = \Gamma_{\text{M}}(P) \cap A^{\text{M}}_{\text{I}}$.

Pour parler des comportements visibles on définit :
– $q \overset{\varepsilon}{\Rightarrow} q' \triangleq q = q' \vee q \xrightarrow{\tau_1 \cdot \tau_2 \cdots \tau_n} q'$ et
– $q \overset{a}{\Rightarrow} q' \triangleq \exists q_1, q_2 \ : q \overset{\varepsilon}{\Rightarrow} q_1 \xrightarrow{a} q_2 \overset{\varepsilon}{\Rightarrow} q'$ où $\varepsilon$ dénote la séquence vide de ${A^{\text{M}}_{\text{VIS}}}^*$.

Les notations précédentes sont étendues aux séquences d'actions visibles ou *traces* par :
– $q \overset{a_1 \cdots a_n}{\Rightarrow} q' \triangleq \exists q_0, \ldots, q_n \ : q = q_0 \overset{a_1}{\Rightarrow} q_1 \cdots \overset{a_n}{\Rightarrow} q_n = q'$ et
– $q \overset{\sigma}{\Rightarrow} \triangleq \exists \ q' \ : q \overset{\sigma}{\Rightarrow} q'$.

L'ensemble des traces possibles à partir d'un état $q$ (resp. de l'état initial d'un IOLTS $M$) sont décrites par
– $Traces(q) \triangleq \{\sigma \in {A^{\text{M}}_{\text{VIS}}}^* \mid q \overset{\sigma}{\Rightarrow}\}$.
– $Traces(M) \triangleq Traces(q^{\text{M}}_0)$ pour un systèmes de transition $M$ d'état initial $q^{\text{M}}_0$.

Enfin, on aura besoin de parler de l'ensemble des états accessibles depuis un état (ou un ensemble d'états) après une trace :
– $q$ after $\sigma \triangleq \{q' \in Q \mid q \overset{\sigma}{\Rightarrow} q'\}$
– $P$ after $\sigma \triangleq \bigcup_{q \in P} q$ after $\sigma$, pour un ensemble d'états $P$.
– pour un systèmes de transition $M$ d'état initial $q^{\text{M}}_0$, on note $S$ after $\sigma$ pour $q^{\text{M}}_0$ after $\sigma$

**Non déterminisme :** Notons que les IOLTS sont *a priori* non déterministes, au sens habituel du terme pour les automates (en interprétant les actions internes comme des $\varepsilon$-transitions). Nous précisons la définition d'un IOLTS déterministe, utile par la suite.

**Définition 3.1.2 (IOLTS déterministe)**
*Un IOLTS* $M = (Q^M, A^M, \rightarrow_M, q_0^M)$ *est déterministe s'il n'a aucune transition étiquetée par une action interne et dans chaque état* $q$*, pour chaque action* $a$*, au plus une transition issue de* $q$ *est étiquetée* $a$*.*
*Formellement,* $\forall q \in Q^M, \forall \sigma \in {A_{VIS}^M}^*$, $Card(q \text{ after } \sigma) \leq 1$.

La figure 3.2 illustre les configurations interdites dans un IOLTS déterministe.

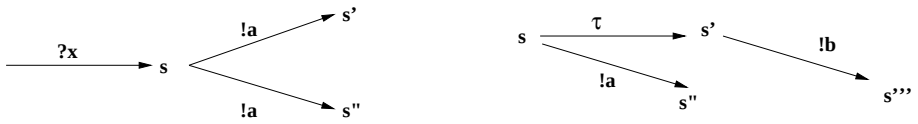



FIG. 3.2 – Non déterminisme au sens des automates

On trouve aussi parfois dans la littérature sur le test le terme "non déterminisme" d'un systèmes ou d'une spécification, utilisé à notre avis à tort ou au moins ambigüe, pour parler de la situation de la figure 3.3.

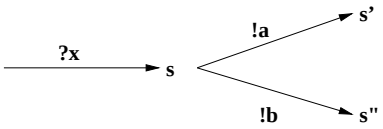



FIG. 3.3 – Non déterminisme observable

.

Cette situation représente un choix non contrôlé par l'environnement et nous l'appelerons non déterminisme observable. Plus généralement, c'est la situation dans laquelle, à partir d'un état, il existe plus d'une sortie possible après un nombre quelconque d'actions internes.

**Définition 3.1.3 (non-déterminisme observable)**
*Un IOLTS* $M = (Q^M, A^M, \rightarrow_M, q_0^M)$ *n'a pas de* non-déterminisme observable *si dans tout état* $q$*,* $Card(Out(q \text{ after } \epsilon)) \leq 1$.

**Complétude en entrée :** Pour certains IOLTS, en particulier les IUT, nous auront besoin de supposer qu'une entrée de l'IUT est toujours possible.

**Définition 3.1.4 (Complétude en entrée)**
*Un IOLTS* $M$ *est* complet en entrée *sur l'alphabet d'entrée* $A_I$ *si dans chaque état elle ne*

*peut refuser aucune de ces entrées après d'éventuelles actions internes i.e.*

$$\forall q \in Q^{M}, \forall a \in A_I, q \overset{a}{\Rightarrow}$$

**Proposition 3.1.1** *Si $M$ est complet en entrée, on a*

$$Traces(M).A_I \subseteq Traces(M)$$

La complétude en entrée est celle présentée par Tretmans [77]) et nommée complétude faible en entrée (*weak input enabling*). Elle est moins forte que la complétude forte en entrée (*strong input enabling*) de Lynch [59] définie par : $\forall q \in Q^{\text{M}}, \forall a \in A_{\text{I}}^{\text{M}}, q \xrightarrow{a}$. Nous exigerons plus loin la complétude en entrée faible pour les implémentations, afin d'assurer que celles-ci acceptent toutes les entrées fournies par un cas de test.

**Modèle de spécification :** un système réactif est généralement décrit dans un langage de description spécialisé (par exemple SDL, Lotos, UML, IF pour ce qui concerne TGV, voir la section 3.4.2). La sémantique opérationnelle de ces langages définit les comportements possibles de la spécification. Dans bien des cas cette sémantique peut être décrite par un LTS où les états représentent les configurations possibles (états de contrôle, valeur des variables, contenu des files, etc) et les actions sont les transitions atomiques. Les simulateurs de tels langages implémentent cette sémantique. Par exemple un simulateur interactif calcule l'état initial et propose l'ensemble des transitions tirables. Le choix d'une transition résulte en un nouvel état, etc. Les transitions atomiques peuvent être complexes et contenir plusieurs actions (par exemple une entrée suivie de sorties et d'actions internes). L'interprétation de la sémantique LTS en IOLTS nécessite parfois un découpage des transitions et d'identifier les entrées, sorties et actions internes. Pour un système réactif, ceci est généralement aisé. Par exemple, Lotos ne différencie pas les entrées et les sorties, mais parle de synchronisation sur des portes. Mais la modélisation en Lotos d'un système réactif fait en général apparaître clairement les entrées et sorties du système par rapport à son environnement.

Nous supposerons donc que les comportements de la *spécification* sont modélisés par un IOLTS $S = (Q^{\text{s}}, A^{\text{s}}, \rightarrow_{\text{s}}, q_0^{\text{s}})$. On supposera seulement que $S$ est non-divergent, i.e il n'a pas de séquence infinie d'actions internes passant par une infinité d'états distincts.

**Modèle d'implémentation :** L'implémentation réelle sous test (IUT) est une boîte noire, logicielle ou matérielle. Ce n'est pas un objet formel, mathématique. Cependant, pour pouvoir raisonner formellement sur la conformité d'une implémentation vis à vis d'une spécification, il faut se baser sur des modèles formels, seule possibilité pour définir une relation mathématique entre implémentation et spécification. On fait donc l'*hypothèse de test* suivante. Bien qu'ils soient inconnus, les comportements possibles de l'implémentation sont supposés modélisables par un IOLTS noté $IUT = (Q^{\text{IUT}}, A^{\text{IUT}}, \rightarrow_{\text{IUT}}, q_0^{\text{IUT}})$ avec $A^{\text{IUT}} = A_{\text{I}}^{\text{IUT}} \cup A_{\text{O}}^{\text{IUT}} \cup I^{\text{IUT}}$.

Dans la suite, on supposera toujours que les alphabets de l'IUT sont *compatibles* avec ceux de $S$ i.e. $A_{\text{I}}^{\text{s}} \subseteq A_{\text{I}}^{\text{IUT}}$, et $A_{\text{O}}^{\text{s}} \subseteq A_{\text{O}}^{\text{IUT}}$.

On supposera de plus que l'IUT est *complète en entrée* (faiblement) sur son alphabet d'entrée (voir définition 3.1.4).

L'hypothèse de compatibilité des alphabets reflète le fait que l'alphabet de l'IUT est inconnu, mais que ses entrées contiennent celles de la spécification, en réalité les seules avec lesquelles on le stimule dans le cadre de la conformité. Par contre on ne peut pas prévoir toutes ses sorties puisque, justement, le test sert à comparer les sorties observées avec celles spécifiées. L'hypothèse de complétude en entrée a ici un but précis que nous verrons plus tard. Informellement, combinée avec une hypothèse plus faible de complétion en entréesur les tests, elle garantit qu'un blocage de l'interaction testeur/IUT induit forcément un verdict (*cf.* section 3.1.3). L'hypothèse de complétude en entrée est raisonnable dans les cas où l'IUT ne refuse aucune entrée invalide ou inopportune, même si elle y répond négativement, par exemple par un message d'erreur. Cependant, pour certains systèmes réactifs, cette hypothèse n'est pas réaliste. Il suffit de penser par exemple à une interface homme machine (IHM) où certaines opérations sont interdites à certains moments. Remettre en cause cette hypothèse nécessite de prendre en compte les *refus* comme dans [21] et implique des changements non négligeables dans la théorie et les algorithmes.

**Comportements visibles :** Dans la pratique, les tests observent les réactions d'un système à des stimuli du testeur, donc les traces d'un système (i.e. les séquences d'actions visibles) et les comparent au comportement attendu, décrits par les traces de la spécification. Il est donc nécessaire d'identifier l'ensemble des traces de la spécification, ou tout au moins les traces prévisibles résultant des stimuli du testeur.

L'ensemble des traces d'un IOLTS (ou d'un LTS) $M$ peut être caractérisé par un IOLTS déterministe (voir la définition 3.1.2) :

**Définition 3.1.5 (Déterminisation)**
*Pour tout IOLTS $M = (Q^M, A^M, \rightarrow_M, q_0^M)$, [1], on peut définir un IOLTS $det(M)$ déterministe et de mêmes traces que $M$ i.e. $Traces(S) = Traces(det(M))$ Il est défini par*

$$det(M) = (2^{Q^M}, A^M_{VIS}, \rightarrow_{det}, q_0^M \textit{ after } \varepsilon)$$
$$\textit{où } P \xrightarrow{a}_{det} P' \iff P, P' \in 2^{Q^M}, a \in A^M_{VIS} \textit{ et } P' = P \textit{ after } a$$

Les états de $det(M)$, qu'on appellera *méta-états* par la suite, sont les parties de $Q^M$. L'état initial $q_0$ after $\varepsilon$ est l'ensemble des états accessibles depuis $q_0^M$ par des actions internes. Il suffit de considérer les méta-états accessibles depuis $q_0$ after $\varepsilon$. Le calcul de $det(M)$ est illustré par la transformation de la figure 3.4.

Signalons que les notions de déterminisme et de déterminisation des IOLTS sont exactement les même que celles des automates d'états finis [39] en interprétant action interne par $\epsilon$-transition. Nous verrons cependant en 3.3.2 un algorithme légèrement différent de l'algorithme classique pour calculer $det(M)$.

**Blocages :** dans la pratique du test, en plus des traces, les tests permettent aussi de détecter les blocages (*quiescence*). On utilise pour cela des temporisateurs (*timers*) qui sont

[1] En toute rigueur, si $Q^M$ est infini dénombrable, $2^{Q^M}$ n'est pas dénombrable et *a priori* $det(M)$ n'est plus un IOLTS au sens strict de la définition. Cependant, si $M$ est à branchement fini, le sous-ensemble de $2^{Q^M}$ accessible par $\rightarrow_{det}$ est dénombrable, ce qui permet de commencer la construction

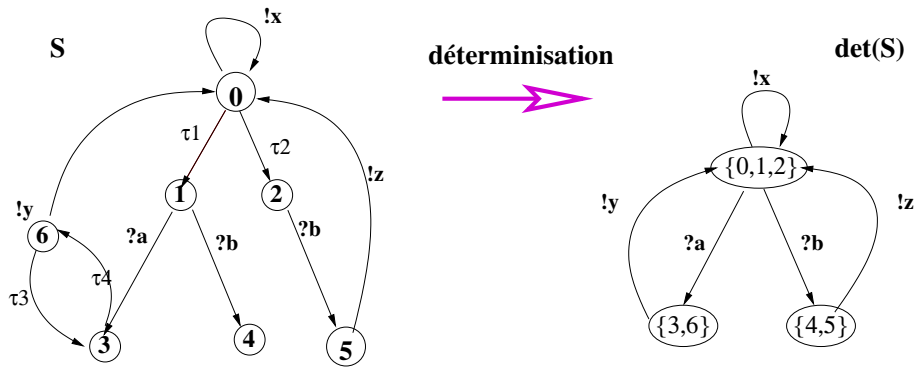



FIG. 3.4 – Déterminisation

armés en attente d'une réponse de l'IUT. On fait alors l'hypothèse que si la valeur du temporisateur est suffisament grande, l'expiration du temporisateur est assimilée à un blocage de l'IUT. Dans le modèle des IOLTS, plusieurs types de blocages peuvent apparaître, comme illustré sur l'IOLTS $S$ de la figure 3.5 :

**le *blocage de sortie* (ou *outputlock*) :** le système est bloqué en attente d'une entrée provenant de l'environnement i.e. $\Gamma(q) \subseteq A_{\mathrm{I}}^{\mathrm{M}}$ (états 1 et 2),

**le blocage complet (ou *deadlock*) :** le système ne peut plus évoluer i.e. $\Gamma(q) = \emptyset$ (état 4 de l'exemple), qui n'est qu'un cas particulier du blocage de sortie,

**le blocage vivant (ou *livelock*) :** le système diverge par une suite infinie d'actions internes, (états 3 et 6). Dans les IOLTS que nous considérons, les seuls livelocks possibles sont caractérisés par une boucle d'actions internes i.e. $\exists \tau_1, \tau_2, \ldots \tau_n, q \overset{\tau_1 \cdot \tau_2 \cdots \tau_n}{\rightarrow} q$. Notons que dans le modèle IOTS, Tretmans ne considère pas les livelocks (il se limite aux systèmes fortement convergents). Il est cependant intéressant de les considérer, car ils ne se différencient pas des deadlocks du point de vue de l'observation.

Pour un IOLTS $M$, notons $deadlock(M)$ l'ensemble de ses états de deadlock, $livelock(M)$ l'ensemble de ses états de livelock, $outputlock(M)$ l'ensemble de ses états de blocage de sortie, et définissons

$$quiescent(M) = deadlock(M) \cup livelock(M) \cup outputlock(M)$$

Un blocage dans la spécification n'est pas forcément une erreur de celle-ci. C'est évident pour les blocages de sortie, mais c'est vrai aussi pour les livelocks (par composition parallèle et masquage des interactions internes par exemple), et même les deadlocks. Par conséquent, un test appliqué à une IUT doit pouvoir distinguer entre un blocage de l'IUT valide (existant dans la spécification) ou non valide. Il faut donc détecter les blocages possibles dans les comportements de la spécification.

Or la déterminisation ne préserve pas nécessairement les blocages, comme l'illustre la figure 3.5. On remarque en effet que $det(S)$ ne présente plus ni deadlock, ni livelock (bien sûr), ni blocage de sortie. En effet, le deadlock de l'état 4 disparait puisque cet état

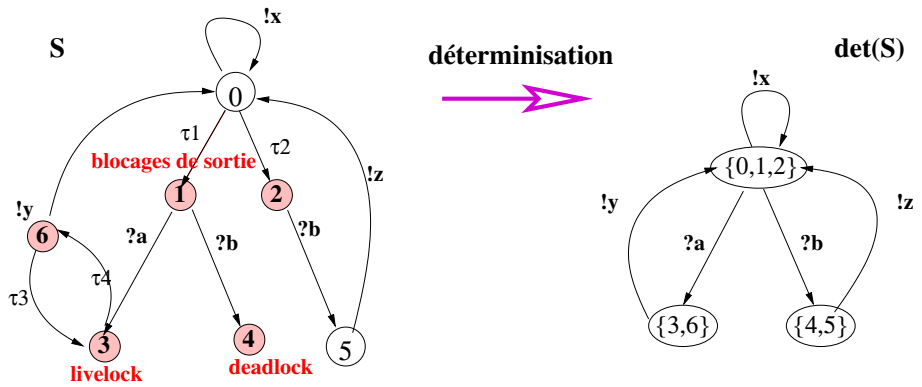



FIG. 3.5 – Perte des blocages par déterminisation

est regroupé dans le méta-état $\{4, 5\}$, or 5 n'a pas de deadlock. De même, les blocages de sortie dans les états 1 et 2 disparaissent dans le méta-état $\{0, 1, 2\}$ car 0 possède une sortie. Les livelocks disparaissent toujours.

Pour conserver les blocages par déterminisation, il faut les expliciter sur l'IOLTS de la spécification, après les avoir détectés. En fait, il suffit de rajouter, dans chaque état de blocage, une boucle étiquetée par une nouvelle action $\delta$. L'action $\delta$ est considérée comme une sortie de la spécification, puisqu'elle est observable mais non contrôlable par l'environnement. Elle sera notée $!\delta$ sur les exemples et l'action d'entrée correspondante dans le test, notée $?\delta$, correspondra à l'expiration d'un temporisateur. L'IOLTS résultant de l'ajout des actions $\delta$ est appelé *automate de suspension*. Sa définition ci-dessous formalise la transformation décrite par la figure 3.6.

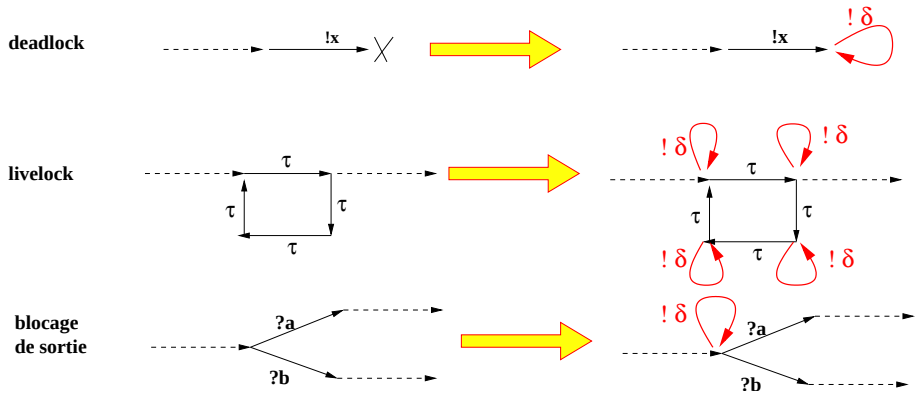



FIG. 3.6 – Construction de l'automate de suspension

**Définition 3.1.6 (Automate de suspension)**
*Soit un IOLTS* $M = (Q^M, A^M, \rightarrow_M, q_0^M)$. *L'*automate de suspension *de* $M$ *est l'IOLTS* $\Delta(M) = (Q^M, A^{\Delta(M)}, \rightarrow_{\Delta(M)}, q_0^{\Delta(M)})$ *tel que* $A^{\Delta(M)} = A^M \cup \{\delta\}$ *avec* $\delta \in A_O^{\Delta(M)}$ *(i.e.* $\delta$ *est une sortie) et la relation de transition* $\rightarrow_{\Delta(M)}$ *est définie par :*

$$\rightarrow_{\Delta(M)} = \rightarrow_M \cup \{q \xrightarrow{\delta} q \mid q \in quiescent(M)\}$$

*i.e.* $\rightarrow_{\Delta(M)}$ *est obtenue à partir de* $\rightarrow_M$ *par ajout de boucles* $q \xrightarrow{\delta} q$ *pour tous les états* $q$ *de blocage.*

> **Note:** *Un livelock étant une boucle, ou plus généralement une composante fortement connexe (CFC, cf. section 3.2) non triviale d'actions internes, il suffit en réalité d'ajouter une boucle* $\delta$ *dans un des états d'une telle CFC pour la rendre tirable au sens la relation* $\Rightarrow$ *depuis tout état de cette CFC.*

La déterminisation *a posteriori* de l'automate de suspension (i.e. après ajout des $\delta$), illustré par la transformation de la figure 3.7, caractérise alors les comportements visibles en conservant toutes les informations de blocages. Ces comportements visibles, traces de l'automate suspendues sont appellées les traces supendues (*suspension traces*) par Tretmanss [77].

**Définition 3.1.7 (Traces suspendues)**
*Les traces suspendues d'un IOLTS* $M$ *notées* $STraces(M)$ *sont les traces de son automate de suspension,* $\delta$ *étant considéré comme une action observable i.e.*

$$STraces(M) \triangleq Traces(\Delta(M))$$

Notons que Tretmans dans [77] définit directement l'automate de suspension comme l'automate déterminisé résultant i.e $det(\Delta(M))$. Il nous a paru plus clair de décrire la construction en deux étapes en séparant l'ajout des $\delta$ et la déterminisation.

**Langages définis par les IOLTS :** dans la suite, il sera parfois utile de raisonner sur les IOLTS en termes de langages. On dotera certains IOLTS d'états marqués afin de les considérer comme des automates, sinon on peut également considérer un IOLTS comme un automate dont tous les états sont marqués. On peut par ailleurs considérer les actions internes comme faisant partie ou non de l'alphabet de l'automate (i.e. les transitions sur des actions internes sont considérées ou non comme des $\varepsilon$-transitions). On distingue alors les *langages des actions* et les *langages de traces*. On obtient donc plusieurs langages différents qu'il est de nécessaire de bien distinguer pour éviter toute ambiguité. Nous ferons en particulier une distinction entre *langage reconnu* (sur des états marqués) et *langage généré* (où tout état est marqué). Dans un même IOLTS, on définira souvent deux ensembles d'états marqués, les états *accepteurs* et les états de *refus*. On raffinera alors le terme langage reconnu en *langage accepté* ou *langage refusé*.

Pour un IOLTS $M = (Q^{\mathrm{M}}, A^{\mathrm{M}}, \rightarrow_{\mathrm{M}}, q_0^{\mathrm{M}})$ et un ensemble $F \subseteq Q^{\mathrm{M}}$ d'états marqués, on définit les langages suivants :

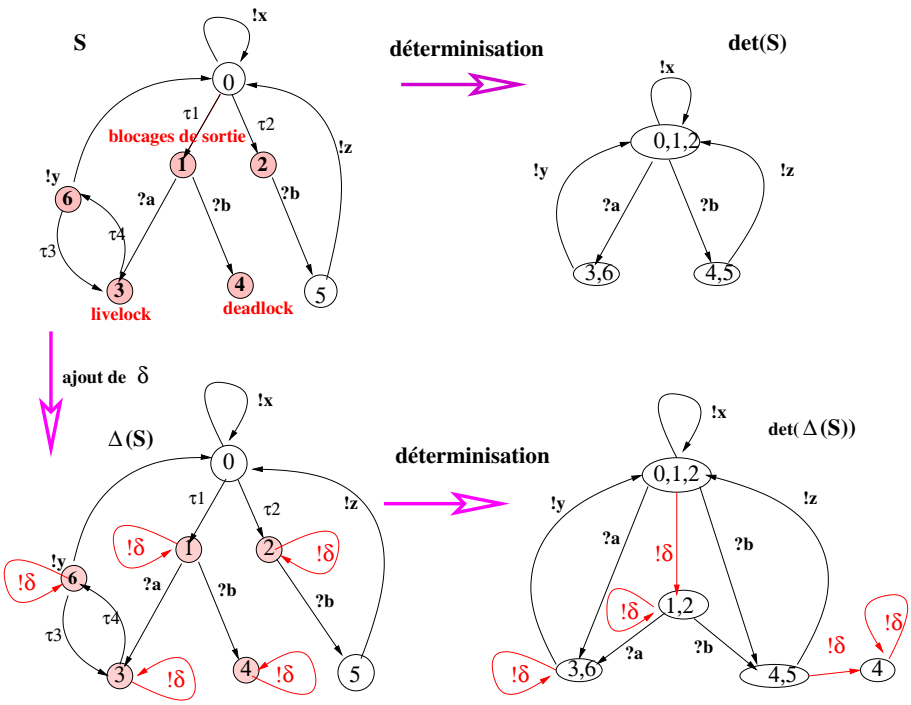



FIG. 3.7 – Comportements visibles et blocages

**Langage des actions reconnu sur** $F$ **:** $L_F(M) \triangleq \{\mu \in A^{\mathrm{M}*} \mid \exists f \in F, q_0^{\mathrm{M}} \xrightarrow{\mu} f\}$
C'est l'ensemble des séquences d'actions sur l'alphabet $A^{\mathrm{M}}$ menant à un état marqué de $F$. Dans ce langage, les actions internes sont considérées comme les autres, non pas comme des $\epsilon$-transitions.
**Langage généré des actions :** $L(M) \triangleq L_{Q^{\mathrm{M}}}(M) = \{\mu \in A^{\mathrm{M}*} \mid q_0^{\mathrm{M}} \xrightarrow{\mu}\}$
C'est donc l'ensemble des séquences d'actions possibles dans $M$. Notons que pour tout $F$, $L_F(M)$ est un sous-ensemble de $L(M)$.

Les traces sont aussi des langages où on restreint les alphabets aux actions visibles, et les actions internes sont considérées comme des $\epsilon$-transitions.

**Langages des traces reconnu sur** $F$ **:** $Traces_F(M) \triangleq \{\sigma \in {A_{\mathrm{VIS}}^{\mathrm{M}}}^* \mid \exists f \in F, q_0^{\mathrm{M}} \overset{\sigma}{\Rightarrow} f\}$
Dans ce langage sur l'alphabet $A_{\mathrm{VIS}}^{\mathrm{M}}$, les actions internes de $I^{\mathrm{M}}$ sont indifférenciées et assimilées à des $\epsilon$-transitions. Notons que le **langage (généré) des traces de** $M$, $Traces(M) = Traces_{Q^{\mathrm{M}}}(M)$.

Bien entendu, si $M$ n'a pas d'action interne, les langages des actions et les traces coïncident. $L(M) = Traces(M)$ et $L_F(M) = Traces_F(M)$.

On utilisera fréquemment la notation $\overline{L}$ pour désigner le complémentaire d'un langage $L$.
**Préfixe :** soient $\omega, \omega' \in A^*$ des mots sur un alphabet $A$. On note $\omega' \leq \omega$ si $\omega'$ est un préfixe de $\omega$ i.e. $\exists \omega'' \in A^*, \omega = \omega'.\omega''$. Si $\omega' \leq \omega$ et $\omega' \neq \omega$, $\omega'$ est un préfixe strict et on note $\omega' < \omega$.
On note $prf_{\leq}(\omega)$ l'ensemble des préfixes de $\omega$, i.e.

$$prf_{\leq}(\omega) \triangleq \{\omega' \in A^* \mid \omega' \leq \omega\}$$

Pour un langage $L \subseteq A^*$, on note $prf_{\leq}(L)$ la *clôture par préfixe* de $L$, i.e.

$$prf_{\leq}(L) \triangleq \bigcup_{\omega \in L} prf_{\leq}(\omega)$$

On notera aussi $prf_{<}(L)$ l'ensemble des préfixes de $L$ qui ne sont pas dans $L$, i.e.

$$prf_{<}(L) \triangleq prf_{\leq}(L) \setminus L$$

Attention, en général $prf_{<}(L)$ n'est pas l'union des préfixes stricts de mots de $L$, voir par exemple le langage $L = \{a.b, a.b.c\}$.

Un langage $L$ est dit *préfixe-clos* si $L = prf_{\leq}(L)$. Notons que le langage généré des actions $L(M)$ et le langage des traces $Traces(M)$ sont préfixe-clos.

On dira qu'un mot $\omega \in L$ est *minimal* pour l'ordre préfixe s'il n'existe pas de préfixe strict de $\omega$ dans $L$ i.e. $\forall \omega' \in L, \neg(\omega' < \omega)$.

### 3.1.2 Relation de conformité

Une *relation de conformité* est une relation entre implémentations et spécifications qui définit exactement l'ensemble des implémentations conformes à une spécification donnée. Rappelons que dans le cas des machines de Mealy, la relation de conformité était l'équivalence d'automates (voir 2.3.1).

Pour les IOLTS, plusieurs relations sont envisageables suivant les observations permises par le test (e.g. traces, blocages, transitions tirables, refus, etc) et la notion de correction voulue. Parmi celles-ci nous avons choisi de retenir la relation de conformité **ioco** définie par Tretmans et al. [77] qui nous semble la plus appropriée à bon nombre de cas. Les observations possibles par un testeur sur une IUT sont les traces suspendues, i.e. les traces de son automate de suspension qui contiennent les entrées, les sorties et les blocages. La relation **ioco** est définie comme suit :

**Définition 3.1.8 (Relation de conformité)**
*Soit $S$ un IOLTS et $IUT$ un IOLTS complet en entrée et compatible avec $S$,*

$$IUT \textbf{ ioco } S \triangleq \forall \sigma \in STraces(S), Out(\Delta(IUT) \textit{ after } \sigma) \subseteq Out(\Delta(S) \textit{ after } \sigma)$$

Bien qu'ayant choisi une expression des modèles et de **ioco** légèrement différentes, cette définition correspond à celle de Tretmans, excepté la prise en compte des livelocks dans les automates de suspension $\Delta(S)$ et $\Delta(IUT)$.

De façon plus intuitive que la définition ci-dessus, une implémentation $IUT$ (complète en entrée et d'alphabet compatible) est conforme à sa spécification $S$ pour **ioco** si après toute trace suspendue $\sigma$ de $S$ (i.e. une trace de son automate de suspension $\Delta(S)$), les sorties de $IUT$ (y compris les blocages puisque $\delta$ est une sortie) sont incluses dans celles de $S$. Ou encore, les traces suspendues de l'IUT qui sont des extensions par des sorties ou blocages de l'IUT, de traces suspendues de $S$ sont aussi des traces suspendues de $S$. Ce qui donne une description alternative de **ioco**, uniquement en termes d'ensembles de traces suspendues, qui sera souvent préférée dans les preuves :

**Proposition 3.1.2** *Soit $S$ une spécification et $IUT$ une implémentation d'alphabet compatible à $S$,*

$$IUT \textbf{ ioco } S \iff STraces(IUT) \cap STraces(S).A_o^{\Delta(IUT)} \subseteq STraces(S)$$

**Preuve :** En effet, par définition de **ioco** :

$$\begin{aligned} IUT \textbf{ ioco } S \quad &\iff \quad \forall \sigma \in STraces(S), Out(\Delta(IUT) \text{ after } \sigma) \subseteq Out(\Delta(S) \text{ after } \sigma) \\ &\iff \quad \forall \sigma \in STraces(S), \forall a \in A_o^{\Delta(IUT)}, \\ &\qquad \sigma.a \in STraces(IUT) \Rightarrow \sigma.a \in STraces(S) \\ &\iff \quad STraces(IUT) \cap STraces(S).A_o^{\Delta(IUT)} \subseteq STraces(S) \end{aligned}$$

□

**Remarque :** il est intéressant de noter qu'être conforme à une spécification est une propriété de sûreté de l'implémentation. En effet, cette propriété peut être invalidée sur un préfixe fini.

La relation **ioco** est illustrée par la figure 3.8 qui va nous permettre de mieux comprendre les conséquences de la définition.

- $IUT_1$ **ioco** $S$ : après les traces de $\Delta(S)$, les sorties de $IUT_1$ sont incluses dans celles de $S$. Par exemple après $\sigma \in (\delta^*.?a.!x.!z)^*.\delta^*.?a$, $\{!x\} \subseteq \{!x, !y\}$. **ioco** permet donc de restreindre les sorties de l'IUT, donc de faire des choix d'implémentation.
- $IUT_2$ **ioco** $S$ : l'état initial de $IUT_1$ permet une entrée supplémentaire $?b$ par rapport à $S$ mais **ioco** l'autorise puisque seules comptent les sorties après les traces de $\Delta(S)$. **ioco** permet donc de décrire des spécifications partielles.
- Par contre $\neg(IUT_3$ **ioco** $S)$ : la sortie $!z$ après l'entrée $?a$ n'est pas autorisée dans la spécification.
- De même $\neg(IUT_4$ **ioco** $S)$ : un blocage (ici un livelock par une boucle d'actions internes) après $?a.!x$ n'est pas autorisé.

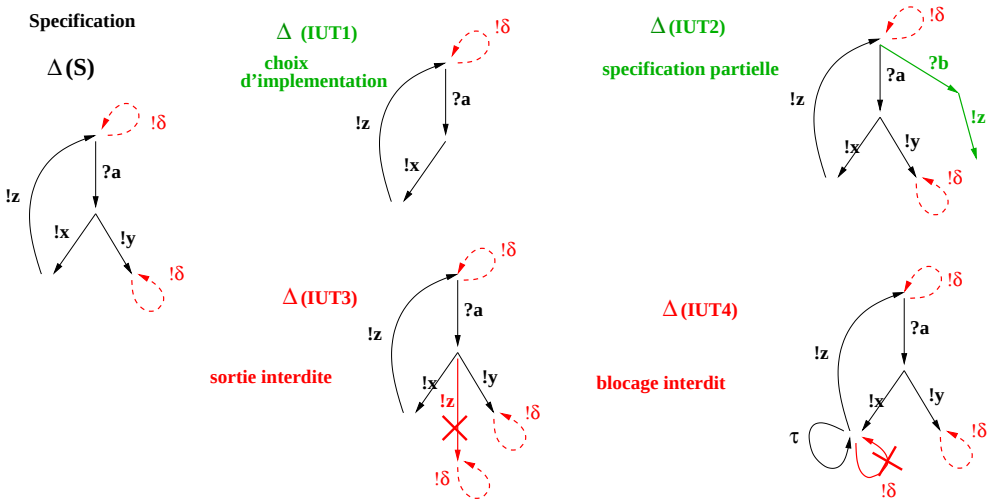



FIG. 3.8 – **ioco** par l'exemple (expliqué sur $\Delta(S)$ et $\Delta(IUT)$)

**Cas des spécifications complètes en entrée :** Rappelons que nous avons jusqu'ici supposé l'IUT complète en entrée, mais pas la spécification. Par contre, quand la spécification est aussi complète en entrée sur l'alphabet d'entrée de l'IUT, la formulation de **ioco** est simplifiée. **ioco** s'exprime alors simplement par inclusion de traces suspendues :

**Proposition 3.1.3** *Soit $S$ et $IUT$ deux IOLTS compatibles, de même alphabet d'entrée et complets en entrée sur cet alphabet,*

$$IUT \textbf{ ioco } S \iff STraces(IUT) \subseteq STraces(S)$$

**Preuve :** ($\Rightarrow$) montrons que $STraces(IUT) \not\subseteq STraces(S) \Rightarrow \neg(IUT$ **ioco** $S)$. Si $STraces(IUT) \not\subseteq STraces(S)$, il existe donc $\sigma$ tel que $\sigma \in STraces(IUT)$ et $\sigma \notin STraces(S)$. La trace $\sigma$ peut être décomposée en $\sigma = \sigma'.a.\sigma''$ où $\sigma'$ est le plus long préfixe de $\sigma$ dans $STraces(S)$, $a \in A^{\Delta(IUT)}$ et $\sigma'' \in (A^{\Delta(IUT)})^*$. Noter que $\sigma'.a \in$

$STraces(IUT)$ par préfixe-clôture. Si $a \in A_{\mathrm{I}}^{\Delta IUT}$ est une entrée, puisque $S$ est complet en entrée sur $A_{\mathrm{I}}^{\Delta IUT}$, on a aussi $\sigma.a \in STraces(S)$. Donc $\sigma$ n'est pas le plus long préfixe recherché dans $STraces(S)$. Par conséquent, $a \in A_{\mathrm{O}}^{\Delta IUT}$ est une sortie ou un blocage. On a donc $\sigma'.a \in STraces(S).A_{\mathrm{O}}^{\Delta IUT}$ et $\sigma'.a \in STraces(IUT)$ mais $\sigma'.a \notin STraces(S)$, ce qui prouve que $\neg(IUT$ **ioco** $S)$ par la proposition 3.1.2.
($\Leftarrow$) Inversement, supposons que $STraces(IUT) \subseteq STraces(S)$. On a immédiatement $STraces(IUT) \cap STraces(S).A_{\mathrm{O}}^{\Delta IUT} \subseteq STraces(S)$, donc $IUT$ **ioco** $S$ par la proposition 3.1.2.
□

De plus, quand la spécification $S$ n'est pas complète en entrée, il est toujours possible de la rendre complète en entrée, sans modifier l'ensemble des implémentations qui lui sont conformes pour des alphabets d'entrées et de sorties donnés. Il suffit d'observer que d'après la définition de **ioco** après une trace suspendue de $S$, et après une entrée de l'$IUT$ non spécifiée dans $S$ après cette trace, l'$IUT$ peut faire ce qu'elle veut. On ne donne pas de définition constructive de la complétion mais seulement une définition de ses traces suspendues. La construction effective passe nécessairement par la construction des traces suspendues, donc de $det(\Delta(S))$, à partir duquel il suffit de rajouter des transitions pour les entrées non spécifiées menant à un état dans lequel toute action visible de $A^{\mathrm{IUT}}$ (y compris $\delta$) boucle.

**Proposition 3.1.4 (Complétion en entrée)**
*Soit $S$ un IOLTS, $A_I^{IUT}$ et $A_O^{IUT}$ des alphabets d'entrée et sorties d'IUT compatibles à $S$ (donc $A_I^s \subseteq A_I^{IUT}$ et $A_O^s \subseteq A_O^{IUT}$). On peut toujours construire un IOLTS complet en entrée sur $A_I^{IUT}$ noté $InComp(S, A_I^{IUT}, A_O^{IUT})$ tel que*

$$STraces(InComp(S, A_I^{IUT}, A_O^{IUT})) =$$
$$STraces(S) \cup [STraces(S).A_I^{IUT} \cap \overline{STraces(S)}].(A_I^{\Delta(IUT)} \cup A_O^{\Delta(IUT)})^*$$

On peut alors montrer que cette complétion en entrée préserve **ioco** sur les implémentations d'alphabets d'entrée et de sortie $A_{\mathrm{I}}^{\mathrm{IUT}}$ et $A_{\mathrm{O}}^{\mathrm{IUT}}$ :

**Proposition 3.1.5** *Soit $S$ une spécification et $IUT$ une implémentation d'alphabet compatible à $S$,*

$$IUT \textbf{ ioco } S \iff IUT \textbf{ ioco } InComp(S, A_I^{IUT}, A_O^{IUT})$$

Nous avons démontré une proposition similaire dans [47], mais pour une relation de conformité plus faible qui ne considère pas les blocages. On donne ici la preuve complète pour **ioco**.

**Preuve :** ($\Rightarrow$) On suppose $IUT$ **ioco** $S$, et on doit montrer $IUT$ **ioco** $InComp(S, A_{\mathrm{I}}^{\mathrm{IUT}}, A_{\mathrm{O}}^{\mathrm{IUT}})$, qui par la proposition 3.1.2, se ramène à prouver l'inclusion

$$(i) : STraces(InComp(S, A_{\mathrm{I}}^{\mathrm{IUT}}, A_{\mathrm{O}}^{\mathrm{IUT}})).A_{\mathrm{O}}^{\mathrm{IUT}} \cap STraces(IUT) \subseteq$$
$$STraces(InComp(S, A_{\mathrm{I}}^{\mathrm{IUT}}, A_{\mathrm{O}}^{\mathrm{IUT}})$$

Par la proposition 3.1.4, on a

$$STraces(InComp(S, A_{\mathrm{I}}^{\mathrm{IUT}}, A_{\mathrm{O}}^{\mathrm{IUT}})) =$$
$$STraces(S) \cup [STraces(S).A_{\mathrm{I}}^{\mathrm{IUT}} \cap \overline{STraces(S)}].(A_{\mathrm{I}}^{\Delta(IUT)} \cup A_{\mathrm{O}}^{\Delta(IUT)})^*$$

En distribuant la concaténation sur l'union, le membre gauche de (i) se récrit en

$$[STraces(S).A_{\mathrm{O}}^{\mathrm{IUT}} \cup [STraces(S).A_{\mathrm{I}}^{\mathrm{IUT}} \cap \overline{STraces(S)}].(A_{\mathrm{I}}^{\Delta(IUT)} \cup A_{\mathrm{O}}^{\Delta(IUT)})^*.A_{\mathrm{O}}^{\mathrm{IUT}} \cap Straces(IUT)$$

En distribuant $\cap$ sur $\cup$, ce membre gauche devient $A \cup B$ avec

$$\begin{aligned} A &= STraces(S).A_{\mathrm{O}}^{\mathrm{IUT}} \cap STraces(IUT) \\ B &= [STraces(S).A_{\mathrm{I}}^{\mathrm{IUT}} \cap \overline{STraces(S)}].(A_{\mathrm{I}}^{\Delta(IUT)} \cup A_{\mathrm{O}}^{\Delta(IUT)})^*.A_{\mathrm{O}}^{\mathrm{IUT}} \cap Straces(IUT) \end{aligned}$$

Dans $B$, on remarque que $(A_{\mathrm{I}}^{\Delta(IUT)} \cup A_{\mathrm{O}}^{\Delta(IUT)})^*.A_{\mathrm{O}}^{\mathrm{IUT}} \subseteq (A_{\mathrm{I}}^{\Delta(IUT)} \cup A_{\mathrm{O}}^{\Delta(IUT)})^*$ et on simplifie pour obtenir

$$\begin{aligned} (\alpha) : A &\subseteq STraces(S) \text{ par prop 3.1.2} \\ (\beta) : B &\subseteq [STraces(S).A_{\mathrm{I}}^{\mathrm{IUT}} \cap \overline{STraces(S)}].(A_{\mathrm{I}}^{\Delta(IUT)} \cup A_{\mathrm{O}}^{\Delta(IUT)})^* \end{aligned}$$

En regroupant les inclusions $(\alpha)$ et $(\beta)$ on obtient

$$\begin{aligned} A \cup B &\subseteq STraces(S) \cup [STraces(S).A_{\mathrm{I}}^{\mathrm{IUT}} \cap \overline{STraces(S)}].(A_{\mathrm{I}}^{\Delta(IUT)} \cup A_{\mathrm{O}}^{\Delta(IUT)})^* \\ &= STraces(InComp(S, A_{\mathrm{I}}^{\mathrm{IUT}}, A_{\mathrm{O}}^{\mathrm{IUT}})) \end{aligned}$$

Ce qui termine la première partie de la preuve.
($\Leftarrow$) Supposons que $\neg(IUT$ **ioco** $S)$. Par définition, il existe $\sigma \in STraces(S)$ et $a \in A_{\mathrm{O}}^{\Delta(IUT)}$ tels que $\sigma.a \in STraces(IUT)$ mais $\sigma.a \notin STraces(S)$. Par la proposition 3.1.4, on a $STraces(S) \subseteq STraces(InComp(S, A_{\mathrm{I}}^{\mathrm{IUT}}, A_{\mathrm{O}}^{\mathrm{IUT}}))$ donc on a le résultat intermédiaire

$$(I) : \sigma \in STraces(InComp(S, A_{\mathrm{I}}^{\mathrm{IUT}}, A_{\mathrm{O}}^{\mathrm{IUT}}))$$

La complétion en entrée n'ajoute jamais de sortie après des traces suspendues de $S$, ce qui s'écrit

$$(II) : STraces(S).A_{\mathrm{O}}^{\mathrm{IUT}} \cap STraces(InComp(S, A_{\mathrm{I}}^{\mathrm{IUT}}, A_{\mathrm{O}}^{\mathrm{IUT}})) \subseteq STraces(S)$$

Si on avait $\sigma.a \in STraces(InComp(S, A_{\mathrm{I}}^{\mathrm{IUT}}, A_{\mathrm{O}}^{\mathrm{IUT}}))$, puisqu'on a $\sigma.a \notin STraces(S)$, l'inclusion $(II)$ impliquerait $\sigma.a \in STraces(S)$. Donc $\sigma.a \notin STraces(InComp(S, A_{\mathrm{I}}^{\mathrm{IUT}}, A_{\mathrm{O}}^{\mathrm{IUT}}))$. Avec $(I)$ et le fait que $\sigma.a \in STraces(IUT)$, ceci montre que $\neg(IUT$ **ioco** $InComp(S, A_{\mathrm{I}}^{\mathrm{IUT}}, A_{\mathrm{O}}^{\mathrm{IUT}}))$.
□

En combinant les propositions 3.1.4 et 3.1.3, dans le cas général où $S$ n'est pas complet en entrée, on peut quand même ramener **ioco** à une inclusion de traces suspendues :

**Proposition 3.1.6 (ioco et inclusion de traces)**

$$IUT \textbf{ ioco } S \iff STraces(IUT) \subseteq STraces(InComp(S, A_I^{IUT}, A_O^{IUT}))$$

**Note:** *On peut noter qu'on n'a encore jamais utilisé l'hypothèse que l'IUT est complète en entrée. En réalité, tous les résultats donnés jusqu'ici sont valides pour des IUT non complètes en entrée. L'hypothèse n'est utile que pour parler de l'exécution des cas de tests sur l'IUT.*

### 3.1.3 Cas de tests et suites de tests : modèles, exécution et propriétés

La relation de conformité **ioco** est une relation entre modèles d'implémentations et modèles de spécifications, tous deux des IOLTS. Dans le cadre du test de conformité, le modèle de l'implémentation n'est pas connu, ce qui interdit de vérifier directement qu'une implémentation est conforme à sa spécification par des techniques de type model-checking. Par contre l'implémentation est un système réactif avec lequel un autre système réactif, un *cas de test*, peut interagir. La relation **ioco** étant une relation sur les traces suspendues de l'implémentation et de la spécification, les cas de test doivent permettre d'observer les traces suspendues de l'implémentation. Nous allons donc modéliser les cas de test par des IOLTS, étendus par la notion de verdicts. Nous définirons également les exécutions des cas de test sur les implémentations, qui définissent les observations, ainsi que la notion de verdict associé. Enfin nous définirons les propriétés que l'on attend des cas de tests que nous souhaitons générer, par rapport à la relation de conformité.

Nous avons vu plus haut que les systèmes réactifs modélisés par des IOLTS ne sont pas toujours contrôlables par l'environnement : pour une même entrée, ils peuvent produire plusieurs sorties différentes. Cette notion a été appelée *non déterminisme observable* pour la différencier du *non déterminisme* des automates. Les *cas de test*, qui décrivent les interactions entre le testeur et l'implémentation, doivent donc prévoir ce non déterminisme observable. Ils doivent prévoir l'observation de toute sortie de l'IUT, correcte ou non, par complétion sur les entrées, relativement à l'alphabet de sortie de l'IUT. Par contre, puisqu'ils contrôlent leurs propres sorties (les entrées de l'IUT), ils n'ont pas besoin de choix sur celles-ci. Comme on s'intéresse à la conformité et non à la robustesse, les tests n'émettront que des actions de l'alphabet d'entrée de la spécification. Enfin, les cas de test sont déterministes (au sens des automates). Comme pour les spécifications et implémentations, on modélise un cas de test par un IOLTS, muni de verdicts et ayant quelques propriétés de structure supplémentaires. C'est un comportement complexe dont la structure est un graphe pouvant posséder des boucles.

**Définition 3.1.9 (Cas de test, suite de test)**
*Un* cas de test *est un IOLTS déterministe* $TC = (Q^{TC}, A^{TC}, \rightarrow_{TC}, q_0^{TC})$ *muni de trois sous-ensembles distincts d'états sans successeur* **Pass**, **Fail** *et* **Inconc** *de* $Q^{TC}$ *caractérisant les verdicts. L'alphabet de TC est* $A^{TC} = A_I^{TC} \cup A_O^{TC}$ *avec* $A_O^{TC} \subseteq A_I^S$ *(i.e.* $TC$ *n'émet que des entrées de la spécification) et* $A_I^{TC} = A_O^{IUT} \cup \{\delta\}$ *(i.e.* $TC$ *doit prévoir la réception de toute sortie et blocage de l'IUT).*

*On suppose (et on assurera par construction) que*

- *(1) TC est* contrôlable *i.e. il n'a jamais le choix entre des sorties ou entre des entrées et des sortie :*

$$\forall q \in Q^{TC}, (\exists a \in A_O^{TC}, q \xrightarrow{a}_{TC} \Rightarrow \forall b \in A^{TC}, (b \neq a \Rightarrow q \not\xrightarrow{b}_{TC}))$$

- *(2) tout état permettant une entrée est* complet en entrées *:*

$$\forall q \in Q^{TC},\ (\exists\ a \in A_I^{TC},\ q \xrightarrow{a}_{TC} \Rightarrow\ \forall\ b \in A_I^{TC},\ q \xrightarrow{b}_{TC})$$

- *(3)* **Fail** *et* **Inconc** *ne sont accessibles en un pas que par des entrées :*

$$\forall (q, a, q') \in \rightarrow_{TC},\ (q' \in \textbf{Inconc} \cup \textbf{Fail} \Rightarrow a \in A_I^{TC})$$

– *(4) de tout état, un verdict est accessible :*

$$\forall q \in Q^{TC},\ \exists \sigma \in A^{TC*},\ \exists q' \in \textbf{Pass} \cup \textbf{Inconc} \cup \textbf{Fail},\ q \xrightarrow{\sigma} q'$$

*On appelle* suite de test *un ensemble (fini ou infini) de cas de test.*

Dans la plupart des articles sur la génération de tests pour des modèles comportementaux, les auteurs se limitent à générer des tests ayant des structures simples. Dans la majorité des cas, ces sont des séquences car on se restreint à des systèmes à *déterminisme observable*. Quand les systèmes exhibent du non-déterminisme observable, les séquences ne sont plus adaptées, car il faut prendre en compte toutes les sorties possibles du système pour une entrée donnée. Les tests sont alors des arbres, qui parfois ne diffèrent d'une séquence que par la prise en compte de sorties alternatives par rapport aux sorties attendues. Mais les tests écrits à la main ont souvent des boucles, par exemple pour ignorer l'occurrence d'une sortie du système non significative pour le test. Les cas de tests produits par TGV auront aussi des boucles, comme le laisse penser la définition 3.1.9, ce qui implique l'existence d'exécutions infinies.

En effet, considérons un cas de test comme un graphe. La définition 3.1.9 stipule que les états de verdit sont des puits, donc des composantes fortement connexes (CFC) maximales, triviales et terminales. Puisque de tout état, un verdict est accessible, le cas de test n'a aucune CFC terminale ne portant pas de verdict. En d'autre termes, le graphe réduit des CFC est un graphe acyclique dont toutes les feuilles portent des verdicts. Dans toute CFC non triviale (ne portant donc pas de verdict), il existe donc au moins un état ayant au moins un successeur hors de cette CFC (en direction d'un verdict), et un autre dans la CFC. D'après la propriété de controlabilité, ce choix ne peut exister qu'entre des entrées du cas de test. Ce qui signifie qu'on ne peut pas contrôler la sortie d'une CFC et qu'on peut y rester infiniment. Pour la même raison, s'il existe des CFC non triviales, on ne peut pas non plus éviter d'y entrer. Les cas de test autorisent donc des exécutions infinies. Or en pratique on veut assurer que l'exécution d'un test soit finie, même bornée. La solution usuelle est d'utiliser un temporisateur global au cas de test, dont l'échéance produirait un verdict *Inconc*. Nous n'avons pas introduit ce temporisateur global dans la modélisation pour ne pas l'alourdir. Au lieu de celà, nous nous limiterons aux traces finies.

> **Note:** *si on avait exigé l'inévitabilité d'un verdict au lieu de l'accessibilité, les cas de tests auraient été des graphes acycliques. Pour les amateurs de logique temporelle, c'est toute la différence entre* $AGAF(\textbf{Pass} \vee \textbf{Fail} \vee \textbf{Inconc})$ *(inévitabilité d'un verdict) et* $AGEF(\textbf{Pass} \vee \textbf{Fail} \vee \textbf{Inconc})$ *(accessibilité à un verdict) en CTL.*

**Exécution des cas de test :** L'exécution d'un cas de test $TC$ sur une implémentation $IUT$ est modélisée par la composition parallèle de $TC$ avec $\Delta(IUT)$ avec synchronisation sur les actions visibles communes, y compris $\delta$, noté $TC \parallel \Delta(IUT)$ .

De manière plus générale, la composition parallèle $P \parallel Q$ de deux IOLTS quelconques $P$ et $Q$, est définie par les trois règles :

$$\frac{p \xrightarrow{a}_P p',\ q \xrightarrow{a}_Q q',\ a \in A^{\mathrm{P}}_{\mathrm{VIS}} \cap A^{\mathrm{Q}}_{\mathrm{VIS}}}{(p,q) \xrightarrow{a}_{P \parallel Q} (p',q')}$$

$$\frac{p \xrightarrow{\tau}_P p',\ \tau \in I^{\mathrm{P}}}{(p,q) \xrightarrow{\tau}_{P \parallel Q} (p',q)} \qquad \frac{q \xrightarrow{\tau}_Q q',\ \tau \in I^{\mathrm{Q}}}{(p,q) \xrightarrow{\tau}_{P \parallel Q} (p,q')}$$

La proposition suivante implique que toute exécution maximale finie d'un cas de test produit un verdict.

**Proposition 3.1.7** *L'exécution d'un cas de test $TC$ sur une une implémentation $IUT$ n'est jamais bloquée, excepté dans un état de blocage de $TC$, état dans lequel un verdict est émis.*

**Preuve :** La demonstration se base sur la définition des cas de test, des implémentations et de la composition parallèle $\|$. Notons d'abord que les sorties (resp. les entrées) de TC sont des entrées (resp. des sorties) de $\Delta(IUT)$. Soit $\mu$ une séquence d'exécution de $TC \parallel \Delta(IUT)$ et notons $(q_{\mathrm{TC}}, q_{\mathrm{IUT}})$ l'état de $TC \parallel \Delta(IUT)$ atteint après $\mu$ depuis l'état initial. Par définition des cas de test, $Q^{\mathrm{TC}}$ est partitionné en états d'émission (où il ne peut recevoir), états de réception (où il ne peut pas émettre) et états de blocage où un verdict est assigné.

- Supposons que $q_{\mathrm{TC}}$ est un état d'émission et soit $a$ une émission dans cet état. L'IUT étant complète en entrée dans chaque état, $q_{\mathrm{IUT}} \overset{a}{\Rightarrow}$. Donc $TC \parallel \Delta(IUT)$ peut progresser (par des actions internes éventuelles) jusqu'à synchronisation sur la sortie $a$ de $TC$, ce n'est donc pas un état de blocage.
- Supposons maintenant que $q_{\mathrm{TC}}$ est un état de réception (complet en entrée). Soit $q_{\mathrm{IUT}}$ a une action interne, soit au moins une sortie (dans tout état, si l'IUT n'a pas de sortie ni d'action interne, $\Delta(IUT)$ a la sortie $!\delta$). Dans le premier cas $TC \parallel \Delta(IUT)$ peut progresser par l'action interne de l'$IUT$, dans le second cas par synchronisation sur une sortie de l'$IUT$ (ou $\delta$) puisque $q_{\mathrm{TC}}$ est complet en entrée. Donc $TC \parallel \Delta(IUT)$ n'est pas dans un état de blocage.
- Enfin, le seul cas restant est celui où $q_{\mathrm{TC}}$ est dans un état de blocage, où un verdict est associé par hypothèse. $TC \parallel \Delta(IUT)$ ne peut alors progresser que par des actions internes de l'$IUT$, invisibles de $TC$.

□

Les actions internes de l'IUT étant invisibles du cas de test, nous considérerons que les observations que fait un cas de test d'une implémentation sont les *traces finies maximales* de $TC \parallel \Delta(IUT)$ [2] du cas de test i.e. les séquences $\sigma \in A^{\mathrm{TC}^*}$ telles que $\Gamma((q_0^{\mathrm{TC}}, q_0^{\Delta(IUT)})$ after $\sigma) = \emptyset$.

On peut alors définir le verdict de l'exécution du cas de test sur l'IUT pour une telle observation.

**Définition 3.1.10 (Verdicts)**
*Soit $\sigma$ une trace finie maximale de $TC \parallel \Delta(IUT)$ (et de $TC$),*

$$\begin{aligned} verdict(\sigma) = \textit{Fail} &\triangleq TC \textit{ after } \sigma \subseteq \textbf{Fail} \\ verdict(\sigma) = \textit{Pass} &\triangleq TC \textit{ after } \sigma \subseteq \textbf{Pass} \end{aligned}$$

[2] Comme évoqué plus haut, on peut éviter le problème des traces infinies dues aux boucles en utilisant un temporisateur global

$$verdict(\sigma) = \textit{Inconc} \quad \triangleq \quad TC \textit{ after } \sigma \subseteq \textbf{Inconc}$$

Ayant défini le verdict d'une exécution particulière d'un cas de test sur une IUT, on peut regarder l'ensemble des exécutions possibles et les verdicts associés pour essayer de définir un verdict global. Malheureusement, à cause du non déterminisme observable de l'implémentation, un même cas de test peut produire plusieurs exécutions possibles sur une même IUT, et potentiellement avec des verdicts différents. Une IUT peut être rejetée ou acceptée ou encore produire un verdict inconclusif pour le même cas de test, suivant les différentes exécutions possibles de $TC \parallel \Delta(IUT)$, comme illustré par la figure 3.9.

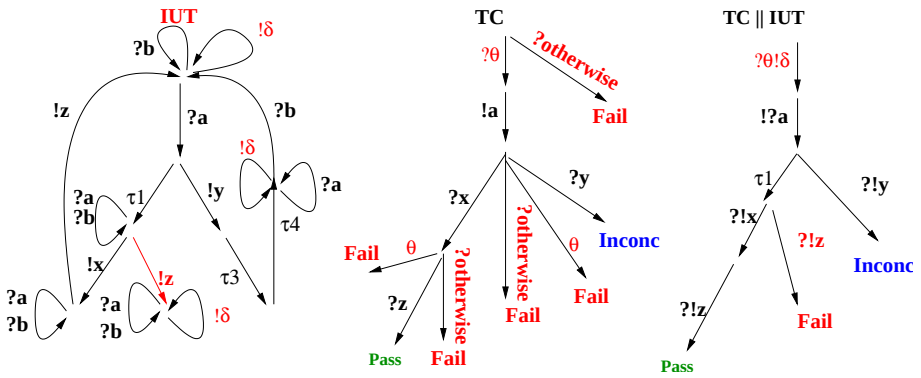



FIG. 3.9 – Exécution d'un cas de test et verdicts

On parlera plutôt de possibilité de rejet, de possibilité de passer un test ou de possibilité de produire un inconclusif :

**Définition 3.1.11 (Rejet, etc)**
*Soient $IUT$ une implémentation et $TC$ un cas de test,*

$$\begin{aligned}
TC \textit{ may reject } IUT &\triangleq \exists \sigma \in Traces(TC \parallel \Delta(IUT)),\ verdict(\sigma) = \textit{Fail} \\
TC \textit{ may pass } IUT &\triangleq \exists \sigma \in Traces(TC \parallel \Delta(IUT)),\ verdict(\sigma) = \textit{Pass} \\
TC \textit{ may inconc } IUT &\triangleq \exists \sigma \in Traces(TC \parallel \Delta(IUT)),\ verdict(\sigma) = \textit{Inconc}
\end{aligned}$$

**Note:** *Tretmans dans [77] n'a pas fait les mêmes choix. D'abord il n'a pas de notion d'inconclusif. Les verdicts sont donc exclusivement Fail ou Pass. Pour lui, une IUT passe un cas de test si toutes les exécutions produisent un verdict Pass. Par opposition, une IUT est rejetée si elle ne passe par le test, donc s'il existe une exécution qui mène à* **Fail**, *ce qui correspond à notre définition de $may\ reject$. Notre définition est donc plus fine en ce qui concerne la possibilité d'obtenir un verdict Pass.*

**Propriétés des cas de test :** Nous avons défini pour l'instant les propriétés structurelles des cas de test, mais nous n'avons pas encore relié ces cas de tests à la relation de conformité. Evidemment, un cas de test est supposé nous renseigner sur la conformité de l'implémentation vis à vis de sa spécification. Il doit donc avoir des propriétés relatives à la conformité. Intuitivement, il faut qu'un verdict **Fail** implique la non-conformité. Inversement, toute non conformité doit pouvoir être détectée par un cas de test (sinon, il serait raisonnable d'affaiblir la relation conformité). Nous faisons ce lien dans les définitions suivantes :

**Définition 3.1.12 (Non-biais, exhaustivité, complétude)**
*Soit $\mathcal{I}_S$ l'ensemble des implémentations compatibles avec une spécification $S$.*
*Un cas de test $TC$ est dit* non biaisé *pour $S$ et* **ioco** *si et seulement si*

$$\forall IUT \in \mathcal{I}_S,\ IUT \textbf{ ioco } S \Rightarrow \neg(TC\ may\ reject\ IUT) \tag{3.1}$$

*Une suite de tests est non biaisée si tous ses cas de tests sont non biaisés.*
*Une suite de test $TS$ est dite* exhaustive *pour $S$ et* **ioco** *si et seulement si*

$$\forall IUT \in \mathcal{I}_S, \neg(IUT \textbf{ ioco } S) \Rightarrow \exists TC \in TS, TC\ may\ reject\ IUT \tag{3.2}$$

*Une suite de test est* complète *si elle est non biaisée et exhaustive.*

Le *non biais* d'une suite de tests $TS$ vis à vis de $S$ et **ioco** signifie que seules les IUTs non conformes doivent être rejetées. En effet la formule 3.1 est équivalente à

$$\forall IUT \in \mathcal{I}_S, TC\ may\ reject\ IUT \Rightarrow \neg(IUT \textbf{ ioco } S) \tag{3.3}$$

C'est une propriété minimale et on verra qu'on peut l'obtenir, mais elle ne suffit pas à avoir des tests utiles. En effet, dans le cas extrême où $TS$ accepte toute IUT, elle est non biaisée, or elle n'apporte aucune information.

Inversement, une suite de tests $TS$ est *exhaustive* pour $S$ et **ioco** si elle peut rejeter toute IUT non conforme. L'exhaustivité seule est aussi insuffisante puisqu'un test qui rejette toute IUT est exhaustive en ce sens.

De plus, il est impossible d'atteindre à la fois le non biais et l'exhaustivité pour une suite de test finie aussitôt que la spécification contient des cycles. Il faudrait une infinité de cas de test ou considérer les comportement infinis des cas de tests.

On cherchera donc à obtenir une propriété d'exhaustivité de la technique de synthèse de test, i.e. l'exhaustivité de l'ensemble infini de tous les cas de tests qui peuvent être synthétisés. Cette propriété permet de dire que, si une IUT est non conforme, il est toujours possible de produire un test qui peut la rejeter.

La définition de non biais s'exprime dans la formule 3.1 sur un cas de test individuel, en fonction de la spécification $S$ et pour une relation de conformité donnée, ici **ioco**. La définition est très générale. Elle manque effectivement de précision puisque, dans sa forme équivalente 3.3, elle ne fait pas de lien explicite entre la trace qui rejette l'IUT et la raison de la non conformité. On imagine mal que ce lien n'existe pas, puisque cela signifierait qu'un cas de test serait doté d'un oracle omniscient qui rejetterait une IUT non conforme sans en regarder les traces responsables. Nous donnons donc une condition suffisante de

non-biais portant sur les traces du cas de test menant à **Fail** et qui explicite ce lien. Cette condition s'exprime indépendemment des implémentations, puisque seul son alphabet de sortie est utile.

**Proposition 3.1.8 (Condition suffisante de non-biais)**
*Soit $TC$ un cas de test et $Traces_{Fail}(TC)$ ses traces menant aux états* **Fail**.
*Si $Traces_{Fail}(TC) \subseteq STraces(S).A_O^{\Delta(IUT)} \cap \overline{STraces(S)}$ alors $TC$ est non-biaisé.*

**Preuve :** En effet avec cette condition, toute trace $\sigma$ de $TC$ menant à **Fail** est une trace qui n'est pas dans $STraces(S)$ et se décompose en une trace $\sigma'$ suivie d'une sortie $a$ de $\Delta(IUT)$. L'exécution de $TC$ sur une implémentation $IUT$ mènera donc à **Fail** si $\Delta(IUT)$ a une telle trace donc si $STraces(IUT) \cap STraces(S).A_O^{\Delta(IUT)} \cap \overline{STraces(S)} \neq \emptyset$, qui implique $STraces(IUT) \cap STraces(S).A_O^{\Delta(IUT)} \not\subseteq STraces(S)$.
Par la proposition 3.1.2, on a alors $\neg(IUT$ **ioco** $S)$.
□

En complément des notions de non biais et d'exhaustivité, il nous a semblé intéressant de définir la propriété de non laxisme des cas de tests qui dit en substance qu'un cas de test doit rejeter toute non conformité observée. Cette propriété peut donc être interprétée comme une certaine exhaustivité limitée aux traces du cas de test, en d'autres termes exhaustive sur les IUT non conformes dont les traces sont incluses dans celles de $TC$.

**Définition 3.1.13 (Non-laxisme)**
*Un cas de test $TC$ d'alphabet d'entrée $A_I^{TC}$ est* non laxiste *si $TC$ peut rejeter toute $IUT$ non conforme sur les traces de $TC$. Formellement,*

$$\forall IUT \in \mathcal{I}_S,\ \forall \sigma \in STraces(S), \forall a \in Out(\Delta(IUT)\ \textit{after}\ \sigma) \setminus Out(\Delta(S)\ \textit{after}\ \sigma),$$
$$\sigma.a \in Traces(TC) \Rightarrow TC\ \textit{after}\ \sigma.a \subseteq \textbf{Fail}.$$

On rappelle que la contrainte sur les alphabets des cas de tests par rapport aux alphabets des implémentations : $A_O^{IUT} \cup \{\delta\} = A_I^{TC}$, impose que le non-laxisme d'un cas de test est toujours par rapport à son alphabet d'entrée.

On peut donner une caractérisation des tests non laxistes qui ne dépende que de $S$, et qui dit en substance qu'après une trace suspendue de $S$, le cas de test produit *Fail* sur toute entrée qui n'est pas une sortie autorisée dans $S$ après cette trace :

**Proposition 3.1.9** *Un cas de test $TC$ est non-laxiste si et seulement si*

$$\forall \sigma \in STraces(S), \forall a \in (A_O^{IUT} \cup \{\delta\}) \setminus Out(\Delta(S)\ \textit{after}\ \sigma),$$
$$\sigma.a \in Traces(TC) \Rightarrow TC\ \textit{after}\ \sigma.a \subseteq \textbf{Fail}$$

Pour finir ce paragraphe, notons qu'avec la condition suffisante de non biais et la caractérisation du non-laxisme en fonction de la spécification, on obtient un moyen de vérifier le non biais et le non-laxisme d'un cas de test (manuel), en comparant ses traces aux traces suspendues de la spécification. C'est ce que nous avions exploité dans [46].

> **Note:** *La littérature sur le test provient de communautés distinctes (protocol testing vs. software testing) et qui se sont trop longtemps mal comprises voire*

*ignorées. Les notions de non biais, d'exhaustivité et de complétude ne sont donc malheureusement pas standard. Pire, les mêmes termes sont parfois utilisés pour des notions différentes. Les discussions et la compréhension entre les communautés ne s'en trouvent donc pas facilitées. Par exemple, Tretmans utilise les termes "exhaustif" et "complet" dans le même sens qu'ici, mais préfère le terme "correct" (*sound*) à non biaisé. Bernot, Gaudel et Marre [4] dans la théorie du test de spécifications algébriques, utilisent "non biais" comme ici, mais utilisent le terme "valide" pour "exhaustif". "exhaustif" implique "complet" mais désigne la suite de test composé de tous les termes clos des axiomes de l'algèbre. En test structurel, le terme "idéal" correspond à "non biaisé" et "complet" a un tout autre sens, relatif à un critère de couverture.*

## 3.2 Algorithmes de synthèse de cas de tests

Dans cette section, nous décrivons trois algorithmes de génération de test. Le premier est l'algorithme de Tretmans [77] qui opère la sélection des tests de manière non déterministe. Les autres algorithmes utilisent des objectifs de test pour guider la sélection. Le deuxième utilise des objectifs de test décrivant des traces suspendues de la spécification. Enfin le troisième utilise des objectifs avec des actions internes, qui autorisent une sélection plus fine sur la base des comportements de $S$ visibles ou non. C'est ce troisième algorithme qui est implémenté dans TGV.

### 3.2.1 Un algorithme de synthèse non déterministe

Dans ce paragraphe, nous expliquons l'algorithme de synthèse de tests de [77]. L'algorithme a l'avantage d'être extrêmement simple et de surcroît, de mon point de vue au moins, facilite la compréhension des algorithmes de TGV.

Si on reprend la définition de **ioco** (voir définiton 3.1.8), la conformité est basée sur la comparaison des sorties de $IUT$ et $S$ après les traces suspendues de $S$, donc les séquences de $det(\Delta(S))$. L'algorithme est donc naturellement fondé sur un parcours de $det(\Delta(S))$. L'IOLTS $det(\Delta(S))$ n'est pas construit entièrement, mais est seulement parcouru durant la génération.

Il reste alors à déterminer un moyen de sélection des tests parmi les comportements de $det(\Delta(S))$. La solution choisie dans cet algorithme, est de baser la sélection des tests sur le *non déterminisme*, au sens où le choix entre entrées et sorties du test, arrêt de l'exploration, et le choix parmi les sorties du test (contrôlables) sont non déterministes. Ce non déterminisme est implémenté dans l'outil TorX [18] par un tirage aléatoire.

Nous exprimons ici cet algorithme de manière itérative et dans le modèle IOLTS. Nous appelerons *gen_test* cet algorithme.

Soit $S = (Q^{\text{S}}, A^{\text{S}}, \rightarrow_{\text{S}}, q_0^{\text{S}})$ une spécification, $\Delta(S)$ son automate de suspension. Le cas de test $TC = (Q^{\text{TC}}, A^{\text{TC}}, \rightarrow_{\text{TC}}, q_0^{\text{TC}}) = gen_test(S)$, avec $A^{\text{TC}} = A_{\text{I}}^{\text{TC}} \cup A_{\text{O}}^{\text{TC}}$ et $A_{\text{O}}^{\text{TC}} \subseteq A_{\text{I}}^{\text{S}}$ $A_{\text{I}}^{\text{TC}} = A_{\text{O}}^{\text{IUT}} \cup \{\delta\}$ et muni de ses ensembles d'états **Fail** et **Pass**, est construit comme suit :

**Initialisation :** $Q^{\text{TC}} = \{P_0^{\text{TC}},\ Fail\}$ où $P_0^{\text{TC}} = q_0^{\text{S}}$ after $\varepsilon$ et $Fail$ est un nouvel état n'appartenant pas à $2^{Q_{\text{S}}}$, $\rightarrow_{\text{TC}} = \emptyset$, **Fail** $= \{Fail\}$, **Pass** $= \emptyset$.

**Tant qu'** il reste des états feuilles dans $TC$ n'appartenant ni à **Pass** ni à **Fail**,
choisir un de ces méta-états $P$ et choisir de façon non déterministe une des règles suivantes, si elle est applicable (la première est toujours applicable) :

1. **Arrêt : Pass** $:=$ **Pass** $\cup \{P\}$,

2. **Choix d'une entrée :** si $In_{\Delta(\text{S})}(P) \neq \emptyset$, choisir une de ces entrées $x$ et faire $Q^{\text{TC}} := Q^{\text{TC}} \cup \{P \text{ after } x\}$ ; $\rightarrow_{\text{TC}} := \rightarrow_{\text{TC}} \cup \{(P, x, P \text{ after } x)\}$.

3. **Toutes les sorties :** si $Out_{\Delta(\text{S})}(P) \neq \emptyset$, faire
$Q^{\text{TC}} := Q^{\text{TC}} \cup \bigcup_{a \in Out_{\Delta(\text{S})}(P)} \{P \text{ after } a\}$ ;
$\rightarrow_{\text{TC}} := \rightarrow_{\text{TC}} \cup \bigcup_{a \in Out_{\Delta(\text{S})}(P)} \{(P, a, P \text{ after } a)\}$
$\cup \bigcup_{a \in A_O^{\Delta(IUT)} \setminus Out_{\Delta(\text{S})}(P)} \{(P, a, Fail)\}$

Remarquer que *after* a pour effet de déterminiser $\Delta(S)$, donc de parcourir $det(\Delta(S))$. Notons aussi qu'aucun verdict **Inconc** n'est produit, cette notion étant liée aux objectifs de test, non utilisés ici. Noter enfin que $A_{\text{O}}^{\Delta(IUT)} \setminus Out_{\Delta(\text{S})}(P)$ n'étant *a priori* pas connu, ses éléments sont représentés par l'abbréviation $otherwise$. On note donc $?otherwise$ la réception de ces sorties.

La figure 3.10 décrit deux cas de test produits par l'algorithme $gen_test$. Dans $TC1$, on choisit d'abord d'émettre $a$, puis d'attendre une réponse, i.e. de recevoir $y$ ou d'observer un blocage $\delta$, ou toute autre émission non conforme de l'IUT menant au verdict *Fail*. Après $y$ ou $\delta$ le calcul continue, jusqu'à ce qu'on décide d'arrêter sur un verdict *Pass*. Dans $TC2$ le choix initial est différent puisqu'on décide ici d'attendre une entrée, donc la réception de $x$, la détection d'un blocage ou n'importe quelle autre émission non conforme. Noter qu'on peut sans problème appliquer l'algorithme à un IOLTS ayant un livelock comme sur l'exemple (états 3 et 5), alors que Tretmans ne les considère pas.

Le théorème suivant est démontré dans [77]

**Théorème 3.2.1** *Pour $S$ donné, l'ensemble (infini) des cas de tests que peut produire l'algorithme* gen_test *est* non biaisé *et* exhaustif.

**Preuve :** Nous n'allons pas donner ici la preuve complète mais juste ses principes. D'après la proposition 3.1.8 montrer le non biais (ou correction, i.e. seules les IUT non conformes sont rejetées) revient à montrer que l'algorithme, ne produit des *Fail* que sur des sorties et blocages non prévues dans $S$ après des traces suspendues de $S$. C'est ce que réalise la règle 3 de l'algorithme.

Pour l'exhaustivité (toute IUT non conforme peut être rejetée par un test), on se donne une IUT non conforme. Par définition de **ioco**, il existe une trace suspendue $\sigma \in STraces(S)$ telle que l'IUT possède une sortie $a$ ou un blocage $\delta$ non autorisée dans $S$. Il suffit alors d'exhiber une exécution de l'algorithme contrainte par $\sigma$ (faisant les choix entrées/sorties en fonction de $\sigma$), terminant par l'observation des sorties, et de constater qu'observer $a$ ou $\delta$ produit *Fail*. Il existe alors une exécution du cas de test produit

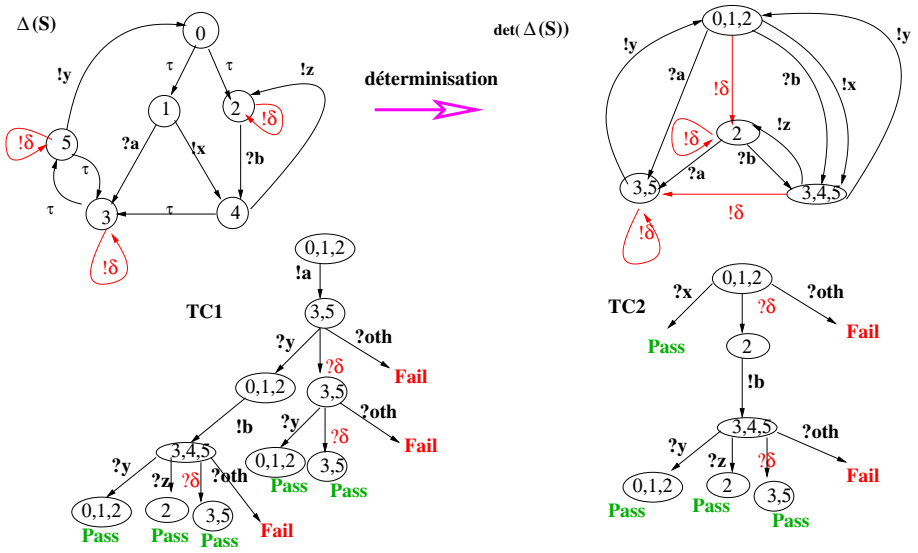



FIG. 3.10 – Algorithme de génération de tests non déterministe : exemple

sur l'IUT qui la rejette.
□

### 3.2.2 Algorithme de génération avec objectifs de test sur les traces suspendues

Dans l'algorithme précédent, la sélection des cas de test est effectuée par la résolution des choix non déterministes. Le problème de la sélection n'est donc pas vraiment traité car il n'y a aucun moyen de cibler les comportements à tester.

Or, en pratique (dans une suite de test TTCN [42] par exemple), les cas de test sont relatifs à des *objectifs de test*, qui décrivent informellement des comportements à tester. Nous avons remarqué qu'en général ces objectifs décrivent une séquence incomplète d'actions, i.e. des séquences dont les actions sont ordonnées causalement mais qui ne décrivent pas nécessairement toutes les actions visibles.

Un des apports principaux de TGV a été de formaliser la notion d'*objectif de test* et de les utiliser pour la sélection des tests. Nous avons choisi de modéliser les objectifs de test par des automates d'état finis (des IOLTS étendus). Dans un premier temps, nous allons expliquer les principes de l'algorithme de génération de tests en considérant des objectifs de test reconnaissant des traces suspendues de la spécification. On peut alors voir cet algorithme comme un raffinement de l'algorithme précédent.

La plupart des auteurs qui les utilisent [34, 19], considèrent les objectifs de test comme une description plus ou moins abstraite des tests. Dans les premiers articles sur TGV, nous avions également adopté ce point de vue [26]. De fait, les objectifs décrivent alors des interactions entre le système et son environnement, et sont donc des traces de $S$ (ou de

$\Delta(S)$). Formellement, ce sont des langages de traces sur les alphabets $(A^{\mathrm{S}}_{\mathrm{VIS}})^*$ ou $(A^{\mathrm{S}}_{\mathrm{VIS}} \cup \{\delta\})^*$. Rien de plus naturel que de choisir alors des automates pour décrire les objectifs.

**Une première formalisation des objectifs de test**

Pour permettre une sélection efficace, nous utilisons deux ensembles distincts d'états marqués servant, soit à accepter, soit à refuser des séquences d'actions de la spécification. Refuser des séquences d'actions permet de limiter l'exploration des comportements de la spécification, ce qui est particulièrement utile pour la génération à la volée (voir 3.3).

**Définition 3.2.1 (Objectif de test externes)**
*Un* objectif de test externe *est un IOLTS* $TP = (Q^{TP}, A^{TP}, \rightarrow_{TP}, q_0^{TP})$ *(TP pour* Test Purpose*) déterministe et* complet *muni de deux ensembles d'*états puits *disjoints* $Accept^{TP} \subseteq Q^{TP}$ *et* $Refuse^{TP} \subseteq Q^{TP}$*, et dont l'alphabet consiste en les actions visibles et blocages de la spécification i.e.* $A^{TP} = A^{S}_{VIS} \cup \{\delta\}$. Complet *signifie que dans tout état, toute action est tirable i.e.* $\forall q \in Q^{TP}, \forall a \in A^{TP}, q \xrightarrow{a}_{TP}$. *Un état* $q$ *est un* puits *s'il boucle pour toute action i.e.* $\forall a \in A^{TP}, q \xrightarrow{a}_{TP} q$.

Un objectif de test est donc ici un automate d'états fini sur l'alphabet des actions visibles de la spécification $A^{\mathrm{S}}_{\mathrm{I}} \cup A^{\mathrm{S}}_{\mathrm{O}} \cup \{\delta\}$. Il définit deux langages de traces reconnus qu'on notera

$$\begin{aligned} Traces_{\mathrm{Accept}}(TP) &= \{\sigma \in (A^{\mathrm{S}}_{\mathrm{VIS}} \cup \{\delta\})^* \mid \exists q_A \in Accept^{\mathrm{TP}},\ q_0^{\mathrm{TP}} \xrightarrow{\sigma}_{\mathrm{TP}} q_A\} \\ Traces_{\mathrm{Refuse}}(TP) &= \{\sigma \in (A^{\mathrm{S}}_{\mathrm{VIS}} \cup \{\delta\})^* \mid \exists q_R \in Refuse^{\mathrm{TP}},\ q_0^{\mathrm{TP}} \xrightarrow{\sigma}_{\mathrm{TP}} q_R\} \end{aligned}$$

Noter que ces deux langages sont disjoints car $Accept^{\mathrm{TP}}$ et $Refuse^{\mathrm{TP}}$ sont disjoints et que $TP$ est déterministe. Noter aussi que les deux langages sont suffixe-clos (tout prolongement d'une séquence acceptée/refusée est acceptée/refusée) car les états de $Accept^{\mathrm{TP}}$ et $Refuse^{\mathrm{TP}}$ sont des états puits. Du coup, ces langages de traces sont déterminés par leurs minimaux pour l'ordre préfixe. On utilisera cette propriété de clôture par suffixe pour arrêter la construction d'une trace de la spécification aussitôt qu'un préfixe est accepté. On s'intéresse donc seulement aux ensembles des traces minimales de $T$ pour l'ordre préfixe $min(Traces_{\mathrm{Accept}}(TP))$ et $min(Traces_{\mathrm{Refuse}}(TP))$ et on a les propriétés

$$\begin{aligned} Traces_{\mathrm{Accept}}(TP) &= min(Traces_{\mathrm{Accept}}(TP)).(A^{\mathrm{TP}})^* \\ Traces_{\mathrm{Refuse}}(TP) &= min(Traces_{\mathrm{Refuse}}(TP)).(A^{\mathrm{TP}})^* \end{aligned}$$

et ce sont les plus petits langages ayant ces propriétés.

Dans la suite, on utilisera principalement $min(Traces_{\mathrm{Accept}}(TP))$.

Concernant l'hypothèse de complétude de $TP$, une alternative aurait été de n'exiger la complétude de $TP$ qu'en dehors de $Accept^{\mathrm{TP}}$ et $Refuse^{\mathrm{TP}}$ et de définir les états $Accept^{\mathrm{TP}}$ et $Refuse^{\mathrm{TP}}$ comme des états sans successeurs. Une raison de notre choix est que, pour toute trace suspendue acceptée, toute prolongation par un postambule reste acceptée, grâce à la propriété de clôture par suffixe. Une autre raison de ce choix apparaîtra aussi clairement dans le cas des objectifs ayant des actions internes.

**Abstraction dans les objectifs de test** Il est important de permettre une certaine abstraction dans la description des objectifs de test par rapport aux comportements de la spécification. En effet, la spécification étant donnée dans un langage de description, il n'est pas toujours aisé d'en inférer les comportements en termes du modèle IOLTS, sans les construire tous. Or dans la définition 3.2.1, les automates sont complets, ce qui peut sembler contradictoire avec l'idée d'abstraction. Ça ne l'est pas en réalité pour plusieurs raisons. D'abord, pour rendre complet un automate, il suffit de disposer d'un opérateur de complémentation d'un ensemble d'actions. C'est ce qui est fait dans TGV, dans lequel on a défini une étiquette spéciale notée "*" qui, si elle étiquette une transition $q \xrightarrow{*} q'$, signifie que pour tout $a$ n'étiquetant pas explicitement une transition d'origine $q$, on a $q \xrightarrow{a} q'$. Cette notation permet de mentionner le complémentaire d'un ensemble d'actions sans l'énumérer. Nous utiliserons cette notation dans les exemples. De plus, par défaut, un objectif de test incomplet est complété implicitement par une boucle dans le même état étiquetée par "*", ce qui permet de décrire des séquences incomplètes d'actions.

**Exemple :** Reprenons l'exemple de la figure 3.10 dont les comportements visibles sont représentés en haut à gauche de la figure 3.11. Imaginons que nous souhaitions tester le système dans les comportements où il exhibe un blocage $!\delta$ et plus tard une émission $!y$, mais sans recevoir $?a$ ni émettre $!y$ avant $!\delta$. L'objectif de test serait formalisé par l'automate $TP$ à droite de la figure 3.11.

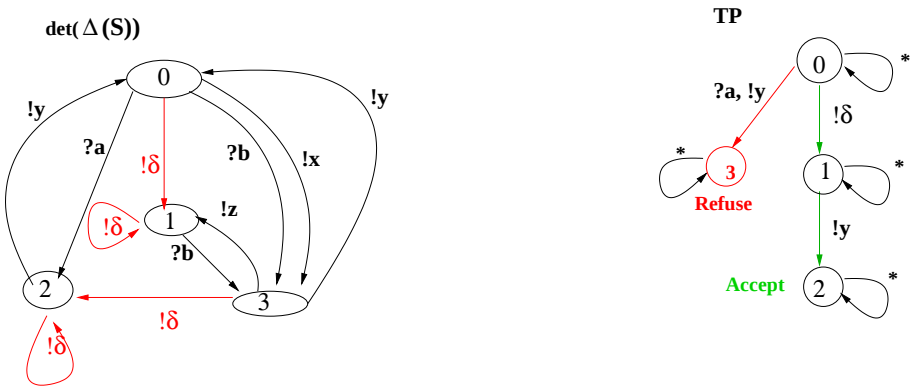



FIG. 3.11 – Spécification et objectif de tests

L'objectif de test reconnait les traces suspendues du langage

$$Traces_{\text{Accept}}(TP) = ((A^{\text{S}}_{\text{VIS}} \cup \{\delta\}) \setminus \{a, y, \delta\})^*.\delta.(A^{\text{S}}_{\text{I}} \cup A^{\text{S}}_{\text{O}} \cup \{\delta\})^*.y.(A^{\text{S}}_{\text{VIS}} \cup \{\delta\})^*$$

et refuse les traces suspendues du langage

$$Traces_{\text{Refuse}}(TP) = ((A^{\text{S}}_{\text{I}} \cup A^{\text{S}}_{\text{O}} \cup \{\delta\}) \setminus \{\delta\})^*.(a + y).(A^{\text{S}}_{\text{VIS}} \cup \{\delta\})^*$$

### Algorithme de génération de tests

Nous décrivons maintenant un algorithme de génération de tests basés sur les objectifs de test externes. Nous appelerons *gen_test_OT1* ce premier algorithme. Les entrées de l'algorithme de génération de test *gen_test_OT1* sont la spécification $S$ qui sert d'oracle, et un objectif de test $TP$ qui sert à la sélection, et on souhaite obtenir en sortie un cas de test $TC = gen_test_OT1(S, TP)$ qui soit au moins correct pour **ioco** et qui focalise le test sur les comportements visibles décrits par $TP$.

L'algorithme *gen_test_OT1* est composé d'un enchaînement d'opérations résumé par la figure 3.12. On commence par calculer l'automate suspendu de $S$ et le déterminiser pour obtenir les comportements visibles $\det(\Delta(S))$. Parmi ces traces suspendues, on souhaite extraire celles qui sont acceptées par $TP$. On effectue alors un produit synchrone avec l'objectif de test $TP$. Le résultat $PS^{\text{VIS}}$ a les mêmes traces que $det(\Delta(S))$, car $TP$ est complet, mais il peut effectuer un dépliage et étiquette certains états par $Accept$ ou $Reject$. L'étape de sélection consiste alors à en extraire un *graphe de test complet* $CTG$. Le graphe de test complet $CTG$ est essentiellement le sous-IOLTS contenant les comportements acceptés minimaux au sens du préfixe. De plus ses entrées et sorties sont inversées par rapport à $PS^{\text{VIS}}$ et il est complété en entrée vers **Fail** afin de déterminer la non-conformité. La dernière étape consiste encore à sélectionner dans $CTG$ un sous-IOLTS assurant la propriété de contrôlabilité. Une alternative à la construction de $CTG$ est de commencer à résoudre les problèmes de contrôlablité durant la sélection pour produire un *graphe de test* $TG$, ce qui réduit mais ne supprime pas la dernière étape.

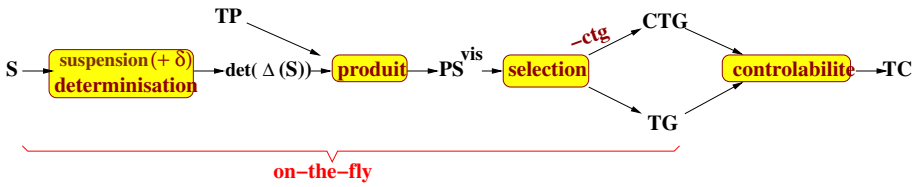



FIG. 3.12 – Résumé des opérations de l'algorithme de synthèse de test *gen_test_OT1*

**Calcul des blocages et déterminisation** Comme $TP$ reconnait des traces sur les actions visibles et blocages de $S$, la première opération consiste à calculer $det(\Delta(S))$. Nous ne détaillons pas cette opération qui est la composition des opérations décrites par les définitions 3.1.6 et 3.1.5. Nous reviendrons plus tard sur son implémentation à la volée. Rappelons juste que le résultat est un IOLTS $det(\Delta(S)) = (Q^{\text{det}}, A_{\text{I}}^{\text{det}} \cup A_{\text{O}}^{\text{det}}, \rightarrow_{\text{det}}, q_0^{\text{det}})$ avec $A_{\text{I}}^{\text{det}} = A_{\text{I}}^{\text{S}}$ et $A_{\text{O}}^{\text{det}} = A_{\text{O}}^{\text{S}} \cup \{\delta\}$ dont le langage de traces généré est exactement l'ensemble des traces suspendues de $S$ : $Traces(det(\Delta(S))) = Traces(\Delta(S)) = STraces(S)$.

L'IOLTS $det(\Delta(S))$ à gauche de la figure 3.11 représente le résultat de cette opération pour la spécification déjà utilisée à la figure 3.10.

**Produit synchrone :** Ensuite, il faut extraire de $det(\Delta(S))$ les traces suspendues acceptées par $TP$ i.e. calculer $STraces(S) \cap Traces_{\text{Accept}}(TP)$. Plus exactement, ces traces min-

imales pour l'ordre préfixe, $STraces(S) \cap min(Traces_{\text{Accept}}(TP))$. Comme d'habitude, l'intersection de langages s'effectue par l'opération classique de produit entre automates $det(\Delta(S)) \times TP$, en considérant tout état de $det(\Delta(S))$ comme état marqué. Le produit a pour effet de déplier $det(\Delta(S))$ et d'étiqueter par $Accept$ ou $Refuse$ certains états du produit, caractérisant ainsi les traces suspendues acceptées ou refusées.

**Définition 3.2.2 (Produit synchrone (I))**
*Soient $S$ un IOLTS et son automate suspendu déterminisé $det(\Delta(S)) = (Q^{det}, A^S_{VIS} \cup \{\delta\}, \rightarrow_{det}, q_0^{det})$ et $TP = (Q^{TP}, A^S_{VIS} \cup \{\delta\}, \rightarrow_{TP}, q_0^{TP})$ un IOLTS déterministe et complet sur $A^S_{VIS} \cup \{\delta\}$ et muni des ensembles d'états puits $Accept^{TP}$ et $Refuse^{TP}$.*
*Le produit synchrone $det(\Delta(S)) \times TP$ est l'IOLTS noté $PS^{VIS} = (Q^{VIS}, A^{VIS}, \rightarrow_{VIS}, q_0^{VIS})$, muni de deux ensembles disjoints d'états $Accept^{VIS}$ et $Refuse^{VIS}$, et défini comme suit :*
- *son ensemble d'états est $Q^{VIS} \triangleq Q^{det} \times Q^{TP}$*
- *son alphabet est $A^{VIS} \triangleq A_I^{VIS} \cup A_O^{VIS}$ avec $A_I^{VIS} \triangleq A_I^S$ et $A_O^{VIS} \triangleq A_O^S \cup \{\delta\}$.*
- *l'état initial est $q_0^{VIS} \triangleq (q_0^{det}, q_0^{TP})$,*
- *la relation de transition $\rightarrow_{VIS}$ est définie par :*
  $(q^{det}, q^{TP}) \xrightarrow{a}_{VIS} (q'^{det}, q'^{TP}) \iff q^{det} \xrightarrow{a}_{det} q'^{det} \wedge q^{TP} \xrightarrow{a}_{TP} q'^{TP}$.
- *$Accept^{VIS} \triangleq Q^{det} \times Accept^{TP}$ et $Refuse^{VIS} \triangleq Q^{det} \times Refuse^{TP}$*

Puisque $TP$ est complet, $PS^{\text{VIS}} = det((\Delta(S))) \times TP$ est bisimilaire à $det(\Delta(S))$ (intuitivement il préserve tous les comportements). Comme $det((\Delta(S)))$ et $TP$ sont déterministes, $PS^{\text{VIS}}$ est aussi déterministe. La bisimilarité est donc l'équivalence de traces $Traces(PS^{\text{VIS}}) = STraces(S)$.

De plus, les traces suspendues de $S$ acceptées (resp. refusées) par $TP$ sont exactement les séquences acceptées (resp. refusées) par $PS^{\text{VIS}}$ sur ses états Accept (resp Refuse) i.e.

$$\begin{aligned} Traces_{\text{Accept}}(PS^{\text{VIS}}) &= STraces(S) \cap Traces_{\text{Accept}}(TP) \\ Traces_{\text{Refuse}}(PS^{\text{VIS}}) &= STraces(S) \cap Traces_{\text{Refuse}}(TP) \end{aligned}$$

De même pour les traces minimales pour l'ordre préfixe :

$$\begin{aligned} min(Traces_{\text{Accept}}(PS^{\text{VIS}})) &= STraces(S) \cap min(Traces_{\text{Accept}}(TP)) \\ min(Traces_{\text{Refuse}}(PS^{\text{VIS}})) &= STraces(S) \cap min(Traces_{\text{Refuse}}(TP)). \end{aligned}$$

Noter qu'on peut restreindre $Q^{\text{VIS}}$ à $reach(q_0^{\text{VIS}})$ i.e. aux états accessibles de $q_0^{\text{VIS}}$ sans modifier ces langages.

La construction de $PS^{\text{VIS}} = det(\Delta(S)) \times TP$ est illustrée par la figure 3.13. Dans cet exemple, la construction a été arrêtée dans les états $Accept$ et $Refuse$ puisqu'on ne s'intéresse au final qu'aux traces acceptées minimales pour l'ordre préfixe. En réalité, il suffirait de recopier $det(\Delta(S))$ à partir des états $Accept$ (0,2) et $Refuse$ (0,3) pour obtenir le graphe complet puisque dans ces deux états $det(\Delta(S))$ est dans son état initial.

**Sélection :** Le langage de traces accepté de $PS^{\text{VIS}}$, minimal pour l'ordre préfixe, noté $min(Traces_{\text{Accept}}(PS^{\text{VIS}}))$ *caractérise* les traces suspendues minimales de $S$ acceptées par $TP$, donc les comportements visibles de $S$ visés par l'objectif de test. Encore faut-il *extraire* de $PS^{\text{VIS}}$ ces traces suspendues.

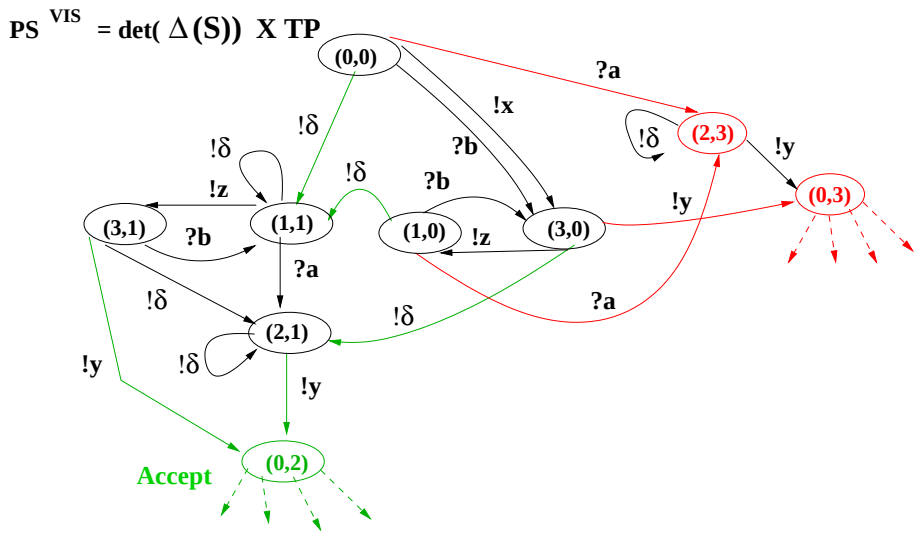



FIG. 3.13 – Produit synchrone $PS^{\text{VIS}} = det(\Delta(S)) \times TP$

La sélection consiste donc essentiellement à extraire de $PS^{\text{VIS}}$, le sous-IOLTS contenant ces traces suspendues acceptées minimales, en ne conservant que les états *utiles* à une trace acceptée. On notera $trim(PS^{\text{VIS}})$ cet IOLTS, par analogie à la théorie du contrôle qui définit un objet similaire (voir par exemple [9]). A partir de $trim(PS^{\text{VIS}})$, on définit un IOLTS $CTG$ pour *graphe de test complet* (il contient tous les tests pour $S$ et $TP$). $CTG$ contient $trim(PS^{\text{VIS}})$, mais inverse les entrées et sorties pour considérer le point de vue du testeur. De plus il complète tous les états en entrées, en distinguant les entrées admissibles (sorties prévues dans $S$ mais ne permettant pas de satisfaire l'objectif) et qui produisent des verdicts *Inconc* de celles qui sont pas admissibles et qui produisent des verdicts *Fail*. Le paragraphe suivant explique enfin comment extraire de $CTG$ des cas de tests.

Dans la définition classique de l'automate $trim$ [9]), l'*émondage* préserve exactement les traces acceptés. Il suffit de remarquer que les états *utiles* sont $reach(q_0^{\text{VIS}}) \cap coreach(Accept^{\text{VIS}})$. Par définition, ces états sont en effet tous les états qui peuvent être parcourus depuis l'état initial jusqu'à un état accepteur, donc sur une trace acceptée.

Dans notre cas, nous souhaitons conserver uniquement les traces *minimales* pour l'ordre préfixe. Une trace minimale s'arrête donc au premier état $Accept$ rencontré. Pour définir les états de l'IOLTS $trim(PS^{\text{VIS}})$ voulu, il ne faut donc pas traverser les états $Accept$. Si on note $\overline{P}$ le complémentaire de l'ensemble $P$, l'ensemble $reach_{|\overline{Accept^{\text{VIS}}}}(q_0^{\text{VIS}})$ est l'ensemble des états accessibles depuis $q_0^{\text{VIS}}$ à travers des états n'appartenant pas à $Accept^{\text{VIS}}$, sauf éventuellement le dernier. Les états de $Accept^{\text{VIS}}$ atteints sont donc les minimaux, i.e. ceux dont un prédécesseur n'est pas dans $Accept^{\text{VIS}}$. On définit donc l'automate $trim(PS^{\text{VIS}})$ comme suit

**Définition 3.2.3 (Automate émondé ou *trim*)**
*Soit $PS^{\text{VIS}} = (Q^{\text{VIS}}, A^{\text{VIS}}, \rightarrow_{\text{VIS}}, q_0^{\text{VIS}})$. Le sous-IOLTS de $PS^{\text{VIS}}$ noté*

$trim(PS^{VIS}) = (Q^{trim}, A^{trim}, \rightarrow_{trim}, q_0^{trim})$ *est défini par*

- $Q^{trim} = L2A = reach_{|\overline{Accept^{VIS}}}(q_0^{VIS}) \cap coreach(Accept^{VIS})$
  *i.e. ses états sont les états de* $Q^{VIS}$ *qui sont à la fois* accessibles *depuis* $q_0^{VIS}$ *à travers des états n'appartenant pas à* $Accept^{VPS}$ *et* coaccessibles *depuis* $Accept^{VIS}$*,*
- *si* $Q^{trim} \neq \emptyset$ *alors* $q_0^{trim} = q_0^{VIS}$*,*
- $\rightarrow_{trim} = \rightarrow_{VIS} \cap (Q^{trim} \setminus Accept^{VIS}) \times A^{VIS} \times Q^{trim}$ *i.e. les transitions sont les transitions de* $PS^{VIS}$ *entre ces états mais dont l'origine n'est pas dans* $Accept^{VIS}$*).*

Dans notre exemple, figure 3.13, $trim(PS^{\text{VIS}})$ correspond à la partie verte du graphe dont les états sont $(0,0)$, $(1,1)$, $(2,1)$, $(3,1)$, $(1,0)$, $(3,0)$, $(0,2)$.

On peut se convaincre aisément que l'ensemble des traces maximales finies de cet IOLTS est exactement le langage $min(Traces_{\text{Accept}}(PS^{\text{VIS}}))$.

La définition de $trim(PS^{\text{VIS}})$ permet d'obtenir les traces voulues, mais ce n'est pas encore un cas de test au sens de la définition 3.1.9. D'abord, il faudrait inverser les entrées et sorties, car les entrées du cas de test sont les sorties et blocages de la spécification, et les sorties sont les entrées de la spécification. Ensuite, les états où une entrée du cas de test (sortie de $S$) sont possibles doivent être complets en entrée sur l'alphabet de l'IUT. Enfin, le cas de test doit être contrôlable, c'est à dire qu'il n'a pas le choix entre une sortie et une entrée ou une sortie.

En relâchant la contrainte de contrôlabilité, nous allons définir un IOLTS appelé *graphe de test complet*, noté $CTG$ (pour *Complete Test Graph*). Cet IOLTS contient tous les tests de $S$ correspondant à l'objectif de test $TP$. Il correspond au testeur canonique [8] de $S$ restreint à $TP$ pour la relation de conformité **ioco**.

**Définition 3.2.4 (Graphe de test complet (I))**
*Soit* $PS^{VIS} = det(\Delta(S)) \times TP$*. Le* graphe de test complet *est l'IOLTS* $select(PS^{VIS}) = CTG = (Q^{CTG}, A^{CTG}, \longrightarrow_{CTG}, q_0^{CTG})$*, muni de trois ensembles d'états sans successeur* **Pass**, **Inconc** *et* **Fail**, *et défini de la façon suivante :*

- *son alphabet est* $A^{CTG} = A_O^{CTG} \cup A_I^{CTG}$ *avec* $A_O^{CTG} = A_I^{VIS} = A_I^S$ *et* $A_I^{CTG} = A_O^{IUT} \cup \{\delta\}$ *(i.e. les sorties de CTG sont les entrées de la spécification, ses entrées sont toutes les sorties possibles de l'IUT, y compris les blocages)*
- *son ensemble d'états est* $Q^{CTG} = L2A \cup$ **Inconc** $\cup$ **Fail**, *avec*
  - $L2A = reach_{|\overline{Accept^{VIS}}}(q_0^{VIS}) \cap coreach(Accept^{VIS})$ *(L2A pour* Leads to Accept*) i.e. l'ensemble des états accessibles depuis* $q_0^{VIS}$ *à travers des états n'appartenant pas à* $Accept^{VIS}$ *et coaccesibles depuis* $Accept^{VIS}$*,*
    *Parmi ceux-ci, on définit l'ensemble des états* **Pass** *par*
    **Pass** $\triangleq L2A \cap Accept^{VIS}$.
  - **Inconc** $= post_{A_O^{VIS}}(L2A) \setminus L2A$ *i.e. les états successeurs immédiats d'un état de* $L2A$ *par une sortie dans* $PS^{VIS}$ *(donc une entrée pour* $CTG$*), mais n'appartenant pas eux-mêmes à* $L2A$
  - **Fail** $= \{Fail\}$ *où Fail* $\notin Q^{VIS}$ *est un nouvel état.*
- *si* $L2A \neq \emptyset$*, l'état initial est* $q_0^{CTG} = q_0^{VIS}$ *sinon* $Q^{CTG}$ *est vide.*
- *la relation de transition est* $\rightarrow_{CTG} = \rightarrow_{L2A} \cup \rightarrow_{\textbf{Inconc}} \cup \rightarrow_{\textbf{Fail}}$ *où*
  - $\rightarrow_{L2A} = \rightarrow_{VIS} \cap ((L2A \setminus \textbf{Pass}) \times A^{CTG} \times L2A)$ *i.e. la restriction de* $\rightarrow_{VIS}$ *aux transistions issues de* $L2A \setminus Pass$ *et menant dans* $L2A$*,*

- $\rightarrow_{\textbf{Inconc}} = \rightarrow_{VIS} \cap ((L2A \setminus \textbf{Pass}) \times A_I^{CTG} \times \textbf{Inconc})$, *i.e. la restriction de* $\rightarrow_{VIS}$ *aux transitions étiquetées par une entrée pour CTG (une sortie ou un blocage de* $S$*) et issues de* $L2A \setminus$ **Pass**, *et manant dans* **Inconc**.
- $\rightarrow_{\textbf{Fail}} = \{(q, a, Fail) \mid q \in L2A \setminus \textbf{Pass} \wedge a \in A_I^{CTG} \wedge q \not\xrightarrow{a}_{VIS}\}$ *ce qui signifie que pour rendre* $CTG$ *complet en entrée et lui permettre de détecter les non conformités, dans chaque état de* $L2A \setminus$ **Pass**, *et pour toute sortie ou blocage non permise dans* $PS^{VIS}$ *(entrée dans CTG), on ajoute une transition vers* **Fail**. *Pour un état* $q$, *on résumera l'ensemble des transitions vers Fail par la notation* $?otherwise$.

Le graphe de test complet $CTG$ est très proche de l'IOLTS $trim(PS^{\text{VIS}})$. En effet en dehors de l'inversion des entrées/sorties, $trim(PS^{\text{vis}})$ est exactement le sous IOLTS de $CTG$ restreint aux états de $L2A$. La différence se situe dans les états *Inconc* et *Fail* et les transitions qui y mènent. La raison est que le graphe de tests doit être complet en entrée. S'il existe une sortie de $S$ qui mène dans un état hors de $L2A$, cette sortie est valide dans $S$ et doit être considérée comme entrée valide du cas de test, puisqu'on ne peut l'interdire. Or, il est impossible en permettant cette entrée, d'atteindre $Accept^{\text{VIS}}$, puisque l'état d'atrrivée n'est pas dans $L2A$. D'où le verdict $Inconclusif$. Enfin, toutes les entrées possibles et non spécifiées, i.e. les sorties de l'IUT non prévues dans $S$, doivent être considérées pour la complétion en entrée, et donc mener à un verdict *Fail*.

La construction de $CTG$ pour notre exemple, est illustrée par la figure 3.14.

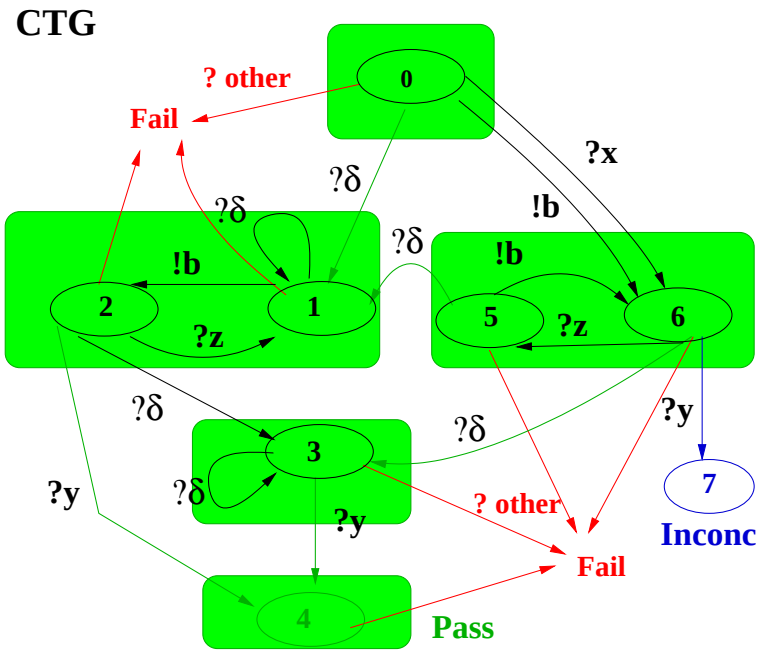



FIG. 3.14 – Graphe de test complet

En termes de langages, si on note $Traces_{\text{Pass}}(CTG)$ les traces suspendues re-

connues dans les états **Pass**, $Traces_{\text{Inconc}}(CTG)$ celles reconnues dans les états **Inconc**, et $Traces_{\text{Fail}}(CTG)$ celles reconnues dans les états **Fail**, on a les caractérisations suivantes :

$$\begin{aligned} Traces_{\text{Pass}}(CTG) &= min(Traces_{\text{Accept}}(PS^{\text{VIS}})) \\ &= STraces(S) \cap min(Traces_{\text{Accept}}(TP)) \end{aligned}$$

i.e. le langage des traces menant aux états **Pass** est composé des traces suspendues de $S$ minimales parmi les traces acceptées par $TP$.

$$\begin{aligned} Traces_{\text{Inconc}}(CTG) &= prf_{<}(Traces_{\text{Pass}}(CTG)).A_{\text{I}}^{\text{CTG}} \cap Traces(PS^{\text{VIS}}) \\ &\quad \cap \overline{prf_{\leq}(Traces_{\text{Pass}}(CTG))} \\ &= prf_{<}(STraces(S) \cap min(Traces_{\text{Accept}}(TP))).(A_{\text{O}}^{\text{S}} \cup \{\delta\}) \end{aligned}$$

i.e. les traces menant aux états **Inconc** sont les préfixes stricts de traces de $S$ acceptées par $TP$ poursuivies par des sorties ou blocages de $S$ possibles dans $S$, mais ne permettant pas de satisfaire l'objectif.

$$\begin{aligned} Traces_{\text{Fail}}(CTG) &= prf_{<}(Traces_{\text{Pass}}(CTG)).(A_{\text{O}}^{\text{IUT}} \cup \{\delta\}) \cap \overline{Traces(PS^{\text{VIS}})} \\ &= prf_{<}(STraces(S) \cap min(Traces_{\text{Accept}}(TP))).(A_{\text{O}}^{\text{IUT}} \cup \{\delta\}) \\ &\quad \cap \overline{STraces(S)} \end{aligned}$$

i.e. les traces menant aux états **Fail** sont les préfixes stricts des traces de $S$ acceptées par $TP$ poursuivies par une sortie ou un blocage de l'IUT non permis dans $S$.

**Élagage des conflits de controlabilité** L'IOLTS $CTG$ possède toutes les propriétés d'un cas de test (Définition 3.1.9), excepté la contrôlabilité : il se peut que dans un état $q$ de $CTG$ on ait le choix entre plusieurs sorties ($!a$ et $!b$ dans l'état 0 de $CTG$) ou entre sorties et entrées (voir la figure 3.15). Noter aussi que $CTG$ est complet en entrée dans tous les états de $L2A \setminus$ **Pass**, alors que dans la définition d'un cas de test, la complétude en entrée n'est exigée que dans les états où une entrée est possible. Forcer la contrôlabilité résoudra cette deuxième différence.

Résoudre ces conflits de contrôlabilité consiste à extraire de $CTG$ un (ou plusieurs) cas de test contrôlables sans remettre en cause les autres propriétés d'un cas de test (i.e. accessibilité à Pass, sauf pour **Inconc** et **Fail**, complétion en entrée, voir définition 3.1.9), et celles acquises par la construction de CTG (en particulier la correction comme nous le verrons plus tard). Dans un état possédant un conflit de contrôlabilité, il faut donc élaguer des transitions de l'une des façons suivantes : soit garder une seule sortie en élaguant les autres sorties et entrées éventuelles (l'initiative est donnée au testeur et il choisit comment contrôler l'$IUT$), soit garder seulement toutes les entrées en élaguant toutes les sorties (l'initiative est donnée à l'$IUT$ et toutes les entrées doivent rester prévues).

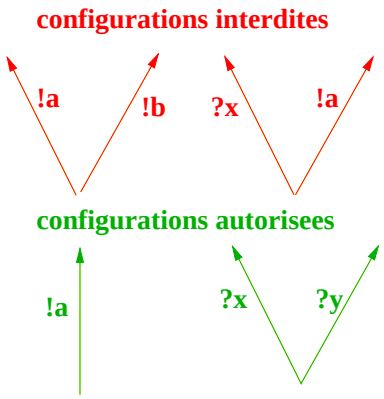



FIG. 3.15 – Conflits de contrôlabilité

Il faut aussi assurer qu'un état qui pouvait mener à un état dans **Pass** dans $CTG$ (les états de $L2A$), s'il est conservé dans le cas de test $TC$, mène toujours à **Pass**. Il faut également que les états restent accessibles depuis l'état initial. Malgré ces contraintes, pour un $CTG$ donné, il existe en général plusieurs cas de test possibles et on n'a plus guère de critères objectifs pour choisir. En effet, les trois IOLTS de la figure 3.16 sont des cas de test possibles obtenus depuis $CTG$.

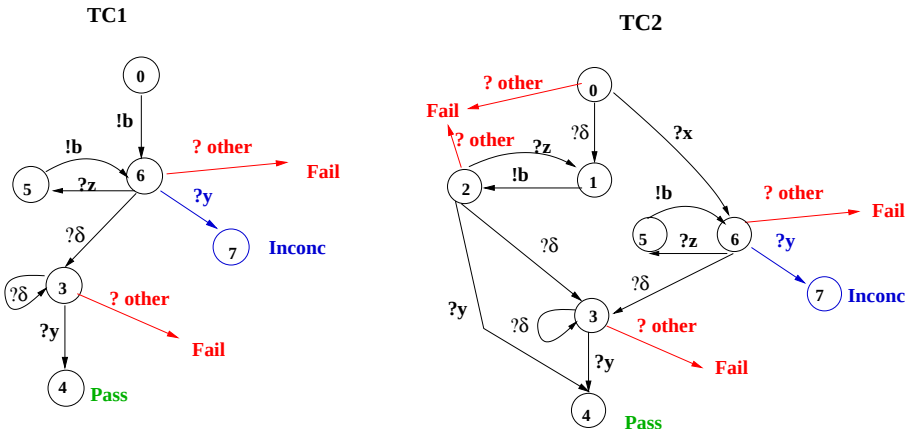



FIG. 3.16 – Cas de tests

Cependant, puisqu'un cas de test $TC$ extrait du graphe de test $CTG$ est un sous-IOLTS de $CTG$, en utilisant les caractérisations des traces menant aux états de **Pass**, **Inconc** et **Fail** dans $CTG$, on peut au moins énoncer les propriétés suivantes sur les cas de test :

**Proposition 3.2.1 (Traces de TC)** *Les traces d'un cas de test $TC$ obtenu par élagage d'un graphe de test complet $CTG$ satisfont :*

$$
\begin{array}{rcl}
Traces_{Pass}(TC) & \subseteq & STraces(S) \cap min(Traces_{Accept}(TP)) \\
Traces_{Inconc}(TC) & \subseteq & prf_{<}(STraces(S) \cap min(Traces_{Accept}(TP))).(A_O^S \cup \{\delta\}) \\
 & & \cap\, STraces(S) \cap \overline{prf_{<}(STraces(S) \cap min(Traces_{Accept}(TP)))} \\
Traces_{Fail}(TC) & \subseteq & prf_{<}(STraces(S) \cap min(Traces_{Accept}(TP))).(A_O^{IUT} \cup \{\delta\}) \\
 & & \cap\, \overline{STraces(S)}
\end{array}
$$

Nous verrons en section 3.3 un algorithme d'élagage adapté du calcul de $coreach$ par parcours en largeur. D'autres stratégies sont possibles pour régler les conflits de contrôlabilité. En particulier, on peut remplacer le parcours en largeur par un parcours en profondeur, ce qui donnerait des cas de tests différents. On peut également générer plusieurs cas de tests pour le même $CTG$, par exemple un par état dans **Pass**, en faisant plusieurs parcours en profondeur depuis chacun de ceux-ci. Il est alors nécessaire de créer de nouveaux états dans **Inconc**, ici des états de $L2A$ qui ne permettent pas d'atteindre l'état de **Pass** sélectionné.

Une autre possibilité est de spécifier une propriété sur les cas de test à extraire du CTG. Par exemple, on peut souhaiter donner une priorité aux entrées du cas de test par rapport aux sorties. Cependant, il se peut qu'il n'y ait pas de solution. La co-accessibilité depuis $Pass$ peut en effet être incompatible avec ce type de propriété.

### Propriétés des tests générés

On a le théorème suivant

**Théorème 3.2.2** *Pour une spécification $S$, pour tout objectif de test $TP$ sur les traces suspendues de $S$, tout cas de test $TC = gen_test_OT1(S, TP)$ produit par l'algorithme* gen_test_OT1 *est* non biaisé *et* non laxiste *pour* **ioco** *et $S$.*

*Pour une spécification $S$ donnée, l'ensemble (infini) de tous les cas de tests que peut produire l'algorithme* gen_test_OT1 *est exhaustif.*

**Preuve :** On peut prouver le non biais et l'exhaustivité de façon très similaires aux preuves de [77] pour l'algorithme *gen_test* présenté en 3.2.1.
*Non biais :* Soit $IUT$ une implémentation d'alphabet compatible à $S$ et soit $TC$ un cas de test généré et supposons que $TC \;\; may \;\; reject \;\; IUT$. Alors $\exists\ \sigma' = \sigma.x \in Traces(IUT \parallel TC)$ tq $TC$ after $\sigma' \in$ **Fail**. Par construction de $\rightarrow_{\text{Fail}}$ dans $TC$ et $CTG$, $\sigma \in Traces(PS^{\text{VIS}}) = Traces(\Delta(S)) = STraces(S)$, mais $\sigma.x \notin Traces(PS^{\text{VIS}})$ et $x \in A_{\text{I}}^{\text{CTG}} = A_{\text{O}}^{\text{IUT}} \cup \{\delta\}$. Il existe donc $\sigma \in STraces(S)$ et $x \in Out(\Delta(IUT)$ after $\sigma)$ tq $x \notin Out(\Delta(S)$ after $\sigma)$. On en déduit donc que $\neg(IUT$ **ioco** $S)$. Rejet par un test généré implique donc non-conformité.

On a cependant une preuve beaucoup plus directe, utilisant le fait que les traces de $TC$ menant à **Fail** (proposition 3.2.1), respectent la condition suffisante de non-biais (proposition 3.1.8). En effet on a

$$
\begin{array}{rcl}
Traces_{\text{Fail}}(TC) & \subseteq & prf_{<}(STraces(S) \cap min(Traces_{\text{Accept}}(TP))).(A_{\text{O}}^{\text{IUT}} \cup \{\delta\}) \\
 & & \cap\, \overline{STraces(S)}
\end{array}
$$

$$\begin{aligned} &\subseteq \quad prf_{\leq}(STraces(S)).A_{\mathrm{O}}^{\Delta(IUT)} \cap \overline{STraces(S)} \\ &\subseteq \quad STraces(S).A_{\mathrm{O}}^{\Delta(IUT)} \cap \overline{STraces(S)} \\ &\qquad \text{puisque } STraces(S) \text{ est préfixe-clos} \end{aligned}$$

Donc $TC$ respecte la condition suffisante de non-biais.

*Non-laxisme :* c'est une conséquence directe de la construction de $\rightarrow_{\text{Fail}}$ dans $CTG$, donc dans $TC$ et du fait que $Traces(PS^{\text{VIS}}) = Traces(\Delta(S)) = STraces(S)$. En effet, soit $\sigma \in STraces(S)$ et $a \in A_{\mathrm{I}}^{\mathrm{TC}}$, tels que $a \notin Out(\Delta(S)$ after $\sigma)$. Si $\sigma.a \in Traces(TC)$ alors $TC$ after $\sigma.a \subseteq$ **Fail**.

*Exhaustivité :* Soit $IUT$ une implémentation d'alphabet compatible à $S$ et supposons que $\neg(IUT$ **ioco** $S)$. Par définition, il existe une séquence $\sigma \in STraces(S)$ et il existe $x \in A_O^{\mathrm{IUT}} \cup \{\delta\}$ tel que $x \in Out(\Delta(IUT)$ after $\sigma)$ et $x \notin Out(\Delta(S)$ after $\sigma)$. Il existe aussi $y \in A_O^{\mathrm{IUT}} \cup \{\delta\}$ tq $\sigma.y \in STraces(S)$ (on sait que $y$ existe toujours car dans tout état $det(\Delta(S))$ a une sortie par l'ajout de $\delta$). Soit $\sigma' = \sigma.y$. Définir alors $TP$ comme l'automate reconnaissant exactement la séquence $\sigma'$. Par construction du $CTG$ produit pour $TP$, on a $CTG$ after $\sigma.y \in$ **Pass**, et $CTG$ after $\sigma.x \in$ **Fail**. $TP$ ne reconnaissant que la séquence $\sigma.y$, $CTG$ n'a qu'un état $Pass$ et l'élagage des conflits de contrôlabilité n'offre aucun choix. $CTG$ est juste complet en entrée dans chaque état alors que $TC$ n'a besoin de l'être que dans les états correspondant aux sorties de $\sigma.y$. En particulier l'élagage de $CTG$ en $TC$ conserve $\sigma.y$ et $\sigma.x$ dans $Traces(TC)$. Il existe donc une exécution de $TC$ sur $IUT$ qui peut produire le verdict **Fail**. Donc $TC\ may\ reject\ IUT$.
□

### 3.2.3 Algorithme de génération de tests avec objectifs de test ayant des actions internes

Nous présentons maintenant un deuxième algorithme de génération de tests basé sur des objectifs de test que nous appelerons *gen_test_OT2*. La différence majeure avec l'algorithme précédent, vient du fait que les objectifs de test permettent plus de précision dans la sélection. Ils reconnaissent, non plus des traces suspendues, mais des séquences d'actions de la spécification, y compris les actions internes. La définition formelle des objectifs s'en trouve modifiée, mais aussi quelques aspects de l'algorithme. C'est cet algorithme qui est implémenté dans l'outil TGV.

#### Justification

Rappelons que le rôle d'un objectif de test est de sélectionner des comportements visibles de la spécification dans un cas de test qui observera les traces suspendues de l'IUT. Le cas de test n'aura donc que des actions visibles définies à partir de celles de la spécification. Mais *a priori* tous les moyens sont bons pour sélectionner des comportements de $S$ afin de focaliser la production d'un cas de test. Il n'y a aucune raison de se limiter à décrire des propriétés sur les traces suspendues de la spécification et on doit pouvoir utiliser toutes les informations contenues dans la spécification. En particulier on peut utiliser des actions internes comme dans TGV, ou des informations sur les états (contrôle et valeurs de variables) comme dans TestComposer [35] ou des informations de couverture.

Nous nous limitons ici à l'usage des actions internes. Cette simple extension par rapport aux objectifs de test externes donne plus de précision à la sélection, puisqu'on peut alors distinguer deux séquences $\mu_1$ et $\mu_2$ de $S$ de même trace $\sigma$ dans l'objectif. Evidemment, au final le test ne parle que de traces, donc la trace $\sigma$ apparaît ou non dans le cas de test et la distinction a disparu. Néanmoins, on peut sélectionner les tests en utilisant les actions internes, sans connaissance *a priori* des actions visibles.

Ceci s'est avéré utile, en particulier dans le cas du *test dans un contexte* (voir figure 3.17). Dans cette architecture de test, le système sous test (SUT) est composé de plusieurs processus. L'implémentation sous test (IUT) est un composant logiciel qui n'est accessible qu'à travers un contexte (e.g. un autre composant logiciel, un médium de communication). On peut supposer que le contexte est implémenté correctement. Bien qu'on souhaiterait tester l'IUT par rapport à sa spécification, les interfaces disponibles pour le test se trouvent entre le contexte et l'environnement. Les comportements visibles du système sont donc à cette interface et ne permettent de tester que le système sous test SUT, c'est à dire l'IUT dans son contexte, par rapport à la spécification de ce composant dans son contexte (ou dans la spécification de celui-ci) ($IUT||C$ **ioco**$S||C$). Si les objectifs permettent de décrire seulements des comportements visibles, il faut pouvoir les imaginer à cette interface, ce qui implique de considérer le comportement du système composé. Mais comme on souhaite cibler le test sur les comportements de l'IUT, l'usage des actions internes permet de décrire des objectifs sur l'IUT, sans faire référence à son contexte.

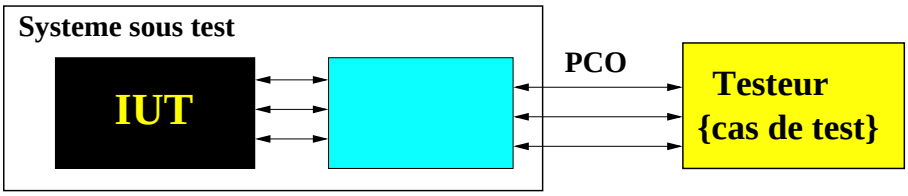



FIG. 3.17 – Test dans un contexte

**Formalisation des objectifs de test avec actions internes**

**Définition 3.2.5 (Objectif de test internes)**
*Un* objectif de test interne *est un IOLTS* $TP = (Q^{TP}, A^{TP}, \rightarrow_{TP}, q_0^{TP})$ *déterministe (sur* $A^{TP}$*) et* complet *muni de deux ensembles d'*états puits *disjoints* $Accept^{TP}$ *et* $Refuse^{TP}$*, et de même alphabet que la spécification i.e.* $A^{TP} = A^S$. Complet *signifie que dans tout état, toute action est tirable (***ici toute action de l'alphabet de** $S$*) i.e.* $\forall q \in Q^{TP}, \forall a \in A^{TP}, q \xrightarrow{a}_{TP}$. *Un état* $q$ *est un* puits *s'il boucle pour toute action (ici toute action de* $S$*) i.e.* $\forall a \in A^{TP}, q \xrightarrow{a}_{TP} q$.

Un objectif de test est donc toujours un automate d'état fini, mais cette fois-ci sur l'alphabet des actions de la spécification $A^S$, non plus seulement sur ses actions visibles et blocages.

Un objectif de test interne définit deux langages reconnus

$$L_{\text{Accept}}(TP) \quad = \quad \{\mu \in (A^S)^* \mid \exists q_A \in Accept, q_0^{TP} \xrightarrow{\mu}_{TP} q_A\}$$

$$L_{\text{Refuse}}(TP) \quad = \quad \{\mu \in (A^{\text{S}})^* \mid \exists q_R \in Refuse, q_0^{\text{TP}} \xrightarrow{\mu}_{\text{TP}} q_R\}$$

Comme pour les objectifs de test externes, ces deux langages sont disjoints car $Accept$ et $Refuse$ sont disjoints et $TP$ est déterministe. Mais attention, ici le déterminisme s'entend par rapport à l'alphabet $A^{\text{S}}$ où les actions de $I^{\text{S}}$ ne sont pas considérées comme des $\varepsilon$-transitions et sont distinguées. On peut également regarder les traces de ces langages, i.e. les projections des langages sur les actions visibles de $A^{\text{TP}}_{\text{VIS}}$ : $Traces(L_{\text{Accept}}(TP))$ et $Traces(L_{\text{Refuse}}(TP))$. On remarque alors que ces deux ensembles de traces ne sont plus disjoints puisqu'une séquence acceptée et une séquence refusée peuvent avoir la même trace.

Un exemple d'objectif de test est décrit par la figure 3.18.

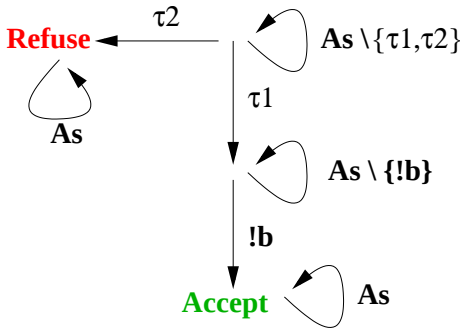



FIG. 3.18 – Exemple d'objectif de test

### Algorithme de génération de tests

Comme précédemment, les entrées de l'algorithme de génération de test *gen_test_OT2* sont une spécification $S$ servant d'oracle, et un objectif de test $TP$ interne servant à la sélection, et on souhaite obtenir en sortie un cas de test $TC = gen_test_OT2(S, TP)$ qui soit au moins correct pour $S$ et **ioco** et qui focalise le test sur les comportements décrits par $TP$.

Comme précédemment, *gen_test_OT2* est composé d'un enchaînement d'opérations, différent de *gen_test_OT1* et résumé par la figure 3.19.

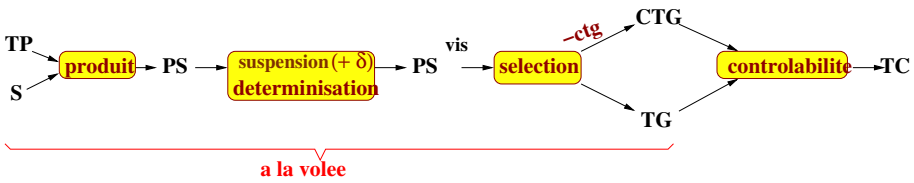



FIG. 3.19 – Résumé des opérations de synthèse de test

Dans le cas où les objectifs de test contiennent des actions visibles, ces objectifs reconnaissent les séquences de $S$, pas ses traces suspendues. On commence donc par calculer un produit $PS$ entre $TP$ et $S$, conservant tous les comportements de $S$ (car $TP$ est complet) mais ayant pour effet d'étiqueter les séquences de $S$ acceptées ou refusées par $TP$. On calcule ensuite les comportements visibles (traces et blocages) de $PS$ (donc de $S$) dans l'IOLTS $PS^{\text{VIS}} = det(\Delta(PS))$. On doit ici définir quelles sont les traces suspendues acceptés ou refusés en fonction des séquences de $S$ acceptées ou refusées par $TP$, par l'intermédiaire de la définition des états Accept et Refuse de $PS^{\text{VIS}}$. L'IOLTS $PS^{\text{VIS}}$ est alors un automate comme dans l'algorithme *gen_test_OT1*. Les étapes de sélection du graphe de test complet ($CTG$) ou d'un sous-graphe de test($TG$), et le calcul du cas de test dans $TC$ dans *gen_test_OT2*. sont alors quasiment identiques (à un détail technique près) aux étapes correspondantes de l'algorithme *gen_test_OT1*.

$S$ et $TP$ décrits en figure 3.20 serviront d'illustration dans toute cette section. Noter que dans cet exemple , l'objectif de test est étiqueté par des expressions régulières, comme cela est permis dans TGV.

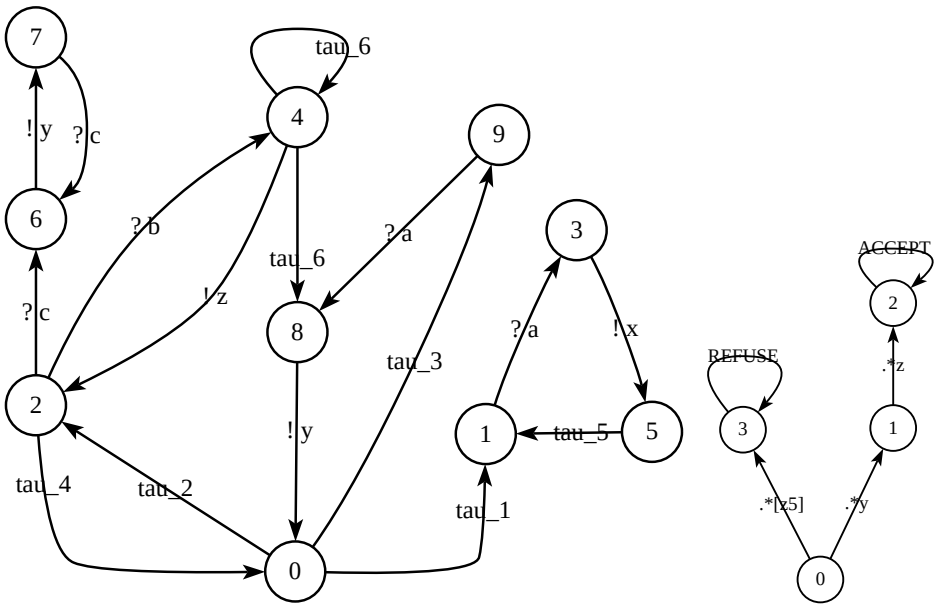



FIG. 3.20 – Spécification $S$ et objectif de test $TP$

**Produit synchrone** On souhaite calculer les séquences de la spécification $S$ acceptés (sur $Accept$) ou refusés (sur $Refuse$) par l'objectif de test $TP$, i.e. calculer le langage $L(S) \cap L(TP)$. Ce problème est à nouveau résolu par un calcul du produit synchrone entre $S$ et $TP$.

**Définition 3.2.6 (Produit synchrone (II))**

*Soient* $S = (Q^S, A^S, \rightarrow_S, q_0^S)$ *un IOLTS et* $TP = (Q^{TP}, A^{TP}, \rightarrow_{TP}, q_0^{TP})$ *un IOLTS déterministe et complet sur* $A^{TP} = A^S$ *et muni des ensembles d'états puits* $Accept^{TP}$ *et* $Refuse^{TP}$*.*
*Le produit synchrone* $S \times TP$ *est un IOLTS noté* $PS = (Q^{PS}, A^{PS}, \rightarrow_{PS}, q_0^{PS})$*, muni de deux ensembles disjoints d'états* $Accept^{PS}$ *et* $Refuse^{PS}$*, et défini comme suit.*

- *son ensemble d'états est* $Q^{PS} \triangleq Q^S \times Q^{TP}$ *item l'état initial est* $q_0^{PS} \triangleq (q_0^S, q_0^{TP})$
- *son alphabet est* $A^{PS} \triangleq A^S (= A^{TP})$.
- *sa relation de transition* $\rightarrow_{PS}$ *est définie par :*
  $(q^S, q^{TP}) \xrightarrow{a}_{PS} (q'^S, q'^{TP}) \iff q^S \xrightarrow{a}_S q'^S \wedge q^{TP} \xrightarrow{a}_{TP} q'^{TP}$.

- $Accept^{PS} \triangleq Q^S \times Accept^{TP}$ *et* $Refuse^{PS} \triangleq Q^S \times Refuse^{TP}$

Puisque $TP$ est complet, $PS = S \times TP$ est bisimilaire à $S$, donc il préserve au moins les traces et blocages. Par conséquent $STraces(PS) = STraces(S)$. En termes de langage des actions, on a bien sûr le résultat escompté i.e. $L_{\text{Accept}}(PS) = L(S) \cap L_{\text{Accept}}(TP)$ et $L_{\text{Refuse}}(PS) = L(S) \cap L_{\text{Refuse}}(TP)$. Noter que ces deux langages sont disjoints puisque $L_{\text{Accept}}(TP)$ et $L_{\text{Refuse}}(TP)$ le sont.

La figure 3.20 décrit un exemple avec arrêt de la construction aux états $Accept$ et $Refuse$. L'effet du produit synchrone est de déplier $S$ en fonction des états de $TP$ (e.g. dans $PS$, 0 = (0,0) et 12 = (0,1)) et d'étiqueter les comportements de $S$ par les $Accept$ (e.g. 20 = (2,2)) et $Refuse$ (e.g. 9 = (2,3) et 13 = (1,3)) de $TP$. $PS$ peut être construit en largeur ou en profondeur à partir de l'état initial.

**Comportements visibles :** Les langages $L_{\text{Accept}}(PS)$ et $L_{\text{Refuse}}(PS)$ définis sur $PS$ identifient des ensembles de séquences d'actions. L'étape suivante de la synthèse de tests consiste à en déduire des comportements visibles (traces suspendues) de la spécification $S$ (qui sont aussi ceux de $PS$). Pour cela, on construit l'IOLTS $PS^{\text{VIS}} = det(\Delta(PS)) = (Q^{\text{VIS}}, A^{\text{VIS}}, \rightarrow_{\text{VIS}}, q_0^{\text{VIS}})$ (voir définitions 3.1.6 pour la définition de l'automate de suspension et 3.1.5 pour sa déterminisation) où $A^{\text{VIS}} = A_{\text{I}}^{\text{VIS}} \cup A_{\text{O}}^{\text{VIS}}$ avec $A_{\text{I}}^{\text{VIS}} = A_{\text{I}}^{\text{S}}$ et $A_{\text{O}}^{\text{VIS}} = A_{\text{O}}^{\text{S}} \cup \{\delta\}$ Par définition on a $Traces(PS^{\text{VIS}}) = STraces(PS) = STraces(S)$.

Le problème qui se pose est de déterminer parmi ces traces suspendues celles qui doivent être acceptées ou refusées. En effet, bien que $L_{\text{Accept}}(PS)$ et $L_{\text{Refuse}}(PS)$ soient disjoints, il n'en est pas de même pour leurs projections sur les actions visibles $Traces(L_{\text{Accept}}(PS))$ et $Traces(L_{\text{Refuse}}(PS))$, ni pour les traces suspendues $Traces(L_{\text{Accept}}(\Delta(PS))$ et $Traces(L_{\text{Refuse}}(\Delta(PS)))$ car une séquence acceptée et une séquence refusée peuvent avoir la même trace. En d'autres termes, un méta-état de $PS^{\text{VIS}}$ peut contenir des états de $Accept^{\text{PS}}$ et de $Refuse^{\text{PS}}$. Il faut donc définir les notions d'états $Accept$ et $Refuse$ dans $PS^{\text{VIS}}$. Nous avons choisi de refuser un comportement visible aussitôt qu'il correspond à au moins un comportement refusé dans $PS$. Ce choix privilégiant les $Refuse$ au détriment des $Accept$ est complètement arbitraire. Cependant, il privilégie la réduction de l'exploration de $PS^{\text{VIS}}$, but recherché quand on utilise les états $Refuse$, et donc utile en pratique.

De ce choix découle la définition des états $Accept$ et $Refuse$ de $PS^{\text{VIS}}$ i.e.

$$\begin{aligned} Refuse^{\text{VIS}} &\triangleq \{P \in Q^{\text{VIS}} \mid P \cap Refuse^{\text{PS}} \neq \emptyset\} \\ Accept^{\text{VIS}} &\triangleq \{P \in Q^{\text{VIS}} \mid P \cap Accept^{\text{PS}} \neq \emptyset\} \setminus Refuse^{\text{VIS}} \end{aligned}$$

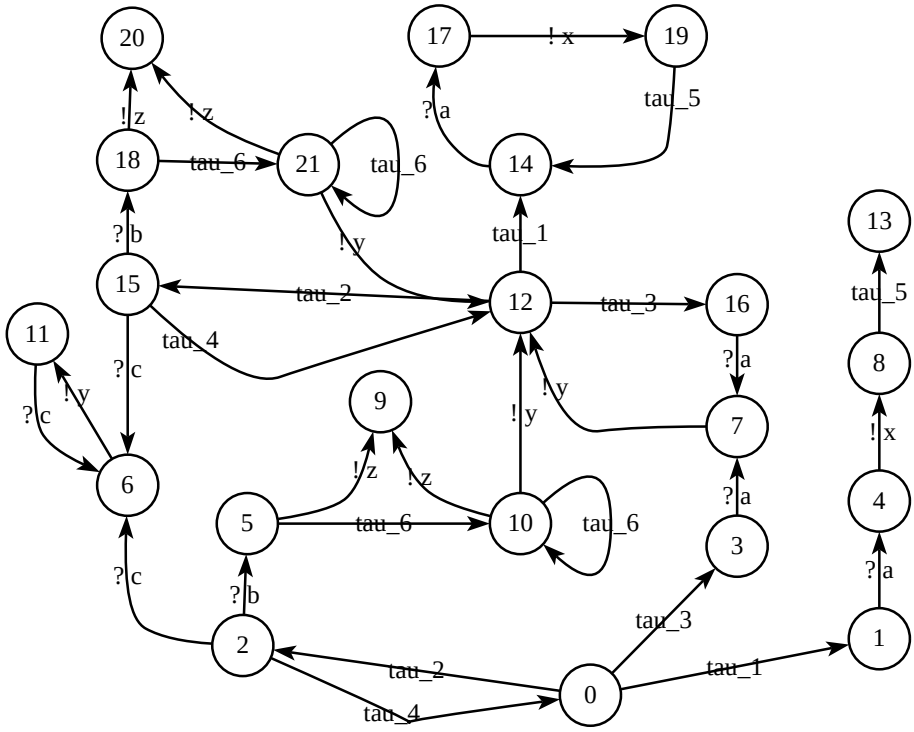



FIG. 3.21 – Produit $PS = S \times TP$ pour la spécification et l'objetif de la figure 3.20

i.e. les méta-états $Refuse^{\text{VIS}}$ sont les méta-états contenant au moins un état $Refuse^{\text{PS}}$ et i.e. les méta-états de $Accept^{\text{VIS}}$ sont les les méta-états qui contiennent au moins un état $Accept$ mais aucun état $Refuse$.

Les traces refusées et acceptées par $PS^{\text{VIS}}$ sont donc :

$$\begin{aligned} Traces_{\text{Refuse}}(PS^{\text{VIS}}) &= Traces(L_{\text{Refuse}}(\Delta(PS))) \\ Traces_{\text{Accept}}(PS^{\text{VIS}}) &= Traces(L_{\text{Accept}}(\Delta(PS))) \setminus Traces(L_{\text{Refuse}}(\Delta(PS))) \end{aligned}$$

Mais contrairement au cas des objectifs de test externes, il est difficile de caractériser plus finement ces traces en fonction des langages de $S$ et $TP$. En effet, on n'a pas de caractérisation de $L_{\text{Accept}}(\Delta(PS))$ en fonction de $L_{\text{Accept}}(PS)$, puisque la définition de $\Delta(PS)$ est structurelle sur $PS$ et ne dépend pas de son langage $L_{\text{Accept}}(PS)$.

Le calcul de $PS^{\text{VIS}}$ à partir de $PS$ se décompose donc en trois opérations : l'ajout des blocages par le calcul de $\Delta(PS)$, la déterminisation par le calcul de $det(\Delta(PS))$, le calcul des ensembles $Refuse^{\text{VIS}}$ et $Accept^{\text{VIS}}$ par simple examen des états constituant chaque méta-état de $det(\Delta(PS))$.

**Exemple :** L'IOLTS de la figure 3.22 représente l'IOLTS $PS^{\text{VIS}}$ obtenu depuis $PS$ à la figure 3.20. Les blocages $\delta$ sont ici différenciés (et notés LIVELOCK et OUTPUTLOCK) grâce à une facilité de TGV. Notons que l'état 5 de $Refuse^{\text{VIS}}$ correspond au refus de séquences contenant $\tau_5$, pourtant invisible dans $PS^{\text{VIS}}$.

**Remarque :** si un objectif de test interne $TP2$ est un automate qui boucle sur toute action interne ($\forall q \in Q^{\text{TP2}}, \forall a \in I^{\text{S}}, q \xrightarrow{a} q$), celui-ci ne peut pas distinguer deux séquences qui ne se différencient que sur des actions internes. A partir de cet objectif, on peut définir un objectif $TP1$ externe type, i.e. sans actions internes, en supprimant simplement ces boucles sur actions internes. En fait $TP1 = det(TP2)$ et on a
$Traces_{\text{Accept}}(TP1) = Traces(L_{\text{Accept}}(TP2))$
i.e. les traces acceptées par $TP1$ sont exactement les traces des séquences acceptées par $TP2$. On peut alors s'interroger sur la relation entre ce que calculent les deux algorithmes : le premier avec $TP1$, le second avec $TP2$. Il s'avère que dans ce cas
$det(\Delta(S \times TP2)) = det(\Delta(S)) \times TP1 = det(\Delta(S)) \times det(TP2)$.
En quelques sortes, on peut dire que la déterminisation se distribue sur le produit. De plus, les traces des séquences acceptées et refusées par $TP$ sont disjointes i.e.
$Traces(L_{\text{Accept}}(TP)) \cap Traces(L_{\text{Refuse}}(TP)) = \emptyset$
donc aucun meta-état de $PS^{\text{VIS}} = det(\Delta(PS)) = det(\Delta(S \times TP2))$ ne peut contenir à la fois un état de $Accept^{\text{PS}}$ et de $Refuse^{\text{PS}}$. Donc les définitions de $PS^{\text{VIS}}$ et de leurs états $Accept^{\text{VIS}}$ et $Refuse^{\text{VIS}}$ dans les deux algorithmes coïncident.

**Sélection :** $PS^{\text{VIS}}$ représente les comportements visibles (traces et blocages) de $S$ et, grâce aux ensembles $Accept^{\text{VIS}}$ et $Refuse^{\text{VIS}}$, identifie les comportements acceptés ou refusés par l'objectif de test $TP$. Comme dans l'algorithme précédent, la sélection consiste à extraire un sous-IOLTS de $PS^{\text{VIS}}$, contenant les comportements acceptés. Comme précédemment, on va d'abord considérer la construction du *graphe de test complet*, noté $CTG$ (pour *Complete Test Graph*).

On pourrait se dire à ce stade qu'il suffit de prendre la même définition que dans l'algorithme précédent. Pour des raisons techniques dues à l'expressivité des objectifs de test,

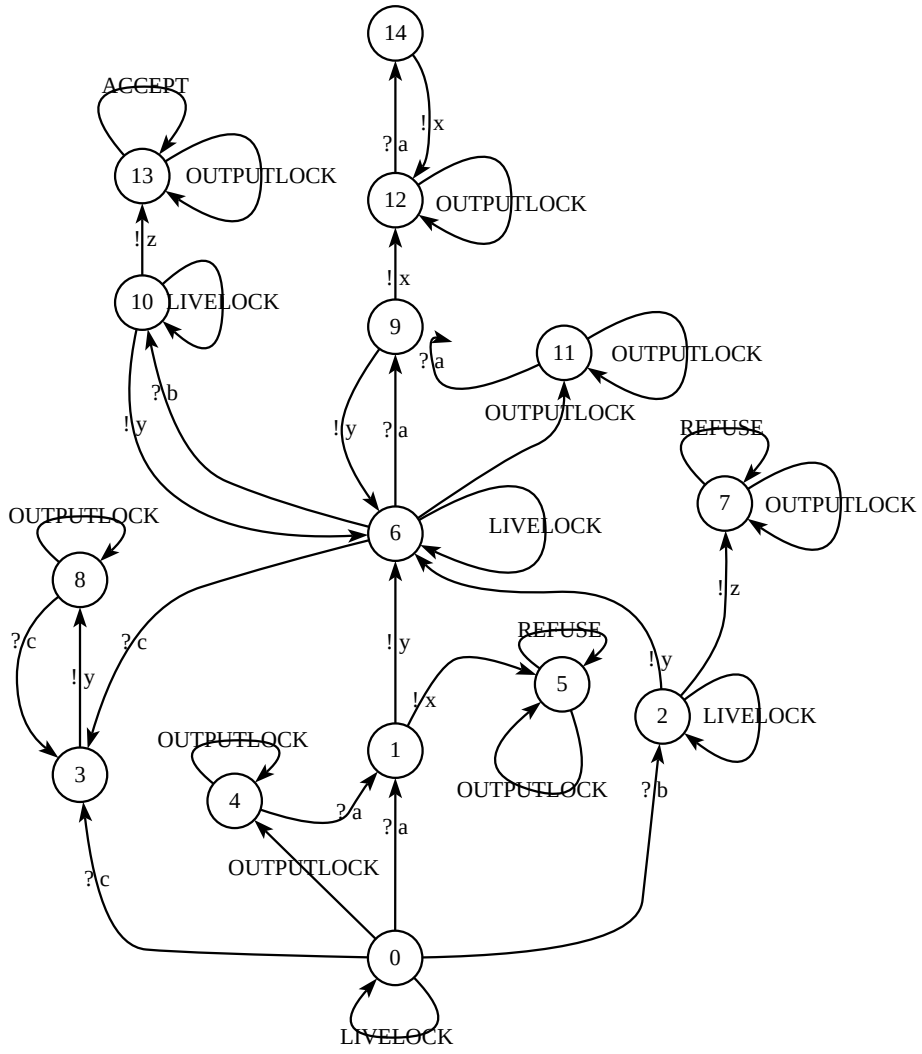



FIG. 3.22 – Comportements visibles $PS^{\text{VIS}} = det(\Delta(PS))$

ceci pourrait faire perdre l'exhaustivité. En effet, notre définition d'objectif ne permet pas d'utiliser les $\delta$ car ceux-ci sont calculés lors de la déterminisation, donc après le produit. On a donc une définition légèrement différente de $CTG$ :

**Définition 3.2.7 (Graphe de test complet (II))**
*Soit* $PS^{VIS} = det(\Delta(S \times TP))$. *Le* graphe de test complet *est un IOLTS* $select(PS^{VIS}) = CTG = (Q^{CTG}, A^{CTG}, \longrightarrow_{CTG}, q_0^{CTG})$, *muni de trois ensembles d'états sans successeur* **Pass**, **Inconc** *et* **Fail**, *et défini de la façon suivante :*

- *Son alphabet est* $A^{CTG} = A_O^{CTG} \cup A_I^{CTG}$ *avec* $A_O^{CTG} = A_I^{VIS}$ *et* $A_I^{CTG} = A_O^{IUT} \cup \{\delta\}$ *(image miroir).*
- *L'ensemble d'états est* $Q^{CTG} = L2A \cup$ **Inconc** $\cup$ **Fail**, *avec*
  - $L2A = (reach_{|\overline{Accept^{VIS}}}(q_0^{VIS}) \cap coreach(Accept^{VIS})) \cup \{q_{AT}\}$, *i.e. l'ensemble des états* accessibles *depuis* $q_0^{VIS}$ *à travers des états n'appartenant pas à* $Accept^{VIS}$ *et* co-accessibles *de* $Accept^{VIS}$, *plus un nouvel état* $q_{AT}$.
    *Parmi ceux-ci, on définit*
    **Pass** $= (L2A \cap Accept^{VIS} \setminus AT) \cup \{q_{AT}\}$
    *avec* $AT = \{q \in Accept^{VIS} \mid \Gamma_{VIS}(q) = \{\delta\}\}$ *l'ensemble des états accepteurs qui ne transitent que par* $\delta$,
  - **Inconc** $= post_{A_O^{VIS}}(L2A) \setminus L2A$ *i.e. les états successeurs immédiats d'un état de* $L2A$ *par une sortie dans* $PS^{VIS}$ *(donc une entrée pour* $CTG$*) mais n'appartenant pas eux-mêmes à* $L2A$
  - **Fail** $= \{Fail\}$ *où Fail* $\notin Q^{VIS}$ *est un nouvel état.*
- *si* $L2A \neq \emptyset$, *l'état initial est* $q_0^{CTG} = q_0^{VIS}$ *sinon* $Q^{CTG}$ *est vide.*
- *La relation de transition est* $\rightarrow_{CTG} = \rightarrow_{L2A} \cup \rightarrow_{\textbf{Inconc}} \cup \rightarrow_{\textbf{Fail}}$ *où*
  - $\rightarrow_{L2A} = (\rightarrow_{VIS} \cap ((L2A \setminus \textbf{Pass}) \times A^{CTG} \times L2A)) \cup (AT \times \{\delta\} \times \{q_{AT}\})$ *i.e. la restriction de* $\rightarrow_{VIS}$ *à* $L2A$ *sauf qu'on déplie les transitions* $\delta$ *de* $AT$ *vers le nouvel état* $q_{AT}$,
  - $\rightarrow_{\textbf{Inconc}} = \rightarrow_{VIS} \cap ((L2A \setminus \textbf{Pass}) \times A_I^{CTG} \times \textbf{Inconc})$,
  - $\rightarrow_{\textbf{Fail}} = \{(q, a, Fail) \mid q \in L2A \wedge a \in A_I^{CTG} \wedge q \not\xrightarrow{a}_{VIS}\}$.

L'état $q_{AT}$ et l'ensemble $AT$ sont ajoutés pour des raisons techniques nécessaires à l'exhaustivité et dues à l'impossibilité d'utiliser $\delta$ dans l'objectif. $AT$ est l'ensemble des états de $Accept^{VIS}$ qui ne transitent que par $\delta$ (e.g. 13 en figure 3.22). Pour tous ces états, on déplie la transition $\delta$ vers un nouvel état $q_{AT} \in$ **Pass**, et on les supprime de **Pass**.

Pour l'exemple de la figure 3.22 la construction de $CTG$ produit le graphe de la figure 3.23.

De la même façon que dans le cas des objectifs externes, on peut définir les traces reconnues dans les états **Pass**, **Fail** et **Inconc**.

$$\begin{aligned} Traces_{\text{Pass}}(CTG) &= min(Traces_{\text{Accept}}(PS^{\text{VIS}})) \\ &= STraces(S) \cap min(Traces_{\text{Accept}}(TP)) \end{aligned}$$

**Élagage des conflits de contrôlabilité :** l'élagage des conflits de contrôlabilité est identique à la version précédente de l'algorithme de génération. Pour le graphe de tests de la

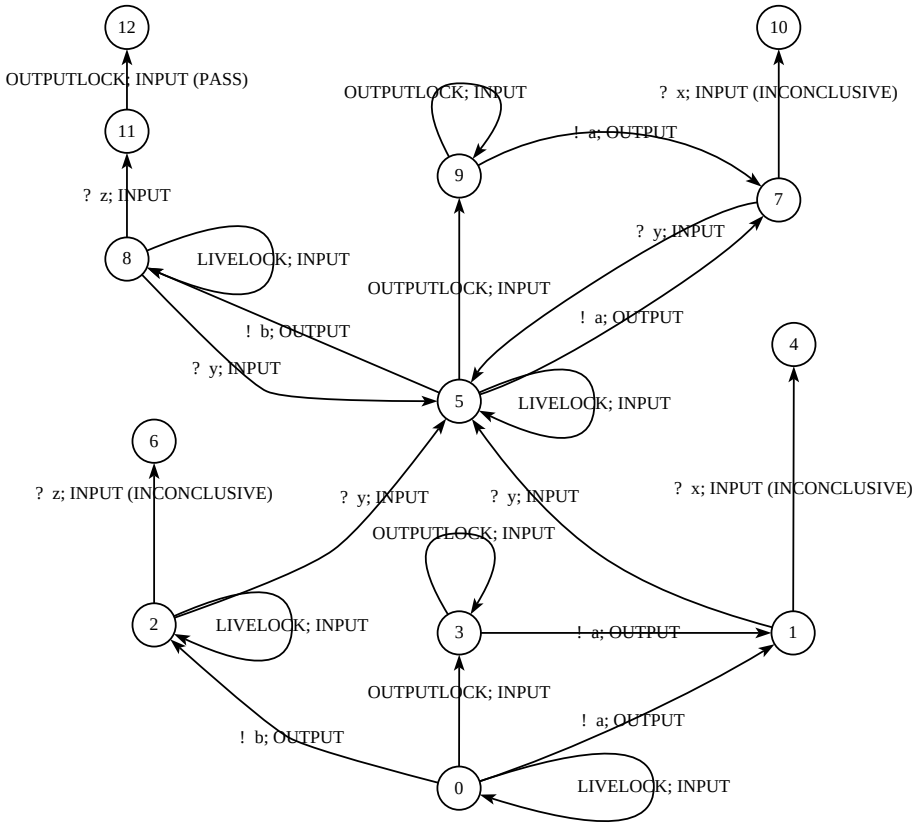



FIG. 3.23 – Graphe de test complet $CTG$

figure 3.23 on peut par exemple produire le cas de test de la figure 3.24.

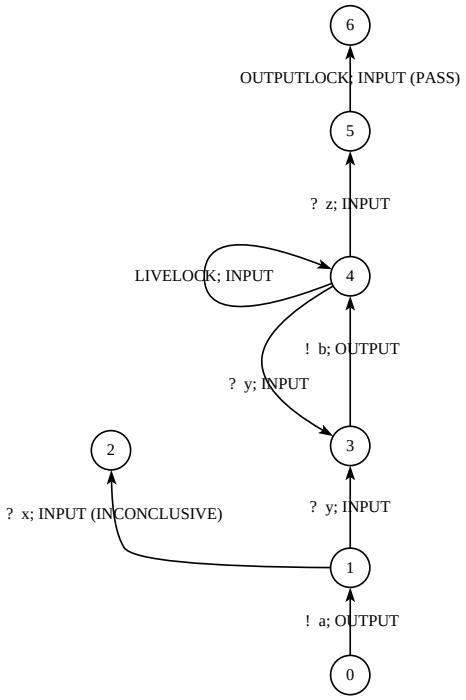



FIG. 3.24 – Cas de test

**Propriétés des cas de tests produits**

Comme pour le précédent algorithme *gen_test_OT1*, on a le théorème suivant pour les tests produits par *gen_test_OT2* :

**Théorème 3.2.3** *Pour une spécification $S$, pour tout objectif de test $TP$ sur $S$, le cas de test $TC = gen_test_OT2(S, TP)$ produit par l'algorithme de génération est* non biaisé, *et* non laxiste *pour* **ioco** *et $S$.*

*Pour toute spécification $S$, l'ensemble (infini) de tous les tests que peut produire l'algorithme* gen_test_OT2 *est exhaustif.*

**Preuve :** Les preuves de non biais et de non-laxisme sont très similaires à celles de l'algorithme précédent. Pour l'exhaustivité, il y a une légère différence.
*Non biais :* Soit $IUT$ une implémentation d'alphabet compatible à $S$ et soit $TC$ un cas de test, et supposons que $TC\ may\ reject\ IUT$. Alors $\exists\ \sigma' = \sigma.x \in Traces(IUT \parallel TC)$ tq $TC$ after $\sigma' \in$ **Fail**. Par construction de $\rightarrow_{\text{Fail}}$ dans $TC$ et $CTG$, $\sigma \in Traces(PS^{\text{VIS}}) = Traces(\Delta(S))$, mais $\sigma.x \notin Traces(PS^{\text{VIS}})$ et $x \in A_I^{\text{CTG}} = A_O^{\text{IUT}} \cup \{\delta\}$. Il existe donc $\sigma \in Traces(\Delta(S))$ et $x \in Out(\Delta(IUT)$ after $\sigma)$ tq $x \notin Out(\Delta(S)$ after $\sigma)$. On en

déduit donc que $\neg(IUT$ **ioco**$S)$.
*Non laxisme :* c'est une conséquence directe de la construction de $\rightarrow_{\text{Fail}}$ dans $CTG$, donc dans $TC$ et du fait que $Traces(PS^{\text{VIS}}) = Traces(\Delta(S))$.
*Exhaustivité :* Soit $IUT$ une implémentation d'alphabet compatible à $S$, et supposons que $\neg(IUT$ **ioco** $S)$. Il existe alors une séquence $\sigma \in Traces(\Delta(S))$ telle que $\exists x \in A_O^{\text{IUT}} \cup \{\delta\}$, $x \in Out(\Delta(IUT)$ after $\sigma)$ et $x \notin Out(\Delta(S)$ after $\sigma)$.

**(i)** s'il existe $y \in A_{\text{O}}^{\text{vis}}$, $y \neq \delta$ tel que $\sigma.y \in Traces(\Delta(S))$, définir $\sigma' = \sigma.y$

**(ii)** sinon on a $Out(\Delta(S)$ after $\sigma) = \delta$ et $x \neq \delta$ et définir $\sigma' = \sigma$ (c'est ce cas qui oblige à introduire $q_{\text{AT}}$ et $AT$ dans $CTG$ afin de rejeter $\sigma.x$ avec $x \neq \delta$).

Soit $\sigma''$ la trace obtenue par abstraction de $\delta$ dans $\sigma'$ (on ne permet pas d'utiliser $\delta$ dans $TP$). Définir alors $TP$ comme l'automate qui reconnait toutes les séquences dont la trace est $\sigma''$ ($TP$ est la trace $\sigma''$ et terminant en $Accept$ avec des boucles sur les actions internes dans chaque état). Par construction du $CTG$ produit pour $TP$, on a $CTG$ after $\sigma.x \in$ **Fail**. Il existe alors au moins une façon d'élaguer $CTG$ en $TC$ de sorte que $\sigma.x \in Traces(TC)$, et donc que $TC$ puisse rejeter $IUT$.
□

### 3.2.4 Complexité des algorithmes

La complexité de l'algorithme de génération de test *gen_test_OT2* est détaillée dans [49, 50]. Le calcul du produit $PS = S \times TP$ est linéaire en temps et espace dans la taille de $S$ et $TP$. Le calcul de $\Delta(PS)$ par ajout de $\delta$ est linéaire dans la taille de $PS$. La déterminisation efffectuée pour le calcul de $PS^{\text{VIS}} = det(\Delta(PS))$ est comme d'habitude exponentielle en temps et espace dans la taille de $PS$, donc de $S$ et $TP$. L'exponentielle est dûe au calcul de sous-ensembles, le coût de l'$\varepsilon$-clôture restant linéaire dans la taille de $PS$. La sélection basée sur une intersection de $reach$ et $coreach$ est linéaire dans la taille de $PS^{\text{VIS}}$. Enfin, la résolution des conflits de contrôlabilité également basée sur un calcul de $coreach$ puis de $reach$ est linéaire dans la taille de $PS^{\text{VIS}}$. Au total, la complexité est donc exponentielle dans la taille de $S$ et $TP$ : $\mathcal{O}(2^{|S| \times |TP|})$.

La complexité de l'algorithme *gen_test_OT1* est plus faible puisque l'exponentielle ne concerne que $S$, pas $TP$ car la déterminisation n'est appliqué qu'à $S$, pas à $S \times TP$. Les autres opérations, sélection et élagage sont quasi identiques et sont de complexité linéaire. Au total la complexité est donc $\mathcal{O}(|TP| \times 2^{|S|})$.

Noter que cette différence de complexité n'intervient que si les objectifs $TP$ de *gen_test_OT2* utilisent vraiment les actions internes, c'est à dire ne bouclent pas dans chaque état pour toute action interne. On a vu en effet que dans ce cas les IOLTS $PS^{\text{VIS}}$ coïncidaient.

### 3.2.5 Alternative

Nous avons jusqu'ici défini deux types d'objectifs de test

- des objectifs de test externes dont l'alphabet porte sur les actions de $det(\Delta(S)$, i.e. sur $A_{\text{VIS}}^{\Delta(S)} = A_{\text{I}}^{\text{S}} \cup A_{\text{I}}^{\text{S}} \cup \{\delta\}$.
- des objectifs de test internes dont l'alphabet porte sur les actions de $S$, i.e. $A^{\text{S}} = A_{\text{I}}^{\text{S}} \cup A_{\text{O}}^{\text{S}} \cup I^{\text{S}}$.

On pourrait aussi définir des objectifs de test portant sur les actions de $\Delta(S)$ i.e. $A^{\Delta(S)} = A^{\mathrm{s}}_{\mathrm{I}} \cup A^{\mathrm{s}}_{\mathrm{O}} \cup I^{\mathrm{s}} \cup \{\delta\}$.

On a vu en effet que le fait de ne pas pouvoir utiliser les $\delta$ dans les objectifs internes, nous ont obligé à un artifice pour obtenir l'exhaustivité de l'algorithme de génération.

Avec cette définition, la génération de test consisterait en les opérations suivantes :

- calculer les blocages de $S$ i.e. $\Delta(S)$,
- effectuer le produit $PS = \Delta(S) \times TP$, où la synchronisation porte sur toutes les actions, y compris les $\delta$. On obtient alors
  $L_{\text{Accept}}(PS) = L(S) \cap L_{\text{Accept}}(TP)$,
  $L_{\text{Refuse}}(PS) = L(S) \cap L_{\text{Refuse}}(TP)$,
- déterminiser en calculant $PS^{\text{VIS}} = det(PS)$, définir les états $Accept$ et $Refuse$ de façon identique au cas des objectifs de test internes. On a alors
  $Traces_{\text{Accept}}(PS) = Traces(L_{\text{Accept}}(PS)) \setminus Traces(L_{\text{Refuse}}(PS))$
- définir le graphe de test complet $CTG$ à partir de $PS^{\text{VIS}}$, cette fois-ci avec la même définition que dans le cas des objectifs externes. Ce qui permet d'obtenir en particulier
  $Traces_{\text{Pass}}(CTG) = min(Traces_{\text{Accept}}(PS))$
  $= min(Traces(L_{\text{Accept}}(PS)) \setminus Traces(L_{\text{Refuse}}(PS)))$
  $= min(Traces(L(S) \cap L_{\text{Accept}}(TP)) \setminus Traces(L(S) \cap L_{\text{Refuse}}(TP)))$

Cet algorithme combine une expressivité maximale des objectifs de test avec une relative simplicité des definitions. Mais comme on le verra plus loin, l'inconvénient de cet algorithme est, comme pour le 1$^{\text{er}}$ avec objectifs les externes, qu'on doive d'abord calculer les $\delta$, puis faire un produit, puis déterminiser. Alors que dans le 2$^{\text{nd}}$ algorithme, ces deux opérations sont consécutives, ce qui permet en particulier de combiner le calcul des livelocks avec la déterminisation, comme on le verra à la section suivante.

## 3.3 Efficacité algorithmique : génération à la volée

Le problème principal de la génération de tests est l'explosion combinatoire, problème habituel de la vérification par model-checking. On sait que si la spécification $S$ est donnée par la composition de processus, la taille de $S$ est au pire exponentielle dans la taille des processus le composant. Cette explosion est encore agravée si les processus manipulent des données, si les processus communiquent par des files de communication. La construction de l'IOLTS $S$ devient alors problématique, voire impossible si $S$ est infini. De plus, les complexité de *gen_test_OT1* et *gen_test_OT2* sont exponentielles dans la taille de $S$.

Or, en général, un cas de test $TC$ est de petite taille par rapport à $S$, grâce à la sélection par l'objectif de test $TP$. Construire complètement $S$ alors qu'on n'en utilise parfois qu'une infime partie dans $TC$ est donc très inefficace, voire impossible si $S$ est infini.

De plus la spécification n'est pas donnée sous la forme énumérée d'un IOLTS $S$, mais dans un langage de spécification. Cette spécification est compilée en un code de simulation qui implémente la sémantique du langage en fournissant les fonctions de construction du système de transitions $S$. Ces fonctions sont, à quelques détails près suivant les compilateurs, le calcul de l'état initial $q^{\mathrm{s}}_0$, le calcul des transitions tirables dans un état $\Gamma(q)$, le calcul d'un état successeur par une des transitions tirables depuis un état source $post_{\{a\}}(q)$,

et la fonction de comparaison entre états $q \stackrel{?}{=} q'$.

Le principe de la synthèse à la volée, implémentée dans TGV, est de réaliser l'algorithme *gen_test_OT2* par une construction paresseuse des parties de $S$, de $PS$ et de $PS^{\text{VIS}}$ nécessaires au calcul de $TC = gen_test_OT2(S, TP)$, i.e sélectionnées par $TP$. Dans tous les algorithmes composant $gen_test_OT2$ (produit, suspension, déterminisation, sélection du CTG, élagage), nous avons décrit le résultat attendu en fonction des entrées, suggérant que les entrées devaient être disponibles, donc complètement calculées, pour effectuer chaque opération. Par ailleurs, la sélection utilise l'opération $coreach$ qu'il est naturel d'implémenter par un parcours arrière en largeur. En réalité, toutes ces opérations (excepté l'élagage) peuvent être implémentés par des parcours en avant et en profondeur, qui permettent une construction paresseuse. On utilise pour cela les propriétés des composantes fortement connexes et les algorithmes permettant leur calcul.

### 3.3.1 Composantes fortement connexes et problèmes d'accessibilité

Commençons par quelques rappels sur les graphes. En effet, les IOLTS ont une structure de graphes dirigés et plusieurs problèmes à résoudre dans la génération de tests peuvent être interprétés comme des problèmes d'accessibilité ou de co-accessibilité dans des graphes. Nous commençons donc par définir les notions utiles sur les graphes (voir par exemple [31]).

**Graphes et CFC :** Soit $G = (V, E)$ un graphe, $V$ son ensemble de nœuds et $E \subseteq V \times V$ son ensemble d'arcs. Soit la relation d'équivalence $\mathcal{R}$ définie sur ce graphe : $v\mathcal{R}w \iff v = w$ ou il existe un chemin de $v$ à $w$ et un chemin de $w$ à $v$. Chaque classe d'équivalence $V_i$ induit un sous-graphe $G_i = (V_i, E_i)$ où $E_i = E \cap (V_i \times V_i)$ restreint $E$ aux arcs entre sommets de $V_i$. Un tel graphe $G_i$ est appelé *composante fortement connexe* (CFC). Elle est *triviale* si elle est réduite à un seul nœud sans boucle. Le quotient $G/\mathcal{R}$ du graphe $G$ par la relation d'équivalence $\mathcal{R}$, définit un graphe acyclique appelé *graphe réduit*, dont les nœuds sont les CFC et un arc existe entre les CFC $G_i = (V_i, E_i)$ et $G_j = (V_j, E_j)$ s'il existe des nœuds $v_i \in V_i$ et $v_j \in V_j$ et un arc de $v_i$ vers $v_j$ dans $E$. Noter que ce graphe n'est pas connexe si $G$ ne l'est pas.

**CFC et accessibilité :** Les CFC ont la propriété suivante : les nœuds $u$ et $v$ sont dans la même CFC maximale si et seulement si, pour tout nœud $w$, il existe un chemin de $w$ à $u$ si et seulement si il existe un chemin de $w$ à $v$. De façon équivalente, les nœuds $u$ et $v$ sont dans la même CFC maximale si et seulement si, pour tout nœud $w$, il existe un chemin de $u$ à $w$ si et seulement si il existe un chemin de $v$ à $w$.

Autrement dit, tous les nœuds d'une même CFC ont les mêmes propriétés d'accessibilité et de co-accessibilité. Or plusieurs problèmes de la synthèse de test sont des problèmes d'accessibilité et de co-accessibilité, qui peuvent se résoudre par calcul de CFC. Un problème d'accessibilité est caractérisé par le calcul de $reach$, alors qu'un problème de co-accessibilité est caractérisé par le calcul de $coreach$.

**Calcul des CFC :** Tarjan [75] décrit un algorithme de complexité linéaire pour calculer les CFC. L'algorithme est fondé sur un parcours en profondeur (DFS pour *Depth First*

*Search*). Grâce à la synthèse d'attributs, il permet d'identifier dans chaque CFC, la *racine* i.e. le premier nœud de la CFC rencontré lors de la DFS. Quand la racine est dépilée, sa CFC est complète. Il est alors possible de superposer à cet algorithme la synthèse de propriétés d'accessibilité et de co-accessibilité. Dans [50, 62], nous en donnons une version itérative générique, instanciée pour plusieurs propriétés d'accessibilité et de co-accessibilité utiles à la synthèse de tests.

### 3.3.2 Utilisation des calculs de CFC

Nous présentons ici comment réaliser pratiquement tous les calculs de l'algorithme *gen_test_OT2* par des parcours en profondeur, certains basés sur les calculs de CFC. Nous illustrerons les algorithmes par l'exemple de la figure 3.20 et les suivantes i.e. figures 3.21, 3.22 et 3.23.

**Produit synchrone :** Dans la définition du produit synchrone $PS = S \times TP$ (voir définition 3.2.6), on peut se limiter à la partie accessible de $Q^{\text{S}} \times Q^{\text{TP}}$ depuis $(q_0^{\text{S}}, q_0^{\text{TP}})$ par la relation $\rightarrow_{\text{PS}}$ définie par $\rightarrow_{\text{S}}$ et $\rightarrow_{\text{TP}}$. Le calcul se résume donc à un calcul de $reach$ sur $PS$ utilisant $reach$ sur chacun des IOLTS $S$ et $TP$. $PS$ se calcule donc par parcours en largeur ou profondeur.

**Calcul de $\Delta(PS)$ :** Il consiste en théorie à l'ajout de boucles $\delta$ dans chaque état de blocage. Pour les deadlocks et blocages de sortie, il suffit d'examiner l'ensemble des transitions sortantes. Pour un livelock, une boucle $\delta$ devrait être ajoutée à chaque état d'une CFC non triviale d'actions internes (appelée $\tau$-CFC par la suite) ($\{0, 2\}$ par exemple dans la figure 3.21). Or, puisqu'une déterminisation suit, par la relation entre CFC et propriété de co-accessibilité évoquée plus haut, il suffit qu'une boucle $\delta$ soit ajoutée à la racine de chaque $\tau$-CFC (0 ici) pour assurer que pour tout état $q$ de la CFC, $q \overset{\delta}{\Rightarrow} q$. Pour des raisons d'efficacité, ceci peut être factorisé avec la déterminisation.

**Déterminisation :** La déterminisation de $\Delta(PS)$ (comme pour un automate, voir par exemple [39]) consiste à construire $PS^{\text{VIS}} = det(\Delta(PS))$ en partant du méta-état initial $q_0^{\text{VIS}} = q_0^{\Delta(\text{PS})}$ after $\varepsilon$ (ici $\{0, 1, 2, 3\}$ dans $PS$ de la figure 3.21) en alternant les deux opérations suivantes : (i) le calcul de sous-ensemble i.e. pour un ensemble d'états $P$ et une action visible $a$ le calcul de $P' = post_{\{a\}}(P) = \{q' | \exists q \in P, q \overset{a}{\rightarrow} q'\}$, l'ensemble d'états accessibles en un pas visible $a$ depuis $P$ (pour $P = \{0, 1, 2, 3\}$, $?a$ donne $P' = \{4, 7\}$ et $?b$ donne $P' = \{5\}$). (ii) l'$\varepsilon$-clôture du résultat $P'$ i.e. pour l'ensemble d'états $P'$, le calcul de $P'$ after $\varepsilon$, l'ensemble des états accessibles depuis $P'$ par des séquences d'actions internes, ou encore $reach_{\text{PS}}(P)$ (donne $\{4, 7\}$ pour $P' = \{4, 7\}$ et $\{5, 10\}$ pour $P' = \{5\}$). L'algorithme de [39], peut présenter des redondances dans l'$\varepsilon$-clôture dans le cas où un état est impliqué dans plusieurs calculs d'$\varepsilon$-clôture (l'état 12 par exemple). On peut éviter cette redondance au prix d'un stockage d'information supplémentaire [49]. Le principe est de remarquer que pour un état $q$, l'ensemble $Tir(q) = \{(a, q') \mid a \in A^{\Delta(\text{PS})} \setminus I^{\Delta(\text{PS})}, q' \in Q^{\Delta(\text{PS})}, q \overset{a}{\Rightarrow} q'\}$ est une propriété de co-accessibilité, donc constant sur toute $\tau$-CFC (e.g. $Tir(0) = Tir(2) = \{(?a, 4), (?a, 7), (?b, 5), (?c, 6), (!\delta, 0)\}$). L'$\varepsilon$-clôture est donc

fondé sur un calcul de graphe réduit de $\tau$-CFC avec synthèse de $Tir(racine(V_i))$ sur chaque $\tau$-CFC $V_i$. Au passage, les blocages sont calculés et les actions $\delta$ ajoutées, un livelock étant une $\tau$-CFC non triviale. Quand un état $q$ est revisité par un nouvel appel d'$\varepsilon$-clôture, la racine de sa $\tau$-CFC fournit $Tir(q)$. Pour un méta-état $P$, l'ensemble $Tir(P) = \bigcup_{V_i \in CFC_{\text{init}}} Tir(racine(V_i))$, où $CFC_{\text{init}}$ est l'ensemble des CFC initiales du graphe réduit de $\tau$-CFC de $P$, fournit les transitions tirables et les états atteints, donc le résultat du calcul de sous-ensemble (e.g. si $P = \{0,1,2,3\}$, $CFC_{\text{init}}(P) = \{\{0,2\}\}$ et $Tir(\{0,1,2,3\}) = \{(?a,4),(?a,7),(?b,5),(?c,6),(!\delta,0)\}$ donc $P \xrightarrow{?a} \{4,7\}$, $P \xrightarrow{?b} \{5\}$, $P \xrightarrow{?c} \{6\}$ et $P \xrightarrow{!\delta} \{0\}$). La complexité de la déterminisation reste exponentielle mais l'algorithme est nettement plus efficace qu'une implémentation naïve de [39].

**Minimisation :** $PS^{\text{VIS}}$ n'est pas minimal pour l'équivalence de traces. Le minimiser nécessiterait de le construire entièrement, les algorithmes de minimisation fonctionnant en arrière, par raffinement de partition. Or, dans la synthèse à la volée (voir 3.3), on veut ne construire $PS^{\text{VIS}}$ que partiellement et en avant. On utilise donc une équivalence plus faible que l'équivalence de traces et on minimise $PS^{\text{VIS}}$ pour cette relation : deux méta-états $P_i$ et $P_j$ de $PS^{\text{VIS}}$ sont *équivalents à profondeur 1* si les ensembles $Tir(P_i)$ et $Tir(P_j)$ sont identiques. Cette minimisation est automatique par codage d'un méta-état $P$ par l'ensemble $Tir(P)$, au lieu d'un codage par les états constituant $P$.

**Sélection :** Le calcul de $CTG$ depuis $PS^{\text{VIS}}$ est aussi un problème d'accessibilité et de co-accessibilité puisque $L2A$ est défini essentiellement par l'intersection de $reach_{|\overline{Accept^{\text{VIS}}}}(q_0^{\text{VIS}})$ et $coreach(Accept^{\text{VIS}})$. Appartenir à chacun de ces ensembles est donc un invariant de CFC. Dans l'exemple de la figure 3.22, $L2A = \{0,1,2,4,6,9,10,11,13\}$). Il suffit donc d'identifier les CFC dont la racine appartient à $L2A$ (e.g. $\{13\}$, $\{6,9,10,11\}$, $\{2\}$, $\{1\}$, $\{4\}$ et $\{0\}$) et d'extraire de $PS^{\text{VIS}}$ le sous-graphe composé des états de ces CFC et des transitions dans, et entre ces CFC. L'algorithme appelé TGVloop adapte l'algorithme de Tarjan en synthétisant l'appartenance à $L2A$ et en construisant les transitions de $\rightarrow_{\text{L2A}}$ pendant le *backtracking*. Le calcul de **Inconc** (e.g. $\{5,7\}$) et $\rightarrow_{\textbf{Inconc}}$ (e.g. $1 \xrightarrow{!x}_{\textbf{Inconc}} 5$ et $2 \xrightarrow{!z}_{\textbf{Inconc}} 7$) est effectué au *backtracking* de transitions de sorties de $PS^{\text{VIS}}$ ($!x$ et $!z$) issues d'états de $L2A$ (e.g. 1 et 2) vers des états hors de $L2A$ (e.g. 5 et 7). Noter que 3 n'est pas dans **Inconc** car atteint par l'entrée $?c$. L'état *Fail* et les transitions de $\rightarrow_{\textbf{Fail}}$ sont implicites, $\rightarrow_{\text{Fail}}$ étant défini par complémentaire de sorties tirables. Comme le calcul de CFC, l'algorithme est de complexité linéaire en temps et en espace. L'algorithme est illustré par le schéma de la figure 3.25.

**Élagage :** On a vu précédemment que plusieurs stratégies étaient possibles pour élaguer le graphe de test complet CTG en un ou plusieurs cas de test. Une possibilité implémentée dans TGV, est de choisir un seul cas de test, de profondeur minimale. Pour ce faire, on adapte le calcul de $coreach(\textbf{Pass})$, qui s'implémente naturellement par un parcours arrière en largeur. Par définition de $CTG$, tous les états de $L2A$ sont dans $coreach(\textbf{Pass})$. A l'étape $n$ de l'algorithme, on atteint éventuellement des nouveaux états dans $new - coreach(n, Pass) = pre^n(\textbf{Pass}) \setminus \cup_{0 \leq i \leq n} pre^i(\textbf{Pass})$ i.e. les états dont la(les) plus courte(s) séquence(s) vers **Pass** est exactement de longueur $n$. L'adaptation consiste à

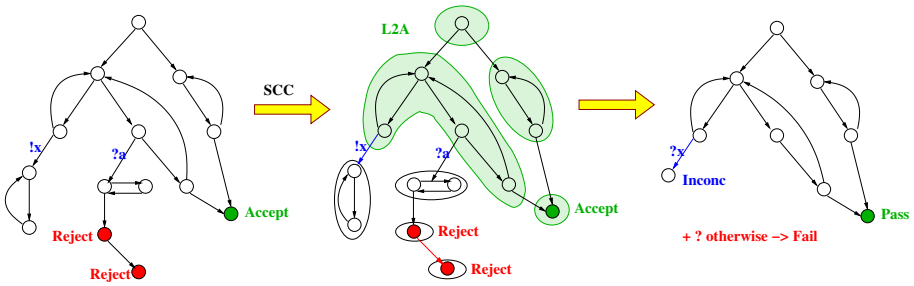



FIG. 3.25 – Calcul du graphe de test complet

chaque itération $n$, et pour chaque état $q$ de $new - coreach(n, \textbf{Pass})$ à ne garder dans $TC$ qu'un sous-ensemble des transitions sortantes de $q$. Puisque $q \in new - coreach(n, \textbf{Pass})$, il existe $a_1, \ldots, a_k \in A^{\text{CTG}}$ tels que $\forall i, q \xrightarrow{a_i} new - coreach(n-1, \textbf{Pass})$. On choisit de régler la contrôlabilité en fonction de la qualité (entrée/sortie) d'une de ces actions choisie arbitrairement (e.g. la première action $a_1$). Si $a_1$ est une sortie de $CTG$, on coupe toutes les autres transitions issues de $q$, i.e. $a_2, \ldots, a_k$ et aussi les transitions $a_{k+1}, \ldots, a_m \in A^{\text{CTG}}$ telles que et $\forall j \geq k+1, q \not\xrightarrow{a_j} new - coreach(n-1, \textbf{Pass})$. Si $a_1$ est une entrée, on coupe toutes les sorties parmi $a_2, \ldots a_k, \ldots a_m$ et on conserve les entrées. On conserve ainsi les entrées menant à **Inconc** s'il y en a, et celles menant à **Fail**. Le fait de couper des transitions peut bien sûr modifier l'accessibilité depuis l'état initial. Les états devenus inaccessibles sont alors supprimés par calcul de $reach(q_0^{\text{TC}})$.

La résolution de conflits de contrôlabilité par parcours en arrière nécessite la construction complète de $CTG$. Mais si on veut calculer un seul cas de test, on peut éviter le calcul complet de $CTG$ en effectuant l'élagage pendant le *backtracking* de la DFS dans TGVloop. Le principe est d'effectuer l'élagage dans un état $q$, sur les transitions explorées ou non, aussitôt que la DFS dépile une transition synthétisant l'accessibilité à $Accept^{\text{VIS}}$. Si la transition est une entrée, on coupe toutes les autres transitions. Si c'est une sortie, on ne conserve que les sorties. Par exemple dans l'état 0, si on explore d'abord $?a$ vers 1, au dépilement de 1, on peut directement élaguer la transition $?b$ et éviter d'explorer 2 et 7. On peut espérer élaguer au plus tôt en explorant en priorité les entrées ou les transitions faisant progresser l'objectif. Cependant, il existe des conflits, dans certains cas de boucles, qui ne peuvent pas être résolus au *backtracking*. En effet, lorsqu'on dépile un successeur $q'$ de $q$, on ne sait pas toujours si $q' \in L2A$. Par exemple pour 6, si depuis 6 on explore $?a$ vers 9 avant $?b$ vers 10. Au pire on ne le sait que lorsqu'on a fini de calculer la CFC de $q'$. On ne peut donc pas appliquer l'élagage et l'IOLTS obtenu n'est alors ni le $CTG$ ni un cas de test. On le note $TG$ pour *test graph* dans la figure 3.19. Dans ce cas, les conflits résiduels seront résolus par l'algorithme d'élagage en arrière *a posteriori* sur $TG$.

### 3.3.3 Enchaînement à la volée

Les algorithmes de la génération de tests à la volée sont exactement ceux présentés plus haut, à ceci près qu'ils ne sont pas appliqués en séquence. Afin de comprendre le fonc-

tionnement global, il faut raisonner en termes des fonctions de construction de chacun des IOLTS $S$, $PS$ et $PS^{\text{VIS}}$. Pour chaque IOLTS, ces fonctions sont similaires à celles produites pour $S$ par le compilateur, auxquelles il faut ajouter les fonctions d'appartenance à $Accept$ et $Refuse$. Par exemple, la fonction de calcul du méta-état initial $q_0^{\text{VIS}}$ de $PS^{\text{VIS}}$ utilise l'algorithme d'ajout des blocages et déterminisation pour calculer l'$\varepsilon$-clôture de l'état initial de $PS = S \times TP$ en explorant les actions internes de $PS$. Le calcul de l'état initial de $PS$ utilise le calcul de l'état initial de $S$ et de $TP$. L'exploration des actions internes de $PS$ utilise l'exploration des actions internes de $S$ et de $TP$ (en ajoutant les $\delta$ nécessaires). L'appartenance de $q_0^{\text{VIS}}$ à $Accept$ ou $Refuse$ dans $PS^{\text{VIS}}$ utilise l'appartenance à $Accept$ et $Refuse$ dans $PS$, qui elle même utilise l'appartenance à $Accept$ et $Refuse$ dans $TP$. Comme le calcul de $q_0^{\text{VIS}}$ calcule également $Tir(q_0^{\text{VIS}})$ en explorant les transitions visibles de $PS$, donc de $S$ et de $TP$, on a directement les transitions visibles tirables depuis $q_0^{\text{VIS}}$. Calculer un méta-état successeur de $q_0^{\text{VIS}}$ revient à effectuer un calcul de sous-ensemble puis $\varepsilon$-clôture comme expliqué plus haut. De même l'appartenance à $Accept$ ou $Refuse$ est propagée. On itère pour ces successeurs pour parcourir $PS^{\text{VIS}}$, en identifiant les états identiques. La sélection dans le calcul de $CTG$ permet d'arrêter la construction sur les états $Accept$ et $Refuse$ de $PS^{\text{VIS}}$.

Au pire, la synthèse à la volée n'apporte aucune réduction à la construction des IOLTS $S$, $PS$ et $PS^{\text{VIS}}$, la complexité théorique est donc inchangée. Mais en pratique la réduction est souvent très importante, d'autant plus que $TP$ est contraignant, en particulier par l'usage d'états $Refuse$. Cette technique nous a souvent permis de synthétiser très rapidement des cas de tests sur des spécifications qu'il était impossible de construire complètement, en particulier dans le cas infini. Cependant, si $S$ est de taille raisonnable, il est préférable de le calculer complètement, de le minimiser pour une équivalence préservant les traces et blocages, et le réutiliser pour différents objectifs de test. On a noté aussi que la synthèse à la volée ne permet pas la minimisation de $PS^{\text{VIS}}$ pour l'équivalence de traces. Il en résulte parfois un dépliage des boucles, répercutée dans les cas de tests. Mais comme ceux-ci sont en général de petite taille, on peut les minimiser pour l'équivalence de traces.

## 3.4 L'outil TGV

### 3.4.1 Architecture de TGV

Elle suit très exactement la découpe fonctionnelle décrite précédemment (voir figure 3.26). TGV est construit par couches communicant par API (*Application Programming Interface*). Chaque couche prend en entrée l'API d'un IOLTS (deux dans le cas du produit), i.e. les fonctions de construction de cet IOLTS, et fournit en sortie l'API d'un autre IOLTS en implémentant un des algorithmes décrits à la section 3.2. TGV utilise aussi des librairies de stockage, de masquage, renommage et d'expressions régulières de la boîte à outils CADP [25]. Grâce à cette architecture, TGV permet de piloter des simulateurs de différents langages de spécification, avec un code source identique, sauf aux API hautes. La cohérence des différentes versions est donc assurée, et le portage sur de nouvelles architectures facilité (TGV fonctionne sur différentes versions d'Unix, Linux et Windows). De plus, certaines parties peuvent être utilisées séparément, ou par d'autres programmes. En particulier, un outil de vérification de cas de test, VTS, détectant et corrigeant le biais et le

laxisme, a été implémenté en remplaçant seulement l'algorithme de sélection TGVloop, et en conservant le produit synchrone et la déterminisation. Cet outil a permis de vérifier des cas de test manuels (voir plus loin) et ceux produits par TGVloop pendant son développement.

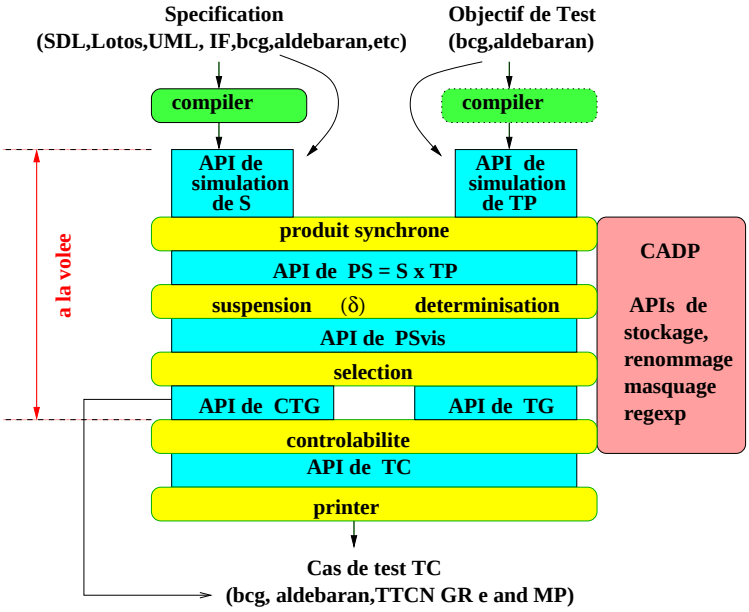



FIG. 3.26 – Architecture de TGV

### 3.4.2 Langages supportés

Grâce à son interfaçage simple avec différents simulateurs, TGV supporte actuellement plusieurs langages de spécification :

**Lotos :** TGV est interfacé avec l'API de simulation fournie par le compilateur CAESAR de la boîte à outils CADP [25] (http ://www.inrialpes.fr/vasy/cadp/). Lotos ne distinguant pas entrées et sorties, celles-ci sont différenciées par des expressions régulières fournies par un fichier additionnel. Cette version de TGV est distribué gratuitement avec la boîte à outils CADP sous la forme d'une librairie.

**SDL :** TGV est interfacé avec le simulateur SDL ObjectGéode (Telelogic) [35]. Il existe deux versions de cet interfaçage. La version académique utilise une interface similaire à celle de CADP et pilote le simulateur ObjectGéode. La version commerciale de TGV est plus intégrée à l'outil TestComposer d'ObjectGéode qui dispose de deux moteurs de synthèse de tests différents. TestComposer est aussi équipé, en amont de la synthèse de test, d'un outil de synthèse d'objectifs de test. Pour un critère de couverture des transitions de la spécification, il produit par simulation des séquences d'actions observables, interprétées ensuite comme objectifs de test.

**UML :** TGV peut produire des tests à partir de modèles UML grâce à son interfaçage avec UMLAUT [37] (http ://www.irisa.fr/UMLAUT), un *framework* de validation de modèles UML développé à l'Irisa dans l'équipe Triskell. UMLAUT utilise les diagrammes de classe, d'objet, de déploiement et les statecharts et donne une sémantique opérationnelle à UML par transformation et compilation du modèle UML. UMLAUT utilise la même librairie TGV que CADP pour Lotos.
**IF :** TGV peut aussi piloter le simulateur de spécifications IF (forme intermédiaire produite depuis SDL et UML) de Vérimag qui fournit une API similaire à CADP. Une nouvelle version de TGV avec une nouvelle interface IF utilisant des objectifs de test de plus haut niveau et des critères de couverture est en cours de développement.
**Autres langages :** enfin, TGV peut synthétiser des tests pour d'autres langages de spécification. Il suffit pour cela de disposer d'un compilateur produisant du code de simulation et de développer une passerelle entre l'interface de simulation produite par le compilateur et l'API attendue par TGV. En sortie de TGV, il est possible de produire des cas de tests sous forme tabulaire en pseudo-TTCN, ou dans les formats de graphe .aut et .bcg de CADP.

### 3.4.3 Autres caractéristiques de TGV

TGV dispose de plusieurs options. En particulier, il permet de produire des tests avec gestion de temporisateurs. Par exemple la cas de test de la figure 3.24 peut être transformé dans le cas de test avec temporisateurs de la figure 3.27.

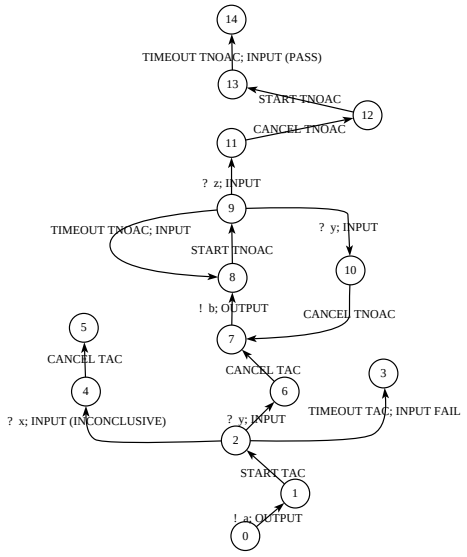



FIG. 3.27 – Cas de test avec temporisateurs

Deux temporisateurs sont gérés, TAC et TNOAC. TAC est démarré en attente d'une ou plusieurs entrées (excepté si $\delta$ est tirable), il est arrêté à l'occurrence d'une entrée ou expire

sinon (*timeout*) en produisant un verdict *Fail*. TNOAC est démarré quand un blocage ($\delta$) est tirable, arrêté si une entrée est observée. Son expiration remplace l'observation de $\delta$, et ne produit pas de *Fail* puisque spécifié. On suppose que la valeur de TNOAC et TAC est suffisamment grande pour que leur expiration soit assimilée à un blocage. On peut aussi borner la profondeur de parcours en termes des actions visibles de $PS^{\text{VIS}}$, une borne en termes d'actions de $S$ pouvant provoquer le biais en cas de non déterminisme. TGV permet le calcul de postambules à partir d'états de **Pass** et **Inconc**. Ces postambules servent à terminer proprement l'exécution d'un cas de test sur l'IUT avant d'enchaîner sur un autre cas de test. Leur calcul consiste à poursuivre le calcul de cas de test jusqu'à des états *stables* de $PS^{\text{VIS}}$, i.e. des états où aucune entrée n'est attendue [3]. Enfin, pour une spécification $S$ donnée, l'architecture de test peut être modulée par la description de $A_I^{\text{s}}$, $A_O^{\text{s}}$ et $I^{\text{s}}$ par des expressions régulières.

### 3.4.4 Études de cas

Différentes versions de TGV ont été utilisées dans des études de cas de taille industrielle, dans des domaines variés et pour tous les langages de spécifications supportés par TGV. Ces études de cas ayant été publiées par ailleurs, nous n'en donnons ici qu'un bref aperçu, en synthétisant les apports ou les limitations de TGV.

**Protocole DREX :** Le protocole DREX, version militaire du protocole ISDN D, a permis la validation des principes de TGV sur une première version ne fonctionnant pas complètement à la volée [27]. Nous avons travaillé sur plusieurs spécifications SDL de 5 services différents du DREX. Chaque spécification consistait en une trentaine de transitions SDL, ce qui représentait environ 1800 lignes de SDL textuel, ou encore 50 planches de SDL graphique. La génération de tests consistait d'abord à calculer les graphes d'états de chaque spécification en le contraignant par un environnement limité. Les graphes produits faisaient entre quelques dizaines et quelques milliers de transitions. Nous appliquions ensuite l'embryon de TGV sur ces graphes explicites. En comparant les résultats produits par l'outil TVéda du CNET [68] et le prototype TOPIC de Vérilog, ainsi que la production manuelle de tests, l'expérimentation a permis de montrer la qualité des tests générés par TGV et son efficacité [20].

**Architectures multiprocesseurs :** Plusieurs expériences [51, 52] ont consisté à étudier la génération de test depuis des spécifications d'architectures multiprocesseurs (Bull) décrites en LOTOS. L'une des spécifications consistait en 2000 lignes de LOTOS avec une partie types de donnée abstrait importante. Ces expériences ont mené à la réalisation de la première version de TGV fonctionnant complètement à la volée et connectée à l'environnement CADP. Nous devons reconnaître quelques difficultés liées principalement à une culture différente du test dans ce domaine d'application par rapport au domaine des télécoms dont nous étions issus. En particulier les tests étaient habituellement constitués d'un ensemble d'entrées et l'attente des sorties correspondantes, et exécutés en batch, alors que

[3] Ce choix est dû à l'impossibilité, en général, de rejoindre l'état initial si $PS^{\text{VIS}}$ n'est pas fortement connexe. C'est vrai en particulier à la volée où la minimisation est partielle.

nous proposions un test plus adaptatif. Cependant un certain nombre de cas de tests produits par TGV ont été exécutés par des ingénieurs Bull sur des simulateurs de l'architecture et l'appréciation des utilisateurs a été positive.

**Le protocole SSCOP :** Le protocole SSCOP est un protocole de communication normalisé par l'ATM Forum qui a servi lors de plusieurs études de cas. Nous disposions d'une spécification SDL consistant en un seul processus (2 dans le cas asynchrone) de 175 transitions, d'une taille de 7000 lignes de SDL textuel, soit 100 pages de SDL graphique. Ce protocole a servi à plusieurs campagnes de synthèse de test. Une cinquantaine d'objectifs de test complexes ont été formalisés et ont servi à la génération des tests correspondants. En variant sur le nombre de PCO, le mode de communication synchrone ou asynchrone entre testeur et IUT, ceci a permis de mettre en valeur les possibilités de TGV dans sa version connectée au simulateur ObjectGéode [5].

Une autre expérience a été menée sur la base de la même spécification et d'une suite de test TTCN de l'ATM Forum écrite manuellement. Nous avons tenté de vérifier automatiquement cette suite de tests (pour l'absence de biais et de laxisme) par notre outil de vérification de tests VTS. Cette expérience a permis de détecter plusieurs erreurs, en particulier dues à l'asynchronisme entre testeur et IUT qui rend très difficile l'écriture manuelle des tests [46].

**Protocole de conférence virtuelle :** TGV a été utilisé sur un protocole de conférence virtuelle [22], principalement sur une spécification LOTOS, mais aussi sur une spécification SDL. Les tests produits par TGV ont été ensuite exécutés sur 28 mutants d'une implémentation. Le but de l'expérience menée avec nos collègues de l'Université de Twente, était de comparer les puissances de détection de non conformité de TGV et de TorX. Les résultats étaient identiques au sens où les deux outils ont permis de détecter tous les mutants non conformes à la spécification. Il faut cependant reconnaître que la difficulté principale pour TGV était d'imaginer des objectifs adéquats qui couvrent suffisamment les comportements de l'IUT. TorX, qui génère et exécute les tests à la volée en se basant sur une génération aléatoire, ne souffre pas de ce problème. Mais on a pu également générer aléatoirement des séquences, utilisés ensuite comme objectifs par TGV pour obtenir un résultat équivalent.

**Contrôleur aérien :** un modèle de contrôleur de trafic aérien décrit en UML sert de démonstrateur de la connexion UMLAUT/TGV. Le modèle UML consiste en un diagramme de classes (6 classes), un diagramme d'objet décrivant l'état initial, un diagramme d'état (statechart) par classe et par acteur ( décrivant et contraignant l'environnement). Quelques objectifs de test ont été formalisés pour montrer la faisabilité de l'approche.

**Gestion de flotte routière :** dans le cadre du projet européen Agedis (http ://www.agedis.de/), TGV a été utilisé sur une spécification IF (retranscrite en SDL à partir d'une spécification UML et compilée en IF) d'un logiciel de gestion de flotte routière. La spécification était constituée de 10 processus de petite taille (quelques transitions simples), environ 700 lignes de code IF. Malgré son apparente taille modeste, la concurrence entre ses 10 processus était telle que le graphe d'état complet était d'environ

50 000 états et approchait 900 000 transitions. La génération des tests à la volée était cependant assez rapide. Cependant, nous avons noté qu'il y avait beaucoup de concurrence entre actions internes, ce qui amène à construire des hypercubes de transitions internes de dimension 10, alors que les actions visibles tirables après ces actions internes concurrentes ne dépend pas de leur ordre. Cette remarque nous amène à l'idée que l'utilisation de techniques d'ordre partiel (voir 5) pour repousser les limites de la génération de test, dans TGV en particulier.

## 3.5 Comparaison avec d'autres techniques et outils

Différentes théories du test et algorithmes de synthèse ont été élaborées depuis quelques années. Nous les comparons ici avec l'approche de TGV.

**Techniques fondées sur les automates :** Nous avons esquissé en section 2.3.1 les techniques de génération de tests fondées sur les machines de Mealy. Revenons un instant sur ses principes pour les comparer aux techniques basées sur les systèmes de transitions. L'argument principal des défenseurs de ces techniques fondées sur les automates est qu'elles sont complètes, alors que les techniques fondées sur les systèmes de transitions ne le sont que si on considère des suites de tests infinies. C'est exact, mais il faut noter que les premières ne sont complètes que sur un ensemble fini d'implémentations, grâce aux hypothèses faites sur celles-ci (mêmes alphabets, borne sur le nombre d'états). Les secondes considèrent *a priori* un ensemble infini d'implémentations, donc des fautes pouvant apparaître après des comportements de taille non bornée, qui nécessitent donc des tests de taille non bornée. Les notions de complétude ne sont donc pas comparables telles quelles. On pourrait faire des hypothèses plus fortes sur les secondes afin de se restreindre à un nombre fini d'implémentations, et obtenir aussi des résultats de complétude avec des suites de test finies, en utilisant également des résultats de la théorie des automates. Mais le point de vue de départ des techniques fondées sur les (IO)LTS est différent, et la complétude (finie) a rapidement été abandonnée pour justement éviter de faire des hypothèses sur l'implémentation, que l'on ne saura de toutes façon pas vérifier. L'autre différence principale est le traitement du non-déterminisme. On trouve dans la littérature des articles dans lesquels on génère des tests sur des machines de Mealy "non-déterminisme"(voir par exemple [58]). Mais il s'agit de non-déterminisme observable (voir 3.1.3), pas de non déterminisme au sens des automates (voir 3.1.2). Les techniques à base d'(IO)LTS traitent les deux, encore au prix d'un affaiblissement de la notion de complétude (toutes les IUT non-conformes *peuvent* être rejetées).

**Test et model-checking :** TGV utilise des principes du model-checking pour synthétiser des tests. De nombreuses autres techniques utilisent directement les outils de model-checking existants (par exemple [24]). Leur principe est en général d'utiliser la faculté de diagnostic des model-checkers, c'est à dire de produire des contre-exemples quand une propriété n'est pas satisfaite. On décrit donc un objectif de test par une propriété $P$ en logique temporelle (CTL, LTL) sur la spécification $S$. On vérifie ensuite si $S$ est un modèle de $\neg P$, i.e. $S \models \neg P$ en utilisant un model-checker (SMV, SPIN, ...). Puisque $P$ est en

général une propriété valide sur $S$ (i.e. $S \models P$), $\neg P$ n'est en général pas valide sur $S$ (i.e. $S \not\models \neg P \iff S \models P$). Le model-checking de $\neg P$ sur $S$ produit alors une (ou plusieurs) séquences contre-exemples de $\neg P$, qui sont donc des séquences témoins de $P$. Un test est alors obtenu par image miroir d'une telle séquence et complétion par des entrées vers **Inconc** correspondant à des sorties de la spécification le long de la séquence. Si toutefois $S \models \neg P$, il n'y a aucune séquence témoin de $P$ dans $S$, donc pas de test pour l'objectif. Ces techniques produisent donc des tests simples, voire simplistes. Elles ne traitent pas les problèmes de non déterminisme (parfois pas non plus de non déterminisme observable) ni d'observation partielle, ce qui restreint leur utilisation sur des spécifications déterministes (et déterministes observables) et dont tous les comportements sont observables. D'autres techniques voisines [3, 38] décrivent non pas des objectifs mais des critères de couverture et produisent un ensemble de séquences pour ces critères. [38] décrit en particulier en CTL des critères de couverture sur les flots de contrôle (couverture d'états de contrôle, de transitions) et de données (définition-utilisation).

On peut remarquer que dans ces travaux utilisant le model-checking pour la génération de tests, les exemples de propriétés donnés sont des propriétés de type accessibilité décrites en CTL par des formules imbriquant les $EF(t \wedge \ldots)$ (il existe un chemin sur lequel $t$ est vrai dans le futur et ...) et les $E(\neg t U t' \wedge \ldots)$ (il existe un chemin où t est faux jusqu'à ce que $t'$ soit vrai et ...). Ces objectifs peuvent donc être facilement transformés en objectifs tels que ceux utilisés par TGV. Notons d'ailleurs que les model-checkers de ce type de formules sont basés sur une utilisation intensive de l'opération $coreach$ puisque si une formule $f$ est satisfaite par un ensemble d'états $F$, l'ensemble des états satisfiant $EFf$ est exactement $coreach(F)$. De même $EfUg$ est calculé par une variante de $coreach$ qui remonte de l'ensemble des états où $g$ est vrai, le long des chemins où $f$ est vrai.

**Autres techniques fondées sur les objectifs de test :** Samstag [34] et sa version commerciale dans l'outil Tau de Telelogic, utilise également des objectifs de test pour la synthèse. Ceux-ci sont décrits par des MSC (Message Sequence Charts) décrivant des séquences complètes d'actions (pas d'abstraction). L'outil n'est pas fondé sur une théorie du test bien établie. Cependant, il s'agit d'extraire des comportements de la spécification, des séquences décrites par l'objectif. Le problème du non déterministe n'est pas résolu car si deux séquences de la spécification ont la même trace visible, la synthèse de tests est impossible. Par manque de possibilité d'abstraction, les tests produits sont des séquences complétées par des Inconclusifs. [70] utilise aussi des objectifs de test, du même type que TGV. L'idée originale est d'utiliser la théorie des jeux pour modéliser l'activité de test. Le test est vu comme un jeu entre l'implémentation et le testeur. Le testeur gagne s'il met l'IUT en faute ou s'il atteint l'objectif de test. Le calcul d'un cas de test consiste à calculer une stratégie gagnante pour le testeur. Pour ceci, on doit optimiser le contrôle qu'a le testeur sur l'IUT. Cette mesure de contrôle, calculée sur un produit entre spécification et objectif, dépend de la distance à une configuration acceptante et du non déterminisme observable (qui correspond aux choix de l'IUT de ne pas satisfaire l'objectif). Malheureusement, la stratégie se calcule en arrière et se prête apparemment peu au calcul à la volée.

**Autres techniques à la volée :** L'outil TGV est proche de l'outil TorX [18]. Ce dernier calcule également des tests à la volée pendant leur exécution sur une IUT, alors que TGV

doit d'abord synthétiser pour éventuellement exécuter. Du coup, TorX explore seulement une séquence visible, puisqu'il choisit en fonction de $det(\Delta(S))$ une sortie à appliquer et compare l'entrée reçue avec celles prévues. TorX est donc pratique pour un test intensif mais nécessite un interfaçage particulier pour chaque IUT testée. L'avantage de TGV est de fournir une solution au problème de la sélection des tests, problème à l'étude aussi pour TorX [19]. Si on revient d'ailleurs un instant sur la différence entre l'algorithme $gen_test$ en 3.2.1 qui est à la base de TorX, et l'algorithme $gen_test_OT1$ par exemple qui est à la base de TGV, on peut noter que le premier est fondamentalement basé uniquement sur l'accessibilité depuis l'état initial, les états accepteurs étant choisis aléatoirement, alors que le second est basé à la fois sur l'accessibilité depuis l'état initial et sur la co-accessibilité aux états accepteurs. $gen_test_OT1$ est donc un raffinement de $gen_test$.

## 3.6 Conclusion sur la génération de tests énumérative à la volée

Dans ce chapitre, nous avons présenté les principes de TGV, la théorie du test sous-jacente, les algorithmes et l'outil. La théorie n'est pas pour l'essentiel notre contribution mais celle de Tretmans et de ses collègues de Twente. Nous en avons cependant compris finement, adapté et amélioré certains aspects. Dans ce document, nous avons voulu présenter la théorie du test d'une manière moins algébrique que dans les travaux de Tretmans et plus accessible au lecteur non familier en particulier des notions de processus et d'équivalences comportementales. Nous nous sommes, le plus souvent possible appuyés sur des notions mieux connues de langages, d'automates qui permettent d'expliquer toute cette théorie de manière plus accessible.

Notre apport à la génération de tests n'est pas dans cette récriture de la théorie. Nous avons surtout travaillé sur les aspects algorithmiques. Au départ, nous présentions la génération de test et ses algorithmes de manière très détaillée, directement en termes d'algorithmique des graphes. Dans ce document, nous avons essayé de prendre de la hauteur sur cet aspect algorithmique. Nous avons d'abord présenté les algorithmes en termes des langages et des traces qu'ils calculent. Puis nous avons présenté de manière plus précise leur implémentation, et en particulier l'aspect de calcul à la volée.

Un autre apport important de notre travail est donc aussi l'implémentation efficace de ces algorithmes de génération de tests. L'efficacité dépend assez étroitement de l'architecture de l'outil TGV que nous avons conçu. Le résultat est un outil performant, qui est capable de synthétiser des tests pour des spécifications de taille industrielle.

Notre travail, et l'outil TGV qui en est issu, ont permis d'améliorer l'état de l'art de la synthèse de test de façon significative. Les succès de TGV dans plusieurs contrats impliquant des industriels ayant été amenés à utiliser différentes versions de TGV en sont lestémoins. Cependant, il reste encore du chemin à parcourir pour rendre la synthèse de tests encore plus utilisable dans le milieu industriel.

Un inconvénient de TGV est l'emploi de techniques énumératives. Or l'énumération peut empêcher de traiter des spécifications manipulant des structures de données ayant des domaines de valeurs importants (voire infinis), même avec des techniques à la volée. Elle interdit aussi de traiter des spécifications paramétrées. De plus, les tests produits sont aussi

énumérés. Une solution que nous avons explorée est l'usage de techniques symboliques qui fait l'object du chapitre suivant.

**Note:** *L'outil TGV est une œuvre commune et de longue haleine. Je tiens donc à remercier tous les participants à sa conception et à son développement, à l'Irisa et à Verimag : J.-C. Fernandez, C. Jard, A. Kerbrat, P. Morel, L. Nedelka, J. Sifakis, S. Simon, C. Viho, V. Tschaen, E. Demairy et quelques stagiaires. Je remercie également L. Mounier et M. Bozga de Verimag pour la connexion à IF, A. Le Guennec pour UMLAUT, H. Garavel et l´équipe VASY de l'Inria Rhônes-Alpes pour leur aide à la connexion de TGV à Lotos, le support et la distribution de TGV dans CADP.*

# Chapitre 4

# Génération de tests symboliques

Les algorithmes de génération de test que nous avons présentés jusqu'ici sont basés sur l'hypothèse que la sémantique opérationelle d'une spécification est donnée en termes de systèmes de transitions. Les spécifications étant données dans des langages de plus haut niveau comme par exemple SDL, UML ou LOTOS, celles-ci manipulent des structures de données complexes, des entrées et sorties portant des paramètres de communication. Les algorithmes "à la TGV" nécessitent une énumération des valeurs de ces variables et paramètres, durant l'exploration du modèle, afin de construire les systèmes de transition. Cette énumération est une cause possible d'explosion combinatoire. De plus, l'énumération complète est impossible aussitôt que les domaines des variables sont infinis, ou que les spécifications contiennent des constantes symboliques [1], ce qui est parfois le cas de spécifications génériques. Les tests produits sont aussi des systèmes de transitions.

Pour limiter cette explosion combinatoire, il peut être intéressant de ne pas travailler au niveau système de transition, mais sur un modèle de plus haut niveau, dans lequel on manipule ces variables, paramètres et constantes symboliques. Au lieu d'énumérer les valeurs de variables, paramètres ou constantes dans les états constituant un système de transition, et d'instancier les transitions, on se propose de manipuler les états de manière symbolique, en caractérisant des ensembles d'états par des formules et en conservant les manipulations de variables, constantes et paramètres, ce qui permet de générer des tests sous forme de programmes.

Dans [72], nous avons proposé un modèle et décrit la génération de tests symboliques pour ce modèle. Ceci a été implémenté depuis dans l'outil STG [13] et utilisé en particulier pour une étude de cas de taille conséquente, le CEPS [12]. Le modèle est basé sur les systèmes de transitions à entrées et sorties, mais étendu par des constantes symboliques, des variables, des paramètres de communication, des gardes et des affectations, des actions d'entrées et sorties portant des paramètres de communications. Le modèle est appelé $\mathcal{IOSTS}$ pour *Input/Output Symbolic Transition System*. La sémantique d'un $\mathcal{IOSTS}$ est définie par un IOLTS. La théorie du test, basée sur la relation de conformité, est définie à ce niveau sémantique, puisque la conformité d'une implémentation se juge sur les com-

[1] On appelle constantes symboliques des constantes dont la valeur est inconnue. On parle parfois de paramètres, mais nous réservons ici ce terme aux paramètres de communications transportés par les actions

portements observables. En particulier, on peut prendre la même relation **ioco** que dans le cas énuméré. On a vu au chapitre précédent que la génération de tests sur les IOLTS est basée sur un certain nombre d'opérations (produit, suspension, déterminisation, sélection). Le traitement symbolique consistera d'abord à définir des transformations syntaxiques sur les $\mathcal{IOSTS}$ qui permettent de réaliser les transformations voulues au niveau sémantique. La phase de sélection du graphe de tests complet nécessitera une analyse sémantique pour identifier les états accessibles et co-accessibles. Malheureusement, contrairement au cas énuméré,ces problèmes sont indécidables. On est donc amené à effectuer une analyse approchée par sur-approximation des états accessibles et co-accessibles. Il faut cependant noter que, bien que l'analyse soit approchée, la sémantique des cas de tests produits est exacte, au sens où les IOLTS qu'ils définissent satisfont les mêmes propriétés de correction que dans le cas énuméré. D'autre part, on sera obligé de faire certaines restrictions sur le non-déterminisme, car on ne sait pas déterminiser de manière effective dans le cas général.

Plusieurs personnes de mon équipe ont participé aux travaux sur la génération de tests symboliques. Initialement Vlad Rusu, Lydie du Bousquet et moi-même avons défini le cadre théorique. Puis Duncan Clarke et Elena Zinovieva nous ont rejoint, ce qui a permis de réaliser une première implémentation et l'étude de cas du CEPS. François-Xavier Ponscarme a ensuite travaillé sur STG pour permettre en particulier l'exécution des tests en Java. Enfin, Bertrand Jeannet et Elena ont interfacé STG avec l'outil NBAC pour permettre une meilleure analyse par interprétation abstraite.

Il n'est pas question ici de faire un exposé très détaillé de la génération de tests symbolique, puisque ceci a fait l'objet de publications et sera relaté dans le document de thèse d'Elena Zinovieva. J'ai choisi ici de faire une présentation synthétique, donnant les éléments principaux de cette technique de génération, en généralisant le modèle et en me basant sur les résultats obtenus dans le cas énuméré.

## 4.1 Le modèle des $\mathcal{IOSTS}$

Le modèle de système de transitions symboliques que nous utiliserons peut être vue comme une extension des systèmes de transitions. Il est plus proche syntaxiquement d'un language de spécification, car on manipule des données (constantes, variables, paramètres de communication) par l'intermédiaire de gardes, d'affectations.

On définit d'abord une syntaxe abstraite des $\mathcal{IOSTS}$ puis sa sémantique en termes de systèmes de transitions. Une syntaxe concrète possible pour les $\mathcal{IOSTS}$, utilisée dans notre outil STG [13], est celle des automates étendus communicants, proche du format intermédiaire IF [6](http ://www-verimag.imag.fr/ async/IF/).

### 4.1.1 Syntaxe des $\mathcal{IOSTS}$

Dans notre modèle, nous utiliserons des données. Une donnée $d$ a un type, noté $type(d)$, qui prend ses valeurs dans un ensemble noté $Dom(d) = Dom(type(d))$. Pour un ensemble $B = \{d_1, \ldots, d_n\}$ de données, on notera $Dom(B)$ le produit $Dom(type(d_1)) \times \cdots Dom(type(d_n))$ des domaine des données $d \in B$. Un élément de $Dom(B)$ est un vecteur de valeurs des données de $B$.

Nous utiliserons des prédicats sur les données pour décrire des gardes. Un prédicat $P$ sur un ensemble de données $B$ définit un sous-ensemble de $Dom(B)$, le sous-ensemble des vecteurs de $Dom(B)$ qui satisfont le prédicat $P$. De façon équivalente, on considérera parfois un prédicat $P$ comme une fonction de $P$ de $Dom(B)$ dans l'ensemble $true, false$ qui rend $true$ pour les éléments de $P$.

Donnons maintenant une définition du modèle des systèmes de transitions symboliques :

**Définition 4.1.1 ($\mathcal{IOSTS}$)** *Un système de transition symbolique à entrée/sortie ($\mathcal{IOSTS}$) $\mathcal{M}$ est un tuple $\langle D, \Theta, \Sigma, T\rangle$ où*

- $D = V \cup C \cup P$ *est un ensemble fini de* données typées *constitué de trois ensembles disjoints de* variables $V$, *de* constantes symboliques $C$ *et de* paramètres de communication $P$. *L'ensemble de variables* $V = V_p \cup V_o$ *est lui-même partitionné en un ensemble de* variables propres $V_p$ *et de* variables observées $V_o$.
- $\Theta$ *appelé* condition initiale, *est un prédicat portant seulement sur les constantes* $C$ *et variables propres* $V_p$. *Donc* $\Theta$ *est assimilé à un ensemble,* $\Theta \subseteq Dom(C) \times Dom(V_p)$.
- $\Sigma = \Sigma_? \cup \Sigma_! \cup \Sigma_\tau$ *est l'*alphabet fini des actions *constitué de trois ensembles disjoints d'*entrées $\Sigma_?$, *de* sorties $\Sigma_!$ *et d'*actions internes $\Sigma_\tau$. *Chaque action* $a \in \Sigma$ *est caractérisée par sa signature* signature $\mathsf{sig}(a)$. *La signature de* $a$ *est un tuple de types* $\mathsf{sig}(a) = \langle \mathsf{t}_1, \ldots, \mathsf{t}_k\rangle$. *La signature d'une action interne* $a \in \Sigma^\tau$ *est le tuple vide* $\langle\rangle$.
- $T$ *est un ensemble fini de* transitions symboliques. *Chaque* transition symbolique $t = \langle a, \pi, G, A\rangle \in T$ *est constituée des éléments suivants :*
  - *une* action $a \in \Sigma$.
  - *un* tuple de paramètres de communication $\pi = \langle p_1, \ldots, p_k\rangle \in P^k (k \in \mathbb{N})$ *porté par l'action* $a$. *Le tuple* $\pi$ *est supposé compatible avec la signature de l'action* $a$, *i.e. (1)* $\mathsf{sig}(a)$ *et* $\pi$ *ont la même longueur, et (2) pour tout* $i \in [1, k]$, $type(p_i) = \mathsf{t}_i$.
  - *une* garde $G$ *est un prédicat portant sur les constantes symboliques* $C$, *les variables* $V$, *et les paramètres* $\pi$ *de la transition (la portée des paramètres est limitée à une transition).*
    *Donc* $G \subseteq Dom(C) \times Dom(V) \times Dom(\pi)$.
  - *une* affectations $A$ *est une fonction de* $Dom(C) \times Dom(V) \times Dom(\pi)$ *dans* $Dom(V_p)$.

**Remarques :**

- Les variables d'un $\mathcal{IOSTS}$ sont partitionnées en variables propres et variables observées. Les variables propres sont les variables modifiables par $\mathcal{M}$. Les variables observées correspondent à des variables de l'environnement que l'IOSTS peut observer. On utilisera cette notion pour les objectifs de tests pour parler des variables propres de l'$\mathcal{IOSTS}$ de la spécification dont l'évolution est observée par $\mathcal{M}$, mais que $\mathcal{M}$ ne peut pas modifier, pour ne pas être intrusif. Plutôt que de donner une définition différente pour les objectifs de test, on a préféré donner une définition générale.
- On peut considérer l'ensemble $V_p$ (respectivement $V_o$) comme un vecteur de variables $V_p = \langle v_{p1}, \ldots v_{pn}\rangle$ (resp. $V_o = \langle v_{o1}, \ldots v_{on}\rangle$) on même comme une seule variable dont le domaine est le produits des domaines des $v_{pi}$ (resp. des $v_{oi}$). On peut

faire de même pour l'ensemble de constantes $C$.
- La condition initiale ne porte que sur les constantes et variables propres. Elle ne contraint donc pas les variables observées $V_o$.
- Une transition ne contient qu'une seule action, réception, émission ou action interne.
- Une transition est gardée par un prédicat dont la valeur de vérité peut dépendre des constantes, des paramètres de communication de l'action, des valeurs de toutes les variables, qu'elles soient propres ou observées.
- Une affectation modifie uniquement les valeurs des variables propres, mais n'affecte pas les variables observées afin de rester non-intrusif. L'affectation étant une fonction, ceci correspond au fait que $A$ contient exactement une affectation de chaque variable propre $v_{pk}$. Dans les exemples, une affectation sera décrite par l'ensemble des affectations de variables de la forme $v_{pk} := A_{v_{pk}}$ où $A_{v_{pk}}$ sera une expression sur $C \cup V \cup \pi$ et on ne décrira que les affectations différentes de $v_{pk} := v_{pk}$. L'affectation d'une variable propre peut dépendre des valeurs des variables (propres et observées) $V$, des constantes symboliques $C$ des paramètres de communication $\pi$ de l'action étiquetant par la transition (la portée de $\pi$ est limitée à une transition).

**$\mathcal{IOSTS}$ avec localités** Le modèle des $\mathcal{IOSTS}$ présenté ici est une généralisation du modèle proposé dans [72]. Une généralisation notable est l'introduction des variables observées, alors que dans le modèle original, celles-ci étaient considérées comme des constantes dans l'objectif de test, ce qui rendait la sémantique peu intuitive.

Nous avons aussi simplifié les notions de condition initiale et de garde, présentées ici par des prédicats, assimilables aux ensembles d'états qui les satisfont.

D'autre part dans le modèle original, on utilise des *localités* (ou points de contrôle) qui explicitent la structure de contrôle. Une transition va alors d'une localité d'origine à une localité but. Notre modèle peut servir pour représenter des localités. En effet, il suffit de considérer une variable propre $l$ sur un domaine fini $L$ de localités. Dans une transition, la localité d'origine $l_i$ est fixée par une garde $G$ de la forme $(l = l_1) \wedge G'$. La localité but $l_2$ est obtenue par l'affectation $l := l_2$. Ce modèle avec localité, que nous notons $\mathcal{IOSTS}_{loc}$, permettra d'instancier notre modèle général, en particulier pour décrire des exemples.

**Exemple :** Un exemple d'$\mathcal{IOSTS}$ dans le sous-modèle $\mathcal{IOSTS}_{loc}$ est représenté par la figure 4.1. Il représente un distributeur de boisson qui peut fournir du café (*COFFEE*) ou du thé (*TEA*). Bien que ceci n'apparaisse pas dans la figure, $C = \{pPrice\}$ et la constante symbolique $pPrice$ est de type entier, $V = V_p = \{vPaid, vBeverage\}$ (il n'y a que des variables propres), la variable $vPaid$ étant entière, et $vBeverage$ de type énuméré $\{TEA, COFFEE\}$ et $P = \{mCoinValue, mRemainingValue, mBeverage\}$ avec $mCoinValue$ et $mRemainingValue$ de type entier, et $mBeverage$ de type énuméré dans $\{TEA, COFFEE\}$. Les deux boissons ont le même prix $pPrice$ supposée strictement positive par la condition initiale. La variable $vPaid$ porte le montant payé par l'utilisateur et la variable $vBeverage$ porte la boisson choisie. Depuis la localité $Begin$, la transition étiquetée $tau; vPaid := 0$ vers $Idle$ est une transition interne qui réinitialise $vPaid$ à 0. Dans la localité $Idle$, l'utilisateur peut, soit appuyer sur le bouton $Cancel$ et passer dans l'état $Return$, soit insérer une pièce par l'entrée $Coin$ avec une paramètre $mCoinValue$ de valeur strictement positive, ce qui incrémente $vPaid$ de ce

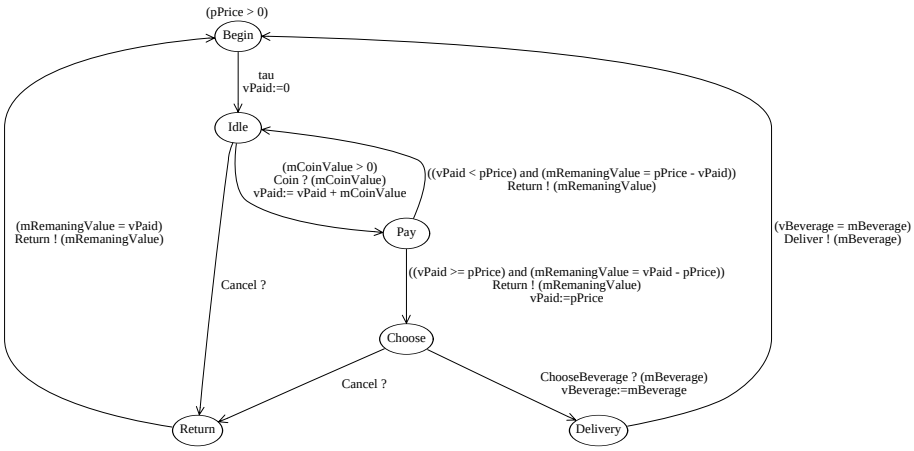



FIG. 4.1 – Exemple d'$\mathcal{IOSTS}_{loc}$

montant et fait passer dans l'état $Pay$. Si la valeur de $vPaid$ est strictement inférieure à $vPaid$, le distributeur retourne à la localité $Idle$ en renvoyant un message $Return$ portant le paramètre $mRemainingValue$ dont la valeur est égale au prix restant à payer, i.e. $mRemainingValue = pPrice - vPaid$, cette contrainte apparaissant dans la garde de la transition. Sinon, i.e. $vPaid \geq pPrice$, on transite vers $Choose$ en renvoyant $Return$ portant la monnaie à rendre i.e. $vPaid - pPrice$. Dans la localité $Choose$, l'utilisateur peut à nouveau appuyer sur le bouton $Cancel$, où choisir une boisson par l'entrée $ChooseBeverage$ dont le paramètre $mBeverage$ (dont la valeur sera COFFEE ou TEA) est affecté à la variable $vBeverage$ et passer dans la localité $Delivery$. Dans la localité $Delivery$, le distributeur délivre une boisson par la sortie $Deliver$ dont le paramètre $mBeverage$ doit satisfaire la contrainte $mBeverage = vBeverage$ pour correspondre à la boisson choisie à la transition précédente. Noter que les paramètres tels que $mBeverage$ ont une portée limitée à une transition. Enfin, dans la localité $Return$, la sortie $Return$ renvoie la sortie $Return$ qui porte le paramètre $mRemainingValue$ dont la valeur est égale à $vPaid$, donc la valeur de ce qui a été payé jusqu'ici par l'utilisateur.

### 4.1.2 Sémantique des $\mathcal{IOSTS}$

La sémantique d'un $\mathcal{IOSTS}$ peut être définie en termes de systèmes de transitions à entrée/sortie (IOLTS). On a besoin de définir les *états*, parmi lesquels on distingue les *états initiaux*, l'*alphabet des actions* et la *relation de transitions* entre états de ces IOLTS. Noter cependant qu'on a besoin d'une définition plus générale des IOLTS, les ensembles d'états et de transition n'étant *a priori* pas dénombrables.

**Définition 4.1.2 (Sémantique IOLTS des $\mathcal{IOSTS}$)** *Pour un $\mathcal{IOSTS}$ $\mathcal{M} = \langle D, \Theta, \Sigma, T \rangle$, la sémantique de $\mathcal{M}$, est définie par un IOLTS noté $[[\mathcal{M}]] = (Q, \Lambda, \rightarrow, Q_0)$ où*

- $Q = Dom(C) \times Dom(V) = Dom(C) \times Dom(V_p) \times Dom(V_o)$ *est l'ensemble des* états.
  *Un état est donc un triplet* $q = (c, v_p, v_o)$ *constituée d'une valuation* $c$ *des constantes symboliques* $C$*, d'une valuation* $v_p$ *des variables propres* $V_p$ *et d'une valuation des variables observées* $V_o$ *(c,* $v_p$ *et* $v_o$ *sont des vecteurs de valeurs de constantes et variables). On notera* $c = val_q(C)$, $v_p = val_q(V_p)$, $v_o = val_q(V_o)$.
- $Q_0 = \Theta \times Dom(V_o^s) \subseteq Q$ *est l'ensemble des* états initiaux. *C'est l'ensemble des états qui satisfont la condition initiale, et laissent libre la valeur des variables observées. Noter que* $Q_0 = proj^{-1}_{Dom(C)\times Dom(V) \to Dom(C)\times Dom(V_p)}(\Theta)$.
- $\Lambda$ *est l'alphabet constitué d'un ensemble d'*actions valuées. *Une action valuée est une paire* $\alpha = \langle a, p\rangle$ *où* $a$ *est une action et* $p \in Dom(\pi)$ *est une valuation du vecteur de paramètres* $\pi$ *porté par cette action. On note* $p = val(\pi)$. $\Lambda$ *est partitionné en* $\Lambda = \Lambda_? \cup \Lambda_! \cup \Lambda_\tau$ *suivant l'appartenance de l'action* $a$ *aux ensembles* $\Sigma_?$, $\Sigma_!$ *ou* $\Sigma_\tau$.
- $\to$ *est la* relation de transition. $\to$ *est l'union, pour toutes les transitions symboliques* $t = \langle a, \pi, G, A\rangle \in T$ *dans* $\mathcal{M}$*, des triplets* $(q, \alpha, q')$ *(on note aussi* $q \xrightarrow{\alpha} q'$*) où*
  - $q = (c, v_p, v_o) \in Q$ *est l'état de départ composé d'une valuation* $c$ *des constantes symboliques (*$c = val_q(C)$*), et d'une valuation* $v_p$ *des variables propres (*$v_p = val_q(V_p)$*), et* $v_o$ *des variables observées (*$v_o = val_q(V_o)$*),*
  - $\alpha = \langle a, p\rangle \in \Sigma \times Dom(\pi)$ *est l'action valuée composée de l'action* $a$ *et d'une valuation* $p = val(\pi) \in Dom(\pi)$ *des paramètres de communication. Si* $a \in \Sigma_\tau$, $Dom(\pi)$ *est vide donc* $val(\pi) = \langle\rangle$.
  - $\langle c, v_p, v_o, p\rangle \in G$ *i.e. que le vecteur de valuations* $\langle c, v_p, v_o, p\rangle$ *doit satisfaire la garde* $G$,
  - $q' = (c, v'_p, v'_o) \in Q$ *est l'état but avec* $c = val_{q'}(C) = val_q(C)$, $v'_p = val_{q'}(V_p) = A(c, v_p, v_o, p)$ *et* $v'_o = val_{q'}(V_o) \in Dom(V_o)$.

**Remarque :** Dans la définition de la relation de transition ci-dessus, les constantes ne sont pas modifiées. La valeur du vecteur de variables propres de l'état d'arrivée $v'_p$ est complètement déterminé par l'affectation $A$ en fonction de l'état de départ $q$ et de la valeur des paramètres de communication $p$. Par contre $v'_o$ peut prendre n'importe quelle valeur dans son domaine. Pour un état de départ $q$ donné, une valeur de paramètres $p$ fixée pour l'action $a$ qui rende la transition $t$ tirable dans $q$, $q$ a autant de successeurs (non-d'terministes) par $\langle a, p\rangle$ que d'éléments dans $Dom(V_o)$. Ceci signifie qu'on ne fait aucune hypothèse sur l'évolution des variables observées puisque celles-ci sont modifiées par l'environnement (l'$\mathcal{IOSTS}$ dont on observe les variables). Ce n'est qu'au moment de la composition par produit synchrone que les valeurs des variables observées seront liées aux valeurs de l'IOSTS observé.

**Exemple :** considérons l'$\mathcal{IOSTS}$ de la figure 4.2. Cet $\mathcal{IOSTS}$ a pour constante $pPrice$ et pour paramètre de communication $mCV$, pour variable propre $vP$ et observe la variable $vPaid$. Supposons que $Dom(vP) = |N$, $Dom(pPrice) = [0, 2]$, $Dom(vPaid) = [0, 3]$, et $Dom(mCV) = [0, 2]$. Un état de l'$\mathcal{IOSTS}$ est un triplet $q = (val(c), val(l), val(vP), val(vPaid))$ où $val(x)$ est la valeur de la donnée $x$. La transition de $s_0$ à $s_1$ étiquetée par $Coin(val(mCV))$ est tirable dans un état $q = (val(c), val(l), val(vP), val(vPaid))$ si $val(l) = s_0$ et $val(vPaid) < pPrice$ et $val(mCV) > 0$. Par exemple dans l'état initial $q0 = (2, s0, 0, 1)$, $Coin(1)$ est tirable ($val(vPaid) = 1 < val(pPrice) = 2$ et $val(mCV) = 1 > 0$). Cette transition

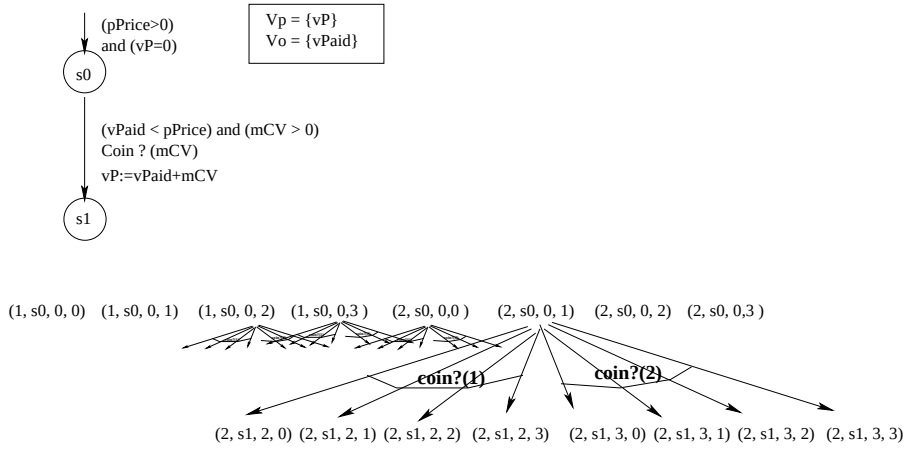



FIG. 4.2 – $\mathcal{IOSTS}$ et sémantique

mène de manière non-déterministe dans les états $q1 = (2, s1, 2, 0)$, $q2 = (2, s1, 2, 1)$, $q3 = (2, s1, 2, 2)$, $q4 = (2, s1, 2, 3)$.

Noter qu'on a utilisé une définition étendue des IOLTS, en permettant un ensemble d'états initiaux au lieu d'un seul état initial. L'ensemble d'états et l'alphabet ne sont pas nécessairement dénombrables. Les IOLTS peuvent etre à branchement infini.

**Instance d'$\mathcal{IOSTS}$** Le fait de disposer de constantes symboliques dans le modèle $\mathcal{IOSTS}$ permet de décrire des spécifications génériques pour un ensemble de valeurs de ces constantes, et, on le verra, de produire des tests eux-mêmes génériques. Pour une spécification $\mathcal{S}$ décrite par un $\mathcal{IOSTS}$, on peut fixer la valeur de toutes ses constantes symboliques, i.e. choisir une valuation $c \in Dom(C)$. Il faut bien sûr que le choix de $c$ permettre toujours de satisfaire la condition initiale i.e. $[\{c\} \times Dom(V_p)] \cap \Theta \neq \emptyset$ ou encore $c \in proj_{\text{Dom(C)}}(\Theta)$. On appelle *instance* de $\mathcal{S}$ et on note $\mathcal{S}(c)$ un tel $\mathcal{IOSTS}$. La sémantique d'une instance $\mathcal{S}(c)$ est un IOLTS dans lequel les états ont une valeur fixe des constantes. On peut également voir l'IOLTS $[[\mathcal{S}]]$ comme l'ensemble des IOLTS $[[\mathcal{S}(c)]]$ pour l'ensemble des valuations $c$ des constantes symboliques $c$ permettant de satisfaire la condition initiale.

**Définition 4.1.3 ($\mathcal{IOSTS}$ fermé et initialisable)** *Un $\mathcal{IOSTS}$ $\mathcal{M}$ est dit* fermé *s'il n'a que des variables propres i.e. $V_o = \emptyset$ et $V = V_p$.*

*Un $\mathcal{IOSTS}$ fermé est dit* initialisable *si, pour toute instance $\mathcal{M}(c)$, il existe une seule valuation $v \in Dom(V)$ des variables $V$ de $\mathcal{M}$ qui satifait la condition initiale $\Theta$ i.e. pour toute instance $\mathcal{M}(c)$, la condition initiale $\Theta$ est réduite à un seul état.*

La sémantique $[[\mathcal{M}(c)]]$ d'un $\mathcal{IOSTS}$ fermé initialisable est un IOLTS ayant un seul état initial.

**Exécutions et langages des $\mathcal{IOSTS}$ :** Etant donné que la sémantique d'un $\mathcal{IOSTS}$ s'exprime par un IOLTS, on peut réutiliser la plupart des concepts introduits sur ce modèle, comme les séquences d'actions (valuées), les blocages (livelock, deadlock, outputlock), les traces (valuées), les traces suspendues, les langages d'actions et de traces associés. Dans les IOLTS, on utilisait la notion de langage pour parler des séquences d'actions, en oubliant le contenu des états. Dans le cas des $\mathcal{IOSTS}$ on peut être plus fin et parler des contenus d'états. En effet, les objectifs de tests définis plus loin permettront non seulement de parler de séquences d'actions des spécifications, mais aussi d'observer les valeurs de variables des spécifications, donc de parler des contenus d'états. Dans le cas des IOLTS les objectifs portaient uniquement sur les séquences d'actions ou les traces. On introduit donc la notion d'*exécution*.

**Définition 4.1.4 (Exécution)** *Une* exécution *de* $[[\mathcal{M}]]$ *est une séquence finie* $\beta^{\rightarrow} = q_0 \stackrel{\alpha_1}{\rightarrow} q_1 \stackrel{\alpha_2}{\rightarrow} \cdots \stackrel{\alpha_n}{\rightarrow} q_n$ *où* $q_0 = (c, v_0) \in Q_0$ *est un état initial, les* $q_i = (c, v_i)$ *sont des états et les* $\alpha_i = \langle a, p \rangle$ *sont des actions valuées, tels que* $\forall i, 1 \leq i \leq 1, (q_{i-1}, \alpha_i, q_i) \in \rightarrow$.

On notera $Run([[\mathcal{M}]])$ l'ensemble des exécutions de $[[\mathcal{M}]]$. $Run([[\mathcal{S}]])$ est aussi un langage de $(Q.\Lambda)^*.Q$ dans lequel on alterne état et action valuée.

**Remarque :** La sémantique des $\mathcal{IOSTS}$ en termes d'IOLTS (definition 4.1.2, définit les états et les transitions possibles de $[[\mathcal{M}]]$, L'ensemble des exécutions $Run([[\mathcal{M}]])$ caractérise uniquement les états accessibles de l'IOLTS $[[\mathcal{M}]]$ et les transitions entre ces états. En d'autres termes, pour deux $\mathcal{IOSTS}$ $\mathcal{M}$ et $\mathcal{M}'$, si $Run([[\mathcal{M}]]) = Run([[\mathcal{M}']])$, les IOLTS $[[\mathcal{M}]] = [[\mathcal{M}']]$ peuvent différer uniquement sur des des états non accessibles. Pour démontrer l'égalité des sémantiques de deux $\mathcal{IOSTS}$, il faut donc démontrer l'égalité des exécutions et des états.

En projetant les exécutions de $Run([[\mathcal{M}]])$ sur les actions valuées $\Lambda$, on obtient le langage d'actions

$$L([[\mathcal{M}]] = proj_{\Lambda}(Run([[\mathcal{M}]]))$$

En projetant seulement sur les actions valuées visibles (entrées et sorties), on obtient les langages de traces

$$Traces([[\mathcal{M}]] = proj_{\Lambda_? \cup \Lambda_!}(Run([[\mathcal{M}]]))$$

On peut également réutiliser les notions de déterminisme, déterminisme observable, complétude en entrée sur les IOLTS. Malheureusement, pour un $\mathcal{IOSTS}$, on ne sait en général pas décider si l'IOLTS définissant sa sémantique possède ces propriétés, sauf en restreignant le modèle. Par contre, on peut trouver des conditions syntaxiques suffisantes sur les $\mathcal{IOSTS}$ pour que les IOLTS exprimant leur sémantique aient ces propriétés.

### 4.1.3 Abstraction d'$\mathcal{IOSTS}$

La génération de tests pour les $\mathcal{IOSTS}$ va reposer sur l'analyse d'accessibilité et de co-accessibilité dans leur sémantique (concrète) IOLTS. Or pour ce modèle, ces questions sont indécidables et il sera nécessaire de faire des analyses approchés. Pour cela, on utilisera des abstractions de la sémantique, conservatives pour les propriétés de sûreté, au sens où les

propriétés vraies sur les sémantiques abstraites restent vraies sur la sémantique concrète. Plus l'abstraction est forte, plus l'analyse est facile, mais moins on est précis.

Une première abstraction qu'on peut utiliser sur les $\mathcal{IOSTS}$ avec localités, est de considérer uniquement la structure de contrôle des $\mathcal{IOSTS}$ en oubliant le traitement des données.

Pour un $\mathcal{IOSTS}$ $\mathcal{M} = \langle D, \Theta, L, L_0, \Sigma, T\rangle$ avec $D = C \cup P \cup V$ du sous-modèle $\mathcal{IOSTS}_{loc}$ avec localité, on peut définir un IOLTS $abs(\mathcal{M}) = (L, \Sigma, \rightarrow_{abs}, L_0)$ où $L$ est l'ensemble des localités, $L_0$ le sous-ensemble des localités d'origine (le plus grand sous-ensemble de $L$ tel que $\Theta \cap (L_0 \times Dom(C) \times Dom(V)) \neq \emptyset$), et $\rightarrow_{abs}$ est l'ensemble $\rightarrow_{abs} = \{abs(t) = (l, a, l') \mid t = \langle l, a, \pi, G, A, l'\rangle \in T\}$. Pour une transition $(l, a, l')$ on note $abs^{-1}(l, a, l')$ l'ensemble des transitions $\langle l, a, \pi, G, A, l'\rangle \in T$ où $\pi$, $G$ et $A$ sont indéterminées.

Dans le cadre de l'interprétation abstraite [16], l'IOLTS $abs(\mathcal{M})$ est une abstraction de l'IOLTS $[[\mathcal{M}]]$. Les états de $[[\mathcal{M}]]$ sont des éléments de $Dom(C) \times L \times Dom(V)$. Le domaine concret est le treillis complet $2^{Dom(C)\times L\times Dom(V)}$ des parties de $Dom(C) \times L \times Dom(V)$ ordonné par inclusion. Les états de $abs(\mathcal{M})$ sont des éléments de $L$. Le domaine abstrait est le treillis complet $2^L$ des parties de $L$, ordonné par inclusion. La fonction d'abstraction est donc simplement la projection $\alpha = proj_L : 2^{Dom(C)\times L\times Dom(V)} \rightarrow 2^L$. des ensembles d'états de $[[\mathcal{M}]]$ sur l'ensemble des localités de ces états. La fonction de concrétisation est la projection inverse $\gamma = proj_L^{-1} : 2^L \rightarrow 2^{Dom(C)\times L\times Dom(V)}$. La paire $(proj_L, proj_L^{-1} :)$ forme une connexion de Galois, en effet :

$$\forall P \in 2^{Dom(C)\times L\times Dom(V)}, \forall Q \in 2^L, \alpha(P) \subseteq Q \iff P \subseteq \gamma(Q)$$

Pour simplifier, l'idée est la suivante. Pour la génération de tests, on a besoin de calculer des plus petits points fixes sur le treillis concret, en particulier l'ensemble $reach_B(P)$ des états accessibles depuis un ensemble d'états $P$ par un ensemble d'actions $B$, et l'ensemble $coreach_B(P)$ des états co-accessibles depuis un ensemble d'états $P$ par l'ensemble d'actions $B$. Or le treillis complet est infini et les calculs de points fixes ne terminent pas nécéssairement. Si $F$ est une fonction croissante sur le treillis concret (e.g. $P \cup post_B(X)$ ou $P \cup pre_B(X)$), si $G$ est croissante sur le treillis abstrait et supérieur'ou égale à $\alpha \circ F \circ \gamma$, alors le plus petit point fixe de $F$ est majoré par la concrétisation du plus petit point fixe de $G$. En pratique, il faut donc se donner des opérateurs abstraits pour $P \cup post_B(X)$ et $P \cup pre_B(X)$. On calcule un point fixe ($reach$ ou $coreach$) sur le treillis abstrait, et on concrétise le résultat dans le treillis concret, ce qui donne une surapproximation du résultat exact.

Les opérateurs abstraits pour $P \cup post_B(X)$ et $P \cup pre_B(X)$ sont $proj_L(P) \cup post_{proj_\Sigma(B)}(X)$ et $proj_L(P) \cup pre_{proj_\Sigma(B)}(X)$. Ils sont donc obtenus par la projection des actions valuées sur l'alphabet $\Sigma$ et l'opérateur $post$ et $pre$ pour la relation de transition de $abs(\mathcal{M})$.

Pour toute exécution $\beta^{\rightarrow} = (c, l_0, v_0) \overset{\langle a_1, p_1\rangle}{\rightarrow} (c, l_1, v_1) \overset{\langle a_2, p_2\rangle}{\rightarrow} (c, l_2, v_2) \ldots (c, l_{n-1}, v_{n-1}) \overset{\langle a_n, p_n\rangle}{\rightarrow} (c, l_n, v_n)$ dans $Run([[\mathcal{M}]])$, $\alpha(\beta^{\rightarrow}) = l_0 \overset{a_1}{\rightarrow} l_1 \overset{a_2}{\rightarrow} l_2 \ldots l_{n-1} \overset{a_n}{\rightarrow} l_n$ est une exécution de $abs(\mathcal{M})$.

On a donc,

$$\alpha(Run([[\mathcal{M}]])) = proj_L \circ proj_\Sigma(Run([[\mathcal{M}]])) \subseteq Run(abs(\mathcal{M}))$$

Et par conséquent,

$$\alpha(L([[\mathcal{M}]]) = proj_{\Sigma}(L([[\mathcal{M}]]) \subseteq L(abs(\mathcal{M})) \quad (4.1)$$
$$\alpha(Traces([[\mathcal{M}]]) = proj_{\Sigma}(Traces([[\mathcal{M}]]) \subseteq Traces(abs(\mathcal{M})) \quad (4.2)$$

On pourrait également montrer que $abs(\mathcal{M})$ simule $proj_{\Sigma}([[\mathcal{M}]])$.

L'IOLTS $M$ est une abstraction brutale puisqu'elle ne préserve que la structure, en perdant toute information sur les données. Les exécutions qu'elle définit par concrétisation sont donc en général beaucoup plus nombreuse que les exécutions réelles de $\mathcal{M}$. Son mérite est d'être simple, donc de permettre une analyse aisée. On peut imaginer d'autres abstractions plus fines, par exemple en énumérant n'importe quelle variable, vecteur de variables, expression sur les variables, si les domaines sont finis. Chacune de ces abstraction a les mêmes propriétés que ci-dessus, pour la paire de Galois adéquate. L'abstraction la plus fine possible est $[[\mathcal{M}]]$ lui-même, où toutes les valeurs des variables sont énumérées, mais son ensemble d'états est infini et les calculs de points fixes ne terminent pas en général. Les abstractions utiles sont celles où les calculs de point fixe terminent. Chaque fois qu'on utilise l'IOLTS $abs(\mathcal{M})$ dans la suite, on pourrait le remplacer par toute autre abstraction finie.

## 4.2 Théorie du test dans le cadre symbolique

La théorie du test que l'on considère est fondée sur la définition de modèles de spécifications et d'implémentations, de relation de conformité, de cas de tests. Plus exactement elle se fonde sur les modèles comportementaux donnés par la sémantique de ces modèles symboliques en termes d'IOLTS. On va donc pouvoir réutiliser la théorie du test dans le cadre des IOLTS. On verra par contre dans la section suivante que la génération de tests s'-effectuera sur les modèles symboliques, les $\mathcal{IOSTS}$, en regardant les propriétés obtenues sur la sémantique en termes d'IOLTS.

**Spécification** On suppose que la spécification est modélisée par un $\mathcal{IOSTS}$ $\mathcal{S} = \langle D^{\mathrm{s}}, \Theta^{\mathrm{s}}, \Sigma^{\mathrm{s}}, T^{\mathrm{s}} \rangle$ fermé et initialisable. On rappelle que fermé signifie que S n'a pas de variable observée i.e. $V_o^{\mathrm{s}} = \emptyset$, i.e. il n'observe pas les variables d'un autre $\mathcal{IOSTS}$. On le suppose initialisable i.e. pour chaque instance $\mathcal{S}(c)$, $[[\mathcal{S}(c)]]$ a un seul état initial.

La sémantique de $\mathcal{S}$ est un IOLTS $[[\mathcal{S}]] = (Q^{\mathrm{s}}, \Lambda^{\mathrm{s}}, \rightarrow_{\mathrm{s}}, q_0^{\mathrm{s}})$ où $\Theta = \{q_0^{\mathrm{s}}\}$ et $Q^{\mathrm{s}} \subseteq Dom(C) \times Dom(V^{\mathrm{s}})$. Pour une instance particulière $\mathcal{S}(c)$, sa sémantique est l'IOLTS $[[\mathcal{S}(c)]] = (Q^{\mathrm{S(c)}}, \Lambda^{\mathrm{S(c)}}, \rightarrow_{\mathrm{S(c)}}, q_0^{\mathrm{S(c)}})$. Étant initialisable, $[[\mathcal{S}]]$ n'a qu'un état initial $q_0^{\mathrm{S(c)}}$.

**Implémentation** Comme dans le cas énuméré, on suppose que l'IUT peut être modélisée, afin de pouvoir raisonner sur la conformité de celle-ci avec une spécification. Reste a choisir le modèle adéquat. Une implémentation sous test n'est pas un $\mathcal{IOSTS}$ au sens général, car les valeurs de ses constantes sont fixées. On pourrait donc supposer que l'IUT est un $\mathcal{IOSTS}$ instancié, fermé et initialisable. En réalité, la conformité ne va s'intéresser qu'aux comportements observables de l'IUT, donc ses traces valuées. On peut donc directement supposer que l'**implémentation est un IOLTS** $IUT = (Q^{\mathrm{IUT}}, A^{\mathrm{IUT}}, \rightarrow_{\mathrm{IUT}}, q_0^{\mathrm{IUT}})$. Comme

dans le cas énuméré, on va faire quelques hypothèses (voir la sous-section 3.1.1). D'abord la conformité d'une IUT sera jugée pour une instance particulière $\mathcal{S}(c)$ de $\mathcal{S}$. Pour cette instance, on supposera la *compatibilité* de l'alphabet $\Lambda_s$ des actions valuées de $[[\mathcal{S}(c)]]$ avec l'alphabet $A^{\text{IUT}}$ de l'$IUT$. Afin d'obtenir de bonnes propriétés de l'exécution des tests sur l'IUT, l'IUT est supposée *complète en entrée* pour son alphabet d'entrée $A_?^{\text{IUT}}$.

**Relation de conformité** On peut maintenant définir la conformité d'une IUT avec une instance de spécification décrite par un $\mathcal{IOSTS}$ :

**Définition 4.2.1 (Relation de conformité)** *On dira qu'une IUT est conforme à une instance $\mathcal{S}(c)$ de $\mathcal{S}$ (avec les hypothèses précédentes sur $\mathcal{S}$ et $IUT$), si $IUT$* **ioco** $[[\mathcal{S}(c)]]$.

Cette définition mérite quelques commentaires. Imaginons que les implémentations aient des constantes correspondant aux constantes symboliques $C$ de la spécification $\mathcal{S}$. Pour une IUT donnée, ces constantes sont fixées à une valeur $c$ et supposée connue. La conformité de l'IUT n'a de sens que vis à vis de la sémantique l'instance $\mathcal{S}(c)$ de la spécification qui fixe la valeur de ses constantes de façon identique.

**Cas de test** Pour une spécification $\mathcal{S}$, on souhaite générer des programmes de tests génériques i.e. dont les constantes ne sont pas fixées, qui manipule ses variables.

**Définition 4.2.2 (Cas de test symbolique)** *Pour une spécification $\mathcal{S}$, un cas de test est un $\mathcal{IOSTS}$ fermé et initialisable $\mathcal{TC} = \langle D^{TC}, \Theta^{TC}, \Sigma^{TC}, T^{TC}\rangle$ muni de trois ensembles d'états puits* **Pass**, **Fail** *et* **Inconc** *dans $C \cup V$.*

- *Les données de $\mathcal{TC}$ sont $D^{TC} = C^{TC} \cup P^{TC} \cup V^{TC}$ et on suppose que $V^{TC}$ n'a que des variables propres, et que $C^{TC} = C^{S}$* [2].
- *Les actions de $\mathcal{TC}$ sont limitées à des entrées et sorties, i.e. $\Sigma^{TC} = \Sigma_!^{TC} \cup \Sigma_?^{TC}$ avec $\delta \in \Sigma_?^{TC}$ représentant l'observation d'un blocage et de signature vide. On suppose que les sorties de $\mathcal{TC}$ sont aussi limitées aux entrées de $\mathcal{S}$ i.e. $\Sigma_!^{TC} \subseteq \Sigma_?^{S}$.*

Les ensembles d'états de verdicts **Pass**, **Fail** et **Inconc** de $C \cup V$ peuvent être définis par des prédicats. Ces ensemble d'états sont des puits i.e. ils n'ont pas de successeurs pour la relation de transition $\rightarrow$ de $[[\mathcal{TC}]]$.

La sémantique d'un cas de test $\mathcal{TC}$ est un IOLTS $[[\mathcal{TC}]]$. Ses ensembles d'états **Pass**, **Fail** et **Inconc** sont les états de $[[\mathcal{TC}]]$ satisfiant les prédicats **Pass**, **Fail** ou **Inconc** de $\mathcal{TC}$. Pour toute instanciation $c \in Dom(C)$ des constantes symboliques $C$, la sémantique de $\mathcal{TC}(c)$ est un IOLTS $[[\mathcal{TC}(c)]]$ avec un seul état initial ($\mathcal{TC}$ est initialisable). On supposera et on assurera par construction que $[[\mathcal{TC}(c)]]$ est déterministe, complet en entrée dans tous ses états.

L'exécution d'une instance de cas de test $\mathcal{TC}(c)$ sur une implémentation $IUT$ peut alors être définie par la même composition parallèle $\|$ que dans le cas des IOLTS, où les actions valuées du cas de test se synchronisent avec les actions observables de l'$IUT$. Il faut cependant supposer que les cas de test acceptent toutes les sorties de l'implémentation

[2] On pourrait généraliser en permettant au cas de test d'observer les valeurs de certaines variables de l'IUT déclarées observables dans la spécification. C'est l'option choisie dans le cadre de l'extension de TGV pour Agedis.

donc $A_{\mathrm{O}}^{\mathrm{IUT}} \subseteq \Lambda_{?}^{\mathrm{TC}}$. Comme dans le cas énuméré, on l'assure simplement par une entrée indéterminée $?otherwise$. Comme dans le cas des IOLTS, l'ensemble des hypothèses faites sur les cas de test et les implémentations assurent que l'exécution des tests ne se bloque que dans les états de verdict **Pass**, **Fail** et **Inconc**.

Les notions de verdict associés aux exécutions maximales, de rejet par un cas de test sont alors identiques au cas des IOLTS.

**Propriétés attendues des cas de test** Nous avons défini la relation de conformité entre une implémentation $IUT$ et une instance $\mathcal{S}(c)$ de l'$\mathcal{IOSTS}$ $\mathcal{S}$. La génération de tests consistera à produire automatiquement des tests ayant de bonnes propriétés vis à vis de la relation de conformité. On pourrait donc vouloir générer des cas de tests pour une instance $\mathcal{S}(c)$ de la spécification $\mathcal{S}$. Au lieu de ça, on se propose de générer des cas de tests symboliques $\mathcal{TC}$, sous la forme d'$\mathcal{IOSTS}$, à partir de la spécification $\mathcal{S}$, en assurant que, pour chaque instanciation $c$, les cas de tests $\mathcal{TC}(c)$ pour cette instanciation, aient de bonnes propriétés pour la relation de conformité, vis à vis de l'instance de $\mathcal{S}(c)$ de la spécification $\mathcal{S}$. La relation de conformité étant définie sur les IOLTS, ces bonnes propriétés s'exprimeront sur les IOLTS modélisant les sémantiques des instances des cas de test, par rapport aux IOLTS modélisant les sémantiques des instances de la spécification. On souhaitera tout simplement les mêmes propriétés que dans le cas des IOLTS, i.e. non-biais et exhaustivité en particulier.

Autrement dit, pour une spécification décrite par un $\mathcal{IOSTS}$ $\mathcal{S}$, on veut produire une suite de tests symboliques $\mathcal{TS}$ i.e. un ensemble de cas de tests symboliques, tel que, pour toute instanciation $c \in Dom(C)$ des constantes symboliques de $C$,

- pour tout cas de test, $\mathcal{TC} \in \mathcal{TS}$, $[[\mathcal{TC}(c)]]$ est non-biaisé par rapport à $[[\mathcal{S}(c)]]$ pour la relation **ioco**.
- la suite de test $TS = \{[[\mathcal{TC}(c)]] | \mathcal{TC} \in \mathcal{TS}\}$ est exhaustive pour $[[\mathcal{S}(c)]]$ et **ioco**.

## 4.3 Principes de la génération de tests symboliques

Comme dans le cas des IOLTS, nous allons baser la génération de tests pour les $\mathcal{IOSTS}$ sur une sélection par des objectifs de test. Nous définissons d'abord la syntaxe de ces objectifs comme des $\mathcal{IOSTS}$ munis d'états accepteurs. Leur sémantique est plus riche que la sémantique des objectifs de tests pour les IOLTS, et ils permettent de reconnaître des exécutions des spécifications pour affiner la sélection. Les tests observant des traces, la génération de tests consistera, à partir des exécutions reconnues par l'objectif, à produire un cas de test qui mènera dans l'état **Pass** sur une trace qui est la projection d'une exécution acceptée par l'objectif. Les cas de test étant des $\mathcal{IOSTS}$, nous devrons donc définir les opérations sur les $\mathcal{IOSTS}$ nécessaires à la génération de tests à partir des $\mathcal{IOSTS}$ de la spécification et de l'objectif de test. Pour chaque opération symbolique $op_{sy}$ sur un $\mathcal{IOSTSM}$, nous essayerons de nous reposer sur l'opération $op$ correspondante sur les IOLTS, de sorte que les exécutions (resp. les traces selon le cas) de l'IOLTS, sémantique de l'IOSTS transformé $[[op_{sy}(\mathcal{M})]]$, soient identiques aux exécutions (resp. traces) du transformé de la sémantique de $\mathcal{M}$, i.e. $op[[\mathcal{M}]]$. On obtiendra donc des propriétés de commutation de diagramme, du type $Run([[op_{sy}(\mathcal{M})]]) = Run(op[[\mathcal{M}]])$ ou

$Traces([[op_{sy}(\mathcal{M})]]) = Traces(op[[\mathcal{M}]])$ qui permettent alors de se reposer complètement sur les propriétés de langages ou de traces obtenues dans le cadre des IOLTS.

### 4.3.1 Objectifs de test

Dans le cas des IOLTS, les objectifs servaient à sélectionner des langages ou des traces de la spécification. Pour les IOSTS, les objectifs vont servir à sélectionner des exécutions de la spécification, en observant à la fois les actions valuées et les contenus des états i.e. en définissant un langage de $(Q.\Lambda)^*.Q$. Ces objectifs doivent observer mais ne pas restreindre les exécutions possibles de la spécification. Pour cela, les IOLTS représentant leur sémantique doivent être complets, non seulement au sens des actions, mais aussi au sens des états de la spécification i.e. complets sur $(Q.\Lambda)^*.Q$.

On souhaite produire des tests symboliques génériques, i.e. pour toutes les instances d'une spécification. Les objectifs de test seront donc des $\mathcal{IOSTS}$ munis d'états accepteurs. Ils manipulent leurs propres variables (pour leurs calculs et décisions) et paramètres de communication (unifiés avec ceux de la spécification lors du produit avec la spécification) et permettent seulement d'observer les valeurs des variables de la spécification, sans les modifier. Pour ce faire, ces variables sont considérées comme des *variables observées*, pouvent prendre une valeur arbitraire dans chaque état. Les objectifs seront donc des observateurs symboliques, génériques pour un ensemble de spécifications.

**Définition 4.3.1 (Objectif de test)** *Soit une spécification $\mathcal{S} = \langle D^s, \Theta^s, \Sigma^s, T^s \rangle$ avec $D^s = V^s \cup P^s \cup C^s$. Un* objectif de test *pour $\mathcal{S}$ est un $\mathcal{IOSTS}$ $\mathcal{TP} = \langle D^{TP}, \Theta^{TP}, \Sigma^{TP}, T^{TP} \rangle$ muni de deux prédicats sur $C \cup V$, $Accept \subseteq Dom(C) \times Dom(V)$ et $Refuse \subseteq Dom(C) \times Dom(V)$ tels que*

- *$Accept$ et $Refuse$ sont des ensembles disjoints d'états puits pour $T^{TP}$ i.e. tout successeur de $Accept$ (resp. $Refuse$) par une action de $\Sigma^{TP}$ est dans $Accept$ (resp $Refuse$).*
- *$C^{TP} = C^s$*
- *$V^{TP} = V_o^{TP} \cup V_p^{TP}$ avec $V_o^{TP} = V^s$,*
- *$P^s \cap P^{TP} = \emptyset$,*
- *$\Sigma^s = \Sigma^{TP}$, donc $Dom(P^s) = Dom(P^{TP})$*
- *$\Theta^{TP} = Dom(C^s) \times \Theta'$ avec $\Theta' \subseteq Dom(V_p^{TP})$.*
- *on suppose que $\mathcal{TP}$ est complet au sens suivant : pour toute action $a \in \Sigma^s$ l'union des gardes des transitions $t = \langle a, \pi, G, A \rangle$ portant l'action $a$ est exactement $Dom(C^s) \times Dom(V_p^{TP}) \times Dom(V^s) \times Dom(\pi)$.*

**Remarques :** la définition des objectifs nécessite quelques commentaires.

- les constantes symboliques de $\mathcal{TP}$ sont identiques à celles de $\mathcal{S}$. Elles pourront donc être utilisées dans les gardes et affectations de $\mathcal{TP}$.
- on rappelle que $\mathcal{S}$ est fermé, c'est à dire qu'il n'a que des variables propres. Comme $V_o^{\text{TP}} = V^{\text{s}}$, l'objectif $\mathcal{TP}$ observe *a priori* toutes les variables de $\mathcal{S}$ (donc aussi ses localités s'il en a). $\mathcal{TP}$ a aussi ses *variables propres* $V_p^{\text{TP}}$. Celles-ci lui permettent d'affiner la sélection, par exemple par l'intermédiaire de compteurs.
- les paramètres de communication de $\mathcal{TP}$ et $\mathcal{S}$ sont disjoints. Ceux-ci seront unifiés lors du produit.

– les alphabets d'actions sont identiques. La signature d'une action étant unique, les types des paramètres de ces actions dans $\mathcal{TP}$ et $\mathcal{S}$ coincident, donc les domaines $Dom(P^{\text{S}})$ et $Dom(P^{\text{TP}})$ également.
– la condition initiale $\Theta^{\text{TP}}$ ne peut contraindre que ses variables propres, mais ni ses constantes symboliques (puisqu'elles sont identiques à celles de $\mathcal{S}$), ni les variables observées (par définition des $\mathcal{IOSTS}$). C'est donc un prédicat $\Theta^{\text{TP}}$ sur $C^{\text{S}} \cup V_p^{\text{TP}}$ qui ne contraint pas la valeur des constantes symboliques $C^{\text{S}}$.
– l'intention de la complétion est que l'IOLTS de la sémantique de $[[\mathcal{TP}]]$ soit complet (au sens des IOLTS). En effet, un état de $[[\mathcal{TP}]]$ est un élément de $Dom(C^{\text{S}}) \times Dom(V_p^{\text{TP}}) \times Dom(V^{\text{S}})$. Pour une action $a$, l'union des gardes des transitions étant $Dom(C^{\text{S}}) \times Dom(V_p^{\text{TP}}) \times Dom(V^{\text{S}}) \times Dom(\pi)$, pour toute valeur $p$ des paramètres $\pi$, l'action valuée $\langle a, p\rangle$ est tirable dans tout état de $[[\mathcal{TP}]]$.
– il est *a priori* difficile de savoir si $\mathcal{TP}$ est complet donc on ne peut pas exiger de l'utilisateur qu'il décrive un objectif complet. Par contre il est facile d'assurer syntaxiquement la completion. Il suffit pour chaque action de rajouter une transition gardée par l'union de la négation des gardes sur cette action [3].

**Exemple :** La figure 4.3 présente un objectif de test dans le modèle avec localités, compatible avec la spécification de la figure 4.1. Cet objectif est ici décrit dans l'outil STG. Implicitement, il est completé dans la localité $S_0$, par une boucle sur $Coin?(mCoinValue)$, sur $ChooseBeverage?$ et sur $(\neg(vPaid < pPrice); Return!(mRemVal)$, et par une transition étiquetée $\neg(mBev = COFFEE) : Deliver!(mBev)$ vers la localité $Reject$. Dans $Accept$ et $Reject$, il est complété par une boucle sur toute action. Cet objectif sélectionne donc des exécutions dans lesquelles on obtient du café en payant en une seule fois (pas de $Return$ avec $vPaid < pPrice$), et sans annulation ($Cancel?$).

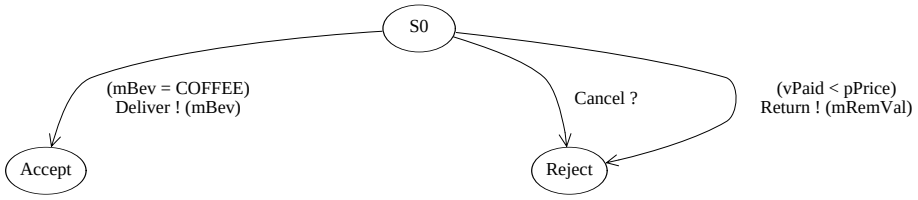



FIG. 4.3 – Objectif de test dans le modèle $\mathcal{IOSTS}_{loc}$

Comme pour tout $\mathcal{IOSTS}$, la sémantique $[[\mathcal{TP}]]$ de $\mathcal{TP}$ est un IOLTS. Ses états sont composés d'une valuation de ses constantes symboliques $C^{\text{TP}} = C^{\text{S}}$ et variables propres et observées de $V^{\text{TP}} = V_p^{\text{TP}} \cup V^{\text{S}}$. Ce sont donc des éléments de $Dom(C^{\text{S}}) \times Dom(V_p^{\text{TP}}) \times Dom(V^{\text{S}})$. De plus $[[\mathcal{TP}]]$ est complet.

De la même manière qu'on a défini des séquences ou traces acceptées pour les IOLTS, nous définissons une *exécution acceptée* (resp. *refusée*) par $[[\mathcal{TP}]]$ comme une exécution

[3] Dans STG, la complétion pour les $\mathcal{IOSTS}_{loc}$ est assurée syntaxiquement de la façon suivante. Si il existe une transition étiquetée par l'action $a$ depuis la localité $l$, qui ne mène pas à $Refuse$, on ajoute une transition vers $Refuse$ gardée par la négation de la disjonction de toutes les gardes des transitions portant cette action. Sinon, on boucle sur $a$ dans $l$

de $[[\mathcal{TP}]]$ finissant dans un état de $Accept$ (resp. $Refuse$). L'ensemble des exécutions acceptées $Run_{\text{Accept}}([[\mathcal{TP}]])$ (resp. refusées $Run_{\text{Refuse}}([[\mathcal{TP}]])$) est alors défini par :

$$Run_{\text{Accept}}([[\mathcal{TP}]]) = \{\beta^{\rightarrow} = q_0 \stackrel{\alpha_1}{\rightarrow} q_1 \stackrel{\alpha_2}{\rightarrow} \cdots \stackrel{\alpha_n}{\rightarrow} q_n \ \mid \ q_n \in Accept\}$$

Le langage accepté et les traces acceptées par $[[\mathcal{TP}]]$ sont obtenus par projection des exécutions acceptées :

$$\begin{array}{rcl} L_{\text{Accept}}([[\mathcal{TP}]]) & = & proj_{\Lambda}(Run_{\text{Accept}}([[\mathcal{TP}]])) \\ Traces_{\text{Accept}}([[\mathcal{TP}]]) & = & proj_{\Lambda_? \cup \Lambda_!}(Run_{\text{Accept}}([[\mathcal{TP}]])) \end{array}$$

Notons $proj_{\text{S}}$ la projection de $Dom(C^{\text{S}}) \times Dom(V_p^{\text{TP}}) \times Dom(V^{\text{S}})$ sur $Dom(C^{\text{S}}) \times Dom(V^{\text{S}})$. La projection $proj_{\text{S}}$ d'un état de $[[\mathcal{TP}]]$ est donc un état de $[[\mathcal{S}]]$. Par ailleurs $\Sigma^{\text{TP}} = \Sigma^{\text{S}}$ implique aussi que $\Lambda^{\text{TP}} = \Lambda^{\text{S}}$ (deux actions identiques ont la même signature et définissent les mêmes actions valuées). Donc $[[\mathcal{TP}]]$ reconnait des séquences d'actions valuées sur l'alphabet $\Lambda^{\text{S}}$ de $[[\mathcal{S}]]$.

On peut aussi étendre la projection $proj_{\text{S}}$ sur les exécutions par

$$proj_{\text{S}}(\beta^{\rightarrow}) = proj_{\text{S}}(q_0) \stackrel{\alpha_1}{\rightarrow} proj_{\text{S}}(q_1) \stackrel{\alpha_2}{\rightarrow} \cdots \stackrel{\alpha_n}{\rightarrow} proj_{\text{S}}(q_n)$$

et sur les ensembles d'exécutions par

$$proj_{\text{S}}(R) = \{proj_{\text{S}}(\beta^{\rightarrow}) \ \mid \ \beta^{\rightarrow} \in R\} \text{ pour un ensemble } R \text{ d'exécutions.}$$

Les exécutions de $\mathcal{S}$ acceptées par $\mathcal{TP}$ sont alors définies par

$$Run([[\mathcal{S}]]) \cap proj_{\text{S}}(Run_{\text{Accept}}([[\mathcal{TP}]]))$$

$[[\mathcal{TP}]]$ étant complet, par définition, ses exécutions (non nécessairement acceptées) sont non restrictives par rapport aux valeurs des constantes symboliques et variables de $\mathcal{S}$. On a donc

$$\begin{array}{c} proj_{\text{S}}(Run([[\mathcal{TP}]])) = \{\beta^{\rightarrow} = q_0 \stackrel{\alpha_1}{\rightarrow} q_1 \stackrel{\alpha_2}{\rightarrow} \cdots \stackrel{\alpha_n}{\rightarrow} q_n \ \mid \\ \forall i, q_i \in Dom(C^{\text{S}}) \times Dom(V^{\text{S}}) \text{ et } \alpha_i = (a_i, p_i), a_i \in \Sigma^{\text{S}}, p_i \in Dom(type(a_i))\} \end{array}$$

En clair, ce sont toutes les exécutions possibles de toutes les spécifications $\mathcal{S}$ pour lesquelles $\mathcal{TP}$ est un objectif. Le langage de $[[\mathcal{TP}]]$ et ses traces sont donc

$$\begin{array}{rcl} L([[\mathcal{TP}]]) & = & proj_{\Lambda}(Run([[\mathcal{TP}]])) = (\Lambda^{\text{S}})^*. \\ Traces([[\mathcal{TP}]]) & = & (\Lambda^{\text{S}}_? \cup \Lambda^{\text{S}}_!)^* \end{array}$$

**Remarque :** Pour une instanciation $c$ des constantes symboliques $C^{\text{S}}$ de $\mathcal{S}$, l'instance $\mathcal{TP}(c)$ est un observateur au sens habituel du terme d'une instance de spécification, similaire par exemple aux observateurs GOAL dans ObjectGéode, pour une spécification SDL.

### 4.3.2 Produit synchrone

Dans le cas énuméré, la génération de tests était basée sur le calcul d'une intersection de langages ou de traces. Pour les $\mathcal{IOSTS}$, nous calculerons l'intersection des exécutions de la spécification et de l'objectif, afin de sélectionner les exécutions à tester i.e. $Run([[\mathcal{S}]]) \cap proj_{\mathrm{S}}(Run_{\mathrm{Accept}}([[\mathcal{TP}]]))$. Nous donc définissons donc une opération de produit synchrone symbolique sur les $\mathcal{IOSTS}$ $\mathcal{S}$ et $\mathcal{TP}$, de sorte à produire un $\mathcal{IOSTS}$ $\mathcal{PS}$ dont les exécutions acceptées, définies sur sa sémantique $[[\mathcal{PS}]]$, sont (à une projection près) les exécutions de $\mathcal{S}$ acceptées par l'objectif de test $\mathcal{TP}$.

**Définition 4.3.2 (Produit synchrone d'$\mathcal{IOSTS}$)** *Soient $\mathcal{S} = \langle D^s, \Theta^s, \Sigma^s, T^s \rangle$ l'$\mathcal{IOSTS}$ d'une spécification, et $\mathcal{TP} = \langle D^{TP}, \Theta^{TP}, \Sigma^{TP}, T^{TP} \rangle$ un objectif de test pour $\mathcal{S}$ muni de ses conditions $Accept$ et $Refuse$.*

*Le produit synchrone $\mathcal{S} \times_{sy} \mathcal{TP}$ est l'$\mathcal{IOSTS}$ $\mathcal{PS} = \langle D^{PS}, \Theta^{PS}, \Sigma^{PS}, T^{PS} \rangle$ muni des conditions $Accept^{PS}$ et $Refuse^{PS}$ et défini comme suit :*

- *$D^{PS} = C^{PS} \cup V^{PS} \cup P^{PS}$ où*
  - *$V^{PS} = V^S \cup V^{TP} = V^S \cup V_p^{TP} = V^{TP}$, et $V_o^{PS} = \emptyset$ ($\mathcal{PS}$ est fermé)*
  - *$C^{PS} = C^S = C^{TP}$*
  - *$P^{PS} = P^S = P^{TP}$*
- *$\Theta^{PS} = \Theta^S \wedge \Theta^{TP}$ porte donc sur $C^S$, $V^S = V_p^S$ et $V_p^{TP}$. Elle définit un sous-ensemble de $Dom(C^S) \times Dom(V^S) \times Dom(V_p^{TP})$.*
- *$\Sigma^{PS} = \Sigma^S = \Sigma^{TP}$,*
- *$T^{PS}$ est le plus petit ensemble de transitions satisfiant la règle :*

$$\frac{\langle a, \pi_1, G_1, A_1 \rangle \in T^S \wedge \langle a, \pi_2, G_2, A_2 \rangle \in T^{TP}}{\langle a, \pi_1, G_1 \wedge G_2[\pi_2/\pi_1], A_1 \cup A_2[\pi_2/\pi_1], \rangle \in T^{PS}}$$

  *où $G_2[\pi_2/\pi_1]$ (resp. $A_2[\pi_2/\pi_1]$) est la garde (resp. l'affectation) de la transition symbolique $t_2$ dans laquelle chaque paramètre $p_2^i \in \pi_2$ porté par l'action $a$ de $t_2$ est remplacé par le paramètre correspondant $p_1^i \in \pi_1$ porté par l'action $a$ de $t_1$. On peut vérifier que $A_1 \cup A_2$ affecte une et une seule fois chaque variable propre, puisque c'est le cas de $A_1$ qui n'affecte que $V^S$ et de $A_2$ qui n'affecte que $V_p^{TP}$ et ces ensembles sont disjoints.*
- *$Accept^{PS} = Accept^{TP}$ et $Refuse^{PS} = Refuse^{TP}$*

La sémantique de $\mathcal{PS}$ est un IOLTS $[[\mathcal{PS}]]$. Ses états sont composés d'une valuation de ses constantes symboliques $C^{\mathrm{PS}} = C^{\mathrm{S}}$ et de ses variables $V^{\mathrm{PS}} = V^{\mathrm{S}} \cup V_p^{\mathrm{TP}} = V^{\mathrm{TP}}$. Ses actions sont sur l'aphabet $\Lambda^{\mathrm{S}}$. On peut définir les exécutions acceptées par $\mathcal{PS}$ et les ensembles d'exécutions acceptées de la même façon que pour les objectifs de test, $Run_{\mathrm{Accept}}([[\mathcal{PS}]])$ et $Run_{\mathrm{Refuse}}([[\mathcal{PS}]])$.

Les langages et traces acceptés par $\mathcal{PS}$ sont définis par projection de $Run_{\mathrm{Accept}}([[\mathcal{PS}]])$ sur les actions valuées ou les actions valuées visibles :

$$\begin{aligned} L_{\mathrm{Accept}}([[\mathcal{PS}]]) &= proj_{\Lambda}(Run_{\mathrm{Accept}}([[\mathcal{PS}]])) \\ Traces_{\mathrm{Accept}}([[\mathcal{PS}]]) &= proj_{\Lambda_? \cup \Lambda_!}(Run_{\mathrm{Accept}}([[\mathcal{PS}]])) \end{aligned}$$

Noter qu'on a pris une définition plus simple des traces acceptées que dans le cas des IOLTS où on définissait celles-ci par complémentaire des traces refusées.

Les variables propres $V_p^{\text{TP}}$ ont servi à contraindre les exécutions acceptées de $[[\mathcal{PS}]]$. La projection $proj_{\text{S}}$ des états de $[[\mathcal{PS}]]$ sur $Dom(C^{\text{S}}) \times Dom(V^{\text{S}})$ abstrait les valeurs des variables propres de $V_p^{\text{TP}}$ dans $[[\mathcal{PS}]]$ et produit des états de $[[\mathcal{S}]]$.

Dans le cas des IOLTS, les objectifs de test externes reconnaissaient des traces, Dans ce cas, les traces acceptées du produit étaient l'intersection des traces de la spécification et des traces acceptées du produit. Les objectifs de test internes étaient plus précis et reconnaissaient des langages d'actions. On avait un résultat similaire pour les langages acceptés. Par contre, on a seulement l'inclusion des traces. L'inclusion vient du fait que la projection ne commute pas avec l'intersection mais préserve seulement l'inclusion : $proj(A \cap B) \subseteq proj(A) \cap proj(B)$.

Dans le cas des $\mathcal{IOSTS}$ la sémantique des objectifs de test est encore plus précise, puisque ceux-ci reconnaissent des exécutions, donc des séquences d'états et d'actions valuées. Deux séquences d'actions identiques peuvent donc être distinguées si elles passent par des états différents. On souhaite obtenir un résultat similaire (à projection près) sur les exécutions acceptées du produit par rapport aux exécutions de la spécification et aux exécutions acceptées de l'objectif. On souhaite donc une égalité pour les exécutions acceptées (à projection près car les exécutions ne portent pas sur les mêmes variables). Pour la même raison que précédemment, on aura seulement inclusion pour les langages acceptés et les traces.

Pour obtenir ce résultat, nous allons devoir caractériser la sémantique du produit d'$\mathcal{IOSTS}$ en fonction de la sémantique de la spécification et de l'objectif. Malheureusement la sémantique du produit d'$\mathcal{IOSTS}$ n'est pas le produit des sémantiques des $\mathcal{IOSTS}$, pour le produit $\times$ défini pour les IOLTS.

Il nous faut donc definir un produit synchrone $\times_e$ sur les IOLTS plus fin que $\times$ (voir 3.2.6 au chapitre 3), c'est à dire qui synchronise sur les actions communes mais seulement quand les valeurs des constantes et des variables de $V^{\text{S}}$ coïncident. La synchronisation est alors définie par l'égalité des projections $proj_{\text{S}}$ des états sur $Dom(C^{\text{S}}) \times Dom(V^{\text{S}})$ et la synchronisation des actions dans $\Lambda$. Nous donnosn ici une définition précise pour des IOLTS $S$ et $TP$.

**Définition 4.3.3 (Produit synchrone étendu d'IOLTS)** *Soient $S = (Q^s, \Lambda, \rightarrow_s, Q_0^s)$ un IOLTS et $TP = ((Q^{TP}, \Lambda, \rightarrow_{TP}, Q_0^{TP})$ un objectif de test compatible avec $S$, i.e. un IOLTS muni d'états $Accept^{TP}$ tq $Q^{TP} = Q^s \times Q_p^{TP}$ et $Q_0^{TP} = Q^s \times Q'^{TP}_0$. On suppose $TP$ complet i.e. $\forall q \in Q^{TP}, \forall a \in \Lambda, q \xrightarrow{a}_{TP}$.*

*Le produit synchrone étendu de $S$ et $TP$ est l'IOLTS $S \times_e TP = (Q, \Lambda, relt_e, Q_0)$ muni de l'ensemble $Accept_e$, avec*

- $Q = Q^{TP}$
- $Q_0 = Q_0^s \times Q_p^{TP} \cap Q_0^{TP}$
- $Accept_e = Accept^{TP}$
- $\rightarrow_e$ *est définie par de la façon suivante :*
  *pour toute paire d'états $q_{TP}, q'_{TP}$ de $TP$, et toute action $\alpha \in \Lambda$,*

$$q_{TP} \xrightarrow{\alpha}_e q'_{TP} \quad \triangleq \quad proj_S(q_{TP}) \xrightarrow{\alpha}_S proj_S(q'_{TP}) \ \wedge \ q_{TP} \xrightarrow{\alpha}_{TP} q'_{TP}$$

Dans le produit synchrone étendu, un état est un état de $TP$, sachant que ceux-ci sont composés d'un état de $Q^{\mathrm{s}}$ et d'un état de $Q_p^{\mathrm{TP}}$. Pour que dans cet état une action soit tirable, il faut que cet état puisse tirer l'action dans $TP$ et que sa projection sur les états de $S$ permette également cette action. On synchronise donc à la fois sur les actions et sur la composante dans $Q^{\mathrm{s}}$ du contenu des états.

On a alors les propriétés suivantes qui caractérisent les exécutions du produit étendu en fonction des exécutions de $S$ et $TP$ :

**Proposition 4.3.1**

$$proj_{\mathrm{s}}(Run(S \times_e TP)) = Run(S)$$
$$proj_{\mathrm{s}}(Run_{Accept}(S \times_e TP)) = Run(S) \cap proj_{\mathrm{s}}(Run_{Accept}(TP))$$

**Preuve :** La première égalité signifie que les exécutions du produit ne restreignent pas les exécutions de $S$. Elle découle d'une part du fait que les condtions initiales de $S$ et $TP$ sont indépendantes (elles ne portent pas sur les mêmes composantes des états, celle de $S$ retreignant $Q^{\mathrm{s}}$ et celle de $TP$ retreignant $Q_p^{\mathrm{TP}}$, et d'autre part de la de la complétude de $TP$.

La deuxième égalité découle de la définition de $\rightarrow_e$ et de $Accept_e$. Une exécution acceptée de $S \times_e TP$ est trivialement une exécution acceptée de $TP$. De plus sa projection est une exécution de $S$.

□

Puisque $TP$ est complet, il est immédiat de constater que les blocages du produit correspondent aux blocages de $S$ i.e. pour tout état $q$ de $S \times_e TP$, $q \in Quiescent(S \times_e TP) \iff proj_{\mathrm{s}}(q) \in Quiescent(S)$

Par conséquent on a le lemme suivant qui servira plus tard dans la suspension symbolique :

**Lemme 4.3.1**

$$STraces(S \times_e TP) = STraces(S)$$

D'après les définitions de $\times_{sy}$ et de $\times_e$, et de la sémantique des $\mathcal{IOSTS}$, il est alors aisé de constater que la sémantique du produit d'$\mathcal{IOSTS}$ est le produit des sémantiques des $\mathcal{IOSTS}$ :

**Proposition 4.3.2**

$$[[\mathcal{S} \times_{sy} \mathcal{TP}]] = [[\mathcal{S}]] \times_e [[\mathcal{TP}]]$$

**Preuve :** La preuve est simple mais fastidieuse. Elle découle directement de la définition du produit synchrone symbolique $\times_{sy}$ (definition 4.3.2), de la sémantique IOLTS des $\mathcal{IOSTS}$(définition 4.1.2) et de la définition du produit synchrone étendu $\times_e$ (définition 4.3.3). Il suffit d'expliciter la sémantique $[[\mathcal{S} \times_{sy} \mathcal{TP}]]$ à partir de la définition de $\mathcal{S} \times_{sy} \mathcal{TP}$, et de la comparer avec le produit synchrone étendu des sémantiques de $\mathcal{S}$ $\mathcal{TP}$ qu'il faut d'abord expliciter.

□

La proposition 4.3.3 suivante est alors une conséquence directe des propositions 4.3.2 et 4.3.1 :

**Proposition 4.3.3 (Propriétés du produit synchrone symbolique)**

$$\begin{array}{rcl}
Run([[\mathcal{S} \times_{sy} \mathcal{TP}]]) & = & Run([[\mathcal{S}]] \times_e [[\mathcal{TP}]]) \\
 & & \textit{par la proposition 4.3.2 d'où} \\
proj_s(Run([[\mathcal{S} \times_{sy} \mathcal{TP}]])) & = & proj_s(Run([[\mathcal{S}]] \times_e [[\mathcal{TP}]])) \\
 & = & Run([[\mathcal{S}]]) \\
 & & \textit{car } [[\mathcal{TP}]] \textit{ est complet} \\
proj_s(Run_{Accept}([[\mathcal{S} \times_{sy} \mathcal{TP}]])) & = & proj_s(Run_{Accept}([[\mathcal{S}]] \times_e [[\mathcal{TP}]])) \\
 & = & Run([[\mathcal{S}]]) \ \cap \ proj_s(Run_{Accept}([[\mathcal{TP}]])) \\
 & & \textit{par la proposition 4.3.1}
\end{array}$$

La sélection sur les exécutions étant plus fine que la sélection sur les langages (ou les traces), cette proposition 4.3.3 n'implique pas l'égalité des langages suivante : $L_{\text{Accept}}([[\mathcal{PS}]]) = L([[\mathcal{S}]]) \cap L_{\text{Accept}}([[\mathcal{TP}]])$ mais seulement l'inclusion :

**Corollaire 4.3.1** *On a les inclusions de langages et de traces acceptées suivantes :*

$$\begin{array}{rcl}
L_{Accept}([[\mathcal{S} \times_{sy} \mathcal{TP}]]) & \subseteq & L([[\mathcal{S}]]) \cap L_{Accept}([[\mathcal{TP}]]) \\
Traces_{Accept}([[\mathcal{PS}]] = Traces_{Accept}([[\mathcal{S} \times_{sy} \mathcal{TP}]]) & \subseteq & Traces([[\mathcal{S}]]) \cap Traces_{Accept}([[\mathcal{TP}]])
\end{array}$$

**Preuve :** Le corollaire 4.3.1 s'obtient depuis la proposition 4.3.3 par projection (sur les actions valuées) :

$$\begin{array}{rcl}
L_{\text{Accept}}([[\mathcal{PS}]]))) & = & proj_\Lambda(Run_{\text{Accept}}([[\mathcal{PS}]]))) \\
 & = & proj_\Lambda(proj_{\text{s}}(Run_{\text{Accept}}([[\mathcal{PS}]]))) \\
 & = & proj_\Lambda(Run([[\mathcal{S}]]) \cap proj_{\text{s}}(Run_{\text{Accept}}([[\mathcal{TP}]])) \\
 & \subseteq & proj_\Lambda(Run([[\mathcal{S}]])) \cap proj_\Lambda(proj_{\text{s}}(Run_{\text{Accept}}([[\mathcal{TP}]]))) \\
 & & \text{par la propriété } proj(A \cap B) \subseteq proj(A) \cap proj(B) \\
 & = & L([[\mathcal{S}]]) \cap L_{\text{Accept}}([[\mathcal{TP}]])))
\end{array}$$

L'inclusion de traces s'obtient de façon identique en remplaçant $proj_\Lambda$ par $proj_{\Lambda_? \cup \Lambda_!}$
□

En termes de langages et traces (non nécessairement acceptées) on a par contre les égalités suivantes qui découlent directement de la proposition 4.3.3 par projection :

**Proposition 4.3.4**

$$\begin{array}{rcl}
L([[\mathcal{S} \times_{sy} \mathcal{TP}]]) & = & L([[\mathcal{S}]]) \\
Traces([[\mathcal{S} \times_{sy} \mathcal{TP}]]) & = & Traces([[\mathcal{S}]])
\end{array}$$

**Exemple** Imaginons par exemple une spécification $\mathcal{S}$ possédant une seule variable $x$, initialisée à 0, et d'unique transition $x = 0; !a; x := 0$. L'ensemble des exécutions de $[[\mathcal{S}]]$ est $Run([[\mathcal{S}]]) = \{(x = 0) \xrightarrow{a} x = 0\}$ et son langage $L([[\mathcal{S}]]) = \{a\}$. Soit $\mathcal{TP}$ l'objectif observant $\mathcal{S}$, sans variable propre, et ayant deux transitions $x = 1; a$ vers $Accept$ et $x \neq 1; a$ vers $Refuse$. Les exécutions de $[[\mathcal{TP}]]$ sont $Run([[\mathcal{TP}]]) = \{(x = m) \xrightarrow{a} (x = n), n, m \in Dom(x)\}$ et ses exécutions acceptées sont $Run_{\text{Accept}}([[\mathcal{TP}]]) = \{(x = 1) \xrightarrow{a} x = n, n \in Dom(x)\}$. Son langage est $L([[\mathcal{TP}]]) = \{a\}$ son langage accepté est $L_{\text{Accept}}([[\mathcal{TP}]]) = \{a\}$. Le produit synchrone $\mathcal{PS}$ aura les transitions $x = 0 \wedge x = 1; a; x := 0$ vers $Accept$ et $x = 0 \wedge x \neq 1; a; x := 0$ vers $Refuse$. Les exécutions de $[[\mathcal{PS}]]$ sont $Run([[\mathcal{PS}]]) = \{(x = 0) \xrightarrow{a} (x = 0)\}$ et ses exécutions acceptées $Run_{\text{Accept}}([[\mathcal{PS}]]) = \emptyset$. Son langage est $L([[\mathcal{PS}]]) = \{a\}$ et son langage accepté est $L_{\text{Accept}}([[\mathcal{PS}]]) = \emptyset$. On a bien

$$\begin{array}{l} proj_{\text{S}}(Run_{\text{Accept}}([[\mathcal{PS}]])) = Run([[\mathcal{S}]]) \cap proj_{\text{S}}(Run_{\text{Accept}}([[\mathcal{TP}]]) = \emptyset, \\ L_{\text{Accept}}([[\mathcal{PS}]]) = \emptyset \subseteq L([[\mathcal{S}]]) \cap L_{\text{Accept}}([[\mathcal{TP}]]) = \{a\}, \text{ et} \\ L([[\mathcal{PS}]]) = L([[\mathcal{S}]]) = \{a\} \end{array}$$

### 4.3.3 Comportements visibles

Pour pouvoir générer des tests symboliques ayant de bonnes propriétés pour la relation de conformité **ioco**, il faut être capable de déterminer les traces suspendues (entrées, sorties et blocages) d'un $\mathcal{IOSTS}$. Pour un $\mathcal{IOSTS}$ $\mathcal{M}$ (ici $\mathcal{PS}$), sa sémantique $[[\mathcal{M}]]$ est un IOLTS, pour lequel on sait calculer son automate de suspension $\Delta([[\mathcal{M}]])$ en ajoutant les boucles de $\delta$ pour tout état de blocage, i.e. dans $quiescent([[\mathcal{M}]])$. On sait ensuite calculer un automate déterministe $det(\Delta([[\mathcal{M}]]))$ ayant les mêmes traces que $\Delta([[\mathcal{M}]])$ i.e. les traces suspendues de $[[\mathcal{M}]]$.

On se propose donc de définir deux opérations symboliques, i.e. définies sur les $\mathcal{IOSTS}$, $\Delta_{sy}$ et $det_{sy}$ telles que, pour tout $\mathcal{IOSTS}$ $\mathcal{M}$, on ait les propriétés

$$\begin{array}{rcl} Traces([[\Delta_{sy}(\mathcal{M})]]) & = & Traces(\Delta([[\mathcal{M}]])) \text{ et} \\ Traces([[det_{sy}(\mathcal{M})]]) & = & Traces(det([[\mathcal{M}]])) \end{array}$$

#### Suspension symbolique

On supposera d'abord que les $\mathcal{IOSTS}$ n'ont pas de boucle d'action internes, plus exactement, que leur sémantique IOLTS ne permet pas de boucles d'actions internes. Cette limitation est usuelle, car la terminaison des boucles est indécidable. Noter que nous ne l'avions cependant pas faite dans les IOLTS. Les seuls blocages sont donc les blocages de sorties (incluant les deadlocks).

Soit $\mathcal{M} = \langle D, \Theta, \Sigma, T \rangle$ un $\mathcal{IOSTS}$ fermé et initialisable et $[[\mathcal{M}]] = (Q, \Lambda, \rightarrow, Q_0)$ sa sémantique IOLTS. Par définition, un état $q = (c, v)$ de $[[\mathcal{M}]]$ est un blocage de sortie s'il n'a ni sortie, ni action interne i.e

$$[\forall \langle a, p \rangle \in \Lambda^{\text{M}}_{!}, \neg(q \xrightarrow{\langle a,p \rangle})] \ \wedge \ [\forall \langle \tau \rangle \in \Lambda^{\text{M}}_{\tau}, \neg(q \xrightarrow{\langle \tau \rangle})]$$

Par définition de la sémantique des $\mathcal{IOSTS}$, ceci se réécrit en

$$\begin{array}{l} [\forall t, t = \langle a, \pi, G, A \rangle \wedge a \in \Sigma_{!}, \forall p \in Dom(\pi), \neg(\langle c, v, p \rangle \in G)] \ \wedge \\ [\forall t, t = \langle \tau, \langle \rangle, G, A \rangle \wedge \tau \in \Sigma_{\tau}, \neg(\langle c, v \rangle \in G)] \end{array}$$

Ou encore

$$[\forall t, t = \langle a, \pi, G, A\rangle \wedge a \in \Sigma_!, \neg(\langle c, v\rangle \in proj_{Dom(C\cup V\cup\pi)\to Dom(C\cup V)}(G))] \ \wedge$$
$$[\forall t, t = \langle \tau, \langle\rangle, G, A\rangle \wedge \tau \in \Sigma_\tau, \neg(\langle c, v\rangle \in G)]$$

On peut donc donner la définition suivante de la construction d'un $\mathcal{IOSTS}$ suspendu :

**Définition 4.3.4 ($\mathcal{IOSTS}$ suspendu)** *L'$\mathcal{IOSTS}$ suspendu de $\mathcal{M}$ est donc défini par l'$\mathcal{IOSTS}$ $\Delta_{sy}(\mathcal{M})$ obtenu à partir de $\mathcal{M}$ en ajoutant à $T$ la transition $t_\delta = \langle \delta, \langle\rangle, G_\delta, A_\delta\rangle$ où*

$$\begin{aligned} G_\delta \quad = \quad & \bigcap_{\{t \in T \mid t = \langle a, \pi, G, A\rangle \wedge a \in \Sigma_!\}} proj_{Dom(C\cup V\cup\pi)\to Dom(C\cup V)}(\overline{G}) \\ & \cap \bigcap_{\{t \in T \mid t = \langle \tau, \langle\rangle, G, A\rangle \wedge \tau \in \Sigma_\tau\}} \overline{G} \end{aligned}$$

*et $A_\delta$ est l'identité i.e. l'ensemble des affectations $v_p := v_p$ pour toutes les variables propres $v_p \in V_p$.*

**Remarques :** Quelques remarques s'imposent pour mieux comprendre cette définition et ses conséquences. Une garde de transition introduit implicitement une quantification existentielle sur la valeur des paramètres $p$ des actions. En effet, pour un état donné, une transition est tirable, s'il existe une valeur des paramètres qui satisfait la garde. La négation de la garde dans $G_\delta$ introduit donc une quantification universelle. Cette quantification universelle peut être vue comme une projection de la garde sur les domaines des constantes et variables. La garde $G_\delta$ est donc un sous-ensemble de $Dom(C) \times Dom(V)$. Elle ne porte donc plus sur les paramètres de communication. Cependant, toute la difficulté consiste à calculer les projections des gardes de toutes les transitions d'émission.

En effet, prenons par exemple une transition portant l'action $a!(\pi)$ et gardée par $G : V = \pi^2 + C$ avec $Dom(\pi) = Dom(V) = Dom(C) = |R$. Elle est tirable en $\langle(c, v)$ s'il existe $p \in |R$ tq $v - c = p^2$. Donc aussitôt que $v - c \geq 0$. Elle n'est pas tirable si $v - c < 0$. La projection de $\overline{G}$ est donc $V - C < 0$ qui sera un des facteurs de la garde $G_\delta$. Par contre si $Dom(\pi) = Dom(V) = Dom(C) = |N$, la projection de $\overline{(G)}$, est beaucoup plus complexe à calculer, puisqu'il s'agit des couples $(c, v)$ d'entiers tels que $v - c$ ne soit pas un carré.

Dans le cas des $\mathcal{IOSTS}$ avec localités, l'unique transition $t_\delta$, peut être remplacée par un ensemble de transitions $t_{\delta,l_i}$, une pour chaque localité $l_i$. En effet une garde $\neg((l = l_1) \wedge G')$ se simplifie en $\neg G'$ si $l = l_1$ et en $false$ sinon.

Notons qu'une définition informelle du calcul de l'automate de suspension symbolique est donnée dans l'article [56].

La propriété suivante nous dit que le calcul symbolique des blocages par l'opération $\Delta_{sy}$ est exactement le résultat recherché puisqu'il coïncide avec le le calcul des blocages sur la sémantique :

**Proposition 4.3.5 (Propriétés de la quiescence symbolique)** *Pour tout $\mathcal{IOSTS}$ $\mathcal{M}$ on a*

$$[[\Delta_{sy}(\mathcal{M})]] = \Delta([[\mathcal{M}]])$$

**Preuve :**Les opérations $\Delta_{sy}$ et $\Delta$ n'ont aucun effet sur les ensemble d'états donc les états de $[[\Delta_{sy}(\mathcal{M})]]$ sont les mêmes que ceux de $\Delta([[\mathcal{M}]])$. Il reste donc à montrer que les relations de transition de $[[\Delta_{sy}(\mathcal{M})]]$ et $\Delta([[\mathcal{M}]])$ coïncident. Comme elles coïncident trivialement sur tous les états pour toute action valuée différente de $\delta$, il suffit de montrer qu'elles coïncident aussi sur $\delta$. En effet

$$\begin{aligned}
q \xrightarrow{\delta}_{[[\Delta_{sy}(\mathcal{M})]]} q \quad &\Longleftrightarrow \quad q \in G_\delta \\
&\Longleftrightarrow \quad q \in \bigcap_{\{t \mid t = \langle a, \pi, G, A\rangle \wedge a \in \Sigma_!\}} proj_{Dom(C \cup V \cup \pi) \to Dom(C \cup V)}(\overline{G}) \cap \\
&\qquad\qquad \bigcap_{\{t \mid t = \langle \tau, \langle\rangle, G, A\rangle \wedge \tau \in \Sigma_\tau\}} \overline{G} \\
&\Longleftrightarrow \quad q \in quiescent([[\mathcal{M}]]) \\
&\Longleftrightarrow \quad q \xrightarrow{\delta}_{\Delta([[\mathcal{M}]])} q
\end{aligned}$$

□

En termes d'exécutions et d'exécutions acceptées, on en déduit les propriétés suivantes :

**Corollaire 4.3.2**

$$Run([[\Delta_{sy}(\mathcal{M})]]) = Run(\Delta([[\mathcal{M}]]))$$

*Pour un $\mathcal{IOSTS}$ avec des états accepteurs on a de plus*

$$Run_{Accept}([[\Delta_{sy}(\mathcal{M})]]) = Run_{Accept}(\Delta([[\mathcal{M}]]))$$

*Par projection, on a les mêmes propriétés sur les langages et langages acceptés, traces et traces acceptées.*

En appliquant à $\mathcal{PS} = \mathcal{S} \times_{sy} \mathcal{TP}$ les propriétés de $\Delta_{sy}$ (proposition 4.3.5) on obtient :

$$[[\Delta_{sy}(\mathcal{PS})]] = \Delta([[\mathcal{PS}]])$$

En termes de traces, ceci permet de déduire en particulier que :

**Corollaire 4.3.3**

$$Traces([[\Delta_{sy}(\mathcal{PS})]]) = STraces([[\mathcal{S}]])$$

**Preuve :** En effet on a

$$\begin{aligned}
Traces([[\Delta_{sy}(\mathcal{PS})]]) = \quad &= \quad Traces(\Delta([[\mathcal{PS}]])) \\
&= \quad STraces([[\mathcal{PS}]] \\
&= \quad STraces([[\mathcal{S} \times_{sy} \mathcal{TP}]])) \\
&= \quad STraces([[\mathcal{S}]] \times_e [[\mathcal{TP}]]) \text{ par prop 4.3.2} \\
&= \quad STraces([[\mathcal{S}]]) \text{ par le lemme 4.3.1}
\end{aligned}$$

□

En d'autres termes, la sémantique de la suspension symbolique du produit symbolique de la spécification et de l'objectif préserve toutes les traces suspendues de la sémantique de la spécification. $\Delta_{sy}\mathcal{PS}$ fournit donc de bonnes bases pour calculer les tests symboliques. Parmi les traces suspendues, certaines seulement sont acceptées et seront utilisées.

**Déterminisation**

La déterminisation des $\mathcal{IOSTS}$ est plus problèmatique. Rappelons d'abord (voir définition 3.1.5) que pour un IOLTS $M = (Q^{\mathrm{M}}, A^{\mathrm{M}}, \rightarrow_{\mathrm{M}}, q_0^{\mathrm{M}})$, on peut définir un IOLTS de mêmes traces que $M$ et déterministe $det(M) = (2^{Q^{\mathrm{M}}}, A^{\mathrm{M}}_{\mathrm{VIS}}, \rightarrow_{\mathrm{det}}, q_0^{\mathrm{M}}$ after $\varepsilon)$ avec $P \xrightarrow{a}_{\mathrm{det}} P'$ si $P' = P$ after $a$. Pour un IOLTS $[[\mathcal{M}]]$ sémantique de $\mathcal{M}$, on peut donc définir l'IOLTS $det([[\mathcal{M}]])$ . Un méta-état de $det([[\mathcal{M}]])$ est alors composé d'un ensemble d'états de l'IOLTS $[[\mathcal{M}]]$ de départ. C'est un ensemble de valuations des constantes symboliques et variables, donc un sous-ensemble de $Dom(C) \times Dom(V)$. Ce méta-état est alors réprésentable par un prédicat sur $C \cup V$.

La déterminisation d'$\mathcal{IOSTS}$ consiste donc à définir une transformation $det_{sy}$ sur les $\mathcal{IOSTS}$ telle que pour tout $\mathcal{IOSTS}$ $\mathcal{M}$, $Traces([[det_{sy}(\mathcal{M})]])$ soit exactement $Traces(det([[\mathcal{M}]]))$ et $[[det_{sy}(\mathcal{M})]]$ soit déterministe. Pour que $[[det_{sy}(\mathcal{M})]]$ ait cette propriété, il faudrait que $det_{sy}(\mathcal{M})$ soit un $\mathcal{IOSTS}$ (avec la définition prise jusqu'ici) qui manipule une variable $pV$ de domaine $2^{Dom(V)}$, c'est à dire dont les valeurs sont des parties de $Dom(V)$ i.e. des prédicats sur $V$. Pour une transition $t = \langle a, \pi, G, A\rangle$, la garde serait alors un prédicat sur $C \cup \pi \cup pV$, donc un sous-ensemble de $Dom(C) \cup Dom(P) \cup Dom(pV)$. L'affectation serait une fonction de $Dom(C) \times Dom(P) \times Dom(pV)$ dans $Dom(pV)$ qui affecte à $pV$ un prédicat sur $V$ en fonction des valeurs des constantes symboliques et paramètres de communication. Remarquons que, comme on n'a fait aucune restriction sur les types de variables des $\mathcal{IOSTS}$ (voir la définition 4.1.1), on peut garder le même modèle pour représenter un $\mathcal{IOSTS}$ obtenu par déterminisation. Cependant en pratique, on sera amené à analyser l'accessibilité et la coaccessibilité des $\mathcal{IOSTS}$. Il faudra faire des restrictions très fortes pour que cette analyse soit possible, même de manière approchée.

On a vu que pour les IOLTS, comme pour les automates, la déterminisation se décompose en deux opérations :

- l'$\epsilon$-clôture, i.e. le calcul pour un état $q$ des états accessibles par des actions internes, i.e. $q$ after $\epsilon$
- le calcul de sous-ensemble i.e. pour un ensemble d'etats $P$ et pour une action $a$, l'ensemble $P\ after\ a$.

C'est le calcul de sous-ensemble qui obligerait à considérer une variable portant sur des ensembles, donc à faire des restrictions fortes sur les modèles. Par contre on peut, sous l'hypothèse qu'un $\mathcal{IOSTS}$ $\mathcal{M}$ n'a pas de cycle d'actions internes, calculer un $\mathcal{IOSTS}$ $\epsilon(\mathcal{M})$ portant sur les mêmes variables que $\mathcal{M}$, qui a les mêmes traces et n'a plus d'actions internes.

**$\epsilon$-clôture symbolique** L'idée de l'$\epsilon$-clôture est de propager l'effet des séquences d'actions internes sur la prochaine action visible. Cet effet va se caractériser par une composi-

tion des gardes et affectations le long de la séquence.

Soit $\mathcal{M} = \langle D, \Theta, \Sigma, T \rangle$ un $\mathcal{IOSTS}$ fermé ($V = V_p$), sans boucle d'actions internes (dont la sémantique $[[\mathcal{M}]]$ n'a pas de boucle d'actions internes). Notons $T_\tau$ le sous-ensemble des transitions de $T$ dont les actions sont internes i.e. dans $\Sigma_\tau$. Pour une transition $t = \langle a, \pi, G, A \rangle$, la garde $G$ est un prédicat sur $C \cup V \cup \pi$ (ou $C \cup V$ pour les actions internes et $\delta$). Par commodité, on considérera ici une garde $G$ comme une fonction de $Dom(C) \times Dom(V) \times Dom(\pi)$ dans $\{true, false\}$ (ou de $Dom(C) \times Dom(V)$ dans $\{true, false\}$). L'affectation $A$ est une fonction de $Dom(C) \times Dom(V) \times Dom(\pi)$ dans $Dom(V)$ (ou de $Dom(C) \times Dom(V)$ dans $Dom(V)$). Pour des raisons de typage de composition, on notera $G(c)$ la fonction $G$ pour $c$ fixé, $G(c)(p)$ la fonction $G$ pour $c$ et $p$ fixés, $A(c)$ la fonction $A$ pour $c$ fixé et $A(c)(p)$ la fonction $A$ pour $c$ et $p$ fixés.

Soit $(c, v_i)$ un état de $[[\mathcal{M}]]$ et $t_i = \langle \tau_i, \langle\rangle, G_i(c), A_i(c) \rangle$ une transition de $T_\tau$. D'après la sémantique des $\mathcal{IOSTS}$, $t_i$ est tirable en $(c, v_i)$ si $G_i(c)(v_i) = true$. L'action valuée $\langle \tau_i \rangle$ mène alors dans l'état $(c, A_i(c, v_i))$. La séquence de transitions $t_1.t_2.\ldots.t_n$ de $(T_\tau)^*$ est tirable en $(c, v_0)$ si $G_1(c)(v_0) \wedge G_2(c) \circ A_1(c)(v_0) \wedge \ldots G_n(c) \circ A_{n-1}(c) \circ \cdots A_2(c) \circ A_1(c)(v_0)$. La séquence d'actions $\tau_1 \cdots \tau_n$ mène alors dans l'état $(c, v_n)$ avec $v_n = A_n(c) \circ \cdots A_2(c) \circ A_1(c)(v_0)$.

Enfin, une transition visible $t = \langle a, \langle \pi, G(c), A(c) \rangle$ est tirable en cet état $(c, v_n)$ si il existe $p \in Dom(\pi)$ tel que $G(c)(p)(v_n) = true$. L'action valuée $\langle a, p \rangle$ mène alors à l'état $(c, v')$ avec $v' = A(c)(p)(v_n)$.

Toute séquence finie de transitions $t_1.t_2.\ldots.t_n.t$ de $(T_\tau)^*.(T \setminus T_\tau)$ peut donc être remplacée par la transition $\langle a, \pi, G_1(c) \wedge G_2(c) \circ A_1(c)(v_0) \wedge \ldots G_n(c) \circ A_{n-1}(c) \cdots A_2(c) \circ A_1(c)(v_0) \wedge G(c) \circ A_n(c) \circ A_{n-1}(c) \circ A_2(c) \circ A_1(c), A \circ A_n(c) \circ A_{n-1}(c) \cdots \circ A_2(c) \circ A_1(c) \rangle$.

Si on opère cette transformation sur toutes les séquences finies de $(T_\tau)^*.(T \setminus T_\tau)$ d'un $\mathcal{IOSTS}$ $\mathcal{M}$, on obtient un $\mathcal{IOSTS}$ $\epsilon(\mathcal{M})$ sans actions internes, et de mêmes traces que $\mathcal{M}$.

Cependant, dans notre modèle d'$\mathcal{IOSTS}$, l'ensemble des séquences finies de $(T_\tau)^*.(T \setminus T_\tau)$ est infini aussitôt qu'il y a une transition interne, ce qui pose problème pour opérer cette transformation de manière effective par énumération de l'ensemble de telles séquences. Par contre, dans le sous-modèle $\mathcal{IOSTS}_{loc}$ des $\mathcal{IOSTS}$ avec localités, on peut facilement trouver des conditions suffisantes pour une construction effective en considérant l'abstraction $abs(\mathcal{M})$. En effet, pour toute séquence $\sigma$ tirable dans $[[\mathcal{M}]]$, son abstraction $\alpha(\sigma)$ sur $\Sigma$ est aussi tirable dans $abs(\mathcal{M})$ (puisque $abs(\mathcal{M})$ est une abstraction de $[[\mathcal{M}]]$). Donc si $abs(\mathcal{M})$ n'a pas de boucle d'actions internes, il en est de même de $[[\mathcal{M}]]$. Si on se restreint donc aux $\mathcal{IOSTS}_{loc}$ $\mathcal{M}$ tels que $abs(\mathcal{M})$ n'a pas de boucles d'actions internes les séquences de $(\Sigma_\tau)^*.(\Sigma \setminus \Sigma_\tau)$ tirables dans $abs(\mathcal{M})$ sont en nombre fini, et on peut les énumérer facilement depuis chaque localité. Il suffit alors d'opérer la transformation sur l'ensemble des séquences de $(T_\tau)^*.(T \setminus T_\tau)$ dont l'image par $abs$ est dans $(\Sigma_\tau)^*.(\Sigma \setminus \Sigma_\tau)$. Puisque $T$ est fini, pour chaque séquence $\tau_1.\ldots.\tau_n.a$ de $(\Sigma_\tau)^*.(\Sigma \setminus \Sigma_\tau)$, le nombre de séquences $t_1.\ldots.t_n.t$ telles que $abs(t_1.\ldots.t_n.t) = \tau_1.\ldots.\tau_n.a$ est fini. Donc l'ensemble des séquences $(T_\tau)^*.(T \setminus T_\tau)$ est fini. En pratique, on n'a pas besoin d'énumérer toutes ces séquences si on prend la peine de factoriser les suffixes. Pour une localité $l$, tous les chemins dans $(T_\tau)^*.(T \setminus T_\tau)$ menant dans $l$ ont *a priori* des conditions différentes.

Ces remarques nous mènent à la définition suivante :

**Définition 4.3.5 ($\epsilon$-clôture symbolique)** *Soit $\mathcal{M} = \langle D, \Theta, \Sigma, T \rangle$ un $\mathcal{IOSTS}_{loc}$ avec lo-*

*calités où $D = C \cup V \cup P$ avec $l \in V$ la variable des localités, de domaine $Dom(l) = L$. Soit $abs_{loc}(\mathcal{M}) = (L, \Sigma, \rightarrow_{abs}, L_0)$ l'IOLTS abstraction de $\mathcal{M}$ sur les localités. Supposons que $abs_{loc}(\mathcal{M})$ n'ait pas de boucle d'actions internes. On définit l'$\epsilon$ clôture symbolique de $\mathcal{M}$ comme l'$\mathcal{IOSTS}$ $\epsilon_{sy}(\mathcal{M}) = \langle D, \Theta, \Sigma_! \cup \Sigma_?, T_\epsilon \rangle$ tel que $T_\epsilon$ est le plus petit ensemble défini comme suit. Pour toute localité $l_0 \in L_0 \cup post_{\Sigma_! \cup \Sigma_?}(L)$ (initial ou successeur d'une action visible) et pour toute exécution $l_0 \xrightarrow{\tau_1} l_1 \xrightarrow{\tau_2} \ldots l_{n-1} \xrightarrow{\tau_n} l_n \xrightarrow{a} l_{n+1}$ de $abs_{loc}(\mathcal{M})$, telle que $\forall i \in [1, n], \tau_i \in \Sigma_\tau$, $a \in \Sigma_! \cup \Sigma_?$, pour toute séquence $t_1.t_2.\ldots.t_n.t$ de transition de $\mathcal{M}$ dans $(T_\tau)^*.(T \setminus T_\tau)$ telle que $\forall i \in [1, n], t_i = \langle l_{i-1}, \tau_i, \pi, G_i, A_i, l_i \rangle \in T$ (donc $abs_{loc}(t_i) = (l_{i-1}, \tau_i, l_i)$) et $t = \langle l_n, a, \pi, G, A, l_{n+1} \rangle$ (donc $abs_{loc}(t) = (l_n, a, l_{n+1})$), la transition $t_\epsilon = \langle l_0, a, G_\epsilon, A_\epsilon, l_{n+1}$ est dans $T_\epsilon$ où $t_\epsilon$ est définie par :*

$$\begin{aligned} G_\epsilon &= G_1 \wedge G_2 \circ A_1 \wedge \ldots \wedge G_n \circ A_{n-1} \circ \cdots A_2 \circ A_1 \wedge G \circ A_n \circ A_{n-1} \cdots \circ A_2 \circ A_1 \\ A_\epsilon &= A \circ A_n \circ A_{n-1} \circ \cdots A_2 \circ A_1 \end{aligned}$$

*Si $\mathcal{M}$ est muni de la condition $Accept$, l'$\mathcal{IOSTS}$ $\epsilon_{sy}(\mathcal{M})$ est muni de la même condition $Accept$.*

Nous n'avons pas défini formellement l'$\epsilon$-clôture en tant que telle pour les IOLTS, celle-ci étant masquée dans la déterminisation. Nous la définissons maintenant pour étudier la relation entre $\epsilon$-clôture des $\mathcal{IOSTS}$ et $\epsilon$-clôture des IOLTS.

Soit $M = (Q, \Lambda, \rightarrow, Q_0)$ un IOLTS, avec $\Lambda = \Lambda_! \cup \Lambda_? \cup \Lambda_\tau$ l'$\epsilon$-clôture de $M$ est l'IOLTS $\epsilon(M) = (Q, \Lambda_! \cup \Lambda_?, \rightarrow_\epsilon, Q_{0_\epsilon})$ où $Q_{0_\epsilon}) = Q_0$ after $\varepsilon$ et pour tout état $q$ dans $Q_{0_\epsilon} \cup post_{\Lambda_! \cup \Lambda_?}(Q)$, $q \xrightarrow{a}_\epsilon q'$ ssi $q' \in q$ after $a$.

Cette définition de l'$\epsilon$-clôture produit un IOLTS sans actions internes, qui préserve les traces de $M$.

**Lemme 4.3.2** $Traces(\epsilon(M)) = Traces(M)$

Si maintenant nous montrons que la sémantique $[[\epsilon_{sy}(\mathcal{M})]]$ de l'$\epsilon$-clôture symbolique d'un IOSTS $\mathcal{M}$ est l'$\epsilon$-clôture $\epsilon([[\mathcal{M}]])$ de la sémantique de cet $\mathcal{IOSTS}$, alors l'$\epsilon$-clôture symbolique réalise bien l'opération voulue i.e. la suppression des actions internes avec préservation des traces.

C'est exactement ce que dit la propriété suivante :

**Proposition 4.3.6 (Propriétés de l'$\epsilon$-clôture)**

$$[[\epsilon_{sy}(\mathcal{M})]] = \epsilon([[\mathcal{M}]])$$

*où $\epsilon$ est l'$\epsilon$-clôture des IOLTS.*

**Preuve :** Les états de $[[\epsilon_{sy}(\mathcal{M})]]$ sont les mêmes que ceux de $[[\mathcal{M}]]$, donc de $\epsilon([[\mathcal{M}]])$.

Par ailleurs, d'après la discussion qui précède sur le tir des séquences de transitions de $(T_\tau)^*.(T \setminus T_\tau)$ dans $[[\mathcal{M}]]$, une action valuée $\langle a, p \rangle$ est tirable depuis un état de $[[\epsilon_{sy}(\mathcal{M})]]$ si et seulement si elle est tirable dans le même état dans $\epsilon([[\mathcal{M}]])$.

□

**Remarque :** pour simplifier, nous avons ici utilisé l'abstraction d'un $\mathcal{IOSTS}_{loc}$ sur ses localités. C'est d'ailleurs celle qui est utilisée dans STG. Une abstraction plus fine pourrait être utilisée, par exemple en expansant toute variable de domaine fini. L'effet d'une telle

abstraction et de son analyse est de découvrir la structure de contrôle masquée dans l'évolution des variables. Cette abstraction induit un dépliage de $\mathcal{M}$ où le contrôle découvert sur l'abstraction est explicité. On se ramène alors au cas de la définition avec un ensemble de localités raffiné. Ce raffinement d'abstraction peut être nécessaire si $abs_{loc}(\mathcal{M})$ contient toujours des boucles d'actions internes. Par ailleurs,dans le cas des $\mathcal{IOSTS}$ sans localité, ce type d'abstraction est nécessaire. Bien entendu, il est possible qu'aucune abstraction ne permette de supprimer les boucles d'actions internes, auquel cas on ne pourra pas calculer d'$\epsilon$-clôture.

De la proposition 4.3.6, on en déduit immédiatement que les exécutions de la sémantique de l'$epsilon$-clôture symbolique d'un $\mathcal{IOSTS}$ sont les exécution de l'$\epsilon$-clôture de la sémantique de cet $\mathcal{IOSTS}$. En remarquant que les états de $[[\epsilon_{sy}(\mathcal{M})]]$ qui satisfont la condition $Accept$ sont les états de $[[\mathcal{M}]]$ qui, à la fois satisfont la condition $Accept$, et sont des états de $[[\epsilon_{sy}(\mathcal{M})]]$ on a une propriété similaire de commutation pour les exécutions acceptées :

**Corollaire 4.3.4**

$$Run([[\epsilon_{sy}(\mathcal{M})]] = Run(\epsilon([[\mathcal{M}]]))$$
$$Run_{Accept}([[\epsilon_{sy}(\mathcal{M})]] = Run_{Accept}(\epsilon([[\mathcal{M}]]))$$

De ce corollaire et en utilisant les propriétés de préservation de traces l'$\epsilon$-clôture des IOLTS 4.3.2, on en déduit finalement les propriétés suivantes sur les traces, et traces acceptées de la sémantique de l'$\epsilon$-clôture symbolique :

**Corollaire 4.3.5**

$$Traces([[\epsilon_{sy}(\mathcal{M})]] = Traces(\epsilon([[\mathcal{M}]])) = Traces([[\mathcal{M}]])$$
$$Traces_{Accept}([[\epsilon_{sy}(\mathcal{M})]] = Traces_{Accept}(\epsilon([[\mathcal{M}]])) = Traces_{Accept}([[\mathcal{M}]])$$

En clair, ce corollaire nous dit que, moyennant les hypothèse sur l'absence de boucles, l'$\epsilon$-clôture symbolique $\epsilon_{sy}$ préserve les traces et traces acceptées des $\mathcal{IOSTS}$. C'est bien le résultat attendu de cette opération.

**Construction de sous-ensemble** Avec les hypotèses précédentes, on a construit un $\mathcal{IOSTS}$ $\epsilon_{sy}(\mathcal{M})$ sans action interne, de mêmes traces que $\mathcal{M}$. Il s'agit maintenant de réaliser le calcul de sous-ensemble sur les IOLTS par une opération sur les $\mathcal{IOSTS}$. Comme on l'a vu, une solution générale au problème passerait par l'introduction d'une variable sur $2^{Dom(V)}$. Cette solution implique, pour les opérations suivantes de sélection de tests, des analyses trop complexes. Nous n'avons pas encore exploré cette piste, mais elle serait certainement intéressante à suivre à l'avenir pour repousser les limites de ce qu'on sait faire.

Nous avons exploré une solution plus simple, qui consiste à restreindre la classe des $\mathcal{IOSTS}$ considérés. Nous allons nous limiter aux $\mathcal{IOSTS}_{loc}$ et nous baser sur l'abstraction sur les localités $abs(\mathcal{M})$. Le problème du calcul symbolique de sous-ensemble pour

des $\mathcal{IOSTS}$ vient du fait qu'un méta-état de l'IOLTS $det([[\mathcal{M}]])$ puisse contenir plusieurs états, avec des valeurs différentes pour la même variable. Or pour que l'opération de déterminisation symbolique soit interne aux modèles $\mathcal{IOSTS}_{loc}$, nous devons assurer que les affectations dans l'$\mathcal{IOSTS}$ déterminisé soient des fonctions, donc que dans un état, une variable n'aie qu'une valeur possible. L'idée que nous allons explorer est de limiter la construction symbolique de sous-ensemble aux localités. Ceci n'est bien sûr pas toujours posssible, et par conséquent, la construction ne s'arrêtera pas toujours.

Soit $\mathcal{M} = \langle L, D, \Theta, \Sigma_! \cup \Sigma_?, T\rangle$ un $\mathcal{IOSTS}_{loc}$ fermé et sans actions internes (comme c'est le cas pour $\epsilon(\mathcal{M})$).

Notons que, si pour chaque localité $l$ et pour chaque action $a$, les gardes des transitions issues de $l$ et portant $a$ sont deux à deux disjointes, i.e. $\forall t_i = \langle l, a, \pi, G_i, A_i, l_i\rangle$, $t_j = \langle l, a, \pi, G_j, A_j, l_j\rangle$, i.e $G_i \cap G_j = \emptyset$, alors pour tout état $\langle c, v, l\rangle$ de $[[\mathcal{M}]]$, les ensembles $\{p \in type(\pi) \mid \langle c, v, l, p\rangle \in G_i\}$ et $\{p \in type(\pi) \mid \langle c, v, l, p\rangle \in G_j\}$ sont disjoints ($c$, $v$ et $l$ sont fixés). L'IOLTS $[[\mathcal{M}]]$ est donc déterministe, le calcul de sous-ensemble est donc l'identité. On ne peut évidemment pas toujours décider si l'intersection de gardes est vide. Il faut rechercher des conditions suffisantes, syntaxiques par exemple. Un cas trivial est le suivant. Soit $abs(\mathcal{M}) = (L, \Sigma_! \cup \Sigma_?, \rightarrow_{abs}, L_0)$ son abstraction sur les localités. Si $abs(\mathcal{M})$ est déterministe, la condition précédente est respectée, donc $[[\mathcal{M}]]$ est déterministe (un argument plus direct est de s'appuyer sur le fait que $abs(\mathcal{M})$ est une abstraction de $[[\mathcal{M}]]$).

Ce cas n'est évidemment pas très satisfaisante puisqu'il restreindrait la génération de tests à des systèmes quasi-déterministes (déterministes après $\epsilon$-clotûre). On peut toutefois faire mieux et traiter certaines spécifications non-déterminsites, dans le cas où on peut résoudre ce non-déterminisme en un nombre borné de pas. Si $\mathcal{M}$ est non-déterministe (ou si on ne sait pas le décider) c'est qu'il existe dans un état deux transitions portant la même action dont l'intersection des gardes est non vide. L'idée est donc de transformer $\mathcal{M}$ en ajoutant une transition gardée par l'intersection des gardes, et menant dans un état composé des états buts des deux transitions. Le problème principal à résoudre est celui des affectations. En effet, les affectations des deux transitions étant *a priori* différentes et on ne sait pas laquelle choisir. L'idéé est de repousser cette affectation au moment où le non-déterminisme est résolu.

Cela amème à proposer une heuristique de calcul de sous-ensemble décrite par la procédure $subset_{sy}$ de la figure 4.4 et que nous commentons ici.

Pour chaque action $a \in \Sigma_! \cup \Sigma_?$, on introduit une nouvelle variable $m_a$ de type $sig(a)$. Cette variable servira à stocker la valeur des paramètres de communication de l'action $a$. En effet, comme on va être amené à repousser les affectations dans le futur, que celles-ci peuvent dépendent des paramètres, et que la portée des paramètre est limitée à une transition, leur valeur doit être stockée. On modifie donc toutes les transitions en ajoutant l'affectation $m_a := m_a$ de sorte que les affectations restent des fonctions sur le domaine augmenté.

La procédure $subset_{sy}$ consiste en un calcul de point fixe. A chaque itération, on examine chaque couple $(l, a)$ de localité $l$ et d'action $a$ et on examine l'ensemble $T_{l,a}$ est l'ensemble des transitions issues de $l$ et portant l'action $a$ et menant aux localités $L_{l,a}$. Si $T_{l,a}$ a plus d'une transition, il y a risque de non-déterminisme. On va transformer cet ensemble de transitions et leurs localités but. en commençant par oter $T_{l,a}$ de $T$. On suppose sans faire de restriction, que tous les paramètres de communication sont identiques à $\pi_a$. Le

remplacement va consister à regarder toutes les parties élémentaires de l'union des gardes de ces transitions. Les transitions étant re-numérotées de 1 à $size(T_{l,a})$, celà va consister à regarder toutes les parties de $\{1, size(T_{l,a})\}$. Chaque partie $P$ représente l'incertitude sur un sous-ensemble de transitions tirées. La condition $G_P$ est l'intersection des gardes $G_i$ des transitions $t_i$ tels que $i \in P$ avec les négations des gardes de $G_j$ des transitions $t_j$ telles que $j \notin P$. Si $G_P$ est non vide, celà signifie qu'il est possible qu'il existe un état $(c, l, v)$ et une valuation $p$ de $\pi_a$ qui permette de franchir toutes les transitions $t_i, i \in P$ et aucune transition $t_i, i \notin P$. Cela suggère une stratégie non détaillée dans l'algorithme qui consisterait à regarder les intersections des gardes en partant des parties les plus petites. On crée alors une nouvelle localité $l_P$ qui code le sous-ensemble des $l_i$ tels que $i \in P$, et on ajoute une transition de $l$ à $l_P$ d'action $a$, de garde $G_P$ et dont l'affectation consiste juste à stocker la valeur du paramètre de communication $\pi_a$. En effet, les affectations des transitions $l_i, i \in P$ étant *a priori* différentes, on ne sait pas produire une affectation unique sur cette transition. On repousse le problème de l'affectation sur les transitions suivantes issues des états $l_i$ dans l'espoir de résoudre le problème au coup suivant, ou plus loin. Pour chaque localité $l_i, i \in P$, on calcule donc l'ensemble de transitions $T_i^+$ des transitions issues de $l_i$. Pour chaque transition $t_{ij}$, on propage donc l'affectation de $t_i$ par composition avec la garde et l'affectation de $t_{ij}$, en prenant soin de remplacer les occurrences de $\pi_a$ dans l'affectation $A_i$ par la variable $m_a$ dans laquelle $\pi_a$ a été stockée.

Les transitions de l'ensemble $T_i^-$ non issues de $l$ mais arrivant dans les états $l_i, i \in P$ ne posent pas de problème. On a conservé les états $l_i$ auxquelles mènent toujours ces transitions, et les transitions de $T_i^+$ qui en sortent.

La condition d'acceptation $Accept^{\mathcal{M}}$ de $\mathcal{M}$ doit être un sous-ensemble de localités, éventuellement fonction des valeurs des constantes symboliques. En effet, $Accept^{\mathcal{M}}$ portait aussi sur les valeurs de variables, la propagation des affectations modifierait les traces acceptées. La condition d'acceptation $Accept^{\text{subset}}$ de $subset_{sy}(\mathcal{M})$ est étendue aux couples $(c, l_P)$ de $Dom(C) \times L'$ tels qu'il existe $l$ dans $l_P$ avec $(c, l) \in Accept^{\mathcal{M}}$.

La procédure $subset_{sy}$ peut ne pas terminer à cause des cycles. En effet, on peut créer du non-déterminisme en recopiant chaque transition entrant dans un état $l_i$ successeur de $l$ par $a$, sur tous les états $l_P$ pour lesquels $i \in P$. Si $\mathcal{M}$ est acyclique, en parcourant les localités de $l_0$ vers les feuilles, on propage le non-déterminisme seulement en avant et on finira par le résoudre. par contre si $\mathcal{M}$ contient des cycles il est possible que le point fixe ne termine pas.

**Proposition 4.3.7** *Si l'algorithme s'arrête,*

$$Traces([[subset_{sy}(\mathcal{M})]]) = Traces([[\mathcal{M}]]) = Traces(subset([[\mathcal{M}]]))$$

*où $subset$ est le calcul de sous-ensemble habituel sur les IOLTS sans action interne.*

*Pour les traces acceptées on a*

$$Traces_{Accept}([[subset_{sy}(\mathcal{M})]]) = Traces_{Accept}([[\mathcal{M}]]) = Traces_{Accept}(subset([[\mathcal{M}]]))$$

**Preuve :** Il suffit de remarquer que chaque itération de la boucle *for each* sur le couple $(l, a)$ ne change que les transitions provenant de $l$ et sortant de ses successeurs. Pour toute séquence de longueur inférieure à 2 ainsi définie entre deux localités $l_1$ et $l_2$. Pour tout état $q_1$ de localité $l_1$ et toute trace de longueur inférieure à 2 atteignant $q_2$ de localité $l_2$, on

*function* $subset_{sy}(\mathcal{M} : \mathcal{IOSTS}) : \mathcal{IOSTS}$
*input* $\mathcal{M} = \langle D, \Theta, L, l_0, \Sigma_! \cup \Sigma_?, T\rangle$
$Accept^{\mathcal{M}} = Dom(V) \times Acc^{\mathcal{M}}$ *with* $Acc^{\mathcal{M}} \subseteq Dom(C) \times L$ *its acceptance condition*
*var* $D', L', T', \Theta'$

$D' := D \cup \{m_a \mid a \in \Sigma_! \cup \Sigma_?\}$ *;*
$T := T$ *where for each* $t = \langle l, a, \pi_a, G_i, A_i, l_i\rangle$ $A_i$ *is augmented with* $\forall a, m_a := m_a$ *;*
$\Theta' := \Theta \wedge \forall a, m_a = null$ *;*
*repeat*
  $L' := L$ *;* $T' := T$ *; (* store* $L, T$ *in* $L', T'$ **)*
  *for each* $l \in L$*, for each* $a \in \Sigma_! \cup \Sigma_?$
    $T_{l,a} := \{t_i = \langle l, a, \pi_a, G_i, A_i, l_i\rangle \mid t_i \in T'\}$ *; (* set of* $a$ *labelled transitions starting in* $l$ **)*
    $L_{l,a} := \{l_i \mid l_i$ *is a goal of* $t_i \in T_{l,a}\}$ *;  (* i.e.* $post_a(l)$ *in* $abs(\mathcal{M})$ **)*
    *if* $size(T_{l,a}) > 1$ *then  (* possible non-determinism for* $a$ **)*
      $T := T \setminus T_{l,a}$ *; (* replace transitions in* $T_{l,a}$ *with ...*)*
      *(* transitions* $t_i$ *in* $T_{l,a}$ *are numbered with* $i \in [1, size(T_{l,a})]$ **)*
      *for each* $P \in 2^{\{1,\ldots,size(T_{l,a})\}}$  *(** $P$ *identifies a subset of* $T_{l,a}$*)*
        $G_P := \bigcap_{i \in P} G_i \cap \bigcap_{i \notin P} \overline{G_i}$ *;*
        *if* $G_P \neq \emptyset$ *then (* non-determinism *)*
          $L := L \cup \{l_P\}$ *; (* new location* $l_P$ *representing the union of* $l_i, i \in P$ **)*
          $T := T \cup \{\langle l, a, \pi_a, G_P, A_{m_a}, l_P\rangle\}$ *;*
          *(* new transition to* $l_P$ *with* $A_{m_a} : m_a := \pi_a$*,* $v := v$ *for other var. *)*
          *for each* $l_i$ *such that* $i \in P$
            ${T_i}^+ := \{t_{ij} = \langle l_i, b_j, \pi_b, G_{ij}, A_{ij}, l_{ij}\rangle \mid t_{ij} \in T'\}$ *;  (* Transitions starting in* $l_i$ **)*
            *for each* $t_{ij} = \langle l_i, b_j, \pi_b, G_{ij}, A_{ij}, l_{ij}\rangle \in {T_i}^+$  *(* propagate assignments*)*
              $T := T \cup \{\langle l_P, b, \pi_b, G_{ij} \circ A_i(m_a/\pi), A_{ij} \circ A_i(m/\pi), l_{ij}\rangle\}$ *;*
              *(* where* $A_i(m_a/\pi)$ *is* $A_i$ *with each occurrence of* $\pi$ *replaced by* $m_a$ **)*

*until* $L' = L$
$subset_{sy}(\mathcal{M}) = (D', \Theta', L, l_0, \Sigma_! \cup \Sigma_?, T\rangle$
$Accept_{subset} = Dom(V) \times Acc^{select}$ *with* $Acc^{select} = Acc^{\mathcal{M}} \cup \{(c, l_P) \mid \exists l \in l_P, (c, l) \in Accept^{\mathcal{M}}\}$

FIG. 4.4 – Heuristique de calcul de sous-ensemble

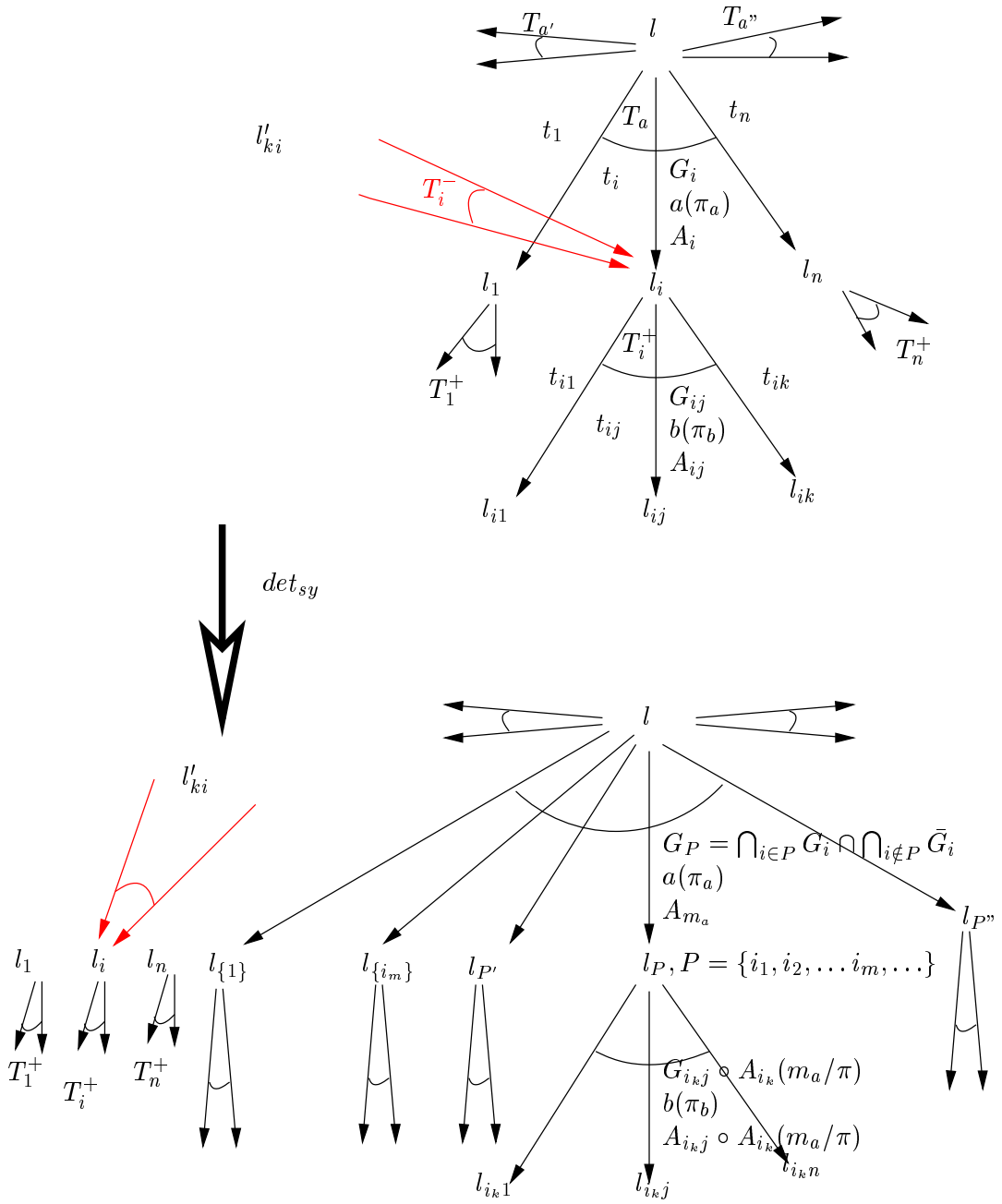



FIG. 4.5 – Calcul de sous-ensemble symbolique

obtient la même trace de longueur 2 dans la relation de transition intermédaire $T$ calculée. Il suffit pour cela d'exhiber les conditions de tir et les affectations résultantes sur 2 transitions consécutives. Chaque itération du point fixe conserve donc toutes les traces. Si l'algorithme termine, on obtient un $\mathcal{IOSTS}$ dont la sémantique IOLTS a les mêmes traces que les traces de l'IOLTS $[[\mathcal{M}]]$ sémantique de l'$\mathcal{IOSTS}$ $\mathcal{M}$ de départ.

Concernant les traces acceptées, si une trace $\sigma$ mène dans $[[\mathcal{M}]]$ dans un état $q$ de $Accept^{\text{M}}$, alors cette même trace mènera dans un état $q'$ de projection $(c, l)$ dans $Dom(C) \times L'$. Si $l \in L$, $(c, l) \in Acc^{\text{M}}$ et $q' \in Accept^{\text{subset}}$. Sinon, $l = l_P$ et $(c, l_P) \in Acc^{\text{subset}}$ par construction, et $q' \in Accept^{\text{subset}}$.
□

Appelons $det_{sy} = subset_{sy} \circ \epsilon_{sy}$ la déterminisation symbolique. On se propose maintenant de grouper les résultats de l'$\epsilon$-clôture $\epsilon_{sy}$ et du calcul de sous-ensemble et de les appliquer à $\mathcal{PS}^{\Delta} = \Delta_{sy}(\mathcal{PS})$.

**Corollaire 4.3.6** *Si $\mathcal{PS}^{\Delta}$ n'a pas de boucle d'action internes, sa condition d'acceptation ne porte pas sur les variables, et si le calcul de $subset_{sy}$ s'arrête,*

$$\begin{aligned} Traces([[det_{sy}(\mathcal{PS}^{\Delta})]]) &= Traces([[\mathcal{PS}^{\Delta}]]) = STraces([[\mathcal{S}]]) \\ &= STraces([[\mathcal{S}]]) \textit{ par le corollaire 4.3.3} \end{aligned}$$

*Les traces acceptées satisfont aussi*

$$Traces_{Accept}([[det_{sy}(\mathcal{PS}^{\Delta})]]) = Traces_{Accept}([[\mathcal{PS}^{\Delta}]])$$

### 4.3.4 Sélection

Soit $\mathcal{PS}^{\text{vis}} = det_{sy}(\epsilon_{sy}(\Delta_{sy}(\mathcal{S} \times_{sy} \mathcal{TP})))$ l'$\mathcal{IOSTS}$ obtenu à partir de la spécification $\mathcal{S}$ et de l'objectif $\mathcal{TP}$ par les opérations de produit symbolique, suspension symbolique, $\epsilon$-clôture symbolique et calcul de sous-ensemble. La sémantique de cet $\mathcal{IOSTS}$ est l'IOLTS $[[\mathcal{PS}^{\text{vis}}]] = (Q^{\text{vis}}, \Lambda, Q_0, \rightarrow_{\text{vis}})$. Ses traces sont exactement les traces suspendues de $[[\mathcal{S}]]$, parmi lesquelles les traces suspendues acceptées sont des projections d'exécutions de $[[\mathcal{S}]]$ acceptées par $[[\mathcal{TP}]]$.

Comme dans le cas des IOLTS, $\mathcal{PS}^{\text{vis}}$ a trop de traces puisque $Traces([[\mathcal{PS}^{\text{vis}}]]) = STraces([[\mathcal{S}]])$. Il faudrait donc extraire de $\mathcal{PS}^{\text{vis}}$ la partie utile, c'est à dire un $\mathcal{IOSTS}$ dont l'ensemble des traces soit exactement l'ensemble des préfixes des traces acceptées c'est à dire les traces de $Trim([[\mathcal{PS}^{\text{vis}}]])$. Ceci repose sur le calcul de $L2A^{\text{vis}} = reach(Q_0^{\text{vis}}) \cap coreach(Accept^{\text{vis}})$. Or un calcul exact de cet ensemble d'états est en général impossible, les problèmes d'accessibilité et de coaccessibilité étant indécidables pour les $\mathcal{IOSTS}$.

Par contre on peut surapproximer cet ensemble. En effet, soit $abs(\mathcal{PS}^{\text{vis}}) = (Q^{\text{abs}}, \Sigma_!^{\text{abs}} \cup \Sigma_?^{\text{abs}}, Q_0^{\text{abs}}, \rightarrow_{\text{abs}})$ muni de l'ensemble d'états $Accept^{\text{abs}}$, une abstraction quelconque de $[[\mathcal{PS}^{\text{vis}}]]$ calculée statiquement sur $\mathcal{PS}^{\text{vis}}$. Soient $(\alpha, \gamma)$ la paire de Galois correspondant à cette abstraction. Supposons qu'on sache calculer $L2A^{\text{abs}} = reach(Q_0^{\text{abs}}) \cap coreach(Accept^{\text{abs}})$. On a $L2A^{\text{vis}} \subseteq \gamma(L2A^{\text{abs}})$. On peut donc définir un /IOSTS/ $Trim_{sy}(\mathcal{PS}^{\text{vis}})$ en surapproximant l'ensemble $L2A^{\text{vis}}$ par l'ensemble $\gamma(L2A^{\text{abs}})$

**Définition 4.3.6 (Automate Trim symbolique)**
*Soit* $abs(\mathcal{PS}^{vis}) = (Q^{abs}, \Sigma^{abs}_! \cup \Sigma^{abs}_?, Q^{abs}_0, \rightarrow_{abs})$ *muni de l'ensemble d'états* $Accept^{abs}$ *une abstraction de* $[[\mathcal{PS}^{vis}]]$. *Notons* $L2A^{abs} = reach(Q^{abs}_0) \cap coreach(Accept^{abs})$.

$Trim_{sy}(\mathcal{PS}^{vis})$ *est l'*$\mathcal{IOSTS}$ $D, \Theta^{trim}, \Sigma, T^{trim}\rangle$ *avec*

- $\Theta^{trim} = \Theta^{vis} \cap \gamma(L2A^{abs})$,
- $Accept^{trim} = Accept^{vis} \cap \gamma(L2A^{abs})$,
- $T^{trim}$ *est défini comme suit : si* $t = \langle a, \pi, G, A\rangle \in T^{vis}$ *alors la transition* $t_{trim} = \langle a, \pi, G \cap \gamma(L2A^{abs}), A\rangle$ *est dans* $T^{trim}$ *sauf si :*
  - *on sait décider* $G \cap \gamma(L2A^{abs}) = \emptyset$.
    *Dans ce cas la garde est vide donc* $t_{trim}$ *peut être supprimée. En fait la transition* $t$ *ne peut être tirée que hors de* $\gamma(L2A^{abs})$ *donc hors de* $L2A^{vis} = reach(Q^{vis}_0) \cap coreach(Accept^{vis})) \subseteq \gamma(L2A^{abs})$.
  - *on sait décider* $A(G \cap \gamma(L2A^{abs})) \subseteq \gamma(\overline{L2A^{abs}})$.
    *Dans ce cas la transition* $t_{trim}$ *peut aussi être supprimée car si ele est tirée, elle mène forcément dans un état hors de* $\gamma(L2A^{abs})$ *donc hors de* $L2A^{vis}$.

**Proposition 4.3.8**

$$\begin{array}{rcl} Traces_{Accept}([[Trim_{sy}(\mathcal{PS}^{vis})]]) & = & Traces_{Accept}([[(\mathcal{PS}^{vis})]]) = Traces_{Accept}(\Delta([[\mathcal{PS}]])) \\ Traces([[Trim_{sy}(\mathcal{PS}^{vis})]]) & \subseteq & Traces([[(\mathcal{PS}^{vis})]]) = STraces([[\mathcal{S}]]) \end{array}$$

**Preuve :** La propriété sur les traces acceptées découle des propriétés de l'abstraction. Si une trace est acceptée dans $[[Trim_{sy}(\mathcal{PS}^{vis})]]$, elle l'est aussi dans $[[(\mathcal{PS}^{vis})]]$ puisqu'on a seulement restreint les gardes. Inversement si une trace $\sigma$ est acceptée dans $[[(\mathcal{PS}^{vis})]]$. Son image $\alpha(\sigma)$ est aussi acceptée dans $abs(\mathcal{PS}^{vis})$. Par définition de $reach$ et $coreach$, elle passe donc uniquement par des états de $L2A^{abs}$. La trace $\sigma$ dans $[[\mathcal{PS}^{vis}]]$ passe donc uniquement par des états de $\gamma(L2A^{abs})$ qui satisfont donc toujours les gardes restreintes. La trace $\sigma$ est donc aussi acceptée dans $[[Trim_{sy}(\mathcal{PS}^{vis})]]$.

De la restriction sur les gardes découle immédiatement la propriétés sur les traces.

□

Il nous reste à définir le graphe de test complet symbolique $\mathcal{CTG} = select_{sy}(\mathcal{PS}^{vis})$. Les différences avec $Trim_{sy}(\mathcal{PS}^{vis})$ sont les suivantes. Il faut inverser les entrées/sorties de sorte à se positionner du point de vue de l'environnement. Il faut définir les ensembles de verdicts. Le verdict **Fail** nous oblige à introduire une variable $verdict$ sur le domaine $\{$**Pass**, **Inconc**, **Fail**$\}$. Contrairement à $Trim_{sy}$ où on coupe toutes les transitions qui sortent de $\gamma(reach(Q^{abs}_0) \cap coreach(Accept^{abs}))$, dans $\mathcal{CTG}$ on ne coupe que celles étiquetées par des sorties de $\mathcal{S}$, les autres menant au vedict **Inconc**. Enfin, $\mathcal{CTG}$ est complété en entrée par des transitions menant à **Fail**.

**Définition 4.3.7 (Graphe de test complet symbolique)**
*Soit* $\mathcal{PS}^{vis} = \langle D^{vis}, \Theta^{vis}, \Sigma_! \cup \Sigma_?, T^{vis}\rangle$ *un* $\mathcal{IOSTS}$ *fermé, muni d'ensembles des conditions* $Accept^{vis}$ *(et* $Refuse^{vis}$*). L'*$\mathcal{IOSTS}$ $\mathcal{PS}^{vis}$ *est obtenu depuis* $\mathcal{S}$ *et* $\mathcal{TP}$ *par produit synchrone, suspension,* $\epsilon$*-clotûre et calcul de sous-ensemble symboliques. Soit* $abs(\mathcal{PS}^{vis}) = (Q^{abs}, \Sigma^{abs}_! \cup \Sigma^{abs}_?, Q^{abs}_0, \rightarrow_{abs})$ *une abstraction finie quelconque de* $[[\mathcal{PS}^{vis}]]$ *munie d'une condition* $Accept^{abs}$. *L'IOLTS* $abs(\mathcal{PS}^{vis})$ *est obtenue à partir de* $\mathcal{PS}^{vis}$ *et est définie par la paire de Galois* $(\alpha, \gamma)$. *On suppose que sur cette abstraction, on sait calculer* $reach(Q_0)$ *et* $coreach(Accept)$.

*Le graphe de test complet symbolique est un $\mathcal{IOSTS}$ qu'on désignera par $\mathcal{CTG} = \langle D^{ctg}, \Theta^{ctg}, \Sigma_!^{ctg} \cup \Sigma_?^{ctg}, T^{ctg} \rangle$ et défini comme suit :*

- $D^{ctg} = D^{vis}$,
- $\Sigma_!^{ctg} = \Sigma_?^{vis}$ *et* $\Sigma_?^{ctg} = \Sigma_!^{vis}$
- $\Theta^{ctg} = \Theta^{vis} \cap \gamma(coreach(Accept^{abs}))$
- **Pass** $= Accept^{vis} \cap \gamma(reach(Q_0^{abs}))$
- **Fail** $= proj^{-1}_{Dom(verdict)}(verdict = fail)$,
- **Inconc** $= \gamma(\overline{reach(Q_0^{abs}) \cap coreach(Accept^{abs})})$
- $T^{ctg}$ *est défini comme suit : si* $t = \langle a, \pi, G, A \rangle \in T^{vis}$ *alors la transition* $t_{ctg} = \langle a, \pi, G \cap \gamma(reach(Q_0^{abs}) \cap coreach(Accept^{abs})), A_v \rangle$ *est dans* $T^{ctg}$ *où*
  - *si* $A(G \cap \gamma(reach(Q_0^{abs}) \cap coreach(Accept^{abs}))) \subseteq$ *Pass*
    *alors* $A_v = verdict := Pass; A$
  - *si* $A(G \cap \gamma(reach(Q_0^{abs}) \cap coreach(Accept^{abs}))) \subseteq$ **Inconc**
    *alors* $A_v = verdict :=$ *Inconc*$; A$

  *et sauf si*
  - *on sait décider* $G \cap \gamma(reach(Q_0^{abs}) \cap coreach(Accept^{abs})) = \emptyset$, *alors* $t_{ctg}$ *peut être supprimée.*
  - *si on sait décider* $A(G \cap \gamma(reach(Q_0^{abs}) \cap coreach(Accept^{abs}))) \subseteq$ **Inconc** *et* $a \in \Sigma_?^{vis}$. *La transition* $t_{ctg}$ *peut être supprimée car c'est une transition portant une émission et menant dans un état* **Inconc**.

  *Enfin, pour toute sortie* $a \in \Sigma_!^{vis}$, *si on note* $UG_a$ *l'union des gardes de transitions portant* $a$, $T^{ctg}$ *est augmenté de la transition* $t_{fail} = \langle a, \pi, \overline{UG_a}, A_{fail} \rangle \in T^{vis}$ *avec* $A_{fail} = verdict :=$ *Fail*$; v := v$ *pour les autres variables.*

On se propose maintenant de caractériser la sémantique de la sélection symbolique par rapport à la sélection exacte, i.e. la sélection sur la sémantique telle que définie par l'opération *select* sur les IOLTS. Pour simplifier, on s'intéresse seulement aux traces menant aux verdicts **Pass**et aux traces menant aux états **Fail** :

**Proposition 4.3.9**

$$\begin{array}{rcl} Traces_{Pass}([[\mathcal{CTG}]]) & = & Traces_{Pass}([[select_{sy}(\mathcal{PS}^{vis})]]) \\ & = & Traces_{Accept}([[\mathcal{PS}^{vis}]]) \\ & = & Traces_{Pass}(select([[\mathcal{PS}^{vis}]])) \\ Traces_{Fail}([[\mathcal{CTG}]]) & \supseteq & Traces_{Fail}(select([[\mathcal{PS}^{vis}]])) \end{array}$$

**Preuve :** La première série d'égalités sur les traces acceptées découle immédiatement de la propriétés sur les traces acceptées de $[[Trim_{sy}(\mathcal{PS}^{vis})]]$ établie en proposition 4.3.8.

La deuxième signifie qu'on peut aller dans **Fail** au moins aussi souvent que dans le cas énuméré. On peut y aller plus souvent, car la surapproximation a conservé des traces qui ne sont pas nécessairement des préfixes de traces acceptées. Attention, ceci ne signifie pas la possibilité de biais des tests, comme il sera démontré plus loin (voir proposition 4.3.10). Une trace $\sigma$ de $Traces_{\text{Fail}}(select([[\mathcal{PS}^{\text{vis}}]]))$ est une trace qui n'est pas dans $Traces([[\mathcal{PS}^{\text{vis}}]])$ et qui peut être décomposée en $\sigma'.a(p)$ où $a(p)$ est une action de sortie valuée de $[[\mathcal{S}]]$ et $\sigma'$ est un préfixe d'une trace acceptée de $[[\mathcal{PS}^{\text{vis}}]]$. $\sigma'$ est donc aussi un

préfixe d'une trace menant à **Pass** dans $[[select_{sy}(\mathcal{PS}^{\text{vis}})]]$. Soit $(c, v)$ l'état de $[[\mathcal{PS}^{\text{vis}}]]$ atteint par $\sigma'$. Si $\sigma'.a(p)$ n'est pas une trace de $[[\mathcal{PS}^{\text{vis}}]]$, $(c, v, p)$ ne satisfait aucune garde de transition de l'action $a$, donc satisfait l'union des négations de ces gardes. Donc $a(p)$ mène dans l'état **Fail** dans $[[select_{sy}(\mathcal{PS}^{\text{vis}})]]$. Donc $\sigma \in Traces_{\text{Fail}}([[select_{sy}(\mathcal{PS}^{\text{vis}})]])$.
□

Cette proposition montre bien les caractéristiques des tests symboliques générés, par rapport aux tests énumérés qu'on pourrait vouloir générer. Ils montrent que, bien que l'-analyse soient basée sur une surapproximation des états accessibles et coaccessibles, les verdicts *Pass* sont produits exactement sur les mêmes traces que sur la solution exacte représentée par $select([[\mathcal{PS}^{\text{vis}}]])$. Par contre, à cause de l'approximation, il est possible qu'on ait conservé dans $[[select_{sy}(\mathcal{PS}^{\text{vis}})]]$ des traces qui ne sont pas préfixes de traces acceptées. Celles-ci ne pourraient être supprimées que par une analyse exacte. En conséquence, on peut produire des verdicts *Fail* après ces traces. Ca ne veut pas dire qu'on perde la correction, comme le montre la proposition 4.3.10. Simplement, qu'on détecte potentiellement plus de non-conformités existantes, mais sur des comportements qui n'auraient pas dû être sélectionnés, car on n'a pas été assez précis dans l'analyse. Plus l'analyse sera fine, plus on évitera ceci.

**Exemple :** Le graphe de test complet de la figure 4.6 est obtenu grâce à l'abstraction sur les localités (les transition vers **Fail** ne sont pas indiquées).

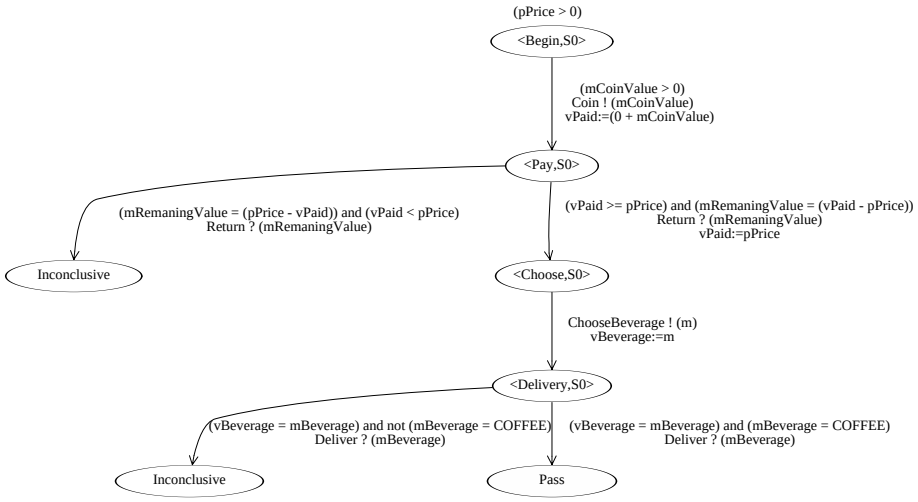



FIG. 4.6 – Graphe de tests complet pour l'abstraction sur les localités

Le graphe de test complet de la figure 4.7 est obtenu grâce à une abstraction plus fine calculée par l'outil NBAC [48](http ://www.irisa.fr/prive/bjeannet/nbac/nbac.html). On a pu ici éliminer les deux localités **Inconc**. La première a été éliminée en restreignant la garde de la première transition (on fournit une pièce ayant une valeur suffisante), de sorte à permettre l'accessibilité à $Accept$. La deuxième utilise le même principe, en forçant à

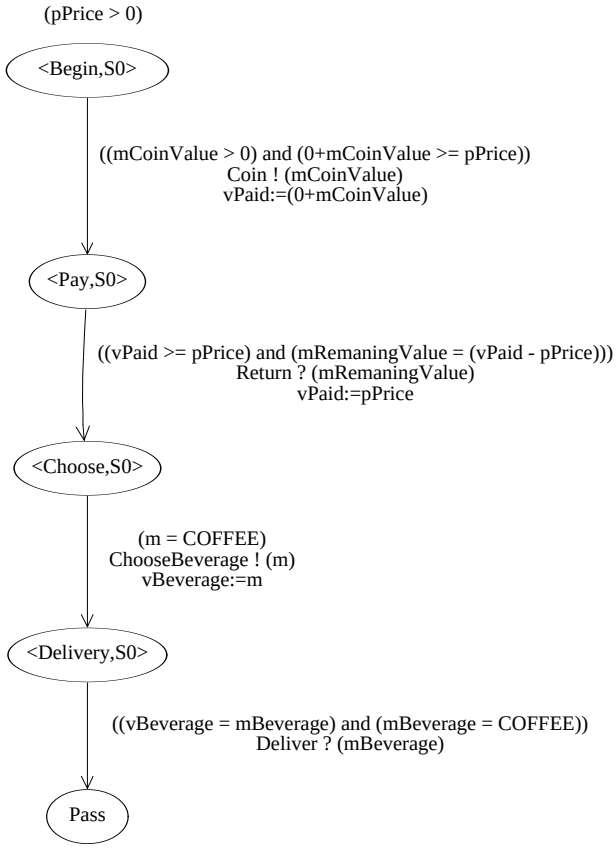



FIG. 4.7 – Graphe de tests complet calculé par NBAC

choisir du café pour obtenir du café.

La proposition suivante à la quelle on souhaitait arriver, démontre que la construction de tests symboliques satisfait les propriétés attendues, avec un petit affaiblissement technique pourl'exhaustivité.

**Proposition 4.3.10 (Non-biais, exhaustivité)** *Soit $\mathcal{S}$ un $\mathcal{IOST}\mathcal{S}$sans boucles d'actions internes, $\mathcal{TS}$ l'ensemble des graphes de tests que peut produire la génération de tests symbolique. $\mathcal{TS}$ est non-baisé et quasi-exhaustif.*

**Preuve :** D'abord notons que toutes les propriétés établies sur les exécutionss, langages et traces restent valides si on remplace un /IOSTS/ $\mathcal{M}$ par une instance $\mathcal{M}(c)$.

**Non-biais** Pour démontrer le non-biais, il suffit de montrer la condition suffisante de non-biais de la proposition 3.1.8 pour chaque instance $\mathcal{CTG}(c)$ de chaque graphe de test $\mathcal{CTG}$, i.e. $Traces_{\text{Fail}}([[\mathcal{CTG}(c)]]) \subseteq Straces([[\mathcal{S}(c)]]).\Lambda_! \cap \overline{Straces([[\mathcal{S}(c)]])}$

La construction des verdicts *Fail* assure qu'on ne peut atteindre **Fail** que sur une trace $\sigma$ de $[[\mathcal{PS}^{\text{vis}}(c)]]$ menant dans un état $(c, v)$ et prolongée par une sortie valuée $a(p)$ telle que

$(c, v, p)$ ne satisfait aucune garde de transition portant $a$. Donc $\sigma.a(p)$ n'est pas une trace de $[[\mathcal{PS}^{\text{vis}}(c)]]$. Par les propriétés des suites de transformation, les traces de $[[\mathcal{PS}^{\text{vis}}(c)]]$ sont des traces suspendues de $[[\mathcal{S}(c)]]$.

**Quasi-exhaustivité** La quasi-exhaustivité signifie l'exhaustivité, sauf qu'on ne peut pas capturer les non-conformités après une trace suspendue de $[[\mathcal{S}]]$ terminant dans un état qui ne peut produire qu'un blocage $\delta$. Dans un cas similaire pour les IOLTS, on avait déplié le graphe de test en ces états afin de vérifier la production d'une sortie $\delta$, car les objectifs ne permettent pas les actions étiquetées par $\delta$. On pourrait faire de même ici, sauf qu'on ne sait pas décider quels sont les états de blocage. On devrait donc le faire systématiquement, ce qui allourdirait la description de $\mathcal{CTG}$.

Pour montrer la quasi-exhaustivité il faut montrer que pour toute spécification $\mathcal{S}$, et pour toute instanciation $c$ de ses constantes symboliques, pour toute IUT $I$ modélisée par un IOLTS et compatible avec $\mathcal{S}(c)$, si $\neg I$ **ioco**$\mathcal{S}(c)$ alors il existe un objectif de test symbolique $\mathcal{TP}$ tel que l'instance $\mathcal{CTG}(c)$ du graphe de test $\mathcal{CTG}$ obtenu à partir de $\mathcal{S}$ et $\mathcal{TP}$ peut rejeter $I$, sauf si la non-conformité est observée après un état de blocage de $[[\mathcal{S}]]$.

Le principe de la preuve est identique au cas des IOLTS. Si $\neg I$ **ioco**$\mathcal{S}(c)$, il existe une trace $\sigma.a$ dans $STraces(I)$ telle que $\sigma \in Straces([[\mathcal{S}(c)]])$ et $\sigma.a \notin Straces([[\mathcal{S}(c)]])$. $\sigma$ peut se prolonger dans $[[\mathcal{S}]]$ en $\sigma.y$ (par hypothèse). On construit alors l'objectif de test comme l'automate qui reconnait toutes les séquences de trace $\sigma.y$ (en bouclant sur les actions internes). On peut alors montrer que le graphe de test produit peut rejeter $I$ sur la trace $\sigma.a$.
□

La suite des opérations symboliques et les propriétés principales de celles-ci sont résumées dans la figure 4.8. La gauche de la figure représente les opérations symboliques et à droite, les opérations correspondantes sur les IOLTS. Pour chaque opération, on a montré une propriété de commutation du diagramme permettant d'obtenir l'égalité des IOLTS, de leurs exécutions ou de leurs traces.

## 4.4 Conclusions sur le test symbolique

Dans ce chapitre, nous avons expliqué les principes de la génération de tests symboliques en ayant en tête plusieurs objectifs :

**Utiliser des modèles assez généraux :** le but de la généralisation est double. D'une part il permet d'aborder une classe assez générale de langages de spécification réels, comme SDL, Lotos, IF, mais aussi des langages synchrones comme Lustre. D'autre part il est inutile de se mettre trop tôt des limites. Le choix d'un modèle général permet de se rendre compte plus précisément des restrictions nécessaires. On peut ainsi mieux entrevoir les difficultés à résoudre et les prolongements possibles de notre approche, en introduisant des hypothèses sur les modèles uniquement quand celles-ci sont nécessaires. En particulier, au lieu de nous restreindre aux $\mathcal{IOSTS}$ avec localités, nous avons défini des $\mathcal{IOSTS}$ sans localités. Les $\mathcal{IOSTS}$ sans localités permettent cependant de simuler les $\mathcal{IOSTS}$ avec localités, par l'utilisation d'une variable spécifique variant sur l'ensemble des localités. Mais ce modèle avec localité est un cas particulier où une structure de contrôle est explicite. Or la strcuture de

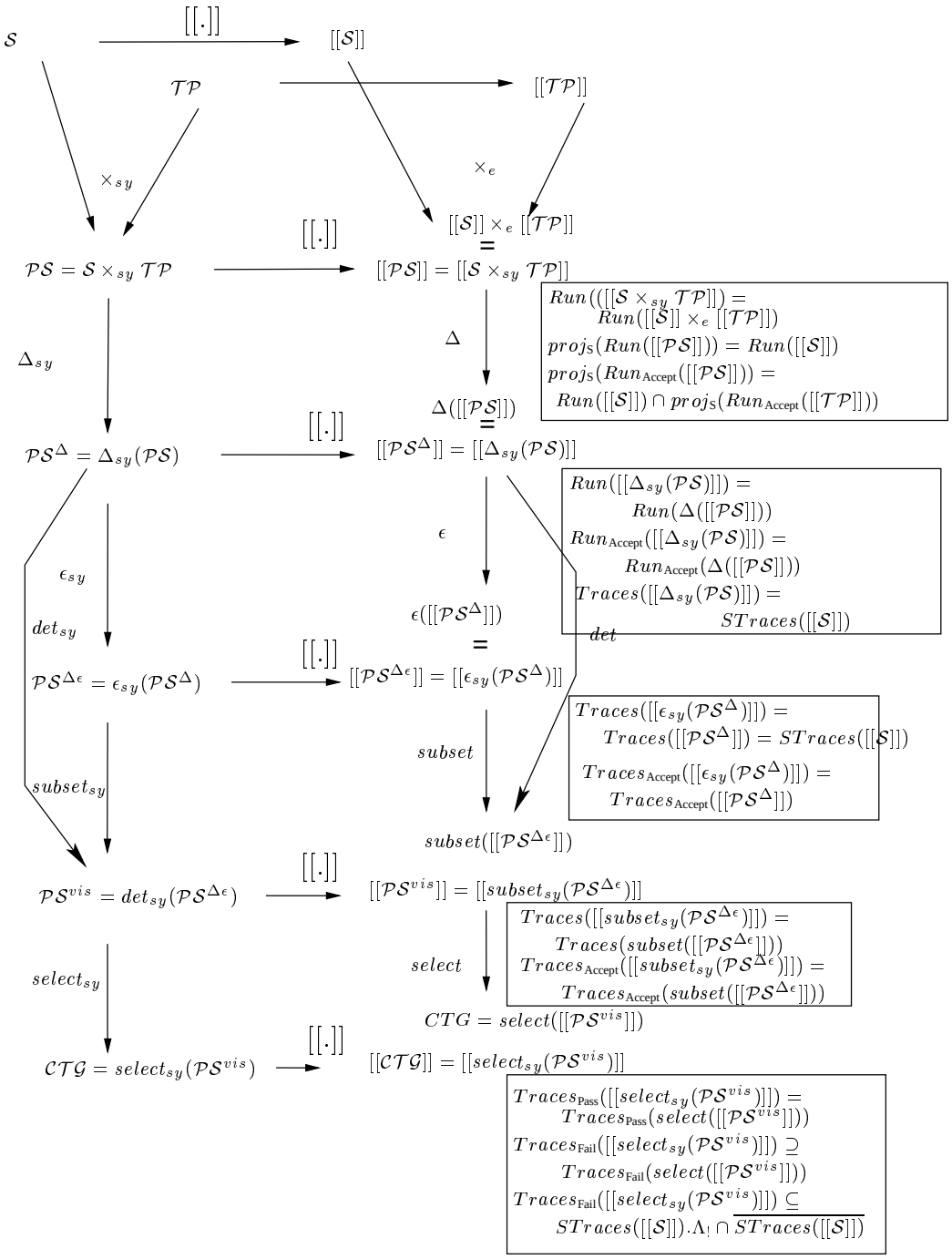



FIG. 4.8 – Schéma général de la génération de tests symbolique

contrôle est parfois implicite dans l'évolution des variables. Un moyen d'expliciter cette structure est l'utilisation de l'interprétation abstraite.

**Distinguer de façon claire les aspects syntaxiques et sémantiques :** les modèles manipulés dans la génération de tests, les $\mathcal{IOSTS}$, sont définis par une syntaxe. Par contre la relation de conformité, les propriétés des cas de tests sont définis sur leur sémantique IOLTS. Nous avons cherché à définir les transformations syntaxiques des modèles $\mathcal{IOSTS}$ qui permettent d'obtenir les propriétés attendues au niveau sémantique.

**S'appuyer sur les résultats déjà acquis pour les IOLTS :** dans le chapitre 3, nous avons défini la génération de tests par des transformations sur les IOLTS et établi des propriétés sur ces transformations. Pour la génération de tests symboliques, nous cherchons à établir des propriétés sur la sémantique IOLTS des $\mathcal{IOSTS}$ calculés. Plutôt que d'établir ces propriétés directement, ce qui est assez pénible, nous nous sommes appuyés au maximum sur des propriétés de commutation de diagramme qui permettent de relier les transformations syntaxiques sur les $\mathcal{IOSTS}$ avec les transformations sémantiques du chapitre 3 et de réutiliser ainsi les propriétés de ces transformations sémantiques pour faciliter les preuves.

**S'appuyer sur des techniques d'interprétation abstraite pour l'analyse :** l'interprétation abstraite est un cadre très général pour l'analyse sémantique des programmes. C'est un outil puisssant pour calculer statiquement des propriétés, en particulier des propriétés d'accessibilité (ou de co-accessibilité) qui sont au cœur de la génération de tests. Toute la difficulté dans l'emploi de ces techniques réside dans le choix des bonnes abstractions. Nous ne sommes pas entré dans le détail des analyses possibles, ni de celles utilisées par l'outil NBAC connecté à l'outil STG, car il aurait fallu pour cela choisir un modèle plus détaillé que celui des $\mathcal{IOSTS}$. Nous avons considéré l'interprétation abstraite comme une boîte noire et seulement utilisé les propriété des abstractions pour établir les propriétés des cas de tests générés en utilisant ces abstractions. Ce choix délibéré laisse la possibilité d'utiliser toute abstraction utile dans l'implémentation effective de la génération de tests et ainsi de profiter de toute avancée dans ces techniques.

Cet exposé s'est bien sûr basé sur nos publications sur le sujet de la génération de tests symboliques, mais un tel effort de formalisation n'avait jusqu'ici pas été mené jusqu'à ce point. Il a nécessité de remettre à plat toutes nos connaissances et interrogations sur la génération de tests symboliques. Il nous a permis d'éclaircir certains points que nous ne maîtrisions pas encore suffisament. Cette partie du document constitue donc un travail original, non encore publié. J'espère qu'il servira de base à nos travaux futurs sur le test symbolique.

### 4.4.1 Travaux connexes

Il existe d'autres travaux sur la génération de tests voisins des techniques symboliques présentées dans ce chapitre. Nous passons en revue quelques unes de ces techniques en essayant de comparer les approches.

### TVéda

TVéda [68] est un outil de génération de tests pour des spécifications Estelle et SDL en utilisant un format intermédiaire d'automates communicants étendus [67, 15]. Plusieurs approches différentes ont été abordées dans TVéda qui ont donné lieu à plusieurs versions de l'outil et plusieurs moteurs de génération. Dans la version V3, une possibilité offerte était basée sur des techniques symboliques. Les hypothèses sur le modèle sont fortes pour appliquer ces techniques. En particulier, les gardes ne peuvent concerner qu'une variable à la fois (e.g. $x < y$ n'est pas autorisé), et les affectations doivent etre réversibles (e.g. $x := y + z$ est interdit) La génération de test consiste alors à recherche un chemin exécutable depuis un ensemble d'état initial vers un ensemble d'état but. Les ensembles d'états initiaux et buts doivent être décrits par un produit de sous-domaines des variables. Le calcul consiste à calculer un chemin symbolique en avant de taille bornée, puis à propager les contraintes en arrière, et enfin à instancier arbitrairement les valeurs des variables dans les domaines calculés. Ce travail, bien qu'ayant de sérieuses limites, était assez précurseur. Il est d'ailleurs intéressant de noter que l'article [15] préconise en conclusion l'utilisation combinée de techniques de résolution de contrainte et d'inteprétation absatraite. . ..

### Test symbolique pour Lotos

Une approche voisine de STG est étudiée dans [28] pour Lotos et déjà évoquée au premier chapitre (section 2.3.3). Elle combine les approches de génération de test pour des spécification algébriques et la simulation symbolique de LOTOS (outil SMILE) étudiee par Eertink dans sa thèse [23], également utilisable pour la génération de tests. Eertink propose un algorithme de génération de test basé sur la simulation symbolique pour Lotos. Pour un processus Lotos, l'algorithme calcule les transitions (symboliques) possibles initialement et applique récursivement l'algorithme sur le résidu du processus après l'exécution de l'action. Le calcul du résidu implique la propagation des contraintes liées à la satisfaction des gardes des transitions tirées. La technique repose donc sur la propagation et la résolution de contrainte sur des chemins. Gaudel et James proposent de combiner cette technique avec les techniques sur les types de données abstraits et les hypothèses de sélection de tests, afin de traiter les aspects du contrôle en même temps que les aspects données. Comme souligné par les auteurs, la génération de tests nécessite non seulement de la simulation symbolique (en avant) mais aussi de la propagation en arrière des contraintes. Ce travail a été poursuivi dans l'article [56] pour la relation de conformité **ioco**.

À notre connaissance, ces propositions n'ont pas encore été implémentées. D'autres part les descriptions restent à un niveau assez abstrait qui ne permet pas de se faire une idée exacte de leur faisabilité. L'idée de combiner les techniques de simulation symbolique de résolution de contraintes et de dépliage sont bien sûr intéressantes et prometteuses, mais il reste des problèmes à traiter. Notament, contraiement aux systèmes déterministe pour lesquels la propagation de contrainte est adaptée, pour des systèmes non-déterministes (les entrées ne définissent pas les sorties de manière unique), la propagation arrière des contraintes ne suffit pas à assurer qu'on atteindra un état donné.

Une différence notable entre ce travail et le nôtre est que les tests qui seraient produits seraient seulement des séquences, et ne considèrent pas vraiment la non contrôlabilité de l'IUT. D'autre part, l'aspect observation partielle ne semble pas être traité, ce qui, on l'a

vu, pose quelques problèmes lors de la déterminisation. Il est donc supposé que toutes les transitions de la spécification comportent des événements visibles.

**GaTeL**

GaTeL est un outil de génération de tests pour des spécifications dans le langage synchrone Lustre. Certains de ses principes sont issus de l'outil Loft, basé sur des spécifications algébriques. Lustre est un langage synchrone déclaratif et flôt de données. Une spécifiaction Lustre est un ensemble d'équations de cycle, qui détermine la valeur d'une variable à un instant en fonction des valeurs de variables à l'instant précédent et des valeurs des variables d'entrées. Un cycle définit donc une relation de transitions. Les spécifications sont déterministes, la conformité est donc naturellent l'égalité des sorties pour les mêmes séquences de vecteur d'entrée. La génération de tests prend en entrée un système Lustre (avec d'eventuelles contraintes d'environnement) et un objectif de test. L'objectif de test décrit une propriété d'accessibilité par un noeud Lustre. Il s'agit donc de rechercher une séquence permettant d'atteindre l'objectif. A noter que la longueur des séquences est bornée par un entier. La recherche de cette séquence est fondée sur un parcours d'un produit entre spécification et objectif. Le parcours est effectué en arrière, dans chaque cycle depuis les sorties vers les entrées, et depuis l'état but à la recherche de l'état initial. La génération de tests est basée sur la propagation et la résolution de contraintes (outil ECLiPSe), reposant sur des heuristiques d'éveil de contraintes et d'instanciation des variables (*labelling*). Si l'état initial est atteint, le chemin calculé n'instancie pas nécessairement toutes les entrées. Une instanciation complète des entrées du chemin est alors effectué. GaTeL se base également sur un dépliage (interactif) des opérateurs Lustre. L'idée est de partitionner les domaines de valeur des entrées en fonction de la satisfaction des conditions. Des hypothèses d'uniformité sont ensuite faites sur les sous-domaines ainsi identifiés afin de choisir une entrée dans chaque sous-domaine le long des séquences de test.

**Agatha**

Agatha est un outil de génération de tests développé au CEA [57]. A l'origine dédié à des spécifications Estelle, puis SDL, il permet aujourd'hui la génération de tests pour UML (diagrammes de classe et d'états) par un codage dans un modèle intermédiaire, les EIOLTS (Extended Input Output Labeled Transition Systems). Les EIOLTS sont assez proches du langage Estelle. Ils sont basés sur des localités. Les transitions entre localités sont étiquetées par une entrée, une garde sur les variables, constantes et paramètres de l'entrée, des affectations des variables et une sortie. Une spécification peut décrire un ensemble de processus communicant par rendez-vous. La génération de tests est basée sur une simulation symbolique de l'espace d'états, avec pour objectif de couvrir chaque chemin symbolique. La simulation symbolique consiste à calculer un graphe (les articles parlent d'arbre) dont les nœuds sont caractérisés par les contraintes sur les chemins qui y mènent. Les contraintes sont simplifiées en utilisant de la récriture (outil Brute). La détection d'inclusion de contrainte, utilisée pour couper l'exploration, est basée sur l'outil Omega. Enfin, une fois le graphe symbolique construit, pour chaque chemin, Agatha recherche une instanciation possible par résolution de contrainte (outils Omega et COn'Flex). Agatha ne traite pas vraiment le non-déterminisme, mais s'il détecte du non-déterminisme observable dans les

réponses possibles de la spécification le long d'un chemin, il produit un verdict **Inconc** sur l'alternative.

**BZ-Testing tool**

L'outil BZ-TT (BZ-Testing Tool) est un outil de génération de tests de conformité à partir de spécification dans les notations B ou Z. Sa version commerciale LTG (Leirios Test Generator) est basée sur les mêmes principes que BZ-TT et admet trois langages de spécification : les Statecharts, UML (diagrammes de classes et d'états, contraintes OCL), et la notation B. Les spécifications sont supposées déterministes, les test générés sont donc des séquences. Le principe de la génération de tests est de calculer des tests aux bornes des domaines de valeurs des variables par l'utilisation de la résolution de contraintes. Le solveur de contrainte utilisé est CLP-BZ qui est implémenté au-dessus de SICStus Prolog. Il prend en entrée un format intermédiaire décrivant un seul processus sous la forme de prédicats pre/post sur des variables de domaines finis. La génération de test est basée sur une simulation symbolique à profondeur bornée, à la recherche d'un état but. La simulation symbolique est une exploration de l'espace d'état depuis l'état initial avec un algorithme *best-first search*, où les ensembles d'états sont représentés par des contraintes. Une fois l'état but atteint, si celui-ci est accessible pour la borne donnée, toutes les opérations dans cet état sont testées avec des valeurs satisfiant ou non les préconditions. Les valeurs ne satisfiant pas les pré-conditions peuvent être vues comme des tests de robustesse.

**Analyse**

D'abord, il faut noter que beaucoup de techniques sont basées sur la résolution de contrainte. Cette technique est aussi beaucoup utilisée pour le test boîte blanche. En particulier Gotlieb et al [33] l'utilise pour générer des tests à partir de code C, Meudec [60] pour du code ADA. L'idée est de trouver un chemin dans le graphe de flot de contrôle menant dans un état donné. Une particularité commune de toutes ces techniques est de se restreindre à des programmes ou spécifications déterministes (au sens déterminisme observable). Ce qui signifie que les sorties sont complètement déterminées par les entrées. Autrement dit le testeur a un contrôle complet sur le système. La génération de tests consiste alors à rechercher des séquences d'entrées qui déterminent les sorties attendues. Les contraintes sont des contraintes sur un seul chemin à la fois. La longueur des chemins est bornée, ce qui permet de déplier les boucles un nombre fini de fois.

Notre technique est assez différente sur ce point, car elle n'est pas limitée à des spécifications déterministes. Les contraintes que nous manipulons par interprétation abstraite (les ensembles d'états) sont des contraintes globales sur tous les passés et futurs possibles. Ce qui nous permet de calculer des tests qui ne contrôlent pas complèteement l'implémentation, mais qui sont réactifs aux actions de l'implémentation. Ce sont des programmes ayant des choix en fonction des réponses possibles de l'implémentation.

Une autre différence notable dans le principe est relatif au *labelling*. Le *labelling* est l'instanciation des valeurs de certaines variables pour permettre de réveiller des contraintes. Il consiste donc à déplier le graphe d'état (ou le chemin) en fonction de la valeur de ces variables. Ceci peut être vu comme une abstraction assez brutale où la structure de contrôle est raffinée par les valeurs possibles de ces variables. Les techniques d'interprétation

abstraites sont plus fines, au sens où la structure de contrôle est raffinée seulement si les comportements passés et futurs divergent.

À notre connaissance, aucun outil de génération de tests n'utilise l'interprétation abstraite pour la génération de tests. Cependant, ces techniques sont utilisées pour des problèmes proches. Par exemple, l'outil de Polyspace technologie (voir http ://www.polyspace.com/), utilise l'interprétation abstraite pour découvrir d'éventuels erreurs d'exécutions à partir de code ADA, C ou C++. Le principe est de calculer une abstraction et de résoudre le problème sur l'abstraction. Si aucun erreur n'existe sur l'abstraction, on est assuré qu'il n'y a pas d'erreur dans le programme. S'il existe une erreur dans l'abstraction, il y a peut-être une erreur dans le programme.

# Chapitre 5

# Conclusion générale et perspectives

Dans le chapitre 3, nous avons décrit les principes de la génération de tests de conformité pour des systèmes réactifs, basée sur un modèle énuméré, les IOLTS. Nous avons décrit une théorie du test bien fondée pour ces modèles et en avons expliqué les aspects principaux. Nous avons ensuite présenté la génération de tests en décrivant les briques de bases en termes d'opérations sur ces modèles IOLTS. Nous avons pu prouver des propriétés de base sur chacune des briques algorithmiques de base, et finalement en inférer des propriétés sur les tests générés. Ces propriétés permettent d'assurer que les tests produits ont un sens précis, ce qui nous semble essentiel pour améliorer la confiance que l'on peut apporter au test. Au delà de ces propriétés, nous avons expliqué l'implémentation efficace de ces algorithmes, basés sur des algorithmes de graphes, et leur enchaînement à la volée.

La limite principale de cette technique de génération de test est l'aspect énuméré. En effet, même si la génération à la volée est souvent efficace, elle peut se heurter à une explosion combinatoire, en particulier par la nécessité d'énumérer les valeurs d'entrée des messages dans leurs domaines. Pire, l'énumération rend impossible le traitement de spécifications où les domaines de valeurs sont infinis, ou lorsque les spécifications ont des constantes non instanciées.

Ce constat nous a mené depuis quelques années, à l'étude de techniques symboliques, élaborées dans le chapitre 4. Nous avons présenté un modèle symbolique, les $\mathcal{IOSTS}$, que nous avons souhaité suffisament général, d'une part pour permettre de modéliser une grande classe de langages de spécification, d'autre part pour ne pas s'imposer d'emblée des restrictions inutiles. La sémantique de ces $\mathcal{IOSTS}$ s'exprime en termes d'IOLTS étendus où on peut distinguer les valeurs des variables constituant les états. La théorie du test reste cependant la même, puisqu'elle s'interprète, comem dans le cas énuméré sur la sémantique des traces et blocages. La génération de tests symbolique consiste à effectuer des opérations au niveau symbolique, c'est à dire sans énumérer les comportements. Afin de démontrer que les tests symboliques produits (leur sémantique) ont bien les propriétés attendues, pour chaque opération sur un $\mathcal{IOSTS}$(ou des $\mathcal{IOSTS}$), nous avons montré une propriété de commutation de l'opération et de la sémantique. Plus précisément nous avons

montré que la sémantique du résultat correspondait au résultat de la transformation correpondante sur la sémantique IOLTS de l'$\mathcal{IOSTS}$ (ou des $\mathcal{IOSTS}$) de départ. Nous avons ainsi pu réutiliser les propriétés des transformations sur les IOLTS pour prouver des propriétés sur les transformations des $\mathcal{IOSTS}$. Dans la dernière opération sur les $\mathcal{IOSTS}$, la sélection, le calcul exact n'est pas possible. Ceci correspond au fait que l'accessibilité est indécidable, ou encore que l'élimination des chemins infaisables est indécidable. Nous utilisons donc un calcul approché par des techniques d'interprétation abstraite qui permettent d'éliminer certains de ces chemins infaisables. La conséquence de cette approximation est que la sélection est moins précise, mais les tests conservent néanmoins leurs bonnes propriétés. Le seul risque est de sortir des chemins que l'ont aurait voulu sélectioner si on avait pu faire un calcul exact. Nous avons réalisé une implémentation de cette technique de génération de tests, sur un modèle moins général que celui présenté ici. Bien que nous n'ayons pas encore traité beaucoup d'études de cas, nous avons pu traiter de manière très satisfaisante une étude de cas de taille conséquente, le CEPS (porte monnaie électronique). Les résultats sont donc tout à fait prometteurs et nous encouragent à poursuivre dans cette voie.

Nous avons fini ce chapitre par une comparaison avec d'autres techniques symboliques existantes. Ces autres techniques sont toutes basées sur des techniques de propagation et résolution de contraintes. Elles ne traitent pas du tout ou très peu les problèmes de non-déterminisme et d'observation partielle. Notre technique ne traite pas tous les problèmes, et en particulier nécessite également l'utilisation de techniques de résolution de contrainte au moment de l'instanciation des tests, que nous retardons à l'exécution. Mais l'utilisation de techniques d'interprétation abstraite et le fait de traiter l'observation partielle et le non-déterminisme sont tout à fait originaux.

Nous dressons maintenant quelques perspectives de travaux futurs en prolongement des travaux présentés ici.

**Extensions des propriétés des objectifs :** Les objectifs de test utilisés dans TGV et STG décrivent des propriétés d'accessibilité. Si l'on considère un objectif de test externe $TP$ (i.e. portant sur les traces suspendues), la génération de tests pour un tel objectif consiste essentiellement en la vérification de cette propriété sur le déterminisé de l'automate de suspension de la spécification $det(\Delta(S))$ (calcul de l'intersection des traces par le produit $TP \times det(\Delta(S))$), et le calcul des témoins de cette propriétés (calcul du graphe de test complet basé sur l'automate *Trim*).

On peut imaginer des objectifs de tests pour des propriétés autres que des propriétés d'accessibilité. En particulier, récemment, nous avons étudié le problème de la génération de tests pour des propriétés de sûreté [73]. On s'intéresse à des propriétés de sûreté $P$ sur les traces suspendues dont la négation peut être décrite par un automate $Obs_{\neg P}$ qui atteint un état *Violate* sur l'occurrence d'une sortie. La propriété de sûreté est satisfaite par la spécification si aucun état *Violate* n'est accessible dans le produit $det(\Delta(S)) \times Obs_{\neg P}$. On souhaite alors générer des tests permettant de tester la violation de la propriété sur une implémentation. On remarque d'abord que la conformité pour **ioco**est une propriété de sûreté dans une classe particulière appelée *propriétés adéquates pour S*, qui se limite aux extensions de traces suspendues de $S$ par des sorties ($STraces(S).A_o$). La description de **ioco**dans la prosition 3.1.2 le caractérise. On peut alors montrer que la conformité est

équivalente à la satisfaction de toute propriété adéquate. On se donne donc une propriété de sûreté satisfaite sur $det(\Delta(S))$. La génération de tests consiste d'abord à calculer l'intersection du langage reconnu par $Obs$ avec $STraces(S).A_o$, ce qui s'effectue en prolongeant $det(\Delta(S))$ par des sorties non spécifiées et en effectuant le produit avec $Obs$. On calcule ensuite le graphe de test complet de manière identique au cas des objectifs d'accessibilité, sauf qu'ici il faut atteindre ún état étiqueté *Violate*. Les tests produits n'ont pas de verdict **Pass**mais un verdict $Violate$ qui signifie en même temps la non-conformité et la violation de la propriété, un verdict **Fail**signifiant la non-conformité (et la violation d'une autre propriété de sûreté), et un verdict **Inconc**signifiant qu'on ne peut plus atteindre à la fois la non-conformité et la violation de la propriété.

Ce travail permet donc d'étendre le type de propriétés utilisables pour la génération de tests et de relier de manière étroite la satisfaction de propriétés et la conformité. Des extensions à ce travail sont envisagées. La première est l'implémentation de cette technique dans TGV. La seconde est l'extension au cas symbolique dans STG. Enfin, la caractérisation précise, en termes de logique temporelle, de ces propriétés de sûreté, et leur compilation en objectifs de test.

**Objectifs *versus* couverture :** Un aspect commun de TGV et STG est l'utilisation d'objectifs de test. C'est un avantage certain par rapport à l'écriture manuelle des cas de test. Les objectifs sont des objets de plus haut niveau, et TGV et STG assurent la correction des tests synthétisés par rapport aux spécifications. Néanmoins, l'effort de formalisation des objectifs reste important et nécessite une certaine expertise. Certains utilisateurs souhaiteraient encore plus d'automatisation, et ne spécifier la sélection des tests que par un critère de couverture, comme cèla est d'usage dans le test structurel (voir par exemple [71]). Certains outils, comme TestComposer fournissent une réponse partielle à cette difficulté, en permettant de synthétiser des objectifs de test simples pour un critère de couverture de branche. Cette couverture est malheureusement souvent insuffisante et il reste nécessaire de construire des objectifs spécifiques. Nous avons amélioré TGV dans le cadre du projet européen Agdis pour permettre la description d'objectifs de couverture sur des expressions quelconques de la spécification, y compris les états et les transitions. Mais ces solutions restent coûteuses car elles sont basées sur une énumération, au moins partielle. Nous sommes cependant persuadés que la recherche sur la couverture des cas de tests pour la conformité est une nécessité. Mais elle doit être adaptée aux spécificités du test de conformité. En test de conformité, le code de l'implémentation est *a piori* inconnu. Ce qui est connu c'est la spécification. Pour la génération de tests, on peut donc parler de couverture de la spécification (de ses traces) par les tests générés. Cependant, il faut tenir compte du non déterminisme observable des systèmes et des tests. Un test de tels systèmes n'est pas une séquence. Ainsi parmi toutes les traces de la spécification couvertes par un test, seulement l'une d'elles est réellement couverte lors d'une exécution du test. De plus, puisqu'on observe des traces, on ne peut pas descendre au niveau des séquences d'actions (internes) de la spécification. Avec des restrictions sur l'ensemble des implémentations possibles (des modèles de ces implémentations), on peut éventuellement parler de couverture potentielle de tests sur cet ensemble d'implémentations. Nous avons commençé à étudier ce problème, en collaboration avec Sophie Pinchinat, en partant du problème récent de la couverture en model-checkin [40, 10]. Le problème de la couverture en model-checking est le suivant.

Étant donné une propriété (de logique temporelle LTL, CTL par exemple) satisfaite par la spécification, quelle sont les états de la spécification qui jouent véritablement un rôle dans la satisfaction de la propriété. Autrement dit, est-ce que la formule est invalidée par un mutant de la spécification dans lequel on mute la valeur de vérité des propositions atomiques dans un état. On peut affiner cette notion de couverture en considérant les dépliages de la spécifications (ses exécutions donc) et en mutant toutes les copies de l'état, ou un sous-ensemble de ses copies. Si on se pose cette question pour des propriétés pour lesquelles on sait générer des tests, par exemple pour les propriétés de sûreté, on peut se demander quelle est la relation entre la couverture des tests générés et la couverture du model-checking de ces propriétés.

**Algorithmique énumérative :** l'efficacité des techniques à la TGV pourrait être améliorée en le combinant avec d'autres techniques de vérification. Une piste intéressante est l'utilisation de techniques dites d'ordre partiel utilisées en model-checking [30, 78, 64]. Leur principe est d'éviter d'énumérer les chemins équivalents à permutation près d'actions concurrentes, si celles-ci ne jouent aucun rôle dans la vérification. Ces techniques peuvent être utilisées, au moins au niveau des actions internes, lors de la déterminisation. En effet, l'ordre de tir des transitions internes concurrentes n'a pas d'importance sur le résultat de la déterminisation (si ce n'est quand elles apparaissent dans les objectifs de test), à partir du moment où on peut calculer les actions visibles après celles-ci. L'application au niveau des actions observables est aussi envisageable mais il est nécessaire cependant de reconstruire les entrelacements sélectionnés dans les cas test, ou de conserver l'information de concurrence dans les tests (voir ci-dessous). Une autre piste que nous avons commencé à exploré, est la combinaison de la génération à la volée et du calcul incrémental des tests dans le cas de spécifications composées.

**Techniques symboliques :** Dans le cadre des techniques symboliques, plusieurs voies de recherches sont envisagées. D'abord, nous essayons d'étendre les modèles symboliques et les analyses par interprétation abstraite sur ces modèles afin de traiter de manière plus précise des classes de spécifications plus riches. En particulier, des travaux de Bertrand Jeannet et Wendelin Serwe ont commencé sur l'analyse par interprétations abstraite de programmes manipulant des types de données pile et tas pour permettre d'analyser les programmes récursifs. Nous envisageons également l'analyse des files de communication dans les automates communicants, afin de traiter de manière plus satisfaisante les modèles de systèmes communiquant de manière asynchrone. D'autre part, nous n'avons pas encore utilisé toutes les possibilités des techniques d'analyse par interprétation abstraite pour la génération de tests. Nous pensons que l'interprétation abstraite, et en particulier des techniques comme le partitionnement dynamique, peuvent être des outils efficaces pour découvrir la structure de contrôle implicite, cachée dans l'évolution des variables. La découverte d'une telle structure de contrôle implicite peut être un préalable fort utile pour sélectionner des tests par des critères de couverture sur la structure implicite, plus précise que la structure de contrôle initiale.

**Test réparti :** Un autre problème important est celui du test réparti. Quand le système à tester est réparti, les cas de tests doivent eux-mêmes être répartis et se synchroniser.

Des langages comme TTCN fournissent cette puissance d'expression. Nous avons étudié ce problème dans [45] en synthétisant un test séquentiel pour la spécification du système réparti et en le distribuant en un ensemble de cas de test en fonction de la localité des interfaces de test, tout en produisant les synchronisations nécessaires entre testeurs. Le problème des choix non locaux est résolu par l'usage d'un service de consensus distribué. L'inconvénient majeur est la perte de la concurrence intrinsèque au système, à cause de la sémantique d'entrelacements, ce qui a pour effet de produire des synchronisations inutiles. Une solution est de préserver l'information de concurrence par l'usage de modèles de vraie concurrence [44]. Les modèles de spécification utilisés sont des ensembles de pièces représentant chacun des comportements concurrents de bases avec pré et post-conditions. La construction des tests est basée sur le calcul d'un dépliage de la structure d'événement sous jacente. Ces modèles de vraie concurrence peuvent aussi être dotés d'actions sur des données. La combinaison des modèles symboliques et des modèles de vraie concurrence pour la génération de tests pourrait permettre de traiter des spécifications plus réalistes. C'est un point que nous souhaitons élaborer en collaboration avec C. Jard.

**Génération de tests et synthèse de contrôleurs** Le problème de la synthèse de contrôleursur des systèmes à événements discrets, que nous étudions par ailleurs dans mon équipe de recherche, est le suivant. Etant données une spécification et une propriété, quelles transitions de la spécification doit-on interdire (par un superviseur) de façon à assurer cette propriété. On pose de plus la contrainte que certaines transitions sont invisibles du superviseur, et parmi les transitions visibles, certaines sont contrôlables (et peuvent être interdites par le superviseur), alors que d'autres sont seulement observables. On cherche de surcroît à minimiser le contrôle.

Depuis plusieurs années, nous nous intéressons aux relations entre synthèse de contrôleurs et génération de tests. On peut constater par exemple que le contrôle pour une propriété d'accessibilité est assez similaire à la génération de tests. Dans les deux cas, on se base sur la construction de l'automate *Trim*. Dans le cas du test, on se permet de sortir de *Trim* par une sortie (incontrôlable par le testeur), et on produit un verdict **Inconc**. Dans le cas de la synthèse de contrôleurs, on est plus sévère. Si on ne contrôle pas le fait de rester dans *Trim*, on remonte en arrière jusqu'à être sûr de controler, au risque de vider *Trim*. Une adaptation de TGV a permis de programmer simplement cette synthèse de contrôleurs dans le cas de l'observation partielle et du contrôle non bloquant.

Par ailleurs, nous nous sommes récemment intéressés au problème du contrôle d'une implémentation pour assurer la conformité pour **ioco** par rapport à une spécification. On considère donc une implémentation (boîte blanche) et un superviseur interne qui peut contrôler les actions internes et sorties de l'implémentation, mais pas ses entrées, de façon à assurer que l'implémentation contrôlée est conforme à une spécification donnée. La conformité étant une propriété de sûreté, il s'agit donc d'un problème de contrôle pour une propriété de sûreté. La seule vraie difficulté est dû aux blocages, car le blocage n'est pas une action de l'implémentation mais une conséquence de l'absence d'actions ou de l'existence de boucles d'actions internes. L'interdiction d'actions peut donc introduire des blocages qui n'existaient pas initialement. Néanmoins, sous l'hypothèse de contrôle des actions internes, on peut synthétiser un superviseur minimal.

Nous poursuivons ces recherches sur les relations entre test et contrôle en essayant en

particulier de faire profiter à l'un des avancées de l'autre. En particulier, nous nous intéressons au contrôle pour des modèles symboliques pour nous rapprocher des applications réelles manipulant des données. Inversement, nous espérons profiter des résultats de nos recherches en cours sur le contrôle hiérarchique, le contrôle modulaire, pour les utiliser dans la génération de tests.

# Bibliographie


[1] S. Abramsky. Observational Equivalence as a Testing Equivalence. *Theoretical Computer Science*, 53(3), 1987.

[2] A. Arnold and M. Nivat. Comportements de processus. In *Colloque AFCET, Les Mathématiques de l'Informatique*, 1982.

[3] M Benjamin, D. Geist, A. Hartman, G. Mas, R. Smeets, and Y. Wolfsthal. A feasibility study in formal coverage driven test generation. In *36th Design Automation Conference, DAC99*, June 1999.

[4] G. Bernot, M. C. Gaudel, and B. Marre. Software Testing Based on Formal Specification : a Theory and a Tool. *Software Engineering Journal*, pages 387–405, November 1991.

[5] M. Bozga, J.-C. Fernandez, L. Ghirvu, C. Jard, T. Jéron, A. Kerbrat, P. Morel, and L. Mounier. Verification and test generation for the SSCOP protocol. *Journal of Science of Computer Programming, special issue on Formal Methods in Industry*, 36(1) :27–52, January 2000.

[6] Marius Bozga, Susanne Graf, and Laurent Mounier. IF-2.0 : A validation environment for component-based real-time systems. *Lecture Notes in Computer Science*, 2404 :343–348, 2002.

[7] Ed Brinksma and Jan Tretmans. Testing transition systems : An annotated bibliography. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, *MOVEP'2k MOdelling and VErification of Parallel processes, Nantes, France*, volume 2067 of *LNCS*, pages 187 – 195. Springer, 2000.

[8] E. Brinskma. A Theory for the Derivation of Tests. In *Protocol Secification, Testing and Verification VIII*, pages 63–74. North-Holland, 1988.

[9] Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, September 1999.

[10] H. Chockler, O. Kupferman, and M.Y. Vardi. Coverage metrics for temporal logic model checking. In *Tools and algorithms for the construction and analysis of systems*, number 2031 in Lecture Notes in Computer Science, pages 528 – 542. Springer-Verlag, 2001.

[11] T.S. Chow. Testing software design by finite state machine. *IEEE Transactions on Software Engineering*, SE-4(3), May 1978.

[12] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. Automated test and oracle generation for smart-card applications. In *International Conference on Research in Smart Cards (e-Smart'01)*, 2001.

[13] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG : a symbolic test generation tool. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS2002, Grenoble, France*, volume 2280 of *LNCS*. Springer-Verlag, april 2002. Tool paper.

[14] E. Clarke and J. Wing. Formal methods : State of the art and future directions. *ACM Computing Surveys*, 28(4), December 1996.

[15] M. Clatin, R. Groz, M. Phalippou, and Richard Thummel. Two approaches linking a test generation tool with verification techniques. In A. Cavalli and S. Budkowski, editors, *Proceedings of IWPTS'95 (8th Int. Workshop on Protocol Test Systems, Evry, France)*. INT, sep 1995.

[16] P. Cousot and R. Cousot. Abstract intrepretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, January 1977.

[17] R. De Nicola and M. Henessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34 :83–133, 1984.

[18] R. G. De Vries and J. Tretmans. On-the-fly conformance testing using SPIN. *Software Tools for Technology Transfer*, 2(4) :382–393, March 2000.

[19] R. G. De Vries and J. Tretmans. Torwards formal test purposes. In E. Brinskma and J. Tretmans, editors, *Workshop FATES'01 : Formal Approaches of Testing of Software*, BRICS Notes Series NS-01-4, 2001.

[20] L. Doldi, V. Encontre, J.-C. Fernandez, T. Jéron, S. Le Bricquir, N. Texier, and M. Phalippou. Assessment of automatic generation methods of conformance test suites in an industrial context. In B. Baumgarten and A. Burkhardt, H.-J.Giessler, editors, *IFIP TC6 $9^{th}$ International Workshop on Testing of Communicating Systems*. Chapman & Hall, September 1996.

[21] K. Drira, P. Azéma, and F. Vernadat. Refusal Graphs for Conformance Tester Generation and Simplification : A Computational Framework. In *Protocol Specification, Testing and Verification XIII*, C-16. IFIP Transactions, North-Holland, 1993.

[22] L. du Bousquet, S. Ramangalahy, S. Simon, Viho. C., A. Belinfante, and R. G. De Vries. Formal test automation : The conference protocol with TGV/TorX. In H. Ural, R.L. Probert, and G. v. Bochmann, editors, *IFIP $13^{th}$ Int. Conference on Testing of Communicating Systems(TestCom 2000)*. Kluwer Academic Publishers, 2000.

[23] E. H. Eertink. *Simulation Techniques for the Validation of LOTOS Specification*. PhD thesis, Universiteit Twente, the Netherlands, mar 94.

[24] A. Engels, L. Feijs, and S. Mauw. Test Generation for Intelligent Networks Using Model-Checking. In *Third Workshop TACAS, Enschede, The Netherlands*, volume 1217 of *LNCS*. Springer-Verlag, 1997.

[25] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP : A Protocol Validation and Verification Toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. of CAV'96 (New Brunswick, New Jersey, USA)*, volume 1102 of *LNCS*, August 1996.

[26] J.-C. Fernandez, C. Jard, T. Jéron, and G. Viho. Using on-the-fly verification techniques for the generation of test suites. In A. Alur and T. Henzinger, editors, *Conference on Computer-Aided Verification (CAV '96), New Brunswick, New Jersey, USA*, volume 1102 of *LNCS*. Springer-Verlag, July 1996. Egalement disponible en rapport de recherche Irisa n$^o$ 1036.

[27] J.-C. Fernandez, C. Jard, T. Jéron, and G. Viho. An experiment in automatic generation of conformance test suites for protocols with verification technology. *Science of Computer Programming*, 29 :123–146, 1997. Egalement disponible en rapport de recherche Irisa n$^o$ 1035 et Inria n$^o$ 2923.

[28] M.-C. Gaudel and P. R. James. Testing algebraic data types and processes : a unifying theory. *Formal Aspects of Computing*, 10(5-6) :436–451, 99.

[29] A. Gill. *Introduction to the Theory of Finite State Machine*. New-York, McGraw-Hill, 1962.

[30] P. Godefroid. Using partial orders to improve automatic verification methods. In *2nd CAV Workshop*, volume 531 of *LNCS*, pages 176–185. Springer Verlag, 1990.

[31] M. Gondran and M. Minoux. *Graphes et algorithmes*. Eyrolles, 1979. Collection de la Direction des Etudes et Recherches d'Electricité de France.

[32] G. Gonenc. A Method for the Design of Fault-Detection Experiments. *IEEE Transactions on Computing*, C-19, June 1970.

[33] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. A CLP framework for computing structural test data. In *Conference on Computational Logic (CL'2000), Imperial College, London, UK*, pages 399–413, 2000.

[34] J. Grabowski, D. Hogrefe, and R. Nahm. Test Case Generation with Test Purpose Specification by MSCs. In Elsevier Science B.V. (North Holland), editor, *6$^{th}$ SDL Forum*, pages 253–266, Darmstadt (Germany), 1993. Færgemand, O. and Sarma, A.

[35] R. Groz, T. Jéron, and A. Kerbrat. Automated test generation from SDL specifications. In Rachida Dssouli, Gregor von Bochmann, and Yair Lahav, editors, *SDL'99 The Next Millenium, 9th SDL Forum, Montréal, Québec*, pages 135–152. Elsevier, June 1999.

[36] F. C Hennie. Fault detecting experiments for sequential circuits. In *Fifth Annual Symposium on Switching Theory and Logical Design*, pages 95–110,, nov 1964.

[37] Wai-Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h. UMLAUT : an extendible UML transformation framework. In *Proc. Automated Software Engineering, ASE'99, Florida*, October 1999.

[38] H. S. Hong, I. Lee, O. Sokolsky, and S. D. Cha. Automatic test generation from statecharts using model-checking. In E. Brinskma and J. Tretmans, editors, *Workshop FATES'01 : Formal Approaches of Testing of Software*, BRICS Notes Series NS-01-4, 2001.

[39] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[40] Yatin Hoskote, Timothy Y. Kam, Pei-Hsin Ho, and Xudong Zhao. Coverage estimation for symbolic model checking. In *Proceedings of the 36th Design Automation Conference (DAC' 99)*, pages 300–305, New York, June 1999. Association for Computing Machinery.

[41] E. P. Hsieh. Checking experiments for sequential machines. *IEEE Transactions on Computers*, C-20(20) :1152–1166, 1971.

[42] ISO. *Information Processing Systems, Open Systems Interconnection, OSI Conformance Testing Methodology and Framework*. ISO 9646.

[43] ITU. ISO/IEC JTC1/SC21 WG7, Information Retrieval, Transfer and Management for OSI ; Framework : Formal Methods in Conformance Testing. Committee Draft CD 13245-1, ITU-T proposed recommendation Z 500, 1996.

[44] C. Jard. Principles of distributed test synthesis based true-concurrency models. In I. Schieferdecker, H. König, and A. Wolisz, editors, *TestCom2002, Berlin*. Kluwer Academic Publishers, March 2002.

[45] C. Jard, T. Jéron, H. Kahlouche, and C. Viho. Towards automatic distribution of testers for distributed conformance testing. In *FORTE/PSTV'98, Paris, France*. Chapman & Hall, November 1998.

[46] C. Jard, T. Jéron, and P. Morel. Verification of test suites. In *TestCom 2000, IFIP TC 6 / WG 6.1, The IFIP 13th International Conference on Testing of Communicating Systems, Ottawa, Ontario, Canada*. Kluwer Academic Publishers, August 2000.

[47] C. Jard, T. Jéron, L. Tanguy, and C. Viho. Remote testing can be as powerful as local testing. In Jianping Wu, Samuel Chanson, and Qiang Gao, editors, *Formal methods for protocol engineering and distributed systems, FORTE XII/ PSTV XIX' 99, Beijing, China*, pages 25–40. Kluwer Academic Publishers, October 1999.

[48] B. Jeannet. Dynamic partitioning in linear relation analysis. application to the verification of reactive systems. *Formal Methods in System Design*, 23(1) :5–37, July 2003.

[49] T. Jéron and P. Morel. Abstraction, $\tau$-réduction et déterminisation à la volée : application à la génération de test. In *CFIP'97, Congrès Francophone sur l'Ingéniérie des Protocoles, Liège, Belgique*. Hermes, September 1997.

[50] T. Jéron and P. Morel. Test generation derived from model-checking. In Nicolas Halbwachs and Doron Peled, editors, *CAV'99, Trento, Italy*, volume 1633 of *LNCS*, pages 108–122. Springer-Verlag, July 1999.

[51] H. Kahlouche, C. Viho, and M. Zendri. An Industrial Experiment in Automatic Generation of Executable Test Suites for a Cache Coherency Protocol. In A. Petrenko and N. Yevtushenko, editors, *IFIP TC6 11$^{th}$ International Workshop on Testing of Communicating Systems*. Chapman & Hall, September 1998.

[52] H. Kahlouche, C. Viho, and M. Zendri. Hardware Testing using a Communication Protocol Conformance Testing Tool. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 315–329. Springer Verlag, March 1999.

[53] R. Langerak. A testing theory for lotos using deadlock detection. In E. Brinskma, G. Scollo, and C. A. Vissers, editors, *Protocol Specification Testing and Verification, PSTV IX*, pages 87–98. North-Holland, 1990.

[54] Lee and Yannakakis. Testing Finite State Machines : State Identification and Verification. *IEEE Trans Computer*, 43(3) :306–320, 1994.

[55] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines - A Survey. *Proceedings of the IEEE*, 84(8) :1090–1123, August 1996.

[56] Grégory Lestiennes and Marie-Claude Gaudel. Testing processes from formal specifications with inputs, outputs and data types. In *13th International Symposium on Software Reliability Engineering (ISSRE'02), Annapolis, Maryland*. IEEE Computer Society Press, nov 2002.

[57] D. Lugato, C. Bigot, and Y. Valot. Validation and automatic test generation on uml models : the AGATHA approach. *Electronics Notes in Theoretical Computer Science*, 66(2), 2002.

[58] Gang Luo, Gregor von Bochmann, and Alexandre Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. *IEEE Trans. Software Eng*, 20(2) :149–162, 1994.

[59] N. A. Lynch. I/O automata : A model for discrete event systems. In *Proc. of 22nd Conf. on Information Sciences and Systems*, pages 29–38, Princeton, NJ, USA, March 1988.

[60] Christophe Meudec. ATGen : Automatic test data generation using constraint logic programming and symbolic execution, June 2000.

[61] Edward. F Moore. *Gedanken Experiments on Sequential Machines*. Princeton University Press, Princeton, New Jersey, automata studies edition, 1956.

[62] P. Morel. *Une algorithmique efficace pour la génération automatique de tests de conformité*. PhD thesis, Université de Rennes I, France, February 2000.

[63] S. Naito and M. Tsunoyama. Fault Detection for Sequential Machines by Transitions Tours. In *Proc. IEEE Fault Tolerant Comput. Symp., IEEE Computer Soc. Press*, pages 238–243, 1981.

[64] D. Peled. Combining partial order reductions with on-the-fly model-checking. In D. L. Dill, editor, *CAV Workshop*, volume 818 of *LNCS*. Springer Verlag, 1994.

[65] A. Petrenko, S. Boroday, and R. Groz. Conforming configurations in efsm. In *Forte-PSTV 99*. Kluwer, oct 1999.

[66] Alexandre Petrenko. Fault model-driven test derivation from finite state models : Annotated bibliography. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, *MOVEP'2k MOdelling and VErification of Parallel processes, Nantes, France*, volume 2067 of *LNCS*, pages 196 – 205. Springer, 2000.

[67] M. Phalippou. *Relations d'implantations et Hypothèses de test sur les automates à entrées et sorties*. PhD thesis, Université de Bordeaux, 1994.

[68] M. Phalippou. Test Sequence Generation Using Estelle or SDL Structure Information. In *FORTE'94*, Berne, October 1994.

[69] Iain Phillips. Refusal testing. *Theoretical Computer Science*, 50 :241–284, 1987.

[70] S. Ramangalahy, P. Le Gall, and T. Jéron. Une application de la théorie des jeux au test de conformité. In André Schaff, editor, *Colloque Francophone sur l'Ingénierie des Protocoles, CFIP 99, Nancy, France*. Hermès, April 1999.

[71] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4) :367–375, 1985.

[72] V. Rusu, L. du Bousquet, and T. Jéron. An approach to symbolic test generation. In *International Conference on Integrating Formal Methods (IFM'00)*, volume 1945 of *LNCS*, pages 338–357. Springer-Verlag, 2000.

[73] V. Rusu, H. Marchand, V. Tschaen, T. Jéron, and B. Jeannet. From safety verification to safety testing. In R. M. Hierons and R. Groz, editors, *Testcom 2004, IFIP TC 6 / WG 6.1 The 16th IFIP International Conference on Testing of Communicating Systems, Oxford, UK*, LNCS. Springer, March 2004. to appear.

[74] R. Segala. Quiescence, fairness and the notion of implementation. In E. Best, editor, *CONCUR'98*, volume 715 of *LNCS*. Springer-Verlag, 1993.

[75] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1 :146–160, 1972.

[76] J. Tretmans. Testing Labelled Transition Systems with Inputs and Outputs. In $8^{th}$ *International Workshop on Protocols Test Systems*, Evry - France, September 1995.

[77] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17, 1996.

[78] A. Valmari. A stubborn attack on state explosion. In *2nd CAV Workshop*, volume 531 of *LNCS*, pages 156–165. Springer Verlag, 1990.

[79] Specification and Description Language SDL, 1988.

[80] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4) :366–427, December 1997.