Numéro d'ordre : 1   112

THÈSE

présentée à

L'UNIVERSITÉ de BORDEAUX I

pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ de BORDEAUX I

Spécialité : Informatique

par

Marc PHALIPPOU

Ingénieur France Télécom - CNET

Relations d'implantation et hypothèses de test sur des automates à entrées et sorties

---

Soutenue le 8 septembre 1994 après avis de :

M.	Ed	BRINKSMA	
Mme.	Rachida	DSSOULI	Rapporteurs

Devant la Commission d'Examen formée de :

MM.	Robert	CORI	Président
	André	ARNOLD	
	Ed	BRINKSMA	
	Richard	CASTANET	
Mme.	Rachida	DSSOULI	
MM.	Roland	GROZ	
	Omar	RAFIQ	

Relations d’implantation et hypothèses de test sur des automates à entrées et sorties

Résumé

Le test de conformité est l’activité qui consiste à s’assurer qu’un produit fini remplit bien les fonctions pour lesquelles il a été conçu. Dans le domaine des télécommunications, la complexité et la variété des systèmes mis en œuvre, ainsi que le haut degré de fiabilité exigé de leur fonctionnement global, justifie le soin apporté à la conception des meilleurs tests possibles.

Cette thèse présente un modèle formel permettant d’étudier et d’optimiser la conception des tests de conformité pour les systèmes de télécommunications.

Dans une première partie, nous étudions quelles sont les propriétés que l’on peut observer au moyen d’un test, en tenant compte notamment du mécanisme d’interaction disponible entre le testeur et l’implantation sous test, ainsi que des différentes architectures de test réalisables en pratique.

Puis dans une deuxième partie nous montrons comment sélectionner un nombre raisonnable de tests en s’appuyant sur un certain nombre d’hypothèses, qui permettent de caractériser de manière rigoureuse ce que l’on sait - et ce que l’on ne sait pas - après avoir passé les tests.

Implementation relations and test hypotheses on input-output automata

Abstract

Conformance testing aims at checking that a product complies to the requirements for which it has been developped. In the telecommunications industry , the complexity and the variety of the systems, together with the high level of fiability which is expected from their global interworking, justify to bring a great care to the design of the best tests.

This thesis presents a formal model for the study and optimisation of the design of conformance test cases for telecommunication systems.

In a first part, we study which properties can be characterized by means of testing, taking into account the interaction mechanism available between the tester and the implementation under test, and the various test architectures that can be used in practice.

Then in a second part we show how to select a reasonable number of test cases by making some test hypotheses, which allow to formulate precisely what we know - and what we do not know - once the test cases have been executed.

Remerciements

Et puisqu'une thèse, malgré tout, n'est pas uniquement une œuvre solitaire, je tiens avant toute chose à remercier

Roland Groz, de m'avoir encouragé à entreprendre ce travail, de m'avoir aidé et conseillé tout au long de son déroulement, et d'avoir eu la patience d'en faire une relecture très minutieuse,

Richard Castanet, d'avoir eu la gentillesse d'accepter de diriger cette thèse malgré la distance séparant Bordeaux de Lannion, et de m'avoir laissé très libre dans le déroulement de mes recherches,

Ed Brinksma et Rachida Dssouli d'avoir consacré une partie de leur précieux temps à la lecture et au jugement de ce travail,

André Arnold, Robert Cori et Omar Rafiq de m'avoir fait l'honneur de faire partie de ce jury ,

enfin tous ceux et toutes celles qui par leur soutien scientifique ou moral m'ont permis de mener à bien la réalisation de cette thèse.

Table des matières

CHAPITRE 1

Une étude formelle du test de conformité

1 Introduction

L'ultime recours pour s'assurer qu'un produit fait bien ce que l'on attend de lui est de le tester . Le test est une pratique courante dans la vie de tous les jours comme dans l'industrie. Il est perçu non pas comme une remise en cause de la compétence des concepteurs du produit, mais comme une dernière occasion de se convaincre que tout a été fait correctement, que rien n'a été oublié. On trouve normal qu'un cuisinier , même expérimenté, goûte ses plats avant de les servir . De même, sans douter du savoir -faire des ingénieurs qui l'ont fabriqué, on est rassuré de savoir que l'avion dans lequel on s'embarque a déjà volé au moins une fois.

En dépit des préjugés tenaces de certaines sectes d'informaticiens les programmes informatiques sont un produit comme les autres, et il est d'usage de les tester avant de les utiliser . L'effort de test représente une part significative du coût de développement d'un programme, et l'industrie du génie logiciel a peu à peu développé et optimisé les méthodes particulières adaptées à ce type particulier de produit. Comme toute activité issue d'un besoin pratique, ceci s'est fait sur des bases empiriques, par accumulation progressive de savoir -faire au rythme des succès et des échecs rencontrés.

Sans négliger aucunement cette forme d'expertise tirée de l'expérience, il faut cependant être conscient de ses limites pour résoudre des problèmes dont la complexité dépasse un certain degré. Il devient alors utile, et parfois nécessaire, de modéliser de façon plus abstraite les questions que l'on se pose, afin de mieux comprendre les mécanismes profonds mis en jeu, et de bénéficier pour trouver des solutions de la puissance et de la variété des outils mathématiques.

Une telle activité de formalisation du test fait l'objet de travaux de recherche depuis une bonne trentaine d'années. Cette thèse contribue à ces travaux. Nous précisons dans ce premier chapitre le sujet exact de la thèse, comment nous situons notre approche par rapport à l'état de l'art du domaine, et quels sont les principaux résultats que nous avons obtenus.

2 Quels tests pour quels systèmes ?

Les problèmes que nous cherchons à résoudre dans cette thèse sont directement issus de ceux rencontrés au CNET pour la validation de systèmes de télécommunications (au CNET , la «validation» est le test effectué sur un équipement avant son déploiement dans le réseau, et c'est dans ce sens que ce terme sera employé tout au long de cette thèse). Ces systèmes peuvent être des équipements physiques isolés, comme des terminaux téléphoniques, des cartes électroniques, des autocommutateurs, mais aussi des services fournis par France Télécom aux usagers et mettant en œuvre, pour leur réalisation, tout un ensemble de fonctions élémentaires réparties sur divers éléments du réseau : un simple appel téléphonique est un bon exemple, mais on peut aussi faire moderne en citant des applications plus sophistiquées comme le transfert d'appel, la carte France Télécom (ex Pastel) ou le téléphone mobile Itinéris.

Comme nous l'avons dit plus haut, tester consiste à vérifier qu'un produit fait bien ce que l'on attend de lui. Pour tester un système, il nous faut donc avant tout être capable de définir ce que nous attendons de lui : c'est ce que nous appelons sa *spécification*. Il nous faut ensuite définir une procédure permettant de s'assurer que le *système sous test* (que nous appelons aussi l'*implantation*) est *conforme* à sa spécification. En pratique, une telle procédure consiste à essayer les fonctions prévues dans la spécification et à vérifier qu'elles marchent correctement.

Le test dont nous parlons dans cette thèse est donc toujours un *test de conformité*. Mais nous donnons ici à ce terme, justifié par l'explication fournie au paragraphe précédent, un sens beaucoup plus large que la signification restreinte qui lui est habituellement assignée dans le domaine du test (par exemple dans [ISO 91]). Pour nous, en effet, relèvent également de la conformité les sujets suivants :

1. le test de *robustesse*. La spécification des systèmes que l'on se propose de tester décrit surtout ce que l'on attend comme fonctionnement dans des conditions normales d'utilisation. Le test de robustesse consiste à vérifier la réaction du système face aux tentatives d'utilisation abusives. Cette notion, informelle comme on le voit, peut aussi bien inclure la réaction face aux pannes que face aux fraudes ou aux simples erreurs de manipulation. Dans cette thèse, nous considérons qu'à partir du moment ou une telle réaction est prévue, elle fait partie intégrante de la spécification du système. Cependant, les particularités du test de robustesse peuvent justifier un traitement spécial au moment de la conception des tests. Ce point sera étudié au chapitre 7.
2. le test de *performances*. Dès lors que ces performances sont exigées par le cahier des charges du système sous test, elles font de toute évidence partie des spécifications. Par contre comme nous allons l'expliquer ci-dessous, les modèles mathématiques utilisés dans cette thèse ne permettent pas de représenter les

performances. Le test de performances est donc bien un test de conformité, mais nous ne l'étudions pas ici.

3. le test d'*interfonctionnement*. Ce type de test consiste à vérifier que deux ou plusieurs équipements peuvent communiquer entre eux afin de fournir un service global aux utilisateurs. En partant de cette définition, il est facile d'interpréter le test d'interfonctionnement comme la vérification de la conformité du système global au service global fourni aux usagers. Cette interprétation, que l'on retrouve dans plusieurs articles traitant du test d'interfonctionnement [Rafiq 90] [Arakawa 92], fait de celui-ci un cas particulier du test de conformité.
4. le test *boîte blanche*. Dans le domaine du test de logiciel on distingue souvent le test dit de boîte blanche (ou test *structurel*), qui consiste à tester un programme en connaissant sa structure interne, du test dit de *boîte noire* (ou test *fonctionnel*) qui consiste à tester un programme en s'appuyant uniquement sur la description des fonctions qu'il est censé fournir à un utilisateur extérieur [Myers 79] [White 87]. Le test de boîte blanche peut être considéré comme un test de conformité pour lequel la spécification du système à tester comprend les fonctions extérieures *et* la structure interne du programme. Nous montrerons au chapitre 8 comment les résultats présentés dans cette thèse s'appliquent à ce type de tests.

L'objet de cette thèse est d'étudier quelques problèmes posés par le test de conformité, en utilisant une approche formelle. Nous avons donc besoin de définir un modèle mathématique de l'activité de test qui nous permette de décrire les spécifications, les implantations, la notion de conformité et la procédure de test. Nous présenterons dans les chapitres suivants les choix que nous avons faits pour ces divers modèles. Examinons seulement ici les conséquences qu'ont ces choix sur les types de systèmes auxquels s'appliquent les résultats de cette thèse.

Pour modéliser notre vaste classe d'applications potentielles, nous proposons d'utiliser les objets mathématiques que sont les automates d'états finis (plus exactement une de leurs nombreuses variantes, qui sera présentée en détail dans le chapitre suivant). La raison principale de ce choix est que ces automates sont très bien adaptés pour modéliser les systèmes auxquels nous nous intéressons. Le passage entre le système de télécommunication et son modèle mathématique sous forme d'automate n'est pas l'objet de notre travail. Il est cependant, bien entendu, un complément nécessaire pour rendre utilisables en pratique les résultats présentés ici. Nous nous contenterons, dans un dernier chapitre en fin de cette thèse, d'indiquer comment des techniques de description formelles comme Estelle [ISO 89a] ou LDS [CCITT 88a] permettent cette formalisation.

En résumé, le domaine d'application de cette thèse est constitué de *tous les systèmes modélisables par des automates d'états finis*.

Ceci inclut donc les systèmes de télécommunication que nous avons mentionnés au début de cette section, mais concerne aussi une classe beaucoup plus large d'applications. En contrepartie, les automates que nous utilisons ne permettent qu'une modélisation incomplète des systèmes que nous nous proposons d'étudier. Par exemple, leur comportement temporel ne peut pas être décrit dans le modèle choisi, ce qui empêche d'étudier les performances des systèmes, ou de s'intéresser au test des programmes temps-réel.

De plus, pour certaines applications, la description sous forme d'automates, bien que possible, n'est pas la mieux adaptée pour formaliser les problèmes que nous nous posons avec un niveau adéquat d'abstraction : par exemple, pour spécifier les propriétés de cohérence requises entre des éléments de bases de données, des techniques axiomatiques (avec préconditions et postconditions par exemple) sont sûrement plus appropriées.

3 Les problèmes à résoudre

Le travail présenté ici s'inscrit dans le cadre d'études menées au CNET afin de rationaliser et d'automatiser la conception de tests de conformité. Il a débuté à la suite d'une étude pratique de génération de tests nommée TVEDA, dont il est un prolongement naturel.

TVEDA [Phalippou 90a] est une expérience visant à automatiser la conception des tests pour un certain type de protocoles (protocole D niveaux 2 et 3 [CCITT 88b] [CCITT 88c], décrivant l'interface entre l'usager et le réseau pour le réseau numérique à intégration de services). Cette expérience s'est faite sur des bases pratiques et empiriques. Nous avons observé comment des experts, prenant en compte les diverses contraintes de mise en œuvre du test dans les laboratoires et leur connaissance des protocoles ci-dessus, conçoivent les suites de tests qu'ils jugent les mieux adaptées. Nous avons en partie formalisé la façon de procéder de ces experts sous forme d'un algorithme, et développé un outil d'aide à la production de tests correspondant.

Très bien, ont conclu lesdits experts. Vous avez mis notre expertise en machine, vous êtes capables d'obtenir peu ou prou les mêmes tests que ceux que nous sommes capables d'imaginer. Le processus est en partie automatisé, c'est moins fatigant et ça va plus vite. Mais pouvez-vous pour autant répondre à nos questions : qu'est-ce que les tests ainsi produits garantissent, une fois qu'on les a passés avec succès ? Pouvez-vous nous fournir un chiffre, une mesure de la *couverture* ainsi obtenue ? Pouvait-on faire mieux, tester plus de propriétés, avec moins d'effort ? Comment croire que notre savoir-faire, si empirique, ait directement abouti au test optimal ? En quelque sorte, une fois l'outil réalisé, il reste à réfléchir pour savoir ce qu'on a fait, et si on n'aurait pas pu faire mieux.

Cette thèse fournit un début de réponse aux questions ci-dessus. Pour cela, nous construisons un modèle mathématique approprié permettant d'analyser la conception et la sélection des tests. Ce modèle et ses propriétés sont étudiés en détail dans les chapitres 2 à 9 suivants. Un dernier chapitre montre comment interpréter précisément au moyen de ce modèle les choix et les algorithmes mis en œuvre dans TVEDA.

Cette thèse étudie principalement deux questions :

1. *Qu'est-ce que le test peut prouver ?* Le test est un mode particulier d'observation d'un système, dans lequel on interagit avec un produit fini, et donc on n'a accès qu'à l'interface externe de ce produit. De plus les contraintes pratiques de mise en œuvre du test dans les laboratoires font que certaines parties de cette interface

externe ne sont même pas accessibles au testeur (ceci sera discuté de manière plus approfondie au chapitre 4). Ceci limite les possibilités d'observation du test. Il est donc important de savoir exactement ce qu'il est possible et ce qu'il n'est pas possible de vérifier au moyen d'un test.

2. *Comment sélectionner un nombre raisonnable de tests ?* Malgré l'automatisation croissante du processus d'exécution des tests dans les laboratoires, des contraintes d'ordre économique limitent fortement la taille des suites de tests qu'il est permis de passer en pratique (au-delà, le coût du test devient excessif). Or la plupart des systèmes que nous nous proposons de tester sont d'une complexité telle que les tester complètement requiert un temps énorme, voire théoriquement infini. Il est donc nécessaire d'étudier les moyens de limiter le nombre de tests, tout en caractérisant de façon précise les conséquences d'une telle limitation sur les résultats fournis par le test.

Certains pourraient s'étonner de la démarche suivie pour aboutir à cette thèse, qui consiste à chercher a posteriori une justification formelle, théorique, à des algorithmes déjà programmés, issus de l'expérience pratique. À ceux qui douteraient que l'on puisse justifier un tel travail par l'aspect indéniablement esthétique de sa démarche, nous proposerons deux arguments plus rationnels :

1. l'étude théorique est le seul moyen pour comprendre vraiment les mécanismes profonds mis en jeu, qui n'apparaissent pas de façon explicite dans les algorithmes empiriques. Elle seule peut permettre d'améliorer les méthodes actuelles et de répondre aux questions que se posent encore les experts du test.
2. partir des problèmes pratiques est un garde-fou salutaire contre les éventuelles dérives d'une théorie pour la théorie et permet d'espérer raisonnablement que les résultats obtenus seront applicables.

4 Choix effectués et thèses défendues

Les chapitres 2 à 4 constituent la première partie de la thèse, et répondent à la première des questions ci-dessus (qu'est-ce que le test peut prouver). Deux choix principaux sont faits :

1. d'une part l'utilisation d'un modèle particulier d'automates, les *automates à entrées et sorties*, que nous définissons au chapitre 2, afin de modéliser les spécifications et les implantations. Ces automates sont une adaptation des systèmes de transition étiquetés qui sont utilisés dans beaucoup de travaux sur le test [Brinksma 88] [T retmans 92b]. Cette adaptation permet d'étudier des systèmes communiquant par entrées et sorties (et non pas par rendez-vous négocié comme dans les systèmes de transitions étiquetés habituels), ce qui, selon nous, modélise mieux les systèmes de télécommunications auxquels nous nous intéressons (ce point est discuté en détail au chapitre 3).

   Remarquons que ce modèle est différent des machines de Mealy [Hopcroft 79] généralement utilisées dans les études sur le test basées sur les automates [Sidhu 89] [Fujiwara 91a]. Nous justifions notre choix au chapitre 2.

2. d'autre part l'utilisation du concept de *relation d'implantation* [Brinksma 88] pour modéliser la conformité. La justification de ce choix vis à vis d'autres approches existantes est faite au chapitre 2.

Le principal résultat obtenu est que le pouvoir d'observation du test est limité, et que la limite que nous devons prendre en compte en pratique est l'*équivalence de traces modulo l'interface de test* (ceci signifiant que nous pouvons distinguer, au mieux, des systèmes dont les traces vues à travers le dispositif de test diffèrent). Ce résultat a été publié dans [Phalippou 92]. De façon plus précise :

1. dans le chapitre 2, nous montrons comment construire une théorie du test adaptée au modèle d'automates choisi. Nous nous plaçons dans le cas idéal où le dispositif de test ne limite pas ce qu'il est possible de tester. Nous montrons la limite intrinsèque du pouvoir d'observation du test, ainsi qu'une manière de concevoir un test optimal compte tenu de cette limite. La démarche suivie est calquée sur les travaux présentés dans [Brinksma 88], mais l'application de cette démarche à des systèmes communiquant par entrées et sorties est une nouveauté de cette thèse. Les résultats que nous présentons rejoignent presque exactement ceux de [Verhaard 92], obtenus par une autre méthode.
2. dans le chapitre 3, nous présentons une transformation formelle entre les systèmes communiquant par entrées et sorties et ceux utilisant un mécanisme de rendez-vous négocié. Cette transformation rend explicites les liens entre la théorie du test présentée au chapitre 2 et celle de [Brinksma 88]. Elle nous sert à discuter de la pertinence (ou plutôt de la non pertinence pratique) de modèles abstraits dans lesquels le test a un pouvoir d'observation supérieur à celui trouvé dans notre modèle.
3. dans le chapitre 4, nous étudions comment dans les configurations de test réelles le dispositif de test limite ce que le test peut prouver. La nouvelle limite compte tenu du dispositif de test est formulée de façon précise et générale. Nous montrons comment la théorie permet de calculer un test optimal en fonction de cette limite. Puis nous montrons comment des méthodes approchées permettent en pratique d'utiliser ce résultat en évitant la complexité rédhibitoire du calcul complet prescrit par la théorie. Enfin, nous examinons les conséquences de ces résultats sur la méthodologie de développement des tests préconisée par la norme ISO 9646 [ISO 91].

Les chapitres 5 à 9 constituent la deuxième partie de la thèse, et répondent à la question : comment sélectionner un nombre raisonnable de tests ?

Nous choisissons d'utiliser le concept d'*hypothèse de test* emprunté à [Bernot 91] [Gaudel 92] pour étudier la sélection des tests. Une hypothèse de test est une supposition a priori faite sur les implantations que l'on teste. Moyennant cette hypothèse, qui restreint le domaine des implantations à considérer, nous pouvons nous contenter d'un nombre réduit de tests pour vérifier la conformité. Nous introduisons dans les différents chapitres différentes sortes d'hypothèses de test utilisables en pratique pour sélectionner des tests et adaptées à notre modèle particulier d'automates :

1. dans le chapitre 5, nous étudions une hypothèse de *régularité*, qui consiste à limiter la complexité des implantations considérées. Ce nom est emprunté à [Bernot 91], bien que la formulation faite ici de ce concept soit fort différente, car selon nous il s'agit bien du même mécanisme profond. Nous montrons que ce type d'hypothèse est dans notre cas une reformulation, adaptée au test, d'un résultat classique sur les automates finis (voir [Gill 62], théorème 5.2) qui sous-tend la plupart des méthodes de génération de tests basées sur les automates [Chow 78] [Fujiwara 91a].
2. dans le chapitre 6, nous introduisons la notion de *test abstrait* comme une méthode pour concevoir des hypothèses de test raisonnables. À partir de la spécification d'un gros automate nous calculons une abstraction, plus petite, qui représente un vue partielle du comportement de cet automate. Puis nous étudions sous quelles hypothèses la conformité entre une implantation et la spécification initiale peut être déduite d'une relation entre leurs abstractions respectives. Cette nouvelle relation abstraite est vérifiée au moyen d'un test concret, appliqué à l'implantation. Nous montrons comment ce mécanisme permet de justifier rigoureusement le test séparé de fonctions *indépendantes* d'un système.
3. dans le chapitre 7, nous appliquons la méthode de test abstrait pour fournir une version adaptée aux automates de l'hypothèse d'*uniformité* [Bernot 91], qui n'est autre qu'une formalisation de la méthode de test par partitionnement largement utilisée pour le test de programmes séquentiels [Rapps 85]. Nous montrons que cette hypothèse de test est bien adaptée pour sélectionner les tests de robustesse, ainsi que les tests portant sur les données transmises par les systèmes de télécommunications, deux sujets qui sont peu ou mal formalisés dans l'état de l'art du domaine qui nous intéresse (tests basés sur des automates finis).
4. dans le chapitre 8, nous présentons une vision originale du concept d'*automate étendu*, qui insiste non pas sur leur aspect opérationnel (instructions classiques de programmation enrichissant un formalisme d'automates, comme en Estelle ou LDS), mais uniquement par leur aspect structurant d'un gros automate sous-jacent. Nous montrons que cette représentation est pertinente pour exploiter les automates étendus à des fins de sélection des tests, en faisant des hypothèses d'uniformité différentes de celles introduites au chapitre précédent.
5. dans le chapitre 9, nous étudions le problème de l'*exécution* des tests. Nous montrons que pour sélectionner un nombre fini de tests dans le cas où les implantations peuvent être non-déterministes, il nous faut introduire une hypothèse d'équité sur les séquences d'actions exécutables par ces implantations. Dans un premier temps, nous raffinons donc notre modèle d'automates pour pouvoir y exprimer le concept d'équité dont nous avons besoin. Puis nous définissons les tests exécutables qui exploitent cette nouvelle hypothèse de test. Remarquons que là encore, nous nous attachons à formaliser une question qui est souvent abordée de façon seulement informelle dans les articles représentatifs de l'état de l'art du domaine.

Si le concept d'hypothèse de test est emprunté à [Bernot 91], comme indiqué plus haut, son application aux spécifications modélisées par des automates est une nouveauté de cette thèse. Certains types d'hypothèse (régularité, uniformité) sont une adaptation de travaux antérieurs sur le test d'automates, ou de travaux sur d'autres domaines, comme le test de programmes séquentiels. Les hypothèses d'indépendance et d'équité sont par contre, à ma connaissance, originales, comme l'est la méthode de test abstrait pour la conception d'hypothèses de test. Une partie de ces travaux a été publiée dans [Phalippou 93] (principalement des extraits des chapitre 5 et 9), une autre dans [Phalippou 94] (des extraits des chapitres 6 et 7).

Enfin, pour conclure cette thèse, le chapitre 10 montre succintement comment les idées qui ont été développées sont liées en une méthode cohérente, et mises en œuvre dans TVEDA. On trouve également dans ce chapitre des idées sur des ouvertures et prolongements possibles.

CHAPITRE 2

Test de conformité des automates à entrées et sorties

1 Introduction

Nous présentons dans ce chapitre une formalisation du test de conformité bâtie sur un modèle particulier d'automates, que nous appelons les automates à entrées et sorties. Nous choisissons ce modèle comme base de notre étude car il est bien adapté pour représenter , avec un niveau de généralité et de simplicité satisfaisant, les systèmes de télécommunications qui nous préoccupent en fin de compte.

Dans un premier temps, nous définissons le modèle d'automates, ainsi que le mode de communication asymétrique par entrées et sorties qui caractérise dans notre approche le dialogue entre le testeur et l'implantation. Nous étudions comment représenter la conformité entre une implantation et une spécification au moyen de la notion de relation d'implantation. Enfin, nous montrons comment modéliser l'activité de test proprement dite, c'est à dire les tests, leur exécution et leurs critères de succès. Remarquons que le même modèle d'automates nous sert à représenter les spécifications, les implantations et les tests.

Nous présentons ensuite un exemple d'application de cette théorie du test. Après avoir choisi une relation d'implantation particulière nous prouvons qu'il est possible de calculer , à partir de toute spécification, un test appelé testeur canonique asynchrone, qui caractérise toute implantation conforme à la spécification, au sens de la relation d'implantation particulière choisie. Cette démarche est calquée sur celle suivie dans [Brinksma 88] à partir d'un autre modèle d'automates.

Enfin nous prouvons une propriété fondamentale du test ainsi formalisé : le mode d'interaction choisi entre le testeur et l'implantation testée limite à priori la conformité que l'on peut vérifier au moyen d'un test. Dans l'exemple d'application présenté ci-dessus, on ne peut pas tester de relation plus forte que l'équivalence de traces.

2 Automates à entrées et sorties

2.1 Définition du modèle

Les systèmes de transitions étiquetées sont un modèle utilisé pour décrire la sémantique d'un certain nombre de langages de spécification, comme CCS [Milner 89], CSP [Hoare 85] ou LOTOS [ISO 89b], car ils représentent bien les processus communicants décrits dans ces langages. Ils sont également utilisés dans certains travaux théoriques sur le test, en particulier dans [Brinksma 88], dont nous reprenons ici la démarche.

Définition 2.1 : un *système de transitions étiquetées* (ou LTS pour *Labelled Transition System*, en anglais) est un quadruplet $\langle S, L, T, s_0 \rangle$ où :

1. S est un ensemble dénombrable non vide d'états;
2. L est un ensemble dénombrable non vide d'interactions;
3. $T \subseteq S \times (L \cup \{\tau\}) \times S$ est la relation de transition.
4. s_0 est l'état de départ du système de transitions étiquetées.

Pour modéliser les spécifications et les implantations, nous utilisons une variante de ce modèle présentant les caractéristiques suivantes :

1. nous nous limitons à des systèmes dont les ensembles d'états et d'interactions sont finis. Cette restriction simplifie plusieurs démonstrations et définitions ultérieures, et ne nous pénalise pas vraiment, car les systèmes réels peuvent toujours être modélisés par des automates finis.
2. les interactions sont réparties en deux sous-ensembles, les *entrées*, c'est à dire les réceptions, et les *sorties*, c'est à dire les émissions. En effet, cette distinction nous parait fondamentale pour étudier les problèmes liés au test de systèmes. Ceci est dû à la nature intrinsèque du test, qui ne peut être mis en œuvre que comme un dialogue, c'est à dire une suite d'émissions et de réceptions, entre un testeur et une implantation. Nous verrons à la section 2.3 comment cette distinction est prise en compte lors de la définition de la communication entre automates. Nous reviendrons en détail au chapitre 3 sur la pertinence d'un modèle à entrées et sorties par rapport à un modèle où les automates communiquent par rendez-vous négocié.

Nous obtenons donc la définition ci-dessous :

Définition 2.2 : un *automate à entrées et sorties* est un quadruplet $\langle S, L, T, s_0 \rangle$ où :

1. S est un ensemble fini non-vide d'états;
2. L est un ensemble fini non vide d'interactions;
3. $T \subseteq S \times ((\{?, !\} \times L) \cup \{\tau\}) \times S$ est la relation de transition. Chaque élément de T est une transition, entre un état de départ et un état d'arrivée. Cette transition

est soit associée à une action observable (émission (!) ou réception (?) d'une interaction), soit une transition interne étiquetée par τ.

4. s_0 est l'état de départ de l'automate.

Nous utilisons parfois l'abréviation anglaise IOSM (pour *Input-Output State Machine*) pour désigner les automates à entrées et sorties. Ces automates modélisent des systèmes dont le fonctionnement peut être interprété de la façon suivante :

1. transition $(s_1, !\text{a}, s_2)$: l'automate qui est dans l'état s_1 émet l'interaction a et passe dans l'état s_2. La décision de déclencher la transition est locale à l'automate.
2. transition $(s_1, ?\text{a}, s_2)$: l'automate qui est dans l'état s_1 reçoit l'interaction a et passe dans l'état s_2. La décision de déclencher la transition est extérieure à l'automate, puisque la transition est déclenchée à la réception de l'interaction.
3. transition (s_1, τ, s_2) : l'automate qui est dans l'état s_1 passe dans l'état s_2 à la suite d'une décision interne, sans émettre d'interaction observable.

De plus, on ne s'intéresse qu'aux états accessibles à partir de l'état initial par un nombre fini de transitions. Nous pouvons donc supprimer de l'ensemble des états d'un automate ceux qui ne remplissent pas cette condition. Cela nous autorise à ne considérer dans la suite de cette thèse que des automates ayant cette propriété, donc connexes. Lorsque des automates seront définis par construction à partir d'autres automates, nous prendrons soin de toujours montrer que cette propriété est vérifiée.

L'évolution du temps n'est pas précisée dans le modèle : nous ne cherchons pas à représenter les caractéristiques temporelles des systèmes. La décision d'émettre se fait au bout d'un temps quelconque et une réception peut survenir au bout d'un temps quelconque également. Par contre les transitions s'exécutent en un temps nul : deux événements ne peuvent pas entrer en collision.

Nous représenterons les transitions par des petits dessins où l'information qui apparaît est plus ou moins complète (quand les flèches ne sont pas dessinées les transitions sont à considérer de gauche à droite et de haut en bas):

Figure 1 : représentation des transitions

2.2 Déterminisme et complétude

Ce paragraphe a pour but de clarifier ce que l'on désigne dans cette thèse par les termes *choix*, *non-déterminisme* et *complétude*, qui sont souvent employés dans la littérature avec des acceptions variables, donc susceptibles de perturber le lecteur.

Lorsque plusieurs transitions sont possibles à partir d'un état donné de l'automate, on dit qu'on a un choix. Ce choix peut se faire entre des transitions qui sont toutes des émissions, toutes des réceptions ou une combinaison des deux. A partir d'un état donné on peut avoir un choix entre des transitions ayant la même interaction.

1. Lorsque les transitions concernées sont toutes des émissions on dit qu'on a un choix interne puisque la décision de déclencher une transition particulière est locale à l'automate.
2. Si on a affaire à un choix entre des réceptions différentes, on dit qu'on a un choix externe : la transition qui est déclenchée est choisie par l'environnement, c'est à dire le monde extérieur qui envoie des interactions à l'automate.
3. On peut aussi avoir un choix entre des émissions et des réceptions. Vu l'interprétation que l'on donne à la progression du temps, on considère qu'une réception est déclenchée dès qu'elle arrive, la machine pouvant décider en interne de faire une émission avant.

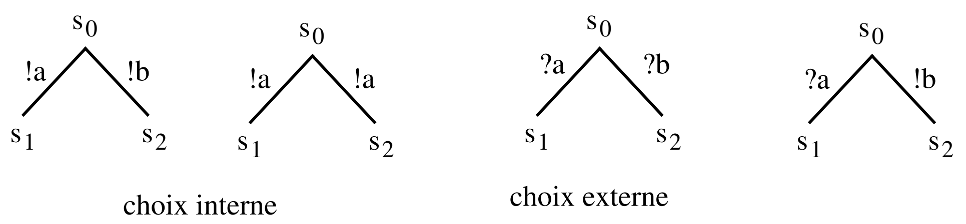


Figure 2 : différents types de choix

Le terme de non-déterminisme est employé en informatique avec des sens différents selon le contexte dans lequel on l'utilise. Intuitivement, un système est non-déterministe lorsqu'il ne permet pas de contrôler ses évolutions futures possibles. Trois formes de non-déterminisme nous intéressent ici (voir figure 3) :

1. dans la théorie des automates, le futur, c'est la transition suivante que l'on peut choisir : on parle de non-déterminisme lorsqu'on a plusieurs transitions partant du même état et étiquetées par la même action [Hopcroft 79]. En suivant cette définition, les IOSM de notre modèle peuvent présenter du non-déterminisme, provenant de deux émissions identiques, deux réceptions identiques, ou deux transitions internes.
2. dans les théories d'algèbres de processus, le futur, c'est l'action observable suivante qui va se produire dans le système. Donc on parle de non-déterminisme dans le cas décrit ci-dessus, mais aussi dans le cas où un choix est possible entre une transition étiquetée par une action observable et une transition étiquetée par l'action interne [Kloosterman 92]. On utilise cette appellation de "non-déterminisme" car l'environnement ne contrôle pas le comportement de l'automate, qui peut décider localement de faire la transition interne.
3. en ce qui concerne le test, le futur c'est : que va répondre le système si on lui envoie telle interaction ? En sus des deux formes de non-déterminisme ci-dessus, qui apparaissent dans notre modèle, on parle donc de non-déterminisme dès que le

comportement du système en réponse à une sollicitation n'est pas unique. C'est le sens utilisé habituellement dans le contexte du test [Fujiwara 91a].

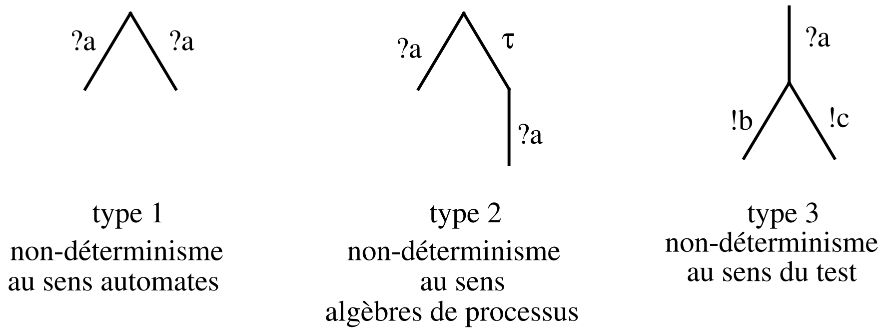


Figure 3 : différents types de non-déterminisme

Nous montrons dans ce chapitre, à la section 6, que seul le troisième type de non-déterminisme doit être pris en compte dans notre théorie du test : nous pouvons transformer tout automate à entrées et sorties en un automate équivalent du point de vue du test qui ne comporte pas les deux premiers types de non-déterminisme. Donc dans la suite de cette thèse, chaque fois que le mot non-déterminisme est employé, et sauf mention explicite contraire, c'est de *non-déterminisme au sens du test* qu'il s'agit. Par contre ceci n'est pas le cas dans l'approche de [Brinksma 88], où la prise en compte du non-déterminisme au sens algèbres de processus est fondamentale. On trouvera plus de détails sur cette approche au chapitre suivant.

D'un point de vue historique, les travaux sur les méthodes de test d'automates finis ont été basés jusqu'à ces dernières années sur des modèles déterministes (voir [Chow 78], [Fujiwara 91a], [Sidhu 89]). Cependant la nécessité de considérer des modèles non-déterministes pour traiter des applications réelles apparaît de plus en plus clairement, ce qui explique une abondance récente de travaux sur ce thème [Fujiwara 91b] [Kloosterman 92] [T ripathy 92].

Nous avons interprété les automates à entrées et sorties comme des machines vers lesquelles on envoie des interactions et qui répondent en retour . Il est clair qu'il est toujours possible dans la réalité d'envoyer vers une implantation n'importe quelle interaction. Nous avons affaire à une boîte noire, qui, dans tous les cas, doit avoir une réaction (qui peut être, bien sûr , de ne rien répondre). Or selon notre définition un automate ne possède pas nécessairement, dans un état donné, de transition de réception pour toutes les interactions du langage L : on a affaire dans ce cas à une spécification incomplète.

Définition 2.3 : une IOSM $\langle S, L, T, s_0 \rangle$ est *complète* si pour chaque état $s \in S$ et pour chaque interaction $a \in L$ il existe une transition correspondant à la réception de a dans s. On dit que l'IOSM est *incomplète* sinon.

Les implantations doivent toujours être modélisées par des automates complets. Par contre les spécifications peuvent être décrites par des automates complets ou incomplets. Nous considérons nécessaire de prendre en compte le cas de spécifications incomplètes, car dans la réalité des applications visées par notre théorie ce phénomène est courant. Les spécifications incomplètes, tout comme les spécifications non-déterministes, sont d'ailleurs un sujet d'études très en vogue actuellement [Petrenko 91] [Luo 93].

Il est important de remarquer que cette définition de la complétude correspond à la notion généralement employée dans le domaine du test, mais diffère de la définition usuelle de la complétude des automates d'états finis. En effet, la complétude ci-dessus est définie uniquement par rapport aux *réceptions* de l'automate à entrées et sorties. Dans la théorie des automates finis un automate ayant un ensemble d'états S et un alphabet L est complet si pour tout $a \in L$ et $s \in S$ il existe une transition issue de s et étiquetée par a [Hopcroft 79]. Par contre notre définition de la complétude est cohérente avec celle utilisée pour les machines de Mealy (voir section 2.6).

2.3 Composition d'automates

Nous modélisons une spécification ou une implantation par un seul automate à entrées et sorties. Par contre, à la section 4, nous choisissons de représenter l'activité de test comme l'exécution en parallèle de l'implantation avec un automate appelé testeur, qui modélise le processus de test. De plus, au chapitre 4, nous étudions le cas où le test se fait à travers une interface de test, dépendante de l'architecture de test choisie, que nous modélisons également par un automate. Il est donc nécessaire pour notre étude de définir une composition parallèle entre automates à entrées et sorties.

Nous avons choisi de faire une composition *synchrone* d'*une* entrée avec *une* sortie, c'est-à-dire qu'une sortie !a d'un automate se synchronise avec l'entrée ?a de son correspondant. Le choix d'un mode d'interaction synchrone comme mécanisme primitif du modèle se justifie par sa simplicité : le fort couplage qui en résulte entre les automates facilite les démonstrations. C'est le choix fait par exemple dans CCS [Milner 89]. De plus il permet de distinguer les notions, fondamentales selon nous, d'entrée et de sortie, de la communication au travers de files d'attente, qui apparaît comme un des cas particuliers possibles d'interface entre le processus testeur et l'implantation (voir ci-dessous). Le choix d'une synchronisation entre deux actions, à la différence du mécanisme de synchronisation multi-partenaires que l'on peut trouver dans LOTOS par exemple, se justifie par le fait qu'il modélise mieux le mécanisme par entrées et sorties que nous voulons représenter.

Une dernière difficulté, purement technique, est de trouver un moyen de représenter la notion d'interface (au sens point d'interaction Estelle) : définir, dès lors qu'il y a plus de deux IOSM, vers quel automate vont les émissions d'un automate donné. Dans la définition ci-dessous, la distinction est uniquement fondée sur les alphabets de messages utilisés. On montre ensuite comment modéliser différents points d'interaction. Cette solution est choisie pour sa simplicité.

Définition 2.4 : la *composition parallèle* de deux IOSM $I_1 = \langle S_1, L_1, T_1, s_{01} \rangle$ et $I_2 = \langle S_2, L_2, T_2, s_{02} \rangle$ est une IOSM $I = \langle S, L, T, s_0 \rangle = I_1 \| I_2$ définie par

1. $S = S_1 \times S_2$
2. $L = L_1 \cup L_2$
3. $s_0 = (s_{01}, s_{02})$
4. les transitions de $I_1 \parallel I_2$ sont exactement celles qui peuvent être obtenues au moyen des règles suivantes :
   a) $(s_1, \tau, s'_1) \in T_1 \Rightarrow \forall s_2 \in S_2, ((s_1, s_2), \tau, (s'_1, s_2)) \in T$
   b) $(s_2, \tau, s'_2) \in T_2 \Rightarrow \forall s_1 \in S_1, ((s_1, s_2), \tau, (s_1, s'_2)) \in T$
   c) si $a \in L_1 \cap L_2$ alors
      $(s_1, ?a, s'_1) \in T_1 \wedge (s_2, !a, s'_2) \in T_2 \Rightarrow ((s_1, s_2), \tau, (s'_1, s'_2)) \in T$ et
      $(s_1, !a, s'_1) \in T_1 \wedge (s_2, ?a, s'_2) \in T_2 \Rightarrow ((s_1, s_2), \tau, (s'_1, s'_2)) \in T$
   d) si $a \in L_1 - L_2$ alors
      $(s_1, ?a, s'_1) \in T_1 \Rightarrow \forall s_2 \in S_2, ((s_1, s_2), ?a, (s'_1, s_2)) \in T$ et
      $(s_1, !a, s'_1) \in T_1 \Rightarrow \forall s_2 \in S_2, ((s_1, s_2), !a, (s'_1, s_2)) \in T$
   e) si $a \in L_2 - L_1$ alors
      $(s_2, ?a, s'_2) \in T_2 \Rightarrow \forall s_1 \in S_1, ((s_1, s_2), ?a, (s_1, s'_2)) \in T$ et
      $(s_2, !a, s'_2) \in T_2 \Rightarrow \forall s_1 \in S_1, ((s_1, s_2), !a, (s_1, s'_2)) \in T$

Dans cette définition, comme dans beaucoup d'autres qui apparaissent dans cette thèse, nous caractérisons les transitions d'un automate par l'expression «sont exactement celles qui peuvent être obtenues au moyen des règles suivantes». Nous entendons par là les transitions qui sont obtenues par application d'un nombre fini quelconque des règles en question. De plus, chaque fois que nous définissons un automate, nous ne considérons que la composante connexe de son graphe contenant l'état initial : ceci concerne à la fois les ensembles d'états et de transitions, qui sont donc seulement des sous-ensembles de ceux décrits par la définition. De façon formelle, cette composante connexe se définit par :

Définition 2.5 : soit $\langle S, L, T, s_0 \rangle$ une IOSM. La *composante connexe* de cette IOSM contenant l'état initial est une IOSM $CC(\langle S, L, T, s_0 \rangle) = \langle S_C, L_C, T_C, s_{0C} \rangle$ définie par :

1. $L_C = L$
2. $s_{0C} = s_0$
3. S_C est défini récursivement par les règles :
   a) $s_{0C} \in S_C$
   b) si $s \in S_C$ et $(s, \mu, s') \in T$ alors $s' \in S_C$.
4. $T_C = \{ (s, \mu, s') \in T, s \in S_C \}$

La définition 2.4 de la composition parallèle permet de modéliser plusieurs interfaces de communication entre différents automates, ou plusieurs points d'interactions au sens Estelle. Prenons l'exemple de la figure 4, qui sera étudié au chapitre 4. Sur cet

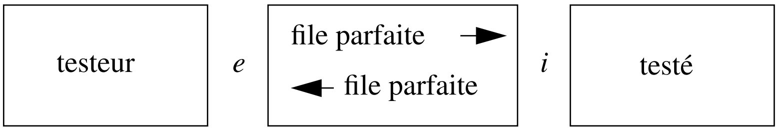


Figure 4 : test à travers un environnement

exemple, on distingue l'interface e (pour externe) entre le testeur et l'environnement de test (i.e. la double file d'attente) et l'interface i (comme interne) entre l'environnement et le testé. Si $L = \{a_1, a_2, \ldots\}$ est un ensemble de messages, on utilise $L_1 = \{e(a_1), e(a_2), \ldots\}$ comme messages du testeur, $L_2 = \{i(a_1), i(a_2), \ldots\}$ comme messages du testé et $L_1 \cup L_2$ comme messages de l'environnement.

Dans ce chapitre, nous étudions uniquement l'exécution en parallèle de deux IOSM (le testeur et le testé), utilisant le même alphabet, de telle sorte que toute émission de l'une est synchronisée avec une réception de l'autre. Le système constitué de ces deux IOSM est un système fermé, qui n'interagit pas avec d'autres automates. D'après la définition ci-dessus, toutes les communications réalisées entre les deux IOSM sont donc masquées sous la forme de l'action interne (inobservable) τ.

Dans le cas général, la composition parallèle d'automates n'est pas associative. Cependant, dans la suite de cette thèse, nous ne considérerons que des systèmes composés d'au plus trois automates, obéissant au schéma de la figure 4, c'est à dire où les ensembles de messages échangés sur les deux interfaces sont disjoints. Dans ce cas particulier, si nous notons respectivement T, E et I les automates modélisant le testeur, l'environnement de test et l'implantation, nous pouvons montrer, d'après la définition de la composition parallèle, que $(T \parallel E) \parallel I = T \parallel (E \parallel I)$ que nous pouvons donc noter plus simplement $T \parallel E \parallel I$.

Enfin, on pourra remarquer que la définition de la composition parallèle ne fait aucune différence entre les entrées et les sorties, qui jouent des rôles exactement symétriques. La différence apparait uniquement par le fait que nous supposons que les IOSM modélisant des entités physiques (implantations, testeurs ou environnements de test) sont complètes, ce qui implique qu'une émission d'un message ne peut jamais être refusée du fait du mécanisme de communication. C'est cette absence de refus qui selon nous caractérise un mode de communication par entrées et sorties, à la différence d'un mode de communication négocié, comme en LOTOS, où la notion de refus est au contraire très importante.

2.4 Traces

La notion de trace est très importante pour notre étude, car, comme nous allons le voir par la suite, les propriétés relatives au test dépendent uniquement des traces des automates manipulés. Nous regroupons dans cette section les principales définitions qui seront utiles pour la suite.

Définition 2.6 : soient $S = \langle S_s, L_s, T_s, s_{0s} \rangle$ et $(\sigma = \mu_1 \ldots \mu_n) \in (\{!, ?\} \times L_s)^*$. Nous notons $\bar{\sigma}$ la séquence obtenue à partir de σ en inversant les ! (émissions) et les ? (réceptions). On introduit les notations suivantes :

1. (s_0, σ, s_n) ssi $(\exists (s_i)_{1 \le i < n} \in S_s^n)$ $(\forall i, 1 \le i \le n)$ $((s_{i-1}, \mu_i, s_i) \in T_s)$
2. (S, σ, s_n) ssi (s_0, σ, s_n)
3. (s_0, ε, s_1) ssi $s_0 = s_1$ ou $(\exists n \ge 1)$ (s_0, τ^n, s_1)
4. $(s_0, \vec{\mu}, s_1)$ ssi $(\exists s_2, s_3 \in S_s)$ $((s_0, \varepsilon, s_2) \wedge (s_2, \mu, s_3) \wedge (s_3, \varepsilon, s_1))$
5. $(s_0, \vec{\sigma}, s_n)$ ssi $(\exists (s_i)_{1 \le i < n} \in S_s^n)$ $(\forall i, 1 \le i \le n)$ $((s_{i-1}, \vec{\mu}_i, s_i) \in T_s)$

Définition 2.7 : une *trace* de S est une séquence d'actions observables $\sigma \in (\{!, ?\} \times L_s)^*$ telle que $(\exists s_n \in S_s)$ $(s_{0s}, \vec{\sigma}, s_n)$. L'ensemble des traces de S est noté $Tr(S)$.

La notion de trace permet de faire abstraction des actions internes τ. Une trace est un comportement observable, c'est à dire visible à l'interface de l'IOSM. La définition de la composition parallèle montre que deux IOSM ne se synchronisent que sur leurs actions observables, donc la notion de trace est étroitement liée à la composition parallèle, comme le montrent les deux propriétés suivantes.

Propriété 2.1 : soient $I_1 = \langle S_1, L_1, T_1, s_{01} \rangle$ et $I_2 = \langle S_2, L_2, T_2, s_{02} \rangle$ deux IOSM et $I_1 \| I_2 = \langle S, L, T, s_0 \rangle$. Si $(\exists (s_{n1}, s_{n2}) \in S)$ $((s_{01}, s_{02}), \tau^n, (s_{n1}, s_{n2}))$ alors $(\exists \sigma \in Tr(I_1))$ $(\bar{\sigma} \in Tr(I_2) \wedge (s_{01}, \vec{\sigma}, s_{n1}) \wedge (s_{02}, \vec{\bar{\sigma}}, s_{n2}))$.

Démonstration : par récurrence sur la longueur n de la séquence d'actions internes de $I_1 \| I_2$.

1. si $n = 0$ la propriété résulte du fait que la séquence vide (également notée ε) est une trace de tout automate (donc $(s_{01}, \vec{\varepsilon}, s_{01})$ et $(s_{02}, \vec{\varepsilon}, s_{02})$) et que $\bar{\varepsilon} = \varepsilon$.
2. on suppose la propriété vraie pour toute séquence de longueur inférieure à n. Si $((s_{01}, s_{02}), \tau^n, (s_{n1}, s_{n2}))$ c'est qu'il existe d'après le point 1 de la définition 2.6 $(s_{(n-1)1}, s_{(n-1)2}) \in S$ tel que $((s_{01}, s_{02}), \tau^{n-1}, (s_{(n-1)1}, s_{(n-1)2}))$ et $((s_{(n-1)1}, s_{(n-1)2}), \tau, (s_{n1}, s_{n2}))$.

   Par hypothèse de récurrence appliquée à la première expression :

   $(\exists \sigma_{n-1} \in Tr(I_1))$ $(\bar{\sigma}_{n-1} \in Tr(I_2) \wedge (s_{01}, \vec{\sigma}_{n-1}, s_{(n-1)1}) \wedge (s_{02}, \vec{\bar{\sigma}}_{n-1}, s_{(n-1)2}))$

   D'après la définition de la composition parallèle, la transition $((s_{(n-1)1}, s_{(n-1)2}), \tau, (s_{n1}, s_{n2}))$ de $I_1 \| I_2$ est obtenue :

a) soit par l'une des règles a) ou b) de la définition 2.4, dans ce cas on pose $\sigma = \sigma_{n-1}$

b) soit par le premier cas de la règle c) : dans ce cas on pose $\sigma = \sigma_{n-1}?a$

c) soit par le deuxième cas de la règle c) : dans ce cas on pose $\sigma = \sigma_{n-1}!a$

Dans tous ces cas cette séquence σ vérifie la propriété. ❑

Propriété 2.2 : soit $\sigma \in Tr(I_1)$ telle que $\bar{\sigma} \in Tr(I_2)$. Alors :
$(\exists (s_{n1}, s_{n2}) \in S) \; (((s_{01}, s_{02}), \varepsilon, (s_{n1}, s_{n2})) \wedge (s_{01}, \sigma, s_{n1}) \wedge (s_{02}, \bar{\sigma}, s_{n2}))$

Démonstration : posons $\sigma = \mu_1 \ldots \mu_n$. Puisque $\sigma \in Tr(I_1)$, alors $(\exists (s_{1i})_{1 \le i \le n} \in S_1^n) \; ((\forall i, 1 \le i \le n) \; (s_{1(i-1)}, \vec{\mu}_i, s_{1i}))$. De même $\bar{\sigma} \in Tr(I_2)$ entraîne $(\exists (s_{2i})_{1 \le i \le n} \in S_2^n) \; ((\forall i, 1 \le i \le n) \; (s_{2(i-1)}, \vec{\bar{\mu}}_i, s_{2i}))$. Donc d'après la définition de $I_1 \| I_2$: $(\forall i, 1 \le i \le n) \; ((s_{(i-1)1}, s_{(i-1)2}), \varepsilon, (s_{i1}, s_{i2}))$, donc $((s_{01}, s_{02}), \varepsilon, (s_{n1}, s_{n2}))$ ❑

Les traces d'un automate sont les mêmes que celles de sa composante connexe contenant l'état initial, ce qui s'exprime par la propriété :

Propriété 2.3 : $Tr(S) = Tr(CC(S))$

Démonstration : découle immédiatement de la définition des traces comme des séquences d'événements observables enchaînés à partir de l'état initial. ❑

À partir de tout automate à entrées et sorties nous pouvons calculer un automate déterministe (au sens des automates), ne comportant pas d'action interne τ et ayant le même ensemble de traces que l'automate initial. Cette construction est similaire à la déterminisation d'un automate non-déterministe (théorème 2.1 de [Hopcroft 79]).

Définition 2.8 : l'*automate de traces* d'une IOSM $I = \langle S, L, T, s_0 \rangle$, noté $TM(I)$ (pour *Trace Machine*) est une IOSM $TM(I) = CC(\langle S_t, L_t, T_t, s_{0t} \rangle)$ définie par :

1. S_t est l'ensemble des parties de S : un état de l'automate est donc un ensemble d'états de la spécification $s_t = \{s_i\}_{1 \le i \le n}$.
2. $L_t = L$
3. $s_{0t} = \{s | (s_0, \varepsilon, s)\}$
4. les transitions de l'automate de traces sont exactement celles obtenues de la façon suivante : pour tout $s \in S_t$ et $\mu \in \{!, ?\} \times L$ soit $s' = \{s_j | (\exists s_i \in s) \; (s_i, \vec{\mu}, s_j)\}$. Si $s' \ne \emptyset$ alors $(s, \mu, s') \in T_t$

L'automate de traces est une IOSM qui "représente" exactement les traces de I, comme le montre la propriété suivante :

Propriété 2.4 : $Tr(TM(I)) = Tr(I)$

Démonstration :

1. Nous montrons par récurrence sur la longueur de la séquence σ que $(s_{0t}, \sigma, s_t) \Rightarrow \{s_t = s \mid (s_0, \vec{\sigma}, s)\} \wedge s_t \neq \emptyset$.
   a) Si $\sigma = \varepsilon$ (séquence vide) alors $s_{0t} = \{s \mid (s_0, \varepsilon, s)\}$ (par définition) et $s_{0t} \neq \emptyset$ car $s_0 \in s_{0t}$.
   b) Sinon, supposons que $(s_{0t}, \sigma\mu, s'_t)$, c'est à dire (s_{0t}, σ, s_t) et (s_t, μ, s'_t). Par hypothèse de récurrence, $s_t = \{s \mid (s_0, \vec{\sigma}, s)\}$. Puisque (s_t, μ, s'_t), par définition de l'automate de traces $s'_t = \{s' \mid (\exists s \in s_t)\ (s, \vec{\mu}, s')\}$ et $s'_t \neq \emptyset$. On en déduit que $s'_t = \{s' \mid (s_0, \vec{\sigma}\vec{\mu}, s')\}$ et $s'_t \neq \emptyset$.

   De cette propriété nous déduisons $\sigma \in Tr(TM(I)) \Rightarrow \sigma \in Tr(I)$.
2. Soit σ une séquence d'événements. Posons $s_t = \{s \mid (s_0, \vec{\sigma}, s)\}$. Nous montrons par récurrence sur la longueur de la séquence σ que $s_t \neq \emptyset \Rightarrow (s_{0t}, \sigma, s_t)$ (au sens de l'automate $\langle S_t, L_t, T_t, s_{0t} \rangle$).
   a) Si $\sigma = \varepsilon$ (séquence vide) alors la propriété est vraie par définition de s_{0t}.
   b) Sinon, posons $\{s'_t = s \mid (s_0, \vec{\sigma}\vec{\mu}, s)\}$ et $\{s_t = s \mid (s_0, \vec{\sigma}, s)\}$. Si $s_t \neq \emptyset$ alors $s'_t \neq \emptyset$, donc par hypothèse de récurrence (s_{0t}, σ, s_t) et $s'_t = \{s \mid (s_t, \vec{\mu}, s)\}$, donc (s_t, μ, s'_t) (définition 2.8), donc $(s_{0t}, \sigma\mu, s'_t)$.

   De cette propriété et de la propriété 2.3 nous déduisons $\sigma \in Tr(I) \Rightarrow \sigma \in Tr(TM(I))$. ❑

De plus l'automate de traces ne contient pas d'action interne, et il est déterministe au sens usuel des automates. Cette dernière propriété étant une conséquence immédiate de la définition. La figure 5 donne quelques exemples d'automates de traces, inspirés de [Drira 92].

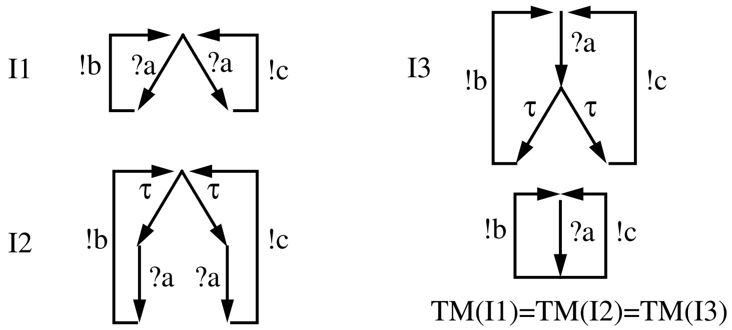


Figure 5 : automates de traces

Remarquons enfin que l'automate de traces est défini pour tout automate à entrées et sorties, celui-ci pouvant très bien être incomplet. La notion d'automate de traces est suffisante pour les études portant sur le pouvoir d'observation du test, c'est à dire pour les chapitres 2 à 4. Dans la deuxième partie de la thèse, nous serons amenés à définir d'autres concepts sur les automates à entrées et sorties, comme les automates minimaux (voir chapitre 5) ou les exécutions équitables sur un automate (voir chapitre 9). Pour ne pas surcharger le lecteur de définitions dès ce premier chapitre, nous présenterons ces concepts seulement lorsque nous en aurons besoin.

2.5 Blocages de sortie et actions observables

Les automates à entrées et sorties peuvent comporter des états à partir desquels il n'existe aucune transition étiquetée par une émission. Nous les appelons des états de *blocage de sortie*, reprenant une terminologie de [Tretmans 92b], en l'appliquant aux IOSM.

Nous verrons par la suite, lors de l'étude de la mise en œuvre concrète des tests (chapitre 9), qu'il est important de savoir si l'absence d'émission d'une implantation est quelque chose d'observable. Cela a en effet une influence directe sur la possibilité de concevoir des testeurs donnant un verdict après un temps fini. Or comme nous l'indiquons à la section 2.1, l'évolution du temps n'est pas représentée dans les automates à entrées et sorties. Il est donc impossible de modéliser explicitement le fait que l'on «attend pour voir si aucune émission ne va se faire».

Dans ces conditions, la seule façon d'inclure dans notre modèle le fait que l'on puisse observer ces absences de sorties est d'en faire une action observable à part entière. Ainsi, c'est le modélisateur, au moment où il représente la spécification (et les implantations possibles) sous forme d'automates à entrées et sorties, qui choisit si l'absence de réponse est observable dans la configuration réelle de test étudiée. En pratique, c'est souvent le cas : on suppose généralement que si l'implantation reste silencieuse pendant un délai suffisant, plus rien ne se passera ensuite. Donc on conclut à l'absence de réponse après avoir attendu en vain pendant le délai en question.

Nous désignons par l'action spéciale $!\delta$ une absence de réponse d'une spécification dans un état donné. Dans le cas où nous choisissons de modéliser le fait que cette absence de réponse est observable, nous rajoutons cette action à l'ensemble des actions, et nous rajoutons à tout état de blocage de sortie s une transition $(s, !\delta, s)$. Cette modélisation explicite est représentée sur la figure 6 ci-dessous.

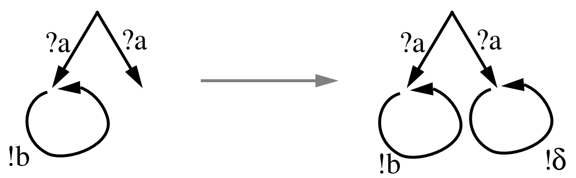


Figure 6 : modélisation explicite des blocages de sortie

Remarquons qu'avec ce choix de modélisation, si l'on suppose que toute absence de réponse est observable, les automates que l'on manipule n'ont plus *aucun état de blocage de sortie*. De plus, l'information sur les possibilités de blocage est conservée dans l'automate de traces, comme illustré par la figure 7 :

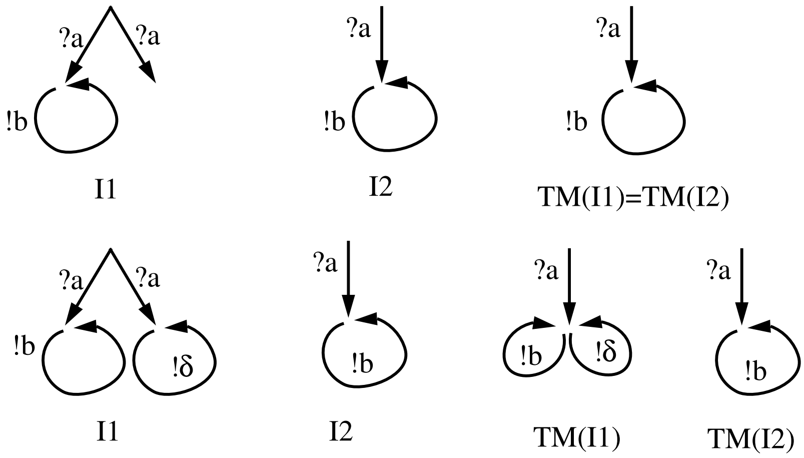


Figure 7 : automates de traces et blocages de sortie

Dans la suite de cette thèse, à l'exception du chapitre 9, la plupart des résultats obtenus porteront sur les traces des automates, sans considérer l'action $!\delta$ comme différente des autres. Par contre, pour la mise en œuvre des tests au chapitre 9, le rôle particulier de $!\delta$ sera bien évidemment pris en compte.

2.6 Comparaison avec d'autres modèles d'automates

Le choix du modèle d'automates à entrées et sorties est très important puisqu'il sert de fondement à toute l'étude exposée dans cette thèse. Il est donc nécessaire de bien justifier ce choix par rapport aux autres modèles couramment employés dans les études sur le test.

Comme nous l'avons dit précédemment, notre modèle est une variante des systèmes de transitions étiquetées, qui sont un des modèles de référence dans les études sur le test [Brinksma 88]. Nous avons seulement adapté ce modèle pour représenter des systèmes communiquant par entrées et sorties, au lieu du mécanisme de rendez-vous négocié généralement associé aux systèmes de transitions étiquetées. Cette proximité entre notre modèle et les systèmes de transitions étiquetées facilite grandement le lien avec les théories du test fondées sur ce dernier modèle, comme nous le verrons au chapitre suivant.

Récemment [T retmans 92b] [V erhaard 92] ont proposé de baser l'étude du test de systèmes communiquant par entrées et sorties au travers de files d'attente sur la version originale des systèmes de transitions étiquetées. Cette approche présente selon nous l'inconvénient de lier le concept d'entrée et de sortie, c'est à dire de communication asymétrique où la notion de refus n'existe pas, avec le concept de file d'attente, qui pour nous n'est pas un mécanisme primitif de la communication, mais une des formes particulières que peut prendre l'interface entre le testeur et l'implantation, comme nous le verrons au chapitre 4. Notre approche est plus proche du modèle des *I/O automata* de [Lynch 87], qui définit également une communication asymétrique basée sur des entrées et sorties comme une propriété des refus, indépendamment d'éventuelles files d'attentes.

Cependant, de nombreux travaux sur le test [Chow 78] [Fujiwara 91a] [Sidhu 89] portant sur des systèmes à entrées et sorties utilisent un autre modèle d'automates, les *machines de Mealy* [Hopcroft 79]. Au sens strict, ces automates, tels qu'ils sont définis dans la référence ci-dessus, sont complets (au sens utilisé dans cette thèse, i.e. vis à vis des interactions d'entrée) et déterministes (au sens du test), mais on trouve dans de nombreux articles des extensions de ce modèle aux automates incomplets ou non déterministes [Luo 93].

La différence principale entre les automates à entrées et sorties et les machines de Mealy est que dans ces dernières une transition est étiquetée par un couple (entrée, sortie). Historiquement, ces automates ont été conçus comme modèles pour étudier les propriétés de circuits électroniques ou les sorties sont synchrones avec les entrées [Gill 62]. Cependant ce choix nous semble moins opportun pour représenter des systèmes de télécommunications comme des réseaux ou des protocoles pour lesquels une telle hypothèse de synchronisme n'est pas réaliste. De plus, notre expérience concrète de la modélisation de tels systèmes montre que l'on a besoin de représenter des états ou un choix est offert entre une entrée et une sortie (la collision de messages dans les protocoles par exemple). Cette situation ne peut être modélisée au moyen de machines de Mealy qu'en admettant que l'interaction vide puisse apparaître aussi bien pour les entrées que pour les sorties. Dans ce cas, une traduction triviale peut être faite entre les deux types d'automates, comme illustré par la figure 8. Une telle traduction sera étudiée de façon formelle au chapitre 5.

Figure 8 : correspondance entre machines de Mealy et automates à entrées et sorties

Les limitations des machines de Mealy pour la description des protocoles ont été analysées en détail dans [Guitton 84]. Elles constituent la principale raison pour laquelle nous avons choisi un autre modèle comme base de notre étude.

3 Modéliser la conformité : relations d'implantation

3.1 Deux représentations de la conformité

Tester a pour but de pouvoir répondre à la question : une implantation I est-elle conforme à une spécification S ? Pour pouvoir donner un sens précis à cette question, il faut définir de façon formelle ce que l'on entend par conformité. Ce problème est étudié en détail dans [Tretmans 92b]. Deux approches sont examinées : le modèle logique et le modèle de comportement.

Dans le modèle *logique*, une spécification est un ensemble de propriétés, i.e. un ensemble de formules du formalisme logique choisi, ce que l'on a coutume d'appeler une théorie. La sémantique du formalisme logique est fournie par une relation de satisfaction, notée $\models$. Elle relie une classe d'objets mathématiques, les modèles du langage, avec les formules du langage, exprimant le fait qu'une formule est vraie pour un modèle particulier. Dans cette approche, les implantations correspondent aux modèles. La relation de satisfaction définit quels modèles satisfont une théorie donnée, c'est à dire quelles implantations sont conformes à la spécification. Cette représentation de la conformité permet une interprétation directe des concepts de la norme 9646 [ISO 91].

Dans le modèle de *comportement*, les spécifications et les implantations sont représentées par une expression décrivant leurs comportements observables respectifs dans des langages de spécifications formels (comme Estelle, LOTOS ou LDS), ou directement par des modèles opérationnels de ces comportements, comme les automates d'états finis ou les systèmes de transitions. La conformité est alors définie comme une relation, dite relation d'implantation, entre les objets modélisant les implantations et ceux modélisant les spécifications. L'équivalence entre ces deux représentations de la conformité est montrée dans [Tretmans 92b].

Dans le cas général, il n'y a aucune raison particulière pour qu'un même formalisme représente les implantations et les spécifications. Dans cette thèse, à l'exception du chapitre 9, nous utiliserons cependant le même modèle, les automates à entrées et sorties, pour représenter ces deux sortes d'objets. Si ce choix paraît assez naturel en ce qui concerne les spécifications, qui sont des objets purement abstraits, il n'en est pas de même pour les implantations. En effet, celles-ci sont des objets physiques (équipements de télécommunications, réseaux), dont on ne peut que fournir un modèle pour les besoins de notre étude. Un modèle trop détaillé n'étant pas réalisable (on ne va quand même pas décrire un téléphone comme l'ensemble des atomes qui le constituent), toute la difficulté est de trouver un niveau pertinent de détail qui représente les propriétés ayant une influence sur le test sans introduire de phénomènes superflus. Nous poserons donc pour l'instant :

Définition 2.9 : une *relation d'implantation* est une relation R sur $IOSM \times IOSM$. Soient $I, S \in IOSM$. Si $R(I, S)$ on dit que I est une implantation *conforme* de S.

Nous verrons cependant au chapitre 9 que le modèle d'automates à entrées et sorties n'est pas suffisant pour représenter les exécutions équitables requises lorsque l'on teste en un temps fini des implantations pouvant être non déterministes. Nous serons amenés alors à utiliser un modèle plus sophistiqué pour les implantations.

Insistons encore une fois sur deux idées qui nous paraissent essentielles :

1. *une spécification seule n'est d'aucune utilité* si elle n'est pas associée à une définition précise de la conformité. Spécifier sert avant tout à étudier des propriété générales, abstraites, des futures implantations à réaliser . Toute spécification doit donc permettre de caractériser les implantations qui lui correspondent (ceci est développé dans [Cavalli 93a]). Dans un contexte plus général que le test, si l'on se réfère à un modèle de développement de logiciel qui procède par raffinements successifs de spécifications, jusqu'à aboutir à l'implantation finale, il est nécessaire de caractériser à chaque étape quelles sont les transformations autorisées entre une spécification plus abstraite et la spécification plus détaillée qui la précise. L'importance de ces transformations, ainsi que les propriétés minimales qu'elles doivent satisfaire, est étudié dans [Leduc 90].
2. *le choix d'une relation d'implantation est arbitraire* , bien que généralement quelques propriétés minimales à respecter soient requises. Dans [T retmans 92b] ou [SPECS 91] plusieurs relations d'implantation adaptées à des systèmes décrits en LOTOS sont présentées, le deuxième de ces documents montrant même que selon l'usage que l'on veut faire de la spécification certaines relations d'implantation sont plus pertinentes que d'autres. Dans [Kristoffersen 92] diverses relations d'implantation adaptées au langage LDS sont définies.

3.2 Quelques relations d'implantation

Nous définissons dans cette section quelques relations d'implantation, adaptées aux automates à entrées et sorties, que nous utiliserons dans cette thèse. Ces exemples illustrent bien la variété des choix arbitraires possibles. Le lecteur pourra aussi se reporter au chapitre suivant où sont définies quelques relations d'implantation en usage dans le monde des systèmes de transitions communiquant à la LOTOS.

Nous nous plaçons dans une situation de test où tout ce que l'on peut faire c'est envoyer des interactions vers une boite noire à tester , et analyser les réponses fournies par la boite noire. Une première idée est de dire qu'une implantation I est conforme à une spécification S si les réponses fournies par *I ne contredisent jamais* ce qui est prévu par S, quand quelque chose est prévu par S. On ne cherche pas à savoir ce qui se passe quand on envoie vers I des interactions appartenant au comportement non spécifié dans S (si S est incomplète).

Pour $\sigma \in Tr(S)$, notons $O(\sigma, S) = \{a \in L \mid \sigma!a \in Tr(S)\}$ les sorties autorisées par la spécification après la trace σ.

Définition 2.10 : $R_1(I, S)$ ssi $(\forall \sigma \in Tr(S))\ (\sigma \in Tr(I) \Rightarrow O(\sigma, I) \subseteq O(\sigma, S))$

Cette relation d'implantation est illustrée par les exemples de la figure 9. Elle n'autorise les extensions de fonctionalité que pour la partie non décrite par la spécification,

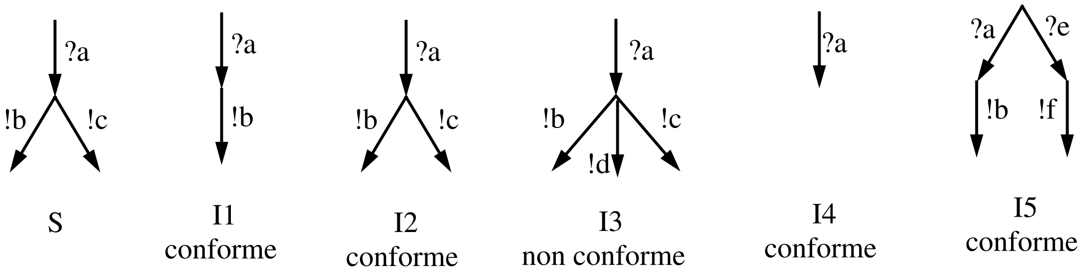


Figure 9 : relation d'implantation R_1

comme le montrent la non-conformité de l'implantation I3 et la conformité de I5. Par contre, elle garantit que tout ce qui peut sortir de l'implantation en réponse à une entrée donnée a été prévu par la spécification. Cependant, cette relation autorise une implantation à ne rien répondre même si la spécification prévoit une ou plusieurs réponses possibles (voir l'exemple I4). Ce qui peut ne pas toujours paraître très naturel. On peut alors considérer :

Définition 2.1 1 : $R_2(I, S)$ ssi
$(\forall\sigma \in Tr(S))\ (\sigma \in Tr(I) \Rightarrow \{O(\sigma, I) \subseteq O(\sigma, S) \wedge (O(\sigma, I) = \emptyset) \Leftrightarrow (O(\sigma, S) = \emptyset)\})$

Cette nouvelle relation d'implantation ne change rien à la conformité des exemples I1 et I2 ni à la non conformité de I3, par contre l'implantation I4 n'est plus conforme à la spécification. A vec cette relation, une implantation est conforme si son non-déterminisme au sens du test est réduit (i.e. moins de sorties possibles) par rapport à celui de la spécification. La réduction du non-déterminisme est une caractéristique courante des relations d'implantation (voir aussi le chapitre suivant). Elle rejoint des notions de raffinements employées pour la conception de systèmes séquentiels qui consistent à affaiblir les préconditions et renforcer les postconditions. Remarquons que dans le cas où les blocages de sortie sont modélisés explicitement, cette deuxième relation d'implantation est équivalente à la première.

Remarquons également que ces définitions sont légèrement compliquées par le fait que les spécifications peuvent être incomplètes. Elles signifient que nous nous désintéressons du comportement de l'implantation en réponse aux entrées qui ne sont pas spécifiées pour un état donné de la spécification. Dans notre modèle en effet, une relation d'implantation sur les traces définie directement comme $R(I, S) \Leftrightarrow Tr(I) \subseteq Tr(S)$ n'est pas pertinente si la spécification n'est pas complète, puisque les implantations sont supposées complètes : pour $\sigma \in Tr(S)$ on peut très bien avoir $\sigma?a \in Tr(I)$ et $\sigma?a \notin Tr(S)$ car cette réception n'est pas spécifiée.

Pour plus de sûreté, nous pouvons imposer un comportement par défaut dans ce cas, par exemple ne rien faire. À partir d'une spécification quelconque S, nous construisons une spécification complète $\hat{S}$ en rajoutant toutes les transitions $(s, ?\mathrm{a}, s)$ pour toutes les réceptions non prévues dans tous les états s. On peut alors dire qu'une implantation I est conforme à une spécification S si les réponses fournies par I ne contredisent jamais ce qui est prévu par la spécification complétée $\hat{S}$. Soit :

Définition 2.12 : $R_{1bis}(I, S)$ ssi $Tr(I) \subseteq Tr(\hat{S})$

Ou dans sa version faisant cas particulier de l'absence de réponses :

Définition 2.13 : $R_{2bis}(I, S)$ ssi
$Tr(I) \subseteq Tr(\hat{S}) \wedge (\forall\sigma \in Tr(\hat{S}))\ (\sigma \in Tr(I) \Rightarrow (O(\sigma, I) = \emptyset) \Leftrightarrow (O(\sigma, S) = \emptyset))$

Selon ces nouvelles relations d'implantation I5 n'est pas conforme à la spécification S. Il est tout à fait possible d'avoir l'avis contraire quant au sens pertinent de l'inclusion à considérer , et de choisir l'une des deux relations d'implantation suivantes :

Définition 2.14 : $R_3(I, S)$ ssi $Tr(S) \subseteq Tr(I)$

Définition 2.15 : $R_4(I, S)$ ssi
$Tr(S) \subseteq Tr(I) \wedge (\forall\sigma \in Tr(S))\ ((O(\sigma, I) = \emptyset) \Leftrightarrow (O(\sigma, S) = \emptyset))$

Ces deux relations expriment que l'implantation doit avoir au minimum toutes les capacités prévues par la spécification, mais a la liberté d'en faire plus. Cette fois, c'est I1 qui n'est pas conforme à S, et I3 qui devient conforme.

Enfin, nous utiliserons souvent dans cette thèse une relation qui exige que l'implantation fasse exactement ce qui est prévu par la spécification, ni plus, ni moins, du moins pour la partie effectivement spécifiée. C'est la conjonction des relations R_2 et R_4 (ou R_1 et R_3) ci-dessus, qui s'écrit directement :

Définition 2.16 : $R_5(I, S)$ ssi $(\forall\sigma \in Tr(S))\ (\sigma \in Tr(I) \wedge (O(\sigma, S) = O(\sigma, I)))$

Parmi les implantations représentées sur la figure 9, seule I2 est conforme à S selon R_5. Sa version pour les spécifications complètes ou complétées est :

Définition 2.17 : $R_{5bis}(I, S)$ ssi $Tr(I) = Tr(\hat{S})$. On notera aussi $R_t(I, S)$ l'équivalence de traces $Tr(I) = Tr(S)$

Historiquement, les études portant sur des modèles à communication par rendez-vous, comme [Brinksma 88], privilégient les relations d'implantation qui s'expriment comme des préordres. C'est le cas des relations R_1 à R_4 ci-dessus. Bien sûr , les relations que nous présentons ici sont adaptées aux systèmes communiquant par entrées et sorties, mais nous pourrons établir une relation formelle avec les relations d'implantation en usage dans le monde LOTOS au chapitre suivant.

Au contraire, la relation la plus étudiée pour le modèle des machines de Mealy est l'équivalence de traces. Si l'on autorise des spécifications incomplètes, cette relation doit être naturellement modifiée pour aboutir à la relation R_5 ci-dessus, que nous avons proposée pour la première fois dans [Phalippou 92], et qui est reprise dans [Luo 93] sous une forme voisine.

On peut légitimement se demander si l'expérience concrète de la spécification et du test de systèmes fournit des arguments pour privilégier l'une de ces différentes relations. En pratique, toutes peuvent présenter un intérêt. Dans la première partie de cette thèse, ce sont surtout les relations R_1 et R_t qui vont intervenir, ainsi que de nouvelles relations, plus faibles, prenant en compte l'influence de l'interface de test. Puis dans la deuxième partie relative à la sélection des tests, la relation R_5 jouera un rôle prépondérant, cette différence étant due principalement à des raisons techniques. Par contre, nous présenterons au chapitre 5 des relations très différentes qui se révéleront beaucoup trop faibles pour pouvoir être utilisées telles quelles en pratique.

4 Modéliser le test : testeurs

4.1 Des suites de tests aux testeurs

Après avoir modélisé les spécifications et les implantations, ainsi que la notion de conformité, nous étudions maintenant comment représenter l'activité de test proprement dite.

Dans la pratique, le test d'un système se fait généralement au moyen d'une suite de tests. Chaque test est défini pour vérifier un but de test bien précis (ceci est vrai au moins sans le cas qui nous intéresse, c'est à dire pour le test de systèmes de télécommunications tel qu'il est pratiqué au CNET. Ce n'est pas la même chose pour des tests en série de circuits intégrés, par exemple). Un test consiste en une procédure finie d'échanges d'interactions entre le testeur et l'implantation testée. Il doit aboutir à un verdict. La norme ISO 9646 [ISO 91], dans le cadre particulier du test de conformité de protocoles, propose des définitions pour ces différents concepts. En particulier, l'exécution d'un test peut conduire à trois sortes de verdicts : *pass* (réussi), *fail* (raté) et *inconclusive* (inconcluant). Ce dernier verdict étant réservé à des exécutions de tests qui, du fait d'un possible non-déterminisme des implantations testées, n'ont pu aboutir à la vérification du but de test, sans pour autant révéler un comportement erroné, c'est à dire en contradiction avec la spécification. Il est alors possible de réessayer le test inconcluant pour tenter d'obtenir l'un des deux autres verdicts.

De cette analyse rapide il ressort que le test global d'un système est décomposé en l'application d'un certain nombre de *cas de tests* individuels, qui eux mêmes sont susceptibles d'être *exécutés* un certain nombre de fois pour aboutir à l'assignation d'un *verdict*. Ces différents concepts sont formalisés dans [T retmans 91].

Une approche différente nous conduit au contraire dans cette thèse à considérer de façon globale l'ensemble des échanges de messages entre le testeur et l'implantation correspondant à toutes les exécutions nécessaires de tous les tests d'une suite de tests donnée. Selon ce principe nous définissons un objet unique, que nous appelons un *testeur*, pour représenter cette vision globale et opérationnelle (car elle inclut les exécutions de tests) du test. Cette approche est celle suivie dans [Brinksma 88] par exemple.

Nous avons choisi ici cette modélisation de l'activité de test car elle offre l'avantage de représenter de façon homogène, par un objet unique, la définition opérationnelle du test, sans devoir passer par les étapes de suite de tests, cas de tests et exécutions d'un cas de test. En particulier, elle inclut dans l'étude des testeurs l'étude de la phase d'exécution des tests (celle-ci sera détaillée au chapitre 9), qui est très peu souvent prise en compte dans les diverses théories connues. De plus cette modélisation est bien dans l'esprit de l'approche algébrique choisie dans cette thèse.

Il faut cependant reconnaître qu'elle présente des inconvénients qui sont la contrepartie inévitable des avantages ci-dessus : il est impossible d'étudier les concepts qui ne sont pas définis dans ce modèle, donc en particulier comment relier les cas de tests individuels aux buts de tests, comment rajouter des tests pour compléter la couverture d'une suite, comment associer le résultat d'un test à un diagnostic, comment interpréter le résultat d'une suite de tests non plus de façon binaire mais comme une collection de résultats des tests individuels, etc. De plus, le testeur étant modélisé par un objet mathématique unique, nous ne pourrons pas étudier les problèmes liés à une exécution distribuée du test, au moyen de différents testeurs agissant sur différentes interfaces (voir la conclusion du chapitre 10).

Pour mettre en œuvre concrètement notre approche, il nous faut définir les testeurs, ainsi que la façon d'associer un verdict de test à un couple (testeur, implantation). Par souci d'homogénéité avec les spécifications et les implantations, nous modélisons les testeurs par des automates à entrées et sorties. Le mécanisme d'assignation du verdict de test est basé sur une vision opérationnelle du test, qui est représenté par une exécution en parallèle du testeur avec l'implantation. Le verdict est calculé en fonction des propriétés de l'automate qui représente cette composition parallèle.

Dans cette thèse, nous construirons deux sortes de testeurs ayant deux mécanismes différents d'assignation de verdicts :

1. les premiers, dits testeurs à verdict *global* (l'explication de ce nom apparaît au chapitre 9), dont nous présentons un exemple à la section suivante, sont des IOSM ayant un état particulier noté *fail* qui indique que le testeur a détecté une erreur. L'échec du testeur T appliqué à l'implantation I est défini par :
   $Fail(T, I)$ ssi $(\exists\sigma \in Tr(T))\ (\bar{\sigma} \in Tr(I) \wedge (T, \sigma, fail))$
   (autrement dit, d'après la propriété 2.2, si un état de la forme $(fail, s)$ appartient au graphe de $T \parallel I$). Le succès $Succ(T, I)$ est défini comme la négation de l'échec.
2. les seconds, dits testeurs à verdict *local*, que nous étudierons surtout au chapitre 9, sont des IOSM ayant aussi un état particulier noté *pass* qui indique que le testeur a correctement vérifié la conformité de l'implantation. Le verdict du testeur T appliqué à l'implantation I est défini par :
   $Succ(T, I)$ ssi $(\exists\sigma \in Tr(T))\ (\bar{\sigma} \in Tr(I) \wedge (T, \sigma, pass))$
   $Fail(T, I)$ ssi $(\exists\sigma \in Tr(T))\ (\bar{\sigma} \in Tr(I) \wedge (T, \sigma, fail))$

(autrement dit, d'après la propriété 2.2, il y a succès si un état de la forme $(pass, s)$ appartient au graphe de $T \parallel I$ et échec si un état de la forme $(fail, s)$ appartient au graphe de $T \parallel I$).

4.2 Comparaison avec des approches plus générales

Nous avons choisi de définir le verdict d'un test comme une propriété de l'automate composé $T \parallel I$, car cela nous paraît modéliser de façon adéquate la pratique du test. Cependant il faut noter que ce n'est que l'une des méthodes possibles pour définir , d'un point de vue théorique, le résultat de l'application d'un test à une implantation. Ce choix introduit des limitations à ce qu'il est possible de vérifier au moyen du test, comme nous le verrons à la section 6 de ce chapitre. Nous présentons brièvement ici une autre approche [Abramsky 87] [De Nicola 84], plus puissante, et les raisons qui nous ont conduit à ne pas l'adopter .

On se donne un ensemble T représentant les tests, un ensemble Q représentant les implantations, et un ensemble V représentant les verdicts. Ici le verdict d'un test doit être compris comme le résultat global de toutes les exécutions possibles de ce test. La signification d'un test (sa sémantique en langage de théoriciens) consiste à définir une fonction $O: T \times Q \to V$, qui indique quel est le verdict résultant de l'application d'un test à une implantation.

Concrètement, les implantations sont des processus, identifiés aux états d'un système de transitions étiquetées. Les verdicts sont l'ensemble $\{\{\bot\}, \{\bot, T\}, \{T\}\}$: le premier élément est interprété comme *fail*, le troisième comme *pass*, le second est la réunion des deux. En fait, le verdict d'un test doit être compris comme étant le bilan de tous les verdicts de toutes les exécutions de ce test, le verdict d'une exécution pouvant être soit *fail* $(\bot)$ soit *pass* (T). La syntaxe des tests est définie par des expressions de test :

$$t ::= \mathrm{SUCC} \mid \mathrm{FAIL} \mid at \mid \tilde{a}t \mid \varepsilon t \mid t_1 \wedge t_2 \mid t_1 \vee t_2 \mid \forall t \mid \exists t$$

Il est alors possible de définir la sémantique des tests de façon dénotationnelle au moyen d'un ensemble de règles qui précisent comment agissent les opérateurs composant une expression de test. Par exemple :

$$\begin{aligned} O(\mathrm{SUCC}, p) &= \{T\} \\ O(t_1 \wedge t_2, p) &= O(t_1, p) \wedge O(t_2, p) \end{aligned}$$

Nous ne donnerons pas ici toutes les règles formulées dans [Abramsky 87], mais nous nous contenterons d'une analyse succincte du test ainsi modélisé. La puissance du test dépend dans cette approche des opérateurs qui sont utilisés pour composer les expressions de test. [Abramsky 87] montre qu'au moyen des tests définis ci-dessus il est possible de caractériser les processus modulo l'équivalence observationnelle [Hennessy 80], ce qui constitue un pouvoir d'observation très fort.

Toute la difficulté, pour mettre en pratique les tests définis ainsi, est de trouver une procédure opérationnelle (i.e. qui puisse correspondre à un algorithme effectif applicable dans une situation de test réelle) pour implanter les opérateurs de test proposés. Si certains sont facilement mis en œuvre, d'autres sont plus complexes à réaliser . Par exemple :

1. at teste que l'implantation peut faire l'action a, puis satisfaire le reste du test t. Dans un environnement où la communication se fait par rendez-vous, cet opérateur est implanté en proposant simplement l'action a à l'implantation.
2. $t_1 \wedge t_2$ ou $t_1 \vee t_2$ signifient que le testeur est capable de combiner les résultats de deux tests s'appliquant à l'état actuel de l'implantation. Pour implanter ces opérateurs, il faut donc être capables d'obtenir à n'importe quel instant des copies de l'implantation testée dans l'état où elle se trouve, pour pouvoir faire séparément un test sur chaque copie, puis combiner les résultats. C'est déjà relativement difficile à mettre en œuvre dans un contexte de test boite noire.
3. $\forall t$ ou $\exists t$ supposent que l'on peut collecter , à un certain point du test, des informations globales portant sur *toutes* les exécutions ultérieures possibles. Implanter en pratique ces opérateurs présente de grandes difficultés. Par exemple, dans le cas où les implantations peuvent être non déterministes, ils signifient que l'on est capable d'observer toutes leurs évolutions possibles. Nous verrons au chapitre 9 comment en pratique cet opérateur n'est réalisé qu'en restreignant le domaine des implantations possibles.

En conclusion, cette approche, puissante et générale, nécessiterait pour être complète l'étude de la mise en œuvre pratique des tests qui sont définis. Au contraire, représenter l'exécution d'un test comme la composition parallèle du testeur et de l'implantation présente l'avantage de pouvoir être mis immédiatement en pratique, surtout si cette composition parallèle utilise un mécanisme de communication par entrées et sorties, qui est généralement le seul mécanisme que l'on sait mettre en œuvre pour dialoguer avec une boite noire.

Le seul problème restant alors pour transformer un testeur en un algorithme est de savoir calculer le verdict de test. Ceci sera étudié au chapitre 9.

5 L'exemple du testeur canonique asynchrone

Suivant la démarche adoptée dans [Brinksma 88], nous présentons dans cette section un premier exemple complet d'application de la théorie du test que nous avons développé. C'est à dire que nous choisissons une relation d'implantation particulière, à savoir la relation R_1 (définition 2.10), et nous montrons comment, à partir de toute spécification calculer un testeur qui caractérise les implantations conformes à cette spécification au sens de la relation R_1.

Pour des raisons historiques (parallèle avec [Brinksma 88], publication antérieure de ce résultat dans [Phalippou 92]), nous appelons ce testeur le *testeur canonique asynchro-*

ne. Remarquons toutefois que ce nom est particulièrement mal choisi : le terme «canonique» emprunté à l'article de Brinksma est un peu prétentieux, puisqu'en réalité ce testeur caractérise seulement une relation d'implantation particulière, ce qui relativise son caractère universel. De plus le terme «asynchrone» introduit dans mon article pour différencier mon testeur de celui de Brinksma est bien mal adapté, puisque la communication entre le testeur et l'implantation se fait par des entrées et des sorties parfaitement synchrones !

5.1 Définition du testeur

Définition 2.18 : soit S une spécification et $TM(S) = \langle S_S, L_S, T_S, s_{0S} \rangle$ son automate de traces. On appelle *testeur canonique asynchrone* de S, une IOSM notée $TCA(S) = T = \langle S_t, L_t, T_t, s_{0t} \rangle$ telle que :

1. $S_t = S_S \cup \{fail\}$
2. $L_t = L_S$
3. $s_{0t} = s_{0S}$
4. les transitions du testeur sont exactement celles obtenues par les règles suivantes :
   a) $(\forall \mu \in \{!, ?\} \times L_S)\ (\forall s, s' \in S_S)\ ((s, \mu, s') \in T_S \Leftrightarrow (s, \bar{\mu}, s') \in T_t)$
   b) $(\forall s \in S_S)\ (\forall a \in L_S)\ (\neg (\exists s')\ ((s, !a, s') \in T_S) \Rightarrow (s, ?a, fail) \in T_t)$

Le verdict fourni par ce testeur est défini comme un verdict global :

Définition 2.19 : $Fail(T, I)$ ssi $(\exists \sigma \in Tr(T))\ (\bar{\sigma} \in Tr(I) \wedge (T, \sigma, fail))$
$Succ(T, I)$ ssi $\neg (Fail(T, I))$

Intuitivement, le testeur canonique asynchrone de S est chargé de vérifier que rien de contraire à ce qui est prévu ne peut arriver. Pour cela, il fournit une image miroir des traces de la spécification (la seule chose qu'il puisse observer, comme nous le verrons à la section 6). On rajoute un mécanisme de détection des erreurs : si le testeur reçoit une interaction qui n'est pas censée pouvoir arriver dans un état donné, il passe dans l'état *fail*. On retrouve donc ce qui est fait dans beaucoup d'approches du test de systèmes communiquant par émissions et réceptions, à savoir une inversion des entrées et des sorties pour obtenir le test à partir de la spécification [Hogrefe 88] [Rouger 89].

La structure obtenue par inversion de l'automate de traces détermine complètement le testeur. Celui-ci est ensuite complété (au sens de la complétude des IOSM de la définition 2.3) par le rajout des transitions de détection d'erreurs. On peut donc parler du testeur en considérant uniquement la structure principale (sans les transitions de détection d'erreurs). Dans ce cas, de la définition comme inversion des entrées et des sorties sur l'automate de traces, il découle :

Propriété 2.5 : $R_t(TCA(TCA(S)), S)$

Le testeur canonique du testeur canonique est la spécification originale (à l'équivalence de traces près) : cette propriété est l'homologue du théorème 2.7 de [Brinksma 88].

5.2 Propriétés du testeur

Nous montrons maintenant pourquoi ce testeur est appelé canonique : c'est parce qu'il permet, parmi toutes les IOSM, de distinguer celles qui sont des implantations conformes de la spécification, au sens de la relation d'implantation R_1.

Le testeur canonique asynchrone est déterministe au sens usuel pour les automates, c'est-à-dire qu'à partir d'un état donné il y a une seule transition avec une étiquette donnée. Ceci découle immédiatement du déterminisme de l'automate de traces et de la définition du testeur .

D'après la propriété ci-dessus à chaque trace du testeur on peut faire correspondre l'état (unique) du testeur à la fin de l'exécution de cette trace. On le note $s(\sigma, T)$ ou plus simplement $s(\sigma)$ s'il n'y a pas de doute sur le testeur dont on parle. On cherche ensuite à caractériser les traces du testeur par rapport à celles de la spécification que l'on teste.

Propriété 2.6 : $(\sigma \in Tr(S)) \Leftrightarrow ((\bar{\sigma} \in Tr(T)) \wedge (s(\bar{\sigma}) \neq fail))$

Démonstration : nous ne détaillerons pas ici la démonstration. Comme d'après la propriété 2.4 $Tr(S) = Tr(TM(S))$, la propriété ci-dessus est une conséquence presque immédiate de la définition du testeur . ❑

Nous pouvons maintenant démontrer la propriété principale du testeur :

Propriété 2.7 : $(\forall I, S \in IOSM)\ (Succ(TCA(S), I) \Leftrightarrow R_1(I, S))$

Démonstration : notons $T = TCA(S)$

1. on montre l'implication $Succ(T, I) \Rightarrow R_1(I, S)$ par sa contraposée. Si $R_1(I, S)$ est faux alors d'après la définition de R_1, il existe $\sigma \in Tr(S)$ telle que $\sigma \in Tr(I)$ et $O(\sigma, I) \not\subseteq O(\sigma, S)$. En d'autres termes, il existe $\sigma \in Tr(S) \cap Tr(I)$ et $a \in L$ tels que $\sigma!a \in Tr(I)$ et $\sigma!a \notin Tr(S)$.
   Puisque $\sigma \in Tr(S)$, alors $\bar{\sigma} \in Tr(T)$ d'après la propriété 2.6, donc $\bar{\sigma}?a \in Tr(T)$ par définition du testeur , car $s(\bar{\sigma}) \neq fail$ et tout état du testeur différent de $fail$ est complet pour les entrées. Comme $\bar{\sigma}?a \in Tr(T)$ et $\sigma!a \notin Tr(S)$, on déduit de la propriété 2.6 que $s(\bar{\sigma}?a, T) = fail$.
   Donc il existe une trace $\bar{\sigma}?a \in Tr(T)$ telle que $\sigma!a \in Tr(I)$ et $(T, \bar{\sigma}?a, fail)$, ce qui par définition 2.19 du verdict du testeur signifie $Fail(T, I)$.
2. on montre l'implication $R_1(I, S) \Rightarrow Succ(T, I)$ par sa contraposée. Si $Fail(T, I)$, par définition 2.19 du verdict du testeur il existe une trace $\sigma \in Tr(T)$ telle que $\bar{\sigma} \in Tr(I)$ et $(T, \sigma, fail)$.

D'après la définition de T seule une réception peut amener le testeur à l'état *fail*, donc cette trace peut s'écrire sous la forme $\sigma = \sigma_2?a$, où $\sigma_2 \in Tr(T)$ et $s(\sigma_2) \neq fail$ (car aucune transition du testeur ne part de l'état $fail$).
Puisque $\sigma_2 \in Tr(T)$ et $s(\sigma_2) \neq fail$ alors $\bar{\sigma}_2 \in Tr(S)$ d'après la propriété 2.6. Toujours d'après cette propriété, si $\sigma \in Tr(T)$ et $s(\sigma, T) = fail$ c'est que $\bar{\sigma} \notin Tr(S)$. Nous avons donc montré qu'il existe $\bar{\sigma}_2 \in Tr(S)$ telle que $\bar{\sigma}_2 \in Tr(I)$, $\bar{\sigma}_2!a = \bar{\sigma} \in Tr(I)$ et $\bar{\sigma} \notin Tr(S)$, donc que $R_1(I, S)$ est faux. ❏

5.3 Exemple

Pour illustrer le fonctionnement du testeur canonique asynchrone, reprenons la spécification S de la figure 9. Nous supposons d'abord que nous sommes dans le cas où les blocages de sortie ne sont pas représentés explicitement. Nous représentons sur la figure 10 le testeur correspondant à cette spécification, ainsi que les graphes composés du testeur avec les implantations. L'ensemble des événements est $L = \{a, b, c, d, e, f\}$. Les états des implantations, n'étant pas importants, ne sont pas numérotés.

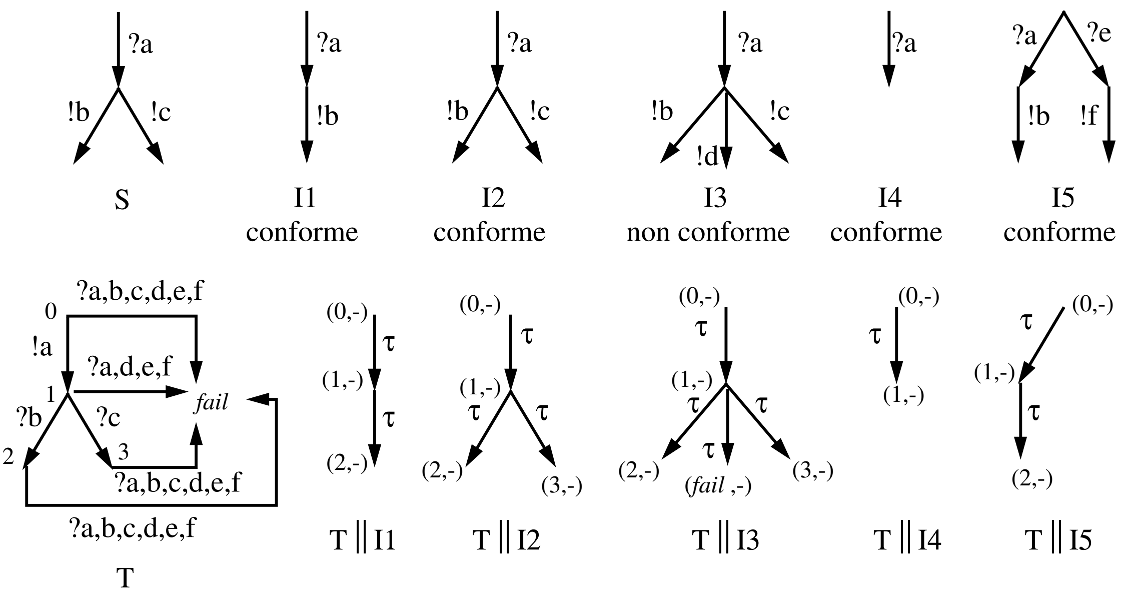


Figure 10 : testeur canonique asynchrone

Cet exemple montre bien que la détermination du verdict revient à savoir si un état de la forme $(fail, s_i)$ se trouve dans le graphe de $T \| I$.

Si les blocages de sortie sont représentés explicitement, alors l'action δ fait partie le l'alphabet : $L = \{a, b, c, d, e, f, \delta\}$. Le testeur est modifié comme indiqué sur la figure 11, et on vérifie bien qu'il détecte la non conformité de l'implantation I6.

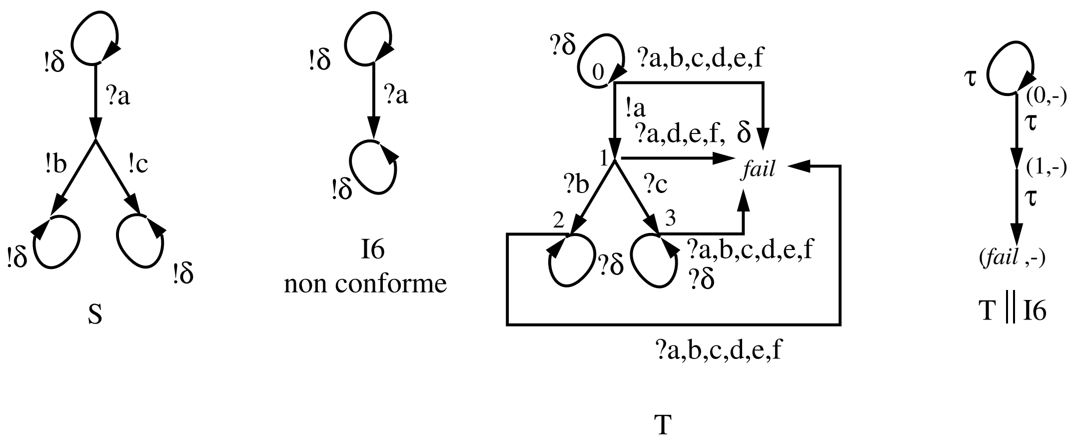


Figure 1 1 : testeur canonique asynchrone (blocages de sortie explicites)

6 Pouvoir maximal de détection du test

Savoir ce qu'il est possible de vérifier au moyen d'un testeur revient à se poser la question suivante : étant donnée une relation d'implantation R et une spécification S, peut-on définir un testeur T (et un verdict associé $Succ(T, I)$) qui caractérise les implantations conformes à la spécification, c'est à dire qui vérifie :

$$(\forall I \in IOSM) \ (Succ(T, I) \Leftrightarrow R(I, S))$$

Nous montrons dans cette section que ce n'est pas possible dans tous les cas. Plus précisément, nous montrons que si l'on garde la même définition du succès du test (définition 2.19) alors une relation d'implantation plus forte que l'équivalence de traces ne peut pas être caractérisée au moyen d'un testeur . Ce résultat montre que le choix d'une procédure particulière de test (définition opérationnelle du succès du test) limite ce qu'il est possible de vérifier au moyen d'un test.

Définition 2.20 : on dit qu'une relation d'implantation R' est *plus forte* qu'une relation R et on note $R' \geq R$ si : $(\forall I, S \in IOSM) \ (R'(I, S) \Rightarrow R(I, S))$

Soit R_t la relation d'implantation correspondant à l'équivalence de traces (définition 2.17). Soit $R' \geq R_t$ une relation d'implantation plus forte que R_t, qui de plus satisfait : pour toute spécification S il existe au moins une implantation I telle que $R'(I, S)$. Soit S une spécification quelconque. Nous allons montrer que quel que soit le testeur T, si T admet comme valides toutes les implantations conformes au sens de la relation R', alors T admet aussi comme valides toutes les implantations conformes au sens de la relation R_t. En d'autres termes, tout testeur qui voudrait caractériser la relation R', et donc qui est assez “large” pour admettre toutes les implantations conformes au sens de cette relation, ne peut pas la caractériser , puisqu'il est trop large : il admet aussi toutes les implantations conformes au sens R_t.

Propriété 2.8 $: \forall T, S \in IOSM,$
$[\,(\forall I \in IOSM)\ (R'\,(I, S) \Rightarrow Succ\,(T, I))\,] \Rightarrow [\,(\forall I \in IOSM)\ (R_t\,(I, S) \Rightarrow Succ\,(T, I))\,]$

Démonstration : nous prouvons la contraposée de cette implication, à savoir que si il existe I telle que $R_t\,(I, S)$ et $Fail\,(T, I)$ alors il existe I' telle que $R'\,(I', S)$ et $Fail\,(T, I')$. Par définition de l'échec du test, si $Fail\,(T, I)$ alors il existe $\sigma \in Tr\,(T)$ telle que $\bar{\sigma} \in Tr\,(I)$ et $(T, \sigma, fail)$. Puisque I et S ont les mêmes traces, $\bar{\sigma} \in Tr\,(S)$.

Soit I' quelconque telle que $R'\,(I', S)$ (par hypothèse il existe au moins une telle implantation). Par définition de la relation $\geq$, puisque $R' \geq R_t$ et $R'\,(I', S)$ alors $R_t\,(I', S)$. Comme $\bar{\sigma} \in Tr\,(S)$, on en déduit que $\bar{\sigma} \in Tr\,(I')$. Nous avons donc trouvé $\sigma \in Tr\,(T)$ telle que $\bar{\sigma} \in Tr\,(I')$ et $(T, \sigma, fail)$, ce qui par définition signifie $Fail\,(T, I')$. ❑

Intuitivement, ce résultat s'explique par la définition du verdict d'un test, basée sur une propriété du graphe de $T \| I$. Le mécanisme choisi pour la composition parallèle fait que le graphe du système composé dépend uniquement des traces de chacun des composants. Quel que soit le testeur , il ne peut pas faire la différence entre deux implantations ayant les mêmes traces. Une conséquence de ce résultat est que, si l'on garde la même définition du succès du test, nous pouvons considérer les automates modulo l'équivalence de traces. Cela nous permet de supposer que les spécifications, comme les implantations, sont des automates de traces.

Il faut noter que bien évidemment ce résultat est valable dans le cas où on modélise explicitement les blocages de sortie par l'action δ, c'est à dire dans le cas où on suppose que de tels blocages sont observables. Dans ce cas, la limite du pouvoir d'observation du test correspond à une *équivalence de traces-blocages* (*trace deadlock equivalence* en anglais).

CHAPITRE 3

Test de conformité et mécanisme de communication

1 Introduction

Nous avons construit au chapitre précédent une théorie du test basée sur des automates communiquant par entrées et sorties. L'activité de test elle même étant représentée par l'exécution en parallèle du testeur et de l'implantation, le mécanisme de communication entre les automates a une influence directe sur le pouvoir d'observation du test, comme le montre la propriété 2.8.

Une question se pose donc : le choix d'une communication par entrées et sorties est-il pertinent ? Nous avons, dans le chapitre précédent, justifié ce choix par des considérations très empiriques, arguant qu'à notre avis ce mécanisme représente mieux notre conception intuitive des systèmes de télécommunications que nous nous proposons d'étudier . Mais notre modélisation n'introduit-elle pas un biais par rapport à la réalité ? Ne risque-t-elle pas de nous laisser croire, à tort, que les limitations trouvées sont caractéristiques de l'activité de test, alors qu'elles pourraient très bien n'être dues qu'au modèle ?

Ce chapitre étudie cette question. Devant les difficultés soulevées par une approche trop générale du problème, nous nous bornons à la comparaison entre notre approche et celle de [Brinksma 88]. Celle-ci est une théorie du test basée sur des systèmes de transitions étiquetées qui communiquent par un mécanisme de rendez-vous négocié. Nous rappelons dans un premier temps les principaux résultats de cette théorie, en particulier que le pouvoir d'observation du test est limité par l'équivalence de refus, c'est à dire que le test est plus puissant que celui étudié au chapitre 2.

Nous montrons ensuite au moyen de transformations explicites entre les deux modèles (rendez-vous et entrées et sorties) que les deux théories peuvent être formellement reliées. En particulier , le testeur canonique asynchrone présenté au chapitre précédent est l'image par cette transformation du testeur canonique (appelé ici «synchrone») de [Brinksma 88]. Ces résultats nous amènent à conclure que la notion de conformité réelle testée est exactement la même dans les deux approches, et que seules diffèrent les formalisations mathématiques basées sur des modèles différents. Nous expliquons enfin pourquoi, à notre avis, l'approche fondée sur les entrées et sorties nous paraît plus judicieuse pour l'étude des problèmes de test.

2 Quelques rappels sur le test synchrone

2.1 Relations d'implantation sur les systèmes de transitions étiquetées

Nous avons introduit au chapitre précédent les systèmes de transitions étiquetées (définition 2.1), que nous désignerons ici par l'abréviation anglaise LTS (pour *Labelled Transition System*). Dans ce chapitre, pour pouvoir établir un lien avec les automates à entrées et sorties, qui sont finis, nous nous limitons aux LTS finis, c'est à dire ayant des ensemble d'états et d'événements finis. Un opérateur de composition parallèle est défini sur ces systèmes de la façon suivante :

Définition 3.1 : la *composition parallèle* de deux LTS $I_1 = \langle S_1, L_1, T_1, s_{01} \rangle$ et $I_2 = \langle S_2, L_2, T_2, s_{02} \rangle$ est un LTS $I = \langle S, L, T, s_0 \rangle = I_1 \| I_2$ défini par

1. $S = S_1 \times S_2$
2. $L = L_1 \cap L_2$
3. $s_0 = (s_{01}, s_{02})$
4. les transitions de $I_1 \| I_2$ sont exactement celles qui peuvent être obtenues au moyen des règles suivantes :
   a) $(s_1, \tau, s'_1) \in T_1 \Rightarrow \forall s_2 \in S_2,\ ((s_1, s_2), \tau, (s'_1, s_2)) \in T$
   b) $(s_2, \tau, s'_2) \in T_2 \Rightarrow \forall s_1 \in S_1,\ ((s_1, s_2), \tau, (s_1, s'_2)) \in T$
   c) si $a \in L_1 \cap L_2$ alors
   $(s_1, \mathrm{a}, s'_1) \in T_1 \wedge (s_2, \mathrm{a}, s'_2) \in T_2 \Rightarrow ((s_1, s_2), \mathrm{a}, (s'_1, s'_2)) \in T$

Nous utiliserons les notations suivantes, pour un LTS $I = \langle S, L, T, s_0 \rangle$ et $\sigma = \mu_1 \ldots \mu_n \in L^n$:

1. (s_0, σ, s_n) si $(\exists (s_i)_{1 \le i < n} \in S^n)\ (\forall i, 1 \le i \le n)\ ((s_{i-1}, \mu_i, s_i) \in T)$
2. (I, σ, s_n) si (s_0, σ, s_n)
3. (s_0, ε, s_1) si $s_0 = s_1$ ou $(\exists n \ge 1)\ (s_0, \tau^n, s_1)$
4. $(s_0, \vec{\mu}, s_1)$ si $(\exists s_2, s_3 \in S_s)\ ((s_0, \varepsilon, s_2) \wedge (s_2, \mu, s_3) \wedge (s_3, \varepsilon, s_1))$
5. $(s_0, \vec{\sigma}, s_n)$ si $(\exists (s_i)_{1 \le i < n} \in S_s^n)\ (\forall i, 1 \le i \le n)\ ((s_{i-1}, \vec{\mu}_i, s_i) \in T_s)$
6. $out(s) = \{\mu \in L \mid \exists s' \in S,\ (s, \vec{\mu}, s')\}$

7. $Tr(I) = \{\sigma \in L^* \mid \exists s' \in S,\ (s_0, \vec{\sigma}, s')\}$

L'ensemble des traces est donc défini de façon semblable pour les LTS et les IOSM. Par contre, l'ensemble $out(s)$ des actions offertes dans l'état s est différent de l'ensemble des sorties $O(s)$ défini sur les IOSM, qui ne concerne que les sorties, pas les entrées.

Dans [Brinksma 88] on trouve la définition de la relation *conf* qui est choisie comme relation d'implantation adaptée aux systèmes de transitions étiquetées.

Définition 3.2 : soient I et S deux systèmes de transitions étiquetées. $I\ conf\ S$ ssi

$$(\forall\sigma \in Tr(S))\ (\forall A \subseteq L)$$
$$\{\ (\exists s_I \in S_I)\ (\forall a \in A)\ (\ (I, \sigma, s_I) \wedge a \notin out(s_I))\ \} \Rightarrow$$
$$\{\ (\exists s_S \in S_S)\ (\forall a \in A)\ (\ (S, \sigma, s_S) \wedge a \notin out(s_S))\ \}$$

Cette définition est celle utilisée dans l'article de Brinksma. Afin de faire apparaître une similarité avec la relation R_1 du chapitre précédent, nous introduisons la notion d'ensemble de refus.

On dit que l'ensemble d'actions $A \subseteq L$ est refusé dans l'état s et on note $s\ ref\ A$ si $A \cap out(s) = \varnothing$. L'ensemble des états atteints par un LTS après une séquence $\sigma \in L^*$ est noté $I\ after\ \sigma = \{s \in S \mid (I, \sigma, s)\}$. Notons que cet ensemble est vide si $\sigma \notin Tr(I)$. Le fait que «le LTS I après la séquence σ peut refuser l'ensemble d'actions A» est noté $I\ after\ \sigma\ ref\ A$ ce qui signifie par définition $(\exists s \in I\ after\ \sigma)\ (s\ ref\ A)$. Enfin, l'ensemble des ensembles d'actions qui peuvent être refusées après une séquence σ est défini par $Ref(\sigma, I) = \{A \subseteq L \mid I\ after\ \sigma\ ref\ A\}$. La relation *conf* peut alors être mise sous la forme :

$$I\ conf\ S \text{ ssi } (\forall\sigma \in Tr(S))\ (\sigma \in Tr(I) \Rightarrow Ref(\sigma, I) \subseteq Ref(\sigma, S))$$

ce qui est à rapprocher de la définition :

$$R_1(I, S) \text{ ssi } (\forall\sigma \in Tr(S))\ (\sigma \in Tr(I) \Rightarrow O(\sigma, I) \subseteq O(\sigma, S))$$

En somme, ces relations ont des définitions similaires, sauf que dans un cas on observe les refus et dans l'autre les sorties, différence que nous justifierons par la différence des pouvoirs d'observation permis par les communications par rendez-vous et par entrées et sorties.

2.2 Testeur canonique synchrone

Il est possible de caractériser la relation d'implantation *conf* au moyen d'un testeur, appelé le *testeur canonique* dans [Brinksma 88], et que nous appelons ici le testeur canonique synchrone pour le distinguer du soit-disant testeur canonique asynchrone défini au chapitre précédent.

Le testeur canonique synchrone (nous utiliserons l'abréviation TCS) d'une spécification S est défini à partir du graphe de refus associé au système de transitions étiquetées S. Nous utilisons ici le terme «graphe» de préférence à «arbre», car ce graphe de refus est construit, comme indiqué ci-dessous, à partir de l'automate des traces de S, qui est un graphe. Nous utilisons ici la définition de [Brinksma 88], de préférence aux graphes de refus définis dans [Drira 92], pour faciliter le parallèle avec la théorie du test synchrone. En tout état de cause, ces définitions sont voisines, la différence résidant principalement dans le fait que les graphes de refus de [Drira 92] éliminent certaines redondances dans les structures que nous utilisons ici.

Définition 3.3 : l'*automate de traces* d'un LTS $I = \langle S, L, T, s_0 \rangle$, noté $TM(I)$ (pour *Trace Machine*) est un LTS $TM(I) = CC(\langle S_t, L_t, T_t, s_{0t} \rangle)$ défini par :

1. S_t est l'ensemble des parties de S : un état de l'automate est donc un ensemble d'états de la spécification $s_t = \{s_i\}_{1 \le i \le n}$.
2. $L_t = L$
3. $s_{0t} = \{s | (s_0, \varepsilon, s)\}$
4. les transitions de l'automate de traces sont exactement celles obtenues de la façon suivante : pour tout $s \in S_t$ et $\mu \in L$ soit $s' = \{s_j | (\exists s_i \in s)\ (s_i, \vec{\mu}, s_j)\}$. Si $s' \neq \emptyset$ alors $(s, \mu, s') \in T_t$

Dans cette définition, nous utilisons la composante connexe contenant l'état initial d'un LTS, que l'on note et que l'on définit comme celle d'un IOSM (voir définition 2.5 au chapitre 2). On remarque que cette définition de l'automate de traces d'un LTS est très voisine de celle de l'automate de traces des IOSM. En particulier, on peut montrer que l'automate de traces a les mêmes traces que le LTS initial et qu'il est déterministe au sens des automates. On notera $s(\sigma, I)$ l'état s (unique) de l'automate de traces $TM(I)$ tel que (s_{0t}, σ, s).

Définition 3.4 : le *graphe de refus* d'un LTS $I = \langle S, L, T, s_0 \rangle$, noté $RG(I)$ (pour *Refusal Graph*) est le quintuplet $\langle S_r, L_r, T_r, R, s_{0r} \rangle$ tel que :

1. $\langle S_r, L_r, T_r, s_{0r} \rangle$ est l'automate de traces du système de transitions étiquetées I.
2. $R : S_r \to \wp(\wp(L_r))$ associe à chaque état de S_r un ensemble d'ensembles d'actions appelés les refus, tels que $R(s(\sigma, I)) = Ref(\sigma, I)$.

Les ensembles de refus vérifient les propriétés suivantes (voir la définition 2.5 de [Brinksma 88]) :

Propriété 3.1 : soit $\langle S_r, L_r, T_r, R, s_{0r} \rangle$ un graphe de refus. Alors :

1. $(\forall s \in S_r)\ (R(s) \neq \emptyset)$
2. $(\forall s \in S_r)\ (X \in R(s) \wedge Y \subseteq X \Rightarrow Y \in R(s))$
3. $(\forall s \in S_r)\ (X \in R(s) \Rightarrow X \cup (L - out(s)) \in R(s))$

Nous pouvons maintenant définir le testeur canonique synchrone.

Définition 3.5 : soit S une spécification (un LTS) et $RG(S) = \langle S, L, T, R, s_0 \rangle$ son graphe de refus. On appelle *testeur canonique synchrone* de S, un LTS noté $TCS(S)$ dont le graphe de refus est $RG(TCS(S)) = \langle S, L, T, R', s_0 \rangle$ où les nouveaux ensembles de refus s'obtiennent par :

$$R'(s) = \{L - A \mid A \notin R(s), A \subseteq L\} \text{ si } L \notin R(s)$$

$$R'(s) = \wp(L) \text{ si } L \in R(s)$$

[Brinksma 88] montre que non seulement il existe des systèmes de transitions étiquetées satisfaisant cette définition (i.e. dont le graphe de refus est égal au graphe de refus transformé défini ci-dessus), mais encore que tous les LTS qui la satisfont sont équivalents pour les refus (*failure equivalent*) entre eux.

Comme dans la théorie présentée au chapitre 2, l'activité de test est modélisée par l'exécution en parallèle du testeur avec une implantation, mais bien sûr en utilisant ici la composition parallèle des LTS. Pour le verdict de test, nous n'utiliserons pas ici la formulation introduite dans [Brinksma 88], qui est limitée aux comportements finis, mais plutôt la définition du verdict basée sur les traces et présentée dans [Tretmans 92b]. Suivant cette approche on dit que le test T d'une implantation I est un succès si quand l'exécution parallèle $T \| I$ peut se bloquer après avoir exécuté la séquence σ alors le test T tout seul peut se bloquer après avoir exécuté la séquence σ. Soit, en utilisant les notations introduites dans ce chapitre :

Définition 3.6 : $Succ(T, I)$ ssi $L \in Ref(\sigma, T \| I) \Rightarrow L \in Ref(\sigma, T)$

Sous cette forme il est difficile de se convaincre que le succès du test effectué par le testeur canonique synchrone va correspondre à la conformité exprimée par la relation $conf$. Nous donnons ci-dessous une interprétation plus intuitive qui permet un parallèle plus aisé avec le fonctionnement du testeur asynchrone.

La relation R_1 utilisée dans le cas asynchrone signifie que après une trace de la spécification, les sorties de l'implantation sont toujours prévues par la spécification. Les transitions du testeur aboutissant à $fail$ caractérisent les sorties non prévues par la spécification. Donc l'absence d'état $fail$ du testeur dans le graphe de $T \| I$ caractérise les implantations conformes. C'est la propriété 2.7.

La relation $conf$ utilisée dans le cas synchrone signifie qu'après une trace de la spécification, les refus de l'implantation sont toujours prévus par la spécification. Supposons qu'une implantation ne soit pas dans ce cas. Il existe alors $\sigma \in Tr(S)$ et $X \subseteq L$ tels que $X \in Ref(\sigma, I)$ et $X \notin Ref(\sigma, S)$. Ceci implique que $L \notin Ref(\sigma, S)$, car les ensembles de refus sont clos pour l'inclusion d'après la propriété 3.1 (donc si $L \in Ref(\sigma, S)$ alors $Ref(\sigma, S) = \wp(L)$). Par définition du testeur, on sait alors que $L - X \in Ref(\sigma, T)$, et donc puisque $X \in Ref(\sigma, I)$ on en déduit assez facilement que $L \in Ref(\sigma, T \| I)$. Or d'après la définition du testeur si $L \notin Ref(\sigma, S)$ alors $L \notin Ref(\sigma, T)$, donc le verdict du test sera $Fail(T, I)$. Ceci correspond à la propriété suivante de [Brinksma 88] :

Propriété 3.2 : $(\forall I, S \in LTS)\ (Succ(TCS(S), I) \Leftrightarrow I\ conf\ S)$

2.3 Pouvoir maximal de détection du test

Pour terminer le parallèle avec la théorie du test présentée au chapitre précédent, signalons qu'il existe dans le cas synchrone un résultat similaire à la propriété 2.8, c'est à dire une limite au pouvoir de détection du test. Si l'on garde la définition du succès du test ci-dessus, c'est à dire que l'observe uniquement les refus du graphe composé de $T \| I$, alors [T retmans 92b] montre qu'il est impossible de caractériser au moyen d'un test une relation d'implantation plus forte que l'équivalence de test, connue également sous le nom d'équivalence de refus [Brinksma 88], définie par :

$$I \approx_{te} S \text{ ssi } Tr(I) = Tr(S) \wedge (\forall \sigma \in Tr(I))\ (Ref(\sigma, I) = Ref(\sigma, S))$$

2.4 Une définition équivalente du TCS

La définition que nous avons donnée du testeur canonique synchrone, à partir des graphes de refus, est directement issue de [Brinksma 88]. Dans cette section nous donnons une définition équivalente qui se prête mieux aux développements qui vont suivre dans la suite de ce chapitre. Au lieu de manipuler des graphes de refus, nous introduisons une structure où les ensembles $out(s)$ des actions offertes dans l'état s apparaissent explicitement, et où les ensembles de refus sont représentés par leurs intersections avec $out(s)$.

Soit $F = \langle S, L, T, R, s_0 \rangle$ un graphe de refus. On définit $\tilde{F} = \langle S, L, T, O, \tilde{R}, s_0 \rangle$ par :

1. $O: S \to \wp(L)$ est la fonction de sortie : $O(s) = out(s)$
2. $\tilde{R}: S \to \wp(\wp(L))$ est l'ensemble des intersections avec $O(s)$ des éléments de $R(s)$: $\tilde{R}(s) = \{X \cap O(s) \mid X \in R(s)\}$.

Les deux représentations F et $\tilde{F}$ sont interchangeables, puisque l'on peut obtenir F à partir de $\tilde{F}$ par la transformation sur les ensembles de refus :

Propriété 3.3 : $R(s) = \{X \cup (L - O(s)) \mid X \in \tilde{R}(s)\}$

Démonstration :

1. si $X \in \tilde{R}(s)$ alors $X \in R(s)$ (d'après la propriété 3.1 alinéa 2 et la définition de $\tilde{R}(s)$). Donc $X \cup (L - O(s)) \in R(s)$ (propriété 3.1 alinéa 3).

   Donc $\{X \cup (L - O(s)) \mid X \in \tilde{R}(s)\} \subseteq R(s)$

2. réciproquement si $Y \in R(s)$ alors $Y \cup (L - O(s)) \in R(s)$ (propriété 3.1 alinéa 3) donc $Y \cap O(s) \cup (L - O(s)) \in R(s)$ où $Y \cap O(s) \in \tilde{R}(s)$.
   Donc $R(s) \subseteq \{X \cup (L - O(s)) \mid X \in \tilde{R}(s)\}$. ❏

Il est alors possible de définir le testeur canonique synchrone à partir de la représentation $\tilde{F}$. Si $\tilde{F}(S) = \langle S, L, T, O, \tilde{R}, s_0 \rangle$ est la nouvelle forme du graphe de refus de la spécification, alors celle du graphe de refus du testeur canonique est
$\tilde{F}(TCS(S)) = \langle S, L, T, O, t(O, \tilde{R}), s_0 \rangle$, où la transformation sur les ensembles de refus est définie par :

$$t(O, \tilde{R})(s) = \{O(s) - X \mid X \notin \tilde{R}(s), X \subseteq O(s)\} \text{ si } O(s) \notin \tilde{R}(s)$$

$$t(O, \tilde{R})(s) = \tilde{R}(s) \text{ si } O(s) \in \tilde{R}(s)$$

Le fait que cette définition est équivalente à celle du testeur canonique (définition 3.5) résulte de la propriété suivante :

Propriété 3.4 : pour tout état s, $t(O(s), \tilde{R}(s)) = \tilde{R}'(s)$

Démonstration :

1. si $L \in R(s)$ alors d'après la définition 3.5 du testeur canonique synchrone $R'(s) = \wp(L)$, donc $\tilde{R}'(s) = \{X \cap O(s) \mid X \in \wp(L)\}$. Par définition de la nouvelle forme des ensembles de refus $\tilde{R}(s) = \{X \cap O(s) \mid X \in R(s)\}$ et de plus $O(s) \in \tilde{R}(s)$ car $L \in R(s)$. Donc $t(O(s), \tilde{R}(s)) = \tilde{R}(s)$. Enfin, les ensembles de refus étant clos pour l'inclusion, puisque $L \in R(s)$ alors $R(s) = \wp(L)$, ce qui entraîne que $t(O(s), \tilde{R}(s)) = \{X \cap O(s) \mid X \in \wp(L)\} = \tilde{R}'(s)$.
2. si $L \notin R(s)$ alors d'après la définition du testeur canonique $R'(s) = \{L - X \mid X \notin R(s), X \subseteq L\}$, donc $\tilde{R}'(s) = \{X \cap O(s) \mid X \in R'(s)\}$ est égal à $\{(L - X) \cap O(s) \mid X \notin R(s), X \subseteq L\}$ ce qui est égal, en posant $Y = X \cap O(s)$ à $\{O(s) - Y \mid Y \notin R(s), Y \subseteq O(s)\}$ (car $X \notin R(s)$ entraîne $Y \notin R(s)$. En effet, si $Y \in R(s)$ alors $Y \cup (L - O(s)) \in R(s)$ d'après la propriété 3.1, donc $X \subseteq Y \cup (L - O(s))$ appartient aussi à $R(s)$ toujours d'après la même propriété).
   Or $\tilde{R}(s) = \{X \cap O(s) \mid X \in R(s)\}$ et $O(s) \notin \tilde{R}(s)$ car $L \notin R(s)$ (encore une conséquence de la propriété 3.1). Donc par la nouvelle transformation $t(O, \tilde{R})(s) = \{O(s) - X \mid X \notin \tilde{R}(s), X \subseteq O(s)\}$. Comme pour tout $X \subseteq O(s)$ on a $X \notin \tilde{R}(s)$ si et seulement si $X \notin R(s)$, cette dernière expression s'écrit aussi $\{O(s) - X \mid X \notin R(s), X \subseteq O(s)\}$, d'où la propriété $t(O(s), \tilde{R}(s)) = \tilde{R}'(s)$. ❏

3 Modélisation du test synchrone par des IOSM

À la section précédente nous avons souligné les similitudes entre la démarche suivie pour modéliser le test des automates à entrées et sorties et celle suivie pour modéliser le test des systèmes de transitions étiquetées. Après avoir présenté ces deux théories, nous montrons maintenant qu'il est possible de les relier formellement.

Dans un premier temps, nous modélisons la théorie du test synchrone dans le formalisme des automates à entrées et sorties. Pour cela, nous définissons deux fonctions AS (pour *Asynchronous Sender*) et AR (pour *Asynchronous Receiver*) qui transforment les graphes de refus en automates de traces (la raison pour laquelle on a recours à deux fonctions sera expliquée plus tard). Puis nous prouvons la propriété représentée par la figure 12, à savoir que l'image par ces fonctions de la transformation sur les graphes de

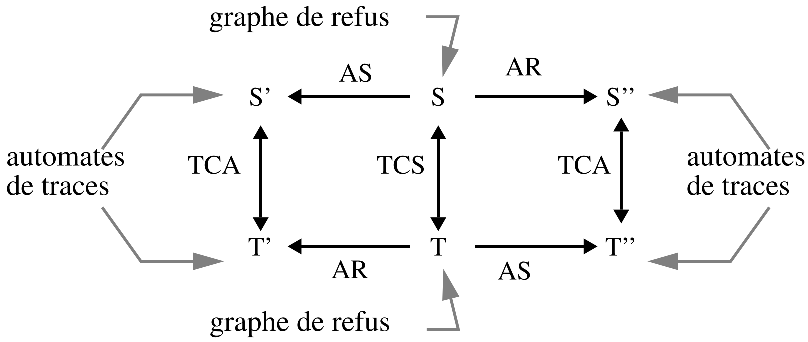


Figure 12 : modélisation du test synchrone par des IOSM

refus qui correspond au testeur canonique synchrone est la transformation sur les automates de traces qui correspond au testeur canonique asynchrone.

Les fonctions AS et AR introduisent une sorte de «protocole» qui implante la communication synchrone par rendez-vous négocié en s'appuyant sur les mécanismes d'entrées et sorties. Que prouve exactement le fait que le diagramme commute ? Schématiquement, le testeur asynchrone teste les traces d'un processus, tandis que le testeur synchrone teste les refus. Donc ça montre :

1. que les refus sont modélisés correctement par les traces du protocole auquel on fait allusion ci-dessus.
2. que la transformation sur les refus, qui est le principe de base du testeur synchrone, est reflétée par la transformation sur les traces (inversion des entrées et sorties) qui est le principe de base du testeur asynchrone.

3.1 Définition de la fonction asynchrone récepteur

Nous voulons montrer comment on peut modéliser dans un contexte asynchrone la communication entre deux processus synchrones (L TS). Pour cela il faut reproduire le comportement d'un L TS par une IOSM.

L'idée de base est de reproduire la communication par rendez-vous négocié du modèle synchrone à l'aide d'entrées et de sorties. Celle-ci est définie classiquement par la règle 4 c) de la définition 3.1, qui dit qu'une communication est possible si les deux partenaires sont d'accord pour participer à cette communication :

$$(s_1, a, s'_1) \in T_1 \wedge (s_2, a, s'_2) \in T_2 \Rightarrow ((s_1, s_2), a, (s'_1, s'_2)) \in T$$

La modélisation de ce mécanisme va donc être basée sur un échange d'information à base d'entrées et de sorties entre deux processus (deux IOSM) pour qu'ils se mettent d'accord sur l'action qui a effectivement lieu. D'après les résultats du chapitre 2 et de la section précédente, nous considérons les IOSM comme des automates de traces et les LTS comme des graphes de refus, exprimés sous la forme définie en 2.4. Les graphes de refus étant déterministes au sens des automates, pour tout état s et pour toute action $a \in O(s)$ il existe un état s' unique tel que $(s, a, s') \in T$. Nous le désignons par $s(s, a)$.

Définition 3.7 : soit $F = \langle S, L, T, O, R, s_0 \rangle$ un graphe de refus sous la forme définie en 2.4. L'*automate asynchrone récepteur* associé à F : $AR(F) = CC(\langle S_s, L_s, T_s, s_{0s} \rangle)$ est une IOSM définie par :

1. $S_s = S \times \wp(L) \times \wp(\wp(L)) \times (L \cup \{\varnothing\})$. Un élément $s_s = (s, O, R, a)$ de cet ensemble est interprété de la façon suivante :
   a) s est un état du graphe de refus.
   b) O est l'ensemble d'interactions que $AR(F)$ est susceptible de se voir proposer dans l'état s_s.
   c) R est l'ensemble d'ensembles de refus qui définit ce que $AR(F)$ est susceptible de refuser dans l'état s_s.
   d) a est égal à $\varnothing$ si $AR(F)$ est en attente (qu'on lui propose une interaction) ou égal à un élément de L si cette interaction a été proposée à $AR(F)$.
2. $L_s = L \cup \{r\}$ où r est une interaction spéciale qui sert à notifier un refus (on suppose que $r \notin L$).
3. $s_{0s} = (s_0, O(s_0), R(s_0), \varnothing)$
4. les transitions de $AR(F)$ sont exactement celles qui peuvent être obtenues au moyen des règles suivantes :
   a) $(\forall a \in O)\ ((s, O, R, \varnothing), ?\mathrm{a}, (s, O, R, a)) \in T_s$
   b) $((s, O, R, a), !\mathrm{a}, (s', O(s'), R(s'), \varnothing)) \in T_s$ où $s' = s(s, a)$
   c) si $(\exists X \in R \mid a \in X)$ alors $((s, O, R, a), !\mathrm{r}, (s, O - \{a\}, R \text{ after}_1 a, \varnothing)) \in T_s$ où $R \text{ after}_1 a = \{X - \{a\} \mid X \in R, a \in X\}$

Le fonctionnement de l'automate asynchrone récepteur est le suivant : dans un état donné correspondant à un état du graphe de refus, il peut recevoir toutes les interactions que le graphe de refus peut effectuer dans cet état, comme indiqué par la règle a) ci-dessus. $AR(F)$ peut accepter ces interactions en les réémettant (règle b). Si le graphe de refus lui permet de refuser une interaction, c'est à dire si elle fait partie de l'un des ensembles de refus, $AR(F)$ peut refuser cette interaction (règle c) en envoyant l'action spéciale !r . La négociation doit alors être poursuivie avec son homologue pour déterminer quelle interaction sera choisie, sachant que $AR(F)$ peut encore refuser toutes les interactions qu'il pouvait refuser en même temps que a, d'où le calcul du nouvel ensemble de refus $R \text{ after}_1\, a$.

$AR(F)$ est appelé automate asynchrone *récepteur* car dans la négociation avec son homologue de l'interaction à choisir c'est l'autre automate qui propose les interactions, et $AR(F)$ les reçoit et décide ou non de les accepter . Quant au terme «asynchrone», il est hélas ici aussi bien mal choisi, puisque la communication se fait par entrées et sorties synchrones, mais tout comme au chapitre précédent au sujet du testeur canonique, nous l'avons gardé pour des raisons historiques.

3.2 Définition de la fonction asynchrone émetteur

Définition 3.8 : soit $F = \langle S, L, T, O, R, s_0 \rangle$ un graphe de refus sous la forme définie en 2.4. L'*automate asynchrone émetteur* associé à F : $AS(F) = CC(\langle S_s, L_s, T_s, s_{0s} \rangle)$ est une IOSM définie par :

1. $S_s = S \times \wp(L) \times \wp(\wp(L)) \times (L \cup \{\varnothing\})$. Un élément $s_s = (s, O, R, a)$ de cet ensemble est interprété de la façon suivante :
   a) s est un état du graphe de refus.
   b) O est l'ensemble d'interactions que $AS(F)$ est susceptible de proposer dans l'état s_s.
   c) R est l'ensemble d'ensembles de refus qui définit ce que $AS(F)$ est susceptible de se voir refuser dans l'état s_s.
   d) a est égal à $\varnothing$ si $AS(F)$ est prêt à proposer une interaction ou égal à un élément de L si cette interaction a été proposée par $AS(F)$.
2. $L_s = L \cup \{r\}$ où r est une interaction spéciale qui sert à notifier un refus (on suppose que $r \notin L$).
3. $s_{0s} = (s_0, O(s_0), R(s_0), \varnothing)$
4. les transitions de $AS(F)$ sont exactement celles qui peuvent être obtenues au moyen des règles suivantes :
   a) $(\forall a \in O)\ ((s, O, R, \varnothing), !\text{a},\ (s, O, R, a)) \in T_s$
   b) $((s, O, R, a), ?\text{a},\ (s', O(s'), R(s'), \varnothing)) \in T_s$ où $s' = s(s, a)$
   c) si $O - \{a\} \notin R$ alors $((s, O, R, a), ?\text{r},\ (s, O - \{a\}, R \text{ after}_2\, a, \varnothing)) \in T_s$ où $R \text{ after}_2\, a = \{X - \{a\} \mid X \in R, a \in X\} \cup \{X \mid X \in R, a \notin X\}$

Le fonctionnement de l'automate asynchrone émetteur est le suivant : dans un état donné correspondant à un état du graphe de refus, il peut envoyer toutes les interactions que le graphe de refus peut exécuter dans cet état (règle a). Envoyer une interaction revient à la proposer comme interaction de rendez-vous possible. Une interaction envoyée peut être acceptée par l'entité homologue pour rendez-vous, cette acceptation étant modélisée par le renvoi de l'interaction proposée (règle b). Mais proposer une interaction revient à proposer de refuser toutes les autres interactions. Donc si $O - \{a\} \notin R$ c'est à dire si $AS(F)$ ne peut pas refuser en bloc $O - \{a\}$, $AS(F)$ doit prévoir un refus potentiel, et donc ensuite proposer d'autres interactions, parmi celles qui restent (règle c).

$AS(F)$ est appelé automate asynchrone *émetteur* car dans la négociation avec son homologue de l'interaction à choisir c'est lui qui propose les interactions, en les émettant vers l'autre automate.

3.3 Exemples

Les figures 13, 14 et 15 montrent quelques exemples d'utilisation des fonctions AS et AR. Les graphes de refus sont représentés comme dans [Brinksma 88] en indiquant pour chaque nœud du graphe les sous-ensembles maximaux (pour l'inclusion) de $O(s)$ qui sont dans $R(s)$. Le premier exemple illustre ce que l'on appelle un choix externe

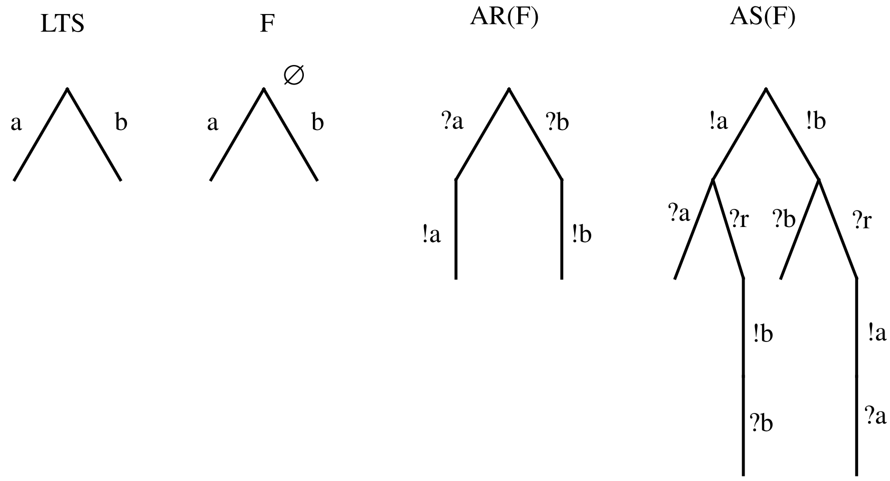


Figure 13 : fonctions AS et AR (exemple 1)

dans la théorie des algèbres de processus (c'est l'environnement qui décide de l'interaction qui va être choisie). Ce choix se modélise bien au moyen de réceptions, moins facilement au moyen d'émissions ($AS(F)$ est plus complexe que $AR(F)$). Le deuxième exemple au présente un choix interne, qui se trouve être, au contraire, plus facilement représenté par des émissions, $AS(F)$ étant cette fois plus simple que $AR(F)$. Enfin, le troisième exemple présente un cas où les automates asynchrones émetteurs et récepteurs ont la même structure, les émissions de l'un correspondant exactement aux réceptions de l'autre. Les trois exemples présentés ici sont inspirés de [Brinksma 88].

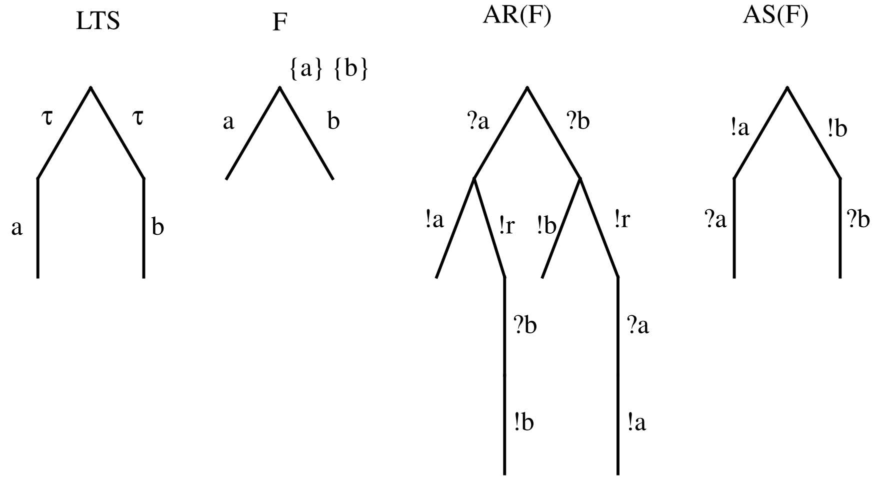


Figure 14 : fonctions *AS* et *AR* (exemple 2)

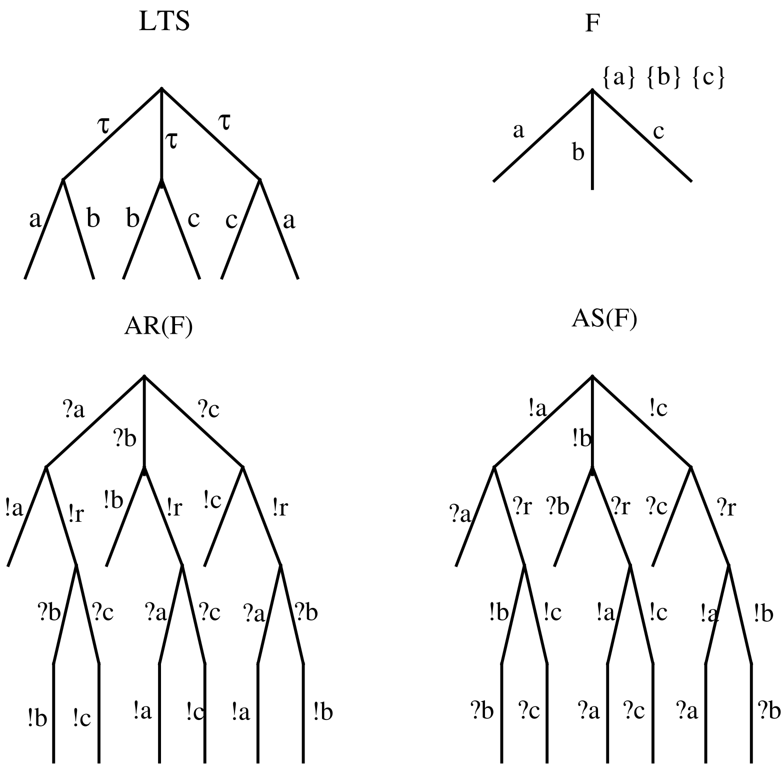


Figure 15 : fonctions *AS* et *AR* (exemple 3)

3.4 Propriétés des fonctions de transformation

Nous montrons maintenant que le diagramme de transformations présenté par la figure 12 commute, c'est à dire que l'image asynchrone du testeur canonique synchrone d'un système de transitions étiquetées est le testeur canonique asynchrone de son image asynchrone (relisez cette phrase si vous avez un doute, mais elle est correcte). Nous identifions ici le testeur canonique synchrone et son graphe de refus. De plus, nous considérons la *structure* du testeur canonique asynchrone, sans prendre en compte les transitions aboutissant à l'état $fail$ qui constituent un mécanisme rajouté afin de fournir le verdict de test.

Propriété 3.5 : pour tout graphe de refus F, $TCA(AS(F)) = AR(TCS(F))$

Démonstration : étant donné les hypothèses faites ci-dessus, et en notant t la transformation sur les graphes de refus, définie à la section 2.4, qui permet d'obtenir le testeur canonique synchrone, nous devons montrer :

$$\text{pour tout graphe de refus } F,\ \overline{AS(F)} = AR(t(F))$$

(nous étendons ici la notation $\overline{}$ aux automates : comme pour les séquences, elle signifie la permutation des entrées en sorties et réciproquement).

Soient $AS(F) = \langle S_1, L_1, T_1, s_{01}\rangle$ et $AR(t(F)) = \langle S_2, L_2, T_2, s_{02}\rangle$ les deux IOSM qui nous intéressent. L'égalité ci-dessus, que nous nous proposons d'établir, doit être comprise comme un isomorphisme, c'est à dire que nous allons définir une bijection f entre les ensembles d'états de ces deux automates, puis montrer que : $S_2 = f(S_1)$, $L_2 = L_1$, $s_{02} = f(s_{01})$ et $(s_1, \mu, s'_1) \in T_1$ si et seulement si $(f(s_1), \bar{\mu}, f(s'_1)) \in T_2$.

Nous définissons à partir de la transformation t sur les graphes de refus une transformation entre les états de l'automate asynchrone émetteur et ceux de l'automate asynchrone récepteur, notée f, telle que :

$$f((s, O, R, a)) = (s, O, t(O, R), a)$$

De la définition de t il découle que $t(O, t(O, R)) = R$, donc la transformation f si-dessus est une bijection de l'ensemble $S_1 = S_2$ sur lui-même telle que $f(f(x)) = x$. Posons $F = \langle S, L, T, O, R, s_0\rangle$. D'après les définitions de f, AS et AR, il est immédiat que $L_1 = L_2 = L$. De plus $s_{01} = (s_0, O(s_0), R(s_0), \varnothing)$ (par définition de l'automate émetteur AS) et $s_{02} = (s_0, O(s_0), t(O(s_0), R(s_0)), \varnothing)$ (par définition de la transformation t et de l'automate récepteur AR), donc $s_{02} = f(s_{01})$.

Il nous reste à montrer que $(s_1, \mu, s'_1) \in T_1$ si et seulement si $(f(s_1), \bar{\mu}, f(s'_1)) \in T_2$. Les transitions des automates $AS(F)$ et $AR(t(F))$ sont définies par induction à partir des états initiaux respectifs de ces deux automates au moyen de trois règles de production (règles a), b) et c) du point 4 de la définition des fonctions AS et AR).

Montrons d'abord que si $(s_1, \mu, s'_1) \in T_1$ alors $(f(s_1), \bar{\mu}, f(s'_1)) \in T_2$. On raisonne par induction sur la construction de la composante connexe de l'état initial des graphes des automates. On sait que pour les états initiaux $s_{02} = t(s_{01})$. Supposons donc la propriété vraie pour toute transition de T_1 obtenue par n applications des règles a) b) ou c) (donc à une distance n de l'état initial), et considérons une transition (s_1, μ, s'_1) obtenue par $n+1$ applications :

1. si (s_1, μ, s'_1) est obtenue par la règle a) de la définition 3.8 alors cette transition est de la forme $((s, O, R, \emptyset), !\mathrm{a}, (s, O, R, a))$. Par hypothèse d'induction $f((s, O, R, \emptyset)) = (s, O, t(O, R), \emptyset)$ appartient aux états accessibles par T_2. Donc d'après la règle a) de la définition 3.7, comme $f((s, O, R, a)) = (s, O, t(O, R), a)$ la transition $(f(s_1), ?\mathrm{a}, f(s'_1)) \in T_2$.
2. si (s_1, μ, s'_1) est obtenue par la règle b) de la définition 3.8 alors cette transition est de la forme $((s, O, R, a), ?\mathrm{a}, (s', O(s'), R(s'), \emptyset))$ où $s' = s(s, a)$. Par hypothèse d'induction $f((s, O, R, a)) = (s, O, t(O, R), a)$ appartient aux états accessibles par T_2. Donc d'après la règle b) de la définition 3.7, comme $f((s', O(s'), R(s'), \emptyset)) = (s', O(s'), t(O(s'), R(s')), \emptyset)$, la transition $(f(s_1), !\mathrm{a}, f(s'_1)) \in T_2$.
3. si (s_1, μ, s'_1) est obtenue par la règle c) de la définition 3.8 alors cette transition est de la forme $((s, O, R, a), ?\mathrm{r}, (s, O - \{a\}, R \text{ after}_2\, a, \emptyset))$ et son existence implique que $O - \{a\} \notin R$.
   Par hypothèse d'induction $f((s, O, R, a)) = (s, O, t(O, R), a)$ appartient aux états accessibles par T_2. Pour conclure d'après la règle c) de la définition 3.7 que $(f((s, O, R, a)), !\mathrm{r}, f((s, O - \{a\}, R \text{ after}_2\, a, \emptyset))) \in T_2$ il suffit donc de montrer :
   a) que $O - \{a\} \notin R$ si et seulement si $(\exists X \in t(O, R) \mid a \in X)$. Si $O - \{a\} \notin R$ alors par définition de la transformation t, $O - (O - \{a\}) = \{a\} \in t(O, R)$ donc $(\exists X \in t(O, R) \mid a \in X)$. Réciproquement si $O - \{a\} \in R$ alors $\{a\} \notin t(O, R)$. Les ensembles de refus étant clos par inclusion, on en conclut qu'il n'existe pas $X \in t(O, R)$ tel que $a \in X$.
   b) que $f((s, O - \{a\}, R \text{ after}_2\, a, \emptyset)) = (s, O - \{a\}, t(O, R) \text{ after}_1\, a, \emptyset)$ dans tous les cas où la règle c) peut être appliquée. Cela signifie montrer que $t(O - \{a\}, R \text{ after}_2\, a) = t(O, R) \text{ after}_1\, a$.
   Si $X \in t(O - \{a\}, R \text{ after}_2\, a)$ et si on est dans le premier cas de la définition de t alors $X = O - \{a\} - Y$ où $Y \notin R \text{ after}_2\, a$ et $Y \subseteq O - \{a\}$. Donc comme $a \notin Y$ alors $Y \notin R$ (sinon on aurait $Y \in R \text{ after}_2\, a$ par définition de after_2). On a donc $X \cup \{a\} = O - Y$ avec $Y \notin R$, ce qui entraîne $X \cup \{a\} \in t(O, R)$ par définition de t, et donc que $X \in t(O, R) \text{ after}_1\, a$ par définition de after_1.
   Le deuxième cas de la définition de t n'est pas possible, car alors on aurait $O - \{a\} \in R \text{ after}_2\, a$, donc nécessairement $O \in R$, donc $O - \{a\} \in R$ car les ensembles de refus sont clos par inclusion, ce qui est en contradiction avec l'hypothèse que la règle c) s'applique.
   Réciproquement, si $X \in t(O, R) \text{ after}_1\, a$ alors $X \cup \{a\} \in t(O, R)$ par définition de after_1, donc si on est dans le premier cas de la définition de t alors

$X \cup \{a\} = O - Y$ avec $Y \notin R$ et $a \notin Y$. Donc $Y \notin R \text{ after}_2\, a$ (sinon comme $Y \notin R$, on aurait $Y \cup \{a\} \in R$ d'après la définition de after_2, ce qui est impossible car les ensembles de refus sont clos par inclusion). Donc $X = O - \{a\} - Y$ où $Y \notin R \text{ after}_2\, a$ et $Y \subseteq O - \{a\}$, ce qui entraîne $X \in t(O - \{a\}, R \text{ after}_2\, a)$.
Là encore, on ne peut pas être dans le deuxième cas de la définition de t, car alors on aurait $O \in R$, donc $O - \{a\} \in R$, ce qui est en contradiction avec l'hypothèse que la règle c) s'applique.

Finalement, on montre de la même façon que l'on a montré l'implication inverse que si $(f(s_1), \bar{\mu}, f(s'_1)) \in T_2$ alors $(s_1, \mu, s'_1) \in T_1$, ce qui termine la preuve de la propriété. ❑

Enfin, pour terminer la démonstration du diagramme commutatif présenté dans l'introduction de ce chapitre, il reste à montrer la propriété symétrique de la propriété 3.5:

Propriété 3.6 : pour tout graphe de refus F, $TCA(AR(F)) = AS(TCS(F))$

Démonstration : il résulte de la propriété 3.5 que $\overline{AS(F)} = AR(t(F))$. Comme $t(F)$ est aussi un graphe de refus, on a $\overline{AS(t(F))} = AR(t(t(F)))$, soit $AS(t(F)) = \overline{AR(F)}$, car les transformations t et $\overline{}$ sont involutives, ce qui prouve la propriété. ❑

On vérifiera que ces deux propriétés sont satisfaites par les exemples introduits à la section précédente, sachant que :

1. le testeur canonique du processus de l'exemple 1 est le processus de l'exemple 2 et vice-versa.
2. le processus de l'exemple 3 est son propre testeur canonique.

4 Modélisation du test asynchrone par des LTS

À l'inverse de ce que nous avons fait à la section précédente, nous montrons maintenant comment nous pouvons représenter (en partie seulement) la théorie du test des automates à entrées et sorties au moyen de systèmes de transitions étiquetées. Pour cela, il nous faut modéliser les entrées et les sorties au moyen de structures utilisant les refus.

La solution que nous présentons ici n'est pas générale, puisqu'elle ne nous permet de représenter que les IOSM où les états comportent soit des entrées, soit des sorties, mais pas les deux à la fois. Cependant cette solution partielle est suffisante pour illustrer la discussion de la section suivante, qui est l'objet principal de ce chapitre, sur l'équivalence entre les modèles synchrone et asynchrone dans la plupart des cas. De plus, nous verrons alors que la difficulté pour représenter les états où coexistent des entrées et des sorties révèle une faiblesse de la théorie synchrone pour modéliser correctement, au moins du point de vue du test, certains mécanismes courants utilisés dans les protocoles, comme les échéances de temporisateurs.

Pour ce sous-ensemble des IOSM, nous définissons une fonction SY (pour *Synchronous*) qui transforme les automates de traces en graphes de refus. Puis nous montrons la propriété représentée par la figure 12:

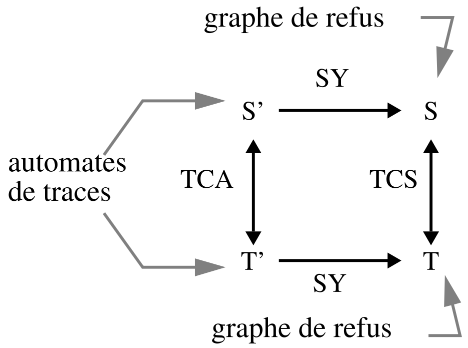


Figure 16 : modélisation du test asynchrone par des LTS

Notons que la transformation SY n'est pas liée aux fonctions AR ou AS. En particulier elle n'est pas l'inverse de ces fonctions, qui ne sont d'ailleurs pas des bijections !

4.1 Définition de la fonction synchrone

Nous considérons un sous-ensemble des automates à entrées et sorties constitué des automates dont les états ne comportent que des entrées (on les appelle états *récepteurs*) ou que des sorties (on les appelle états *émetteurs*). Nous notons IOSM1 ce sous-ensemble. Pour une entrée $\mu = ?a$ ou une sortie $\mu = !a$ nous notons $|\mu| = a$ l'interaction associée.

Définition 3.9 : soit $T = \langle S, L, T, s_0 \rangle$ un automate de traces d'un élément de IOSM1. L'*automate synchrone* associé à T est défini par son graphe de refus $SY(T) = \langle S_f, L_f, T_f, R_f, s_{0f} \rangle$ tel que :

1. $S_f = S$
2. $L_f = L$
3. $T_f = \{ (s, |\mu|, s') \mid ((s, \mu, s') \in T) \}$
4. $s_{0f} = s_0$
5. R_f est définie par :
   a) si s est un état récepteur, $R_f(s) = \{\varnothing\}$
   b) si s est un état émetteur, $R_f(s)$ est engendré par $\{L - \{a\} \mid a \in O(s)\}$ (i.e. contient ces ensembles et vérifie les alinéas 2 et 3 de la propriété 3.1).

Intuitivement, une réception est modélisée par une action que l'on ne peut pas refuser, tandis qu'une émission est modélisée comme la possibilité de refuser toutes les autres interactions du langage. Il est facile de montrer que la définition ci-dessus est bien celle d'un graphe de refus, en construisant explicitement un système de transitions étiquetées correspondant. Sans définir formellement ici cette construction, indiquons simplement qu'elle consiste à transformer respectivement les états du graphe de refus de la façon suivante :

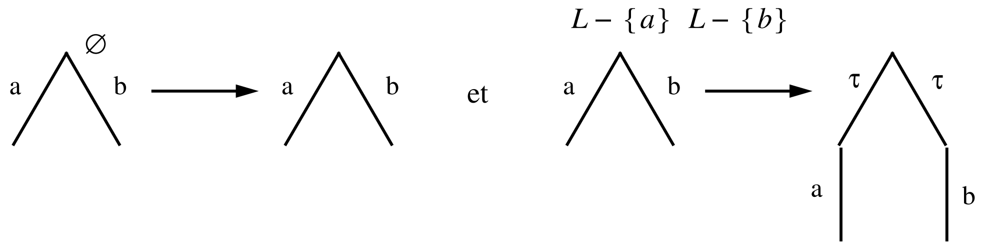


Figure 17 : graphe de refus et système de transitions associé

4.2 Exemples

La figure suivante présente deux exemples d'utilisation de la fonction SY, modélisant respectivement des réceptions et des émissions.

Figure 18 : exemples de la fonction SY

4.3 Propriété de la fonction SY

Nous montrons maintenant que le diagramme présenté à la figure 12 commute, à savoir que l'image synchrone du testeur canonique asynchrone d'un automate de traces est le testeur canonique synchrone de son image synchrone. Comme à la section précédente, nous identifions ici le testeur canonique synchrone et son graphe de refus, et nous ne considérons que la structure du testeur canonique asynchrone, sans prendre en compte les transitions qui aboutissent à l'état *fail* .

Propriété 3.7 : pour tout automate de traces T correspondant à un élément de IOSM1, $SY(TCA(T)) = TCS(SY(T))$

Démonstration : d'après les définitions des testeurs canoniques synchrone et asynchrone, et en notant TCS la transformation sur les graphes de refus correspondant au testeur canonique synchrone (définition 3.5), prouver la propriété revient à prouver :

$$\text{pour tout automate de traces } \quad T,\ SY(\bar{T}) = TCS(SY(T))$$

Notons $T = \langle S, L, T, s_0 \rangle$, $SY(T) = \langle S_3, L_3, T_3, R_3, s_{03} \rangle$, $SY(\bar{T}) = \langle S_1, L_1, T_1, R_1, s_{01} \rangle$ et $t(SY(T)) = \langle S_2, L_2, T_2, R_2, s_{02} \rangle$. D'après les définitions de SY, de TCS et de $\bar{T}$, on a immédiatement : $S_1 = S_2 = S$, $L_1 = L_2 = L$, $T_1 = T_2 = \{ (s, |\mu|, s') \mid ((s, \mu, s') \in T) \}$ et $s_{01} = s_{02} = s_0$. Pour prouver la propriété, il nous reste donc à montrer que $R_1 = R_2$.

1. si s est un état émetteur de T alors s est un état récepteur de $\bar{T}$, donc $R_1(s) = \{\varnothing\}$ (par définition de SY) et (toujours par définition de SY) $R_3(s)$ est engendré par $\{L - \{a\} \mid (a \in O(s))\}$.
   Donc par définition de TCS, $R_2(s) = \{L - A \mid A \notin R_3(s), A \subseteq L\}$. Comme $L - \{a\} \in R_3(s)$ et que les ensembles de refus vérifient la propriété 3.1 le seul sous-ensemble de L qui n'est pas dans $R_3(s)$ est L lui-même, donc $R_2(s) = \{\varnothing\}$, ce qui signifie que $R_1(s) = R_2(s)$.
2. si s est un état récepteur de T alors s est un état émetteur de $\bar{T}$, donc $R_1(s)$ est engendré par $\{L - \{a\} \mid (a \in O(s))\}$ (définition de SY) et $R_3(s) = \{\varnothing\}$ (définition de SY). Donc $R_2(s) = \{L - A \mid (A \notin R_3(s), A \subseteq L)\}$ par définition de TCS. Puisque aucun sous-ensemble non vide de L n'appartient à $R_3(s)$, $R_2(s)$ contient toutes les parties de L sauf $L - \varnothing = L$, donc $R_2(s)$ est engendré par $\{L - \{a\} \mid (a \in O(s))\}$ (i.e. contient ces ensembles et vérifie la propriété 3.1), ce qui là aussi signifie que $R_1(s) = R_2(s)$, ce qui termine la démonstration. ❑

5 Quel modèle de communication choisir ?

Munis des résultats des sections précédentes, nous sommes maintenant en mesure d'aborder l'objet principal de ce chapitre, à savoir une comparaison détaillée entre les notions de conformité et les tests qui sont respectivement représentés dans les modèles synchrone et asynchrone.

Nous basons notre analyse sur l'étude de quelques exemples. Nous allons montrer que dans certains cas (qui selon nous représentent la majeure partie des systèmes susceptibles de nous intéresser), si les modélisations synchrone et asynchrone diffèrent, la notion réelle de conformité étudiée est exactement la même, et les tests calculés selon les deux théories sont en quelque sorte équivalents : s'il est vrai que le modèle synchrone utilise des relations d'implantation plus puissantes, il ne faut pas oublier que celles-ci s'appliquent à des modélisations différentes des systèmes réels.

Nous présenterons cependant un cas (mécanisme d'échéance de temporisateur) où les tests prédits dans les deux théories ne sont pas équivalents. Dans ce cas précis, le modèle asynchrone nous parait mieux adapté aux besoins pratiques du test.

5.1 Modélisation de systèmes communiquant par entrées et sorties

Certains systèmes, par exemple la plupart des protocoles utilisés dans les télécommunications, fonctionnent au moyen d'échanges de messages (les unités de données de protocole, ou *PDU*) entre les différentes entités de protocoles. La spécification informelle en langage naturel de tels systèmes fait donc naturellement référence aux émissions et aux réceptions de ces messages. Par exemple, on peut imaginer le protocole suivant (que les fanatiques du modèle OSI ne trouveront pas réaliste, mais dont le but est uniquement ici d'illustrer un concept) :

> «sur réception d'une demande de connexion (message *Conreq*), l'entité de protocole peut répondre par une acceptation (message *Conconf*) ou par un refus (message *Disind*) selon l'état de ses ressources internes»

La modélisation de ce morceau de protocole au moyen d'automates à entrées et sorties est immédiate (voir figure 19). Mais il est aussi possible de faire la modélisation en uti-

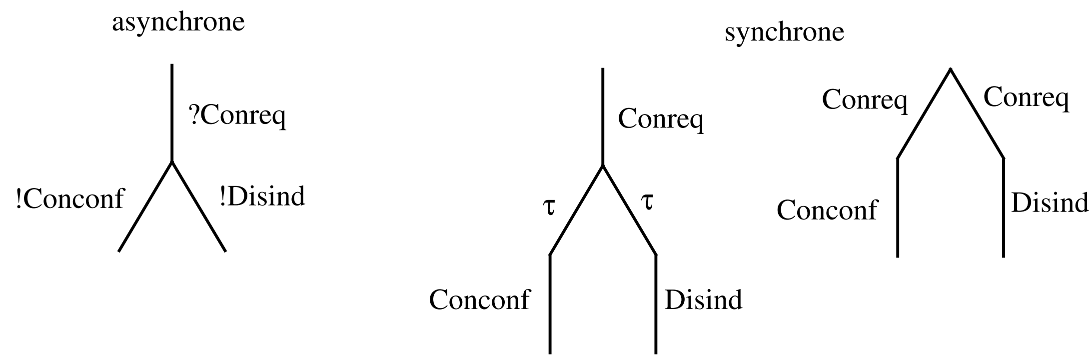


Figure 19 : modélisations de l'exemple 1

lisant des systèmes de transitions étiquetées. C'est la modélisation qui résulterait de l'utilisation du langage de spécification LOTOS par exemple. Dans ce cas, afin de représenter le fait que la décision de renvoyer une acceptation ou un refus est un choix interne de l'entité de protocole, l'action interne τ peut être utilisée; mais ce n'est pas la seule modélisation possible, le non-déterminisme pouvant aussi être utilisé. Ces représentations apparaissent sur la figure 19.

Une implantation sera conforme si après avoir reçu une demande de connexion, elle peut répondre par l'acceptation et par le refus, et par aucun autre message, et si cette décision dépend uniquement de facteurs internes à l'entité de protocole. Cette conformité que l'on veut vérifier est l'équivalence de traces sur le modèle asynchrone. Par contre, sur le modèle synchrone, la même notion de conformité correspond à l'équivalence de refus. En effet, l'implantation présentée à la figure 20, qui a les mêmes traces

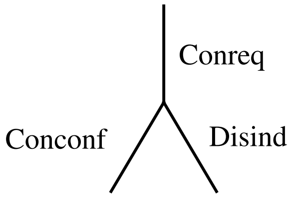


Figure 20 : implantation non conforme de l'exemple 1

que la spécification, n'est cependant pas conforme, car dans cette implantation la décision d'accepter ou de refuser la connexion n'est pas interne à l'entité de protocole. Une autre façon d'exprimer ce phénomène est que la structure des refus est nécessaire pour modéliser la localité des décisions, qui est caractéristique des concepts d'entrées et sorties : une sortie est décidée localement, tandis qu'une entrée est reçue, donc décidée extérieurement.

L'équivalence de traces dans le modèle asynchrone représente donc la même notion de conformité que l'équivalence de refus dans le modèle synchrone. Ce n'est pas un paradoxe puisque *les traces du modèle asynchrone sont plus riches en information que celles du modèle synchrone, car elles indiquent la nature - entrées ou sorties - des interactions*.

Sur cet exemple, les modèles asynchrone et synchrones sont reliés par la transformation SY introduite à la section précédente. Il est donc naturel, en vertu des résultats de cette section (propriété 3.7), que les tests calculés au moyen des deux approches soient équivalents. Ceci signifie que les testeurs canoniques synchrone et asynchrone, représentés sur la figure 21, sont deux modélisations, l'une synchrone, l'autre asynchrone,

Figure 21 : testeurs canoniques correspondant à l'exemple 1

d'un même processus : le test à effectuer est le même, mais exprimé dans des formalismes différents. En conclusion, pour ce type de systèmes faisant référence à des envois et des réceptions de messages, on obtient exactement les mêmes résultats en utilisant les modèles synchrone ou asynchrone. Cependant il parait plus naturel d'utiliser directement le modèle asynchrone, plus proche dans ce cas des spécifications en langage naturel.

5.2 Utilisation de la communication synchrone comme abstraction

Dans l'exemple précédent la communication synchrone est utilisée de façon un peu artificielle, pour modéliser des phénomènes d'entrées et de sorties. Mais il est aussi des cas où la communication synchrone est utilisée comme une abstraction qui sert à décrire une interaction entre deux entités à un niveau plus abstrait que le détail des échanges de messages qui a lieu dans l'implantation. C'est d'ailleurs cette utilisation de la communication synchrone qui est la plus naturelle, celle qui fait son intérêt pour la spécification concise et abstraite de systèmes distribués.

Prenons l'exemple d'un guichet bancaire automatique. La spécification informelle prévoit entre autres :

> «l'utilisateur , après avoir tapé son code personnel, peut demander à consulter le solde de son compte. Pour réaliser cette opération, le guichet doit se connecter à l'ordinateur de la banque. Cette connexion pouvant échouer , l'opération solde peut éventuellement être refusée à l'utilisateur»

Une modélisation de ce mécanisme au moyen de systèmes de transitions étiquetées, ainsi que le graphe de refus correspondant, est représentée à la figure 22. V ue de façon

Figure 22 : modèle synchrone de l'exemple 2 et son graphe de refus

abstraite, l'opération *solde* est modélisée comme une interaction entre l'utilisateur et le guichet, celle-ci pouvant être refusée pour des raisons internes, que nous n'avons pas à décrire ici (d'où l'utilisation de l'action interne τ).

En réalité, cette représentation constitue une vue abstraite du phénomène. Le dialogue réel entre l'utilisateur et le guichet (c'est à dire l'implantation) doit nécessairement faire intervenir trois messages : l'utilisateur fait une demande de solde (message *demande-solde*), puis il reçoit la réponse (message *réponse-solde*) ou l'indication de l'impossibilité de l'opération (message *solde-impossible*). De même on peut supposer que la validation du code est implantée sous forme d'un dialogue : l'utilisateur tape son code (message *tape-code*) et reçoit confirmation (message *code-bon*). Nous obtenons donc une modélisation moins abstraite du même système au moyen de l'automate à entrées et sorties de la figure 23.

À nouveau sur cet exemple nous constatons que la conformité que nous voulons modéliser , à savoir que le mécanisme décrit par la spécification informelle est reproduit par l'implantation, se représente par l'équivalence de refus sur la spécification synchrone, et par l'équivalence de traces sur la spécification asynchrone. De plus, les modèles syn-

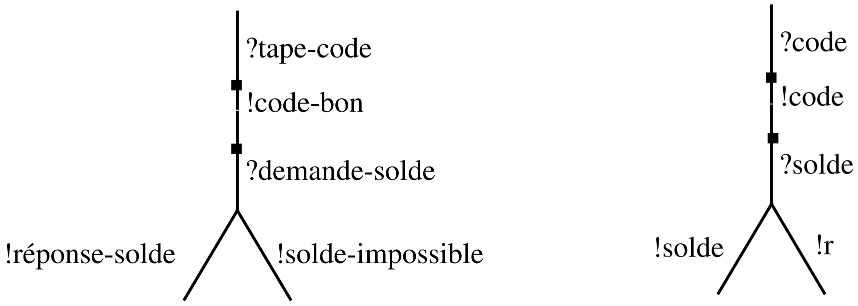


Figure 23 : modèle asynchrone de l'exemple 2 et automate asynchrone récepteur

chrone et asynchrone ci-dessus sont reliés, à un renommage des actions près, par la transformation AR introduite à la section 3.1. Conformément aux résultats de cette section (propriété 3.5), les tests calculés au moyen des deux approches sont reliés par la transformation AS et peuvent être considérés comme deux modélisations, l'une synchrone, l'autre asynchrone, du même test informel (voir la figure 24).

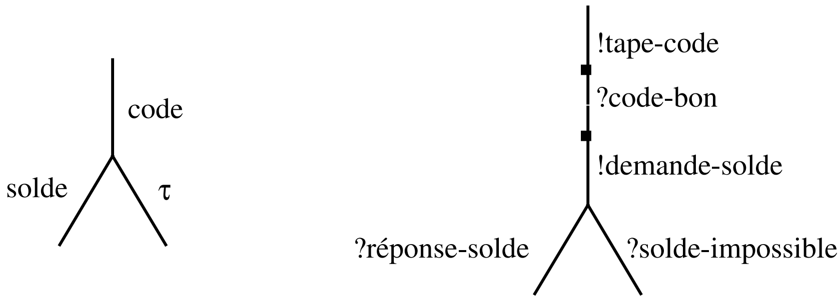


Figure 24 : testeurs canoniques de l'exemple 2

En conclusion, pour ce deuxième type de systèmes, l'utilisation du modèle synchrone aboutit au même résultat que celle du modèle asynchrone. On peut cependant remarquer que le test calculé par la théorie synchrone fait le même usage abstrait de la communication synchrone que la spécification. La mise en œuvre concrète de ce test, c'est à dire son implantation, doit avoir recours à des émissions et des réceptions de messages. Si l'on part du modèle synchrone qui fournit une spécification plus simple, on a le choix entre calculer d'abord le testeur synchrone, puis son implantation asynchrone, ou bien calculer d'abord la version asynchrone de la spécification avant de calculer le testeur. Dans le premier cas le calcul du test est plus compliqué, mais le testeur manipulé est plus simple, car plus abstrait.

5.3 Mécanisme d'échéance de temporisateur

Considérons enfin un dernier exemple illustrant la collision d'un événement spontané qui se déclenche à l'intérieur d'un système avec la réception par ce même système d'un message venant de l'extérieur . Ce phénomène est courant dans les systèmes de communication :

> «après avoir envoyé des données (message *dt*) à l'entité homologue, un entité de protocole arme un temporisateur . Si un acquittement (message *ak*) est reçu avant l'échéance de la temporisation, le protocole termine. Sinon, l'entité de protocole envoie une demande de déconnexion (message *dis*)»

La modélisation asynchrone de ce protocole est immédiate, puisque la description informelle fait directement référence à des émissions et des réceptions. Pour la modélisation synchrone, ce qu'il est important de représenter c'est le caractère local de la décision d'envoi de déconnexion, et au contraire le fait que la réception d'acquittement est décidée par l'extérieur . On obtient donc naturellement les deux descriptions représentées sur la figure 25.

Figure 25 : modélisations de l'exemple 3

Remarquons que dans ce cas ces deux représentations ne sont reliées par aucune des transformations introduites dans ce chapitre; en fait, nous sommes dans le cas où la transformation *SY* ne peut pas être appliquée, car l'automate à entrées et sorties a un état où coexistent une émission et une réception. L'application des théories synchrone et asynchrone à ce troisième exemple conduit aux testeurs canoniques représentés sur la figure 26.

Figure 26 : testeurs canoniques de l'exemple 3

On constate cette fois-ci que ces deux automates *ne représentent pas* le même processus informel. En effet, pour le testeur canonique asynchrone la réception du message de déconnexion est décidée extérieurement. Au contraire, la déconnexion résulte d'un choix interne pour le testeur canonique synchrone. Dans ce cas, nous n'obtenons donc pas les mêmes résultats, c'est à dire le même test, selon que nous utilisons un modèle plutôt qu'un autre.

Comme indiqué par leurs théories respectives, ces deux tests possèdent chacun leur logique propre, et en particulier chacun permet de vérifier la conformité pour laquelle il a été conçu. Cependant, lorsque l'on vise une implantation concrète du test, qui, in fine, doit s'exprimer avec des envois et réceptions de messages, il nous faut un testeur utilisant des entrées et des sorties. L 'examen de l'exemple ci-dessus montre qu'il nous faut alors partir directement de la représentation asynchrone du protocole, car l'utilisation du modèle synchrone, suivi d'une traduction du testeur obtenu vers un modèle asynchrone, ne donne pas le résultat logiquement attendu.

En conclusion de ce chapitre, nous constatons donc que dans la plupart des cas les théories synchrone et asynchrone donnent les mêmes résultats. Cela nous conforte dans notre conviction que la notion de conformité que nous étudions est bien celle qu'il est pertinent d'étudier en pratique. En particulier , les conformités plus fortes que l'équivalence de refus dans le modèle synchrone, que nous avons mentionnées au chapitre précédent, section 4.2, ne semblent pas intéressantes (outre le fait que leur vérification exige un pouvoir d'observation très fort sur les systèmes sous test, qu'il est difficile de mettre en œuvre en pratique).

Enfin, le fait que l'implantation des tests se fait au moyen d'émissions et de réceptions, ainsi que les meilleurs résultats obtenus par la théorie asynchrone dans le dernier cas étudié, plaident pour une utilisation de cette théorie pour l'étude du test. N'oublions pas que le langage le plus utilisé dans le monde des télécommunications pour décrire les tests, TTCN [ISO 91], utilise un mode de communication par entrées et sorties.

CHAPITRE 4

Test de conformité et architectures de test

1 Introduction

Au cours des deux chapitres précédents nous avons étudié le pouvoir d'observation du test, ainsi que la manière de construire un test optimal, en nous plaçant dans l'hypothèse idéale où le couplage entre le test et l'implantation sous test est modélisé par une synchronisation parfaite, soit à base d'entrées et de sorties, soit à base de rendez-vous négocié.

Une telle situation est hélas difficile à réaliser en pratique. Les contraintes de mise en œuvre du test dans les laboratoires, ainsi que l'accès souvent limité, par les implantations, aux modules que l'on veut tester , rendent irréaliste cette hypothèse de synchronisation parfaite. Il est donc nécessaire de prendre en compte dans notre étude l'influence des diverses architectures de test sur la conception et le pouvoir d'observation des tests.

Ce chapitre est consacré à l'étude de ce problème. Après une analyse des contraintes pratiques de mise en œuvre du test et des principales architectures de test à prendre en compte, nous montrons comment leur effet peut être étudié dans le cadre du modèle défini au chapitre 2. Nous expliquons sur quelques exemples pourquoi les résultats obtenus dans le cas idéal ne sont plus valables. Puis nous présentons une solution très générale, fournie par la théorie, aux deux questions suivantes : quelle relation de conformité peut-on espérer montrer lorsque le test est fait à travers une architecture donnée ? Quel testeur utiliser pour cela ?

Cependant le traitement d'exemples élémentaires montre que le testeur ainsi défini est souvent trop complexe à calculer pour que les résultats théoriques puissent être utilisés directement. Nous proposons donc une autre approche, basée sur l'utilisation de règles empiriques, qui permet de calculer simplement une approximation du testeur précédemment défini, tout en restant dans un cadre rigoureusement justifié par la théorie. Cette approche rejoint de façon naturelle celle utilisée dans TVEDA.

Enfin, nous montrons que les résultats de ce chapitre, qui rejoignent d’ailleurs remarquablement les conclusions obtenues à partir d’un autre modèle [T retmans 92b] [V erhaard 92], ont une influence sur la méthodologie de conception des tests. Ils permettent en particulier de donner une nouvelle interprétation, formelle, à certains concepts définis dans la norme ISO 9646 [ISO 91].

2 Problèmes posés par les architectures de test

2.1 Mise en œuvre et architectures de test

Il est extrêmement rare, dans la pratique, que l’on puisse réaliser une synchronisation parfaite entre l’outil de test et l’implantation que l’on teste. Donc la modélisation du test que nous avons adoptée jusqu’à présent, représentée par la figure 27, n’est pas réa-

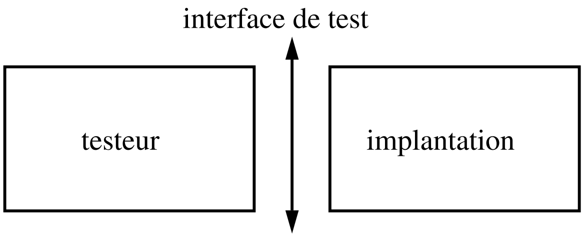


Figure 27 : synchronisation parfaite entre testeur et implantation

liste dans la plupart des cas. Diverses raisons empêchent de mettre en œuvre l’architecture de test ci-dessus :

1. l’équipement physique testé ne fournit pas d’interface directe permettant l’accès à certains des protocoles qu’il implante. Prenons l’exemple d’un terminal téléphonique Numéris : celui-ci présente deux interfaces accessibles, l’une étant l’interface utilisateur (constituée d’un clavier et d’un écran pour l’affichage de certaines informations), l’autre étant la prise de branchement sur le réseau. L’architecture en couches du protocole D implanté dans ce terminal fait que pour accéder à la couche réseau de ce protocole il faut dialoguer avec la couche physique sur une interface, avec la couche application sur l’autre. Ce dialogue indirect ne peut qu’affaiblir les possibilités de contrôle et d’observation de cette couche réseau (figure 28).
2. certaines implantations ne peuvent être testées, en pratique, que de manière distante, à travers un réseau de communication. C’est par principe le cas du service fourni par un centre de test à distance, dont les clients se connectent à travers un réseau (téléphonique, de transmission de données) pour avoir accès à ses services. C’est souvent le cas de fonctions de la couche application, comme un serveur de données par exemple, accessible à travers un réseau (figure 29). Là encore, les possibilités de contrôle et d’observation sur le système sont réduites, en particulier en ce qui concerne les performances du système sous test (temps de réponse)

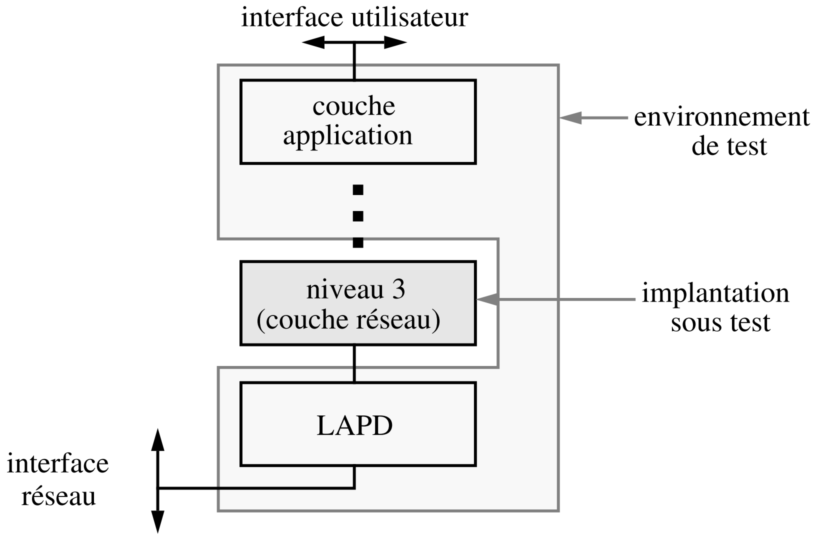


Figure 28 : test à travers des couches de protocoles

dont l'observation est fortement gênée par les délais de transmission des informations dans le réseau, difficiles à connaître exactement.

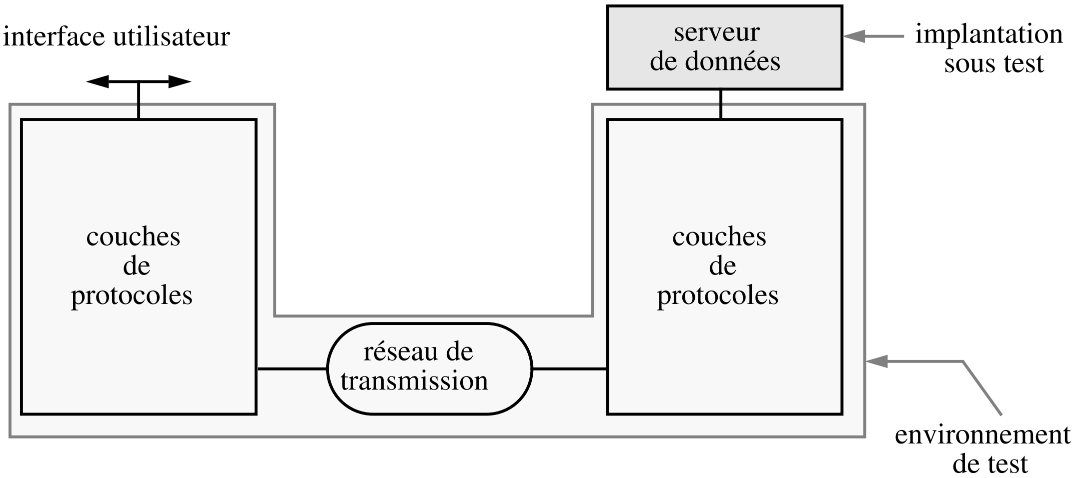


Figure 29 : test à travers un réseau

3. certains systèmes distribués, mettant en jeu des composants physiquement distants, ne peuvent être testés qu'à travers plusieurs interfaces physiquement distantes. C'est le cas par exemple d'un système de communication avec des mobiles, comme le réseau Itinéris, qui met en œuvre simultanément des terminaux mobiles, des autocommutateurs du réseau téléphonique et des bases de données spécialisées, tous spatialement distribués, à l'échelle d'un pays ou d'un continent (figure 30). T ester le fonctionnement global de certains protocoles mis

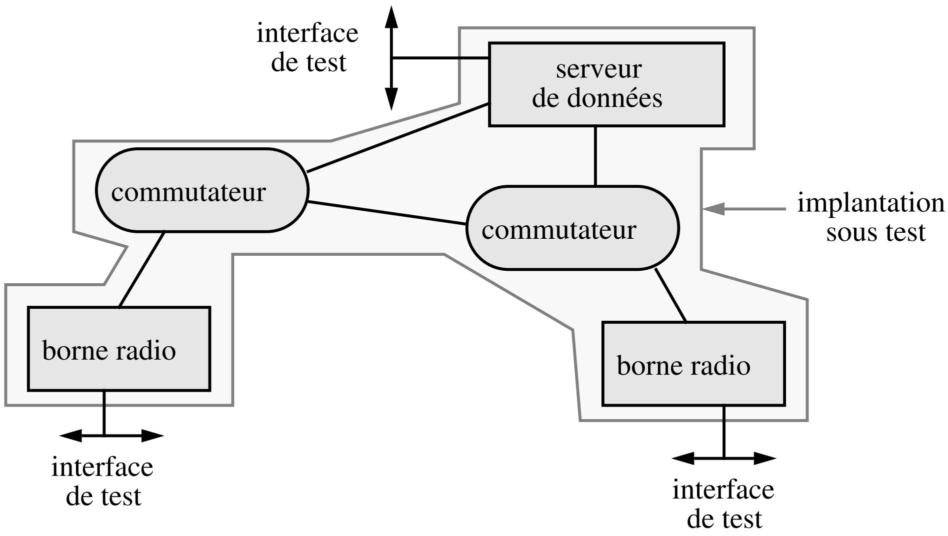


Figure 30 : test à travers plusieurs interfaces

en œuvre nécessite de synchroniser les interactions émises ou reçues sur ces différentes interfaces, ce qui n'est pas toujours possible. En particulier , il peut être difficile ou même impossible de reconstituer l'ordre des événements ayant lieu sur des interfaces différentes, ce qui limite le pouvoir d'observation du test [Bochmann 89] [Dssouli 86].

Dans la suite de ce chapitre, nous appelons :

1. *implantation sous test* (ou IUT) le module ou la fonction particulière dont on veut tester la conformité. Elle peut ne pas avoir de réalisation physique propre (dans l'exemple de la figure 28 elle est généralement incluse dans un composant qui réalise aussi d'autres fonctions), elle peut correspondre à une entité physique bien identifiée (cas de la figure 29), ou elle peut être constituée de plusieurs composants physiques distincts (figure 30).
2. *environnement de test* les modules, fonctions ou équipement à travers lesquels se fait l' accès à l'implantation sous test. Il est généralement imposé par les contraintes pratiques de mise en œuvre du test. Dans certains cas, cet environnement peut être vide : on a alors un véritable accès direct à l'implantation sous test (cas de la figure 27).
3. *interface de test* les points d'accès qui permettent de tester le système. Cette notion correspond à celle de *point de contrôle et d'observation* (PCO) définie dans la norme ISO 9646. Cependant, à la différence de cette norme, nous autorisons un nombre quelconque de PCO pour un système donné. Attention, on trouve parfois dans la littérature le terme interface de test pour désigner ce que nous appelons ici environnement de test [T retmans 92b]. En l'absence d'appellations officielles reconnues, nous préférons nous en tenir aux choix ci-dessus, qui nous paraissent plus proches du sens usuel français des termes employés.

4. *architecture de test* la définition d'un environnement de test et de ses interfaces de tests, adoptés pour la mise en œuvre du test d'un système. Dans le cadre particulier du test de protocoles OSI, la norme ISO 9646 standardise un certain nombre d'architectures (test local, réparti, coordonné, distant). Des architectures diverses ont été proposées pour la réalisation concrète des testeurs (test à cheval, ferry , ferry clip). Pour une synthèse des travaux sur les architectures de tests, le lecteur pourra se reporter à [Rafiq 85] [Rafiq 91] [Sarikaya 89].

Nous étudions dans ce chapitre les conséquences de la présence d'un environnement de test sur ce qu'il est possible de tester , et sur la meilleure façon de le faire. Pour cela, nous modélisons une architecture de test quelconque de la façon représentée par la figure 31. Le testeur , l'environnement et l'implantation sont modélisés par des automa-

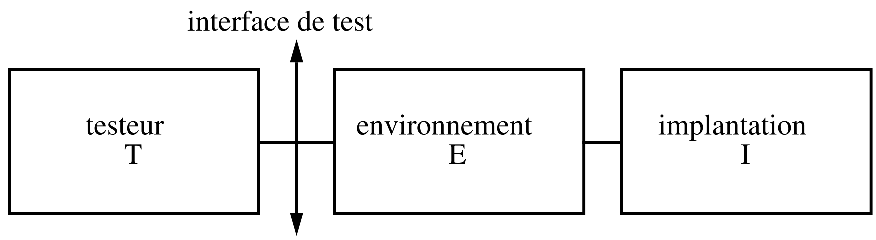


Figure 31 : test à travers un environnement de test

tes à entrées et sorties. Ces automates sont synchronisés par la composition parallèle décrite au chapitre 2. Nous utilisons des ensembles disjoints pour modéliser les messages échangés sur les deux interfaces, l'interface de test entre le testeur et l'environnement, et l'interface entre l'environnement et l'implantation. De cette façon, comme nous l'avons indiqué au chapitre 2, section 2.3, on peut montrer , d'après la définition de la composition parallèle, que $(T \parallel E) \parallel I = T \parallel (E \parallel I)$ que nous pouvons donc noter plus simplement $T \parallel E \parallel I$.

Par contre, l'effet sur le test de la présence de plusieurs interfaces de test au lieu d'une seule n'est pas étudié dans cette thèse. Pour l'étude de cette question, on pourra se reporter aux travaux de R. Dssouli [Dssouli 86] [Bochmann 89] ou de l'équipe de C. Jard [Diehl 93].

Les deux sections suivantes présentent, de manière informelle, quelques uns des problèmes liés à la présence d'un environnement de test.

2.2 Nous ne pouvons plus utiliser le même test

Comme il découle de l'architecture présentée sur la figure 31, le test en présence d'un environnement va s'exprimer en fonction des interactions échangées à l'interface entre le testeur et l'environnement, et non plus au moyen des interactions directement émises ou reçues par l'implantation. Pour obtenir le nouveau test, il faut donc procéder au moins à un renommage des actions. Par exemple, pour tester de façon locale un protocole de la couche transport (côté appelé), on utilisera les messages *ConInd* et *ConResp* .

Par contre, si le test est fait à travers un réseau, en architecture distante, on utilisera les messages *ConReq* et *ConConf* (figure 32).

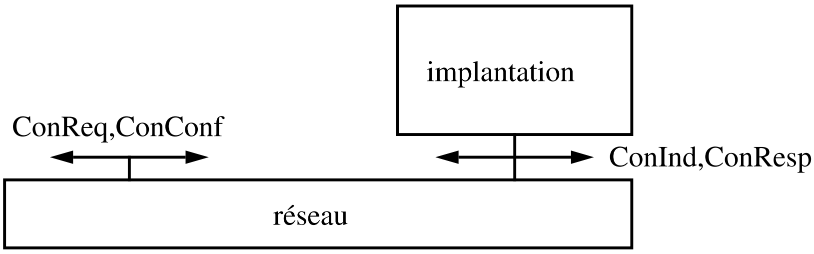


Figure 32 : renommage des actions

Mais dans le cas général l'effet de l'environnement ne se limite pas à un simple renommage. Considérons le cas de la spécification représentée sur la figure 33. Son testeur ca-

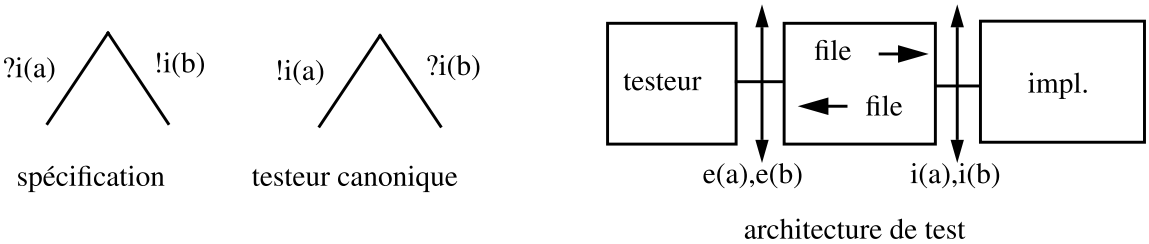


Figure 33 : problèmes posés, exemple 2

nonique est calculé comme indiqué au chapitre 2. L'implantation est testée à travers une double file d'attente (bornée, pour pouvoir être modélisée par une IOSM), qui constitue donc l'environnement de test. Pour les distinguer des interactions i(a) et i(b) échangées à l'interface de l'implantation, notons e(a) et e(b) les mêmes messages échangés à l'interface de test. La trace ?e(a)!e(b) est une trace valide du système composé de l'implantation et de l'environnement. En effet, le message ?e(a) est reçu par la file d'attente, mais avant qu'il parvienne à l'implantation sous test celle-ci émet !i(b), qui est donc transmis sous la forme !e(b) au testeur . Or si on utilise le testeur canonique ci-dessus modulo un simple renommage des actions, cette trace aboutit à un verdict *fail* . Cet exemple montre qu'un effet de désynchronisation introduit par l'environnement peut rendre nécessaire une modification de la structure du testeur canonique.

Enfin, considérons le cas extrême où l'environnement peut fortement perturber les messages transmis entre le testeur et l'implantation. C'est le cas par exemple où le test est fait à travers un canal de transmission non fiable, susceptible de perdre des messages ou de créer des messages aléatoires. Alors, même si l'implantation est parfaitement correcte, le testeur pourra détecter des erreurs, engendrées par l'environnement défectueux : pour ne pas rejeter à tort l'implantation, nous devons là aussi modifier le testeur .

2.3 Nous ne pouvons plus vérifier la même relation

Au chapitre 2 nous avons montré comment définir un testeur canonique caractérisant la relation d'implantation R_1 (définition 2.10). Cette relation exprime le fait que les messages émis par l'implantation sont toujours prévus par la spécification, sauf éventuellement pour des parties non spécifiées du comportement du système.

Si, comme à la section précédente, nous prenons l'exemple d'un canal non fiable, susceptible de perdre des messages, comme environnement de test, alors il est possible que des émissions erronées de l'implantation ne parviennent jamais au testeur . Dans ce cas, le testeur conclura à tort à la conformité de l'implantation. Il est donc évident qu'en présence d'un tel environnement, il est vain de chercher à vérifier une relation d'implantation comme R_1.

Pour un exemple plus réaliste, reprenons la spécification et l'architecture de test présentées à la figure 33. Considérons alors l'implantation suivante :

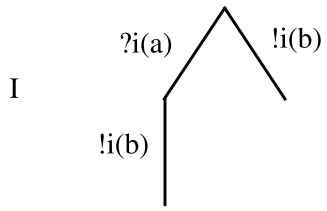


Figure 34 : implantation non conforme

La trace ?i(a)!i(b) de l'implantation montre que celle-ci n'est pas conforme à la spécification au sens de la relation d'implantation R_1 (d'ailleurs, en utilisant le testeur canonique décrit à la figure 33, on peut constater que celui-ci aboutirait à un état *fail*). Mais nous avons vu ci-dessus que la trace ?e(a)!e(b) est une trace valide fournie par une implantation conforme en présence de l'environnement de test. Il est donc impossible de mettre en évidence la non conformité de l'implantation de la figure 34 à travers la double file d'attente. Là encore, en présence de l'environnement imposé, il est vain de chercher à tester la relation d'implantation R_1.

3 Résultats théoriques

3.1 Pouvoir maximal du test

De la même manière qu'au chapitre 2 nous avons montré qu'il y a une limite à la relation d'implantation que l'on peut tester , liée au mécanisme d'interaction entre le testeur et l'implantation, nous montrons ici qu'il y a une limite à la relation d'implantation que l'on peut vérifier à travers un environnement de test donné.

Soit E un automate à entrées et sorties. Nous définissons l' *équivalence de traces modulo l'environnement* E :

Définition 4.1 : $R_t(E)\ (I, S)$ ssi $Tr(E \| I) = Tr(E \| S)$

L'exécution d'un test, en présence d'un environnement, est modélisée par la composition parallèle décrite à la figure 31. Par analogie avec ce qui est fait au chapitre 2 (définition 2.19), le verdict fourni par le test est ici défini (dans le cas d'un verdict global) par :

Définition 4.2 : $Fail_E(T, I)$ ssi $(\exists \sigma \in Tr(T))\ (\bar{\sigma} \in Tr(E \| I) \wedge (T, \sigma, fail))$
$Succ_E(T, I)$ ssi $\neg(Fail_E(T, I))$

Nous allons montrer que si l'on garde la définition ci-dessus du succès d'un test, alors on ne peut pas définir de testeur qui caractérise, à travers l'environnement E, une relation d'implantation plus forte (au sens de la définition 2.20) que $R_t(E)$. Soit $R' \geq R_t(E)$ une relation d'implantation qui de plus satisfait : pour toute spécification S il existe au moins une implantation I telle que $R'\ (I, S)$.

Propriété 4.1 : $\forall T, S \in IOSM$,
$(\forall I \in IOSM)\ (R'\ (I, S) \Rightarrow Succ_E(T, I)) \Rightarrow (\forall I \in IOSM)\ (R_t(E)\ (I, S) \Rightarrow Succ_E(T, I))$

Démonstration : elle est calquée sur celle de la propriété 2.8. Nous prouvons la contraposée de cette implication, à savoir que si il existe I telle que $R_t(E)\ (I, S)$ et $Fail_E(T, I)$ alors il existe I' telle que $R'\ (I, S)$ et $Fail_E(T, I')$. Par définition de l'échec du test, si $Fail_E(T, I)$ alors il existe $\sigma \in Tr(T)$ telle que $\bar{\sigma} \in Tr(E \| I)$ et $(T, \sigma, fail)$. Puisque $E \| I$ et $E \| S$ ont les mêmes traces, $\bar{\sigma} \in Tr(E \| S)$.

Soit I' quelconque telle que $R'\ (I', S)$ (par hypothèse il existe au moins une telle implantation). Par définition de la relation $\geq$, puisque $R' \geq R_t(E)$ et $R'\ (I', S)$ alors $R_t(E)\ (I', S)$. Comme $\bar{\sigma} \in Tr(E \| S)$, on en déduit que $\bar{\sigma} \in Tr(E \| I')$. Nous avons donc trouvé $\sigma \in Tr(T)$ telle que $\bar{\sigma} \in Tr(E \| I')$ et $(T, \sigma, fail)$, ce qui par définition signifie $Fail_E(T, I')$. ❑

Il faut noter que la propriété ci-dessus n'est pas exactement l'application de la propriété 2.8 au cas où la spécification est $S \| E$, car on fait ici l'hypothèse supplémentaire que l'implantation de l'environnement E est égale à sa spécification. Cette hypothèse est couramment faite en pratique. Lorsque nous voulons étudier le cas d'environnements ayant des comportements bizarres (comme ceux des exemples de la section précédente), ceux-ci sont décrits explicitement par la spécification de l'environnement.

Il découle de la propriété ci-dessus que le pouvoir d'observation du test ne peut qu'être réduit lorsqu'on teste à travers un environnement, ce qui correspond bien à l'intuition.
En effet :

Propriété 4.2 : $(\forall E \in IOSM)\ (R_t(E) \leq R_t)$

Démonstration : soient E, I et S des IOSM quelconques. De la définition de la composition parallèle sur les automates à entrées et sorties, on déduit que $Tr(I) = Tr(S)$ implique $Tr(E \| I) = Tr(E \| S)$, donc que $R_t(I, S)$ implique $R_t(E)\ (I, S)$. ❑

3.2 T esteur canonique

Une application immédiate des résultats du chapitre 2 permet de construire une théorie du test adaptée au cas où la communication entre le testeur et l'implantation se fait à travers un environnement. Comme nous venons de le montrer , il nous faut dans ce cas considérer une nouvelle relation d'implantation, plus faible. Soient I, E, S trois IOSM. Nous définissons la relation d'implantation suivante:

Définition 4.3 :$R_1(E)\ (I, S)$ ssi
$(\forall \sigma \in Tr(S \| E))\ (\sigma \in Tr(I \| E) \Rightarrow O(\sigma, I \| E) \subseteq O(\sigma, S \| E))$

On s'aperçoit sur cette définition que $R_1(E)\ (I, S) \Leftrightarrow R_1(I \| E, S \| E)$. Nous déduisons alors de la propriété 2.7 que le testeur $TCA(E \| S)$ est un testeur canonique permettant de caractériser cette relation d'implantation. La définition du succès du test est celle de la définition 4.2.

Propriété 4.3 : $(\forall I, S, E \in IOSM)\ (Succ_E(TCA(E \| S), I) \Leftrightarrow R_1(E)\ (I, S))$

Démonstration : en appliquant la propriété 2.7 à la spécification $E \| S$, on déduit : $(\forall I \in IOSM)\ (Succ(TCA(E \| S), I) \Leftrightarrow R_1(I, E \| S))$. Ceci est vrai en particulier pour toutes les implantations de la forme $E \| I$, donc $Succ(TCA(E \| S), E \| I) \Leftrightarrow R_1(E \| I, E \| S)$, ce qui d'après les définitions de $R_1(E)$ et de $Succ_E$ signifie $Succ_E(TCA(E \| S), I) \Leftrightarrow R_1(E)\ (I, S)$. ❑

Il est important de souligner que les résultats obtenus dans les sections 3.1 et 3.2 sont vrais quel que soit l'environnement de test (à condition que celui-ci puisse être modélisé par une IOSM). Ils ont donc une portée très générale, plus large par exemple que les travaux similaires mentionnés dans la section 3.4.

3.3 Exemples

Pour bien comprendre la signification pratique des résultats ci-dessus, reprenons la spécification présentée à la figure 33. Dans un premier temps, supposons qu'on a oublié de brancher les fils entre le testeur et l'implantation sous test : ce cas trivial correspondant à un environnement trou noir , qui ne laisse passer aucun message dans aucune direction. Un tel environnement est modélisé sur la figure 35. D'après la définition de la composition parallèle, quelle que soit l'implantation I considérée, les traces de $E \| I$

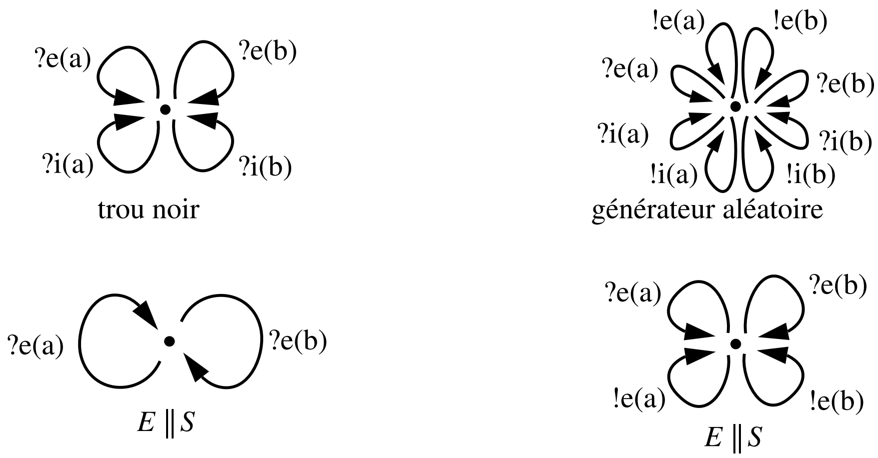


Figure 35 : deux environnements extrême et leurs effets visibles

sont toutes les séquences possibles de ?e(a) et ?e(b). De cette propriété et du calcul de $E \| S$ (voir figure 35), on déduit que :

1. le testeur canonique est très sévère : on envoie toutes les séquences possibles à l'implantation, et dès que l'on reçoit une réponse, on crie à l'erreur !
2. quelle que soit l'implantation que l'on teste on la trouvera conforme à la spécification.

Un autre exemple trivial est celui évoqué à la section 2.2 où l'environnement est un canal peu fiable susceptible de perdre des messages ou de créer des messages aléatoires (mais, tout de même, appartenant à l'alphabet autorisé). Un tel environnement peut être modélisé par le générateur aléatoire représenté sur la figure 35. On s'aperçoit dans ce cas que quelle que soit l'implantation I considérée, sa composition avec l'environnement est égale à l'automate $E \| S$ représenté sur la figure. On en déduit que :

1. le testeur canonique est très libéral : il accepte tout de la part du système sous test, et ne signale jamais d'erreur .
2. en conséquence, toutes les implantations sont là aussi déclarées conformes à la spécification.

Prenons maintenant un exemple plus sérieux, celui où l'environnement est la double file d'attente bornée décrite à la section 2.2. Pour pouvoir représenter cet environnement, nous considérons des files d'attente à une case seulement. On trouve sur la figure 36 les représentations de l'environnement, de sa composition avec la spécification (celle de la figure 33), l'automate de traces correspondant, et le testeur canonique qui en résulte. On peut constater que malgré la simplicité de l'environnement choisi, les calculs sont déjà relativement compliqués. On constate alors que l'implantation décrite à la figure 34, qui était non conforme modulo la relation d'implantation R_1, doit maintenant être considérée comme conforme, ce qui est logique puisqu'on ne peut pas la distinguer , à travers l'environnement, d'autres implantations qui sont naturellement conformes.

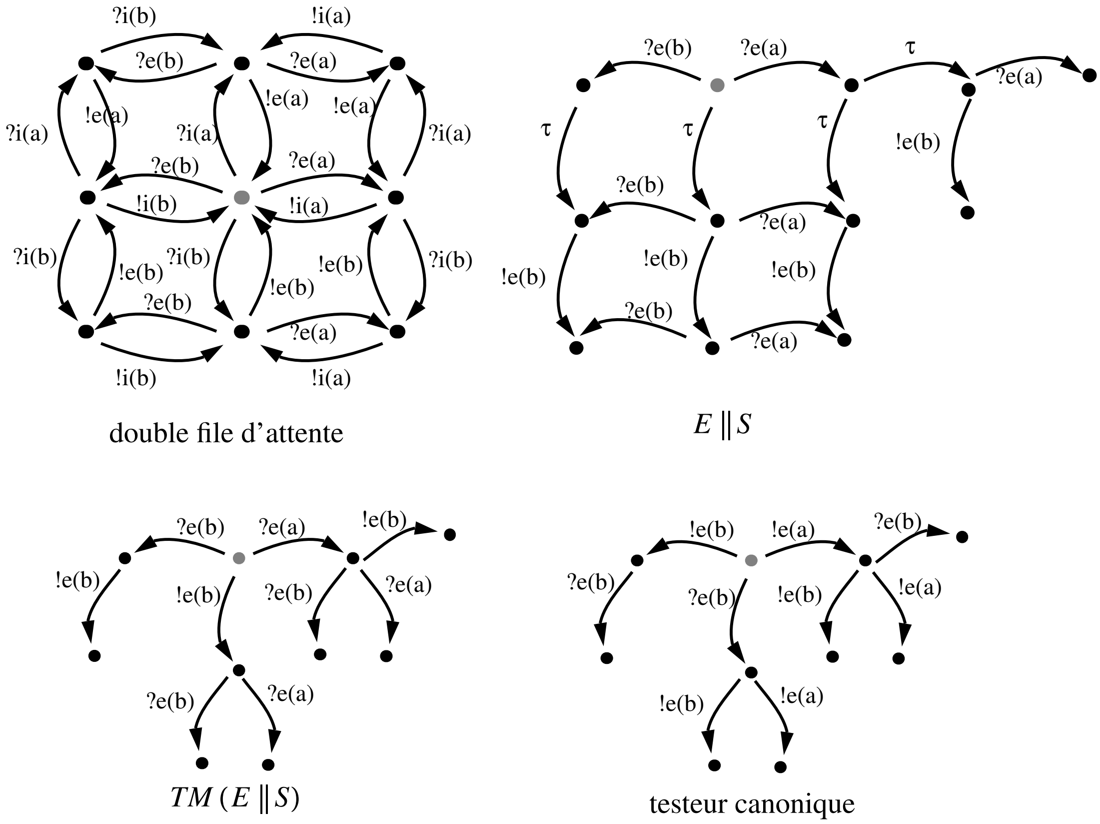


Figure 36 : exemple de la double file à une case

Le nouveau testeur canonique est donc plus tolérant, il accepte plus d'implantations. Par contre, il ne signale plus d'erreurs à tort, comme celle analysée à la section 2.2.

3.4 Comparaison avec d'autres travaux sur le sujet

Il existe des travaux, basés sur le modèle synchrone (les systèmes de transitions étiquetées présenté au chapitre 3), qui visent également à étudier les conséquences de la présence d'un environnement de test. Nous résumons ici brièvement ces travaux, dont les résultats présentent une grande similitude avec ceux que nous avons obtenus. Pour plus de détails, se reporter aux articles [T retmans 92a] [V erhaard 92] [Kristoffersen 92] ou à la thèse [T retmans 92b].

Il s'agit d'étudier quelles modifications apporter aux résultats de la théorie du test synchrone (voir chapitre 3 section 2) lorsque le testeur et l'implantation communiquent au travers de files d'attentes non bornées. C'est donc un cas particulier d'environnement de test tel que nous l'avons défini dans ce chapitre. Stricto sensu, nous ne pouvons pas modéliser une file non bornée par des IOSM (le nombre d'états est infini). Cependant, les résultats obtenus peuvent être comparés avec ceux que nous obtenons dans le cas d'une file d'attente bornée, étudié à la section précédente.

L'approche suivie dans ces travaux consiste à modéliser explicitement deux files d'attentes non bornées par des systèmes de transitions étiquetées, et à appliquer les résultats de la théorie du test synchrone au système composé de la spécification et des deux files d'attente. On montre ensuite comment la vérification de la relation d'implantation *conf* à l'interface externe du système global revient à vérifier une nouvelle relation d'implantation, plus faible, entre l'implantation et la spécification. Nous ne rapportons pas ici les détails techniques de ces études, mais il est important de noter les principaux résultats :

1. la relation d'implantation utilisée dans le cadre d'une communication directe entre le testeur et l'implantation doit être abandonnée, car on ne peut plus la vérifier, le pouvoir d'observation du test étant perturbé par l'environnement. De nouvelles relations d'implantation spécifiques à la communication à travers l'environnement sont introduites.
2. de même, le testeur canonique ne peut plus être utilisé tel quel, car il donne de faux verdicts d'erreur . Ces résultats rejoignent exactement ceux que nous avons exposés à la section 2.
3. la nouvelle relation d'implantation est basée sur les *sorties* et les *blocages* observables après une trace du système composé de la spécification et de l'environnement. Selon que l'on étudie la relation d'équivalence ou les préordres correspondants, on retrouve, en nous plaçant dans le cas où les blocages de sortie sont modélisés explicitement par l'action δ, la relation $R_1(E)$ étudiée dans ce chapitre, ou les relations $R_5(E)$ ou $R_3(E)$ qu'il est facile de définir par extrapolation, à partir de celles définies au chapitre 2.

4 Application pratique

La section précédente nous fournit un moyen théorique de calculer un testeur canonique pour une spécification S que l'on teste à travers un environnement E. Cependant, en pratique, un tel calcul pose problème :

1. dans des cas réels, modéliser l'environnement est pénible, quand ce n'est pas impossible pour des raisons de coût (pensons à un empilement de couches OSI par exemple). De plus, dans certains cas de systèmes très complexes, comme un autocommutateur ou un réseau complet de transmission par exemple, on ne connaît pas forcément l'environnement de test exact.
2. même dans le cas où la modélisation de l'environnement est réalisable, le calcul de sa composition avec la spécification est souvent compliqué, comme le montre l'exemple de la file d'attente bornée présenté à la section précédente. On peut pourtant, dans la pratique, difficilement considérer un environnement plus simple !

Or ce qui importe, lors du calcul du test, c'est d'appréhender l'effet de l'environnement sur le testeur que l'on cherche à définir . Dans cette section, nous allons montrer que certaines caractéristiques des environnements couramment mis en œuvre peuvent être prises en compte au niveau du testeur en définissant celui-ci à partir d'une version transformée de la spécification. Intuitivement, cette version transformée modélise la spécification *telle qu'on peut l'observer* à travers l'environnement de test. Les transformations que nous allons définir constituent un ensemble d'heuristiques, ou règles empiriques, qui permettent de calculer un testeur tenant compte de la présence de l'environnement, sans modéliser explicitement cet environnement sous la forme d'un automate ni avoir à calculer sa composition parallèle avec la spécification. Nous résumons cette démarche par le tableau ci-dessous :

	spécification	*environnement*	*calcul à faire*	*testeur*
théorie	S	E	$E \parallel S$	$TCA(E \parallel S)$
application pratique	S	règles empiriques R	$S' = R(S)$	$TCA(S')$

Dans le reste de cette section nous allons donner des exemples concrets d'application de cette méthode. Nous essayons, sur des cas extrêmement simples, de justifier l'utilisation des règles empiriques, en montrant leur équivalence avec le calcul exact exigé par la théorie. Puis nous commentons un exemple plus réaliste, issu de l'expérience TVEDA [Phalippou 91].

Il faut souligner que les règles empiriques étudiées ici sont celles définies dans [Phalippou 91] pour le calcul d'une abstraction de la spécification servant à la définition des tests. Il est intéressant de noter que lors de l'étude TVEDA ces règles ont été obtenues à partir de l'observation des méthodes utilisées manuellement pour produire les tests, et ce n'est qu'a posteriori, au cours de la formalisation théorique faite dans le cadre de cette thèse, qu'on a mis en évidence la possibilité de les justifier formellement. Il y a sur ce point une bonne convergence entre la pratique et la théorie.

4.1 Renommage des actions

Reprenons l'exemple de test à travers un réseau présenté à la figure 29. Pour simplifier, supposons que le réseau n'introduit pas de délai de transmission des messages, donc qu'il n'y a pas possibilité de croisement entre interactions (l'effet d'un tel croisement sera discuté à la section 4.5). Le réseau effectue donc un simple renommage des actions, et peut être modélisé par l'automate représenté sur la figure 37. La règle empirique traduisant l'effet d'un tel environnement consiste en un renommage des actions (ConInd devient ConReq et ConResp devient ConConf). On peut alors vérifier , pour la spécification représentée sur la figure ci-dessus, que $TCA(E \parallel S) = TCA(R(S))$.

En d'autres termes, pour calculer le testeur théoriquement correct en présence de l'environnement E, tout se passe donc comme si on avait d'abord appliqué un renommage aux actions de la spécification, suivi du calcul du testeur canonique simple par inversion des entrées et des sorties. D'où la définition :

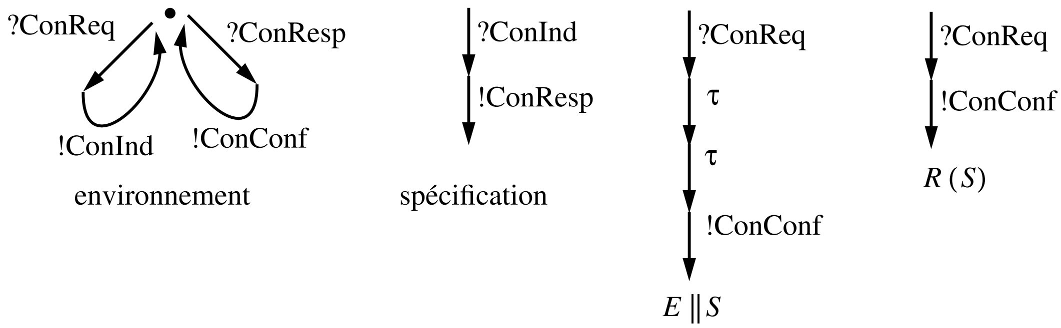


Figure 37 : renommage des actions

Règle empirique 1 : appliquer un renommage (mécanisme d'alias) sur les actions de la spécification.

4.2 Événements non observables

Certains environnements de test ne permettent pas d'observer toutes les interfaces de la spécification à tester . Prenons l'exemple, étudié pour l'expérience TVEDA, du test du protocole LAPD [CCITT 88b]. Ce protocole communique avec trois entités distinctes, mais pour la définition de la suite de test normalisée du LAPD, on se place dans le cas où l'interface avec le module de gestion n'est pas accessible au testeur . De plus, les actions correspondant à des manipulations sur des temporisateurs (armement, réarmement) sont des actions internes au protocole et ne sont pas directement visibles du testeur , même si elles sont explicitement modélisées dans l'automate représentant le protocole. Enfin, dans le cas où l'on emploie la méthode de test distante (l'une des architectures de test normalisées par l'ISO), le testeur n'a pas accès à l'interface supérieure du protocole testé. T out ceci est illustré sur la figure ci-dessous.

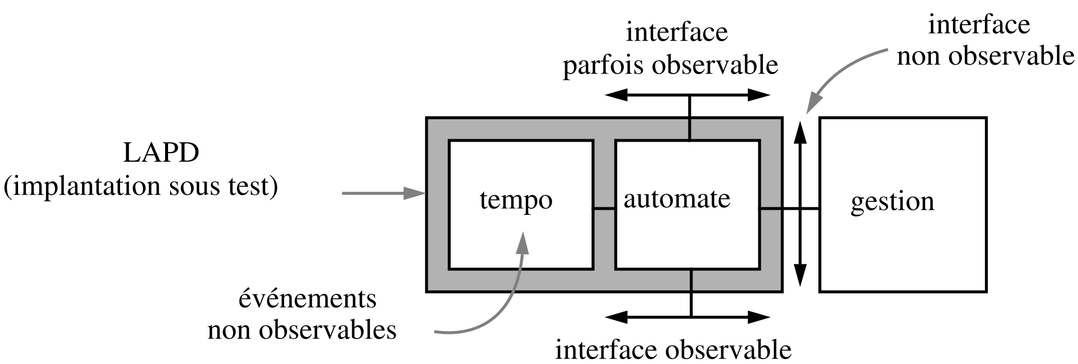


Figure 38 : événements non observables (les architectures en cause)

Considérons par exemple l'environnement de test représenté sur la figure 38. Il transmet, en les renommant comme dans l'exemple de la section précédente, les messages

ConReq et ConResp, mais il masque le message InfGest, qui est émis par la spécification vers l'interface avec le module de gestion, non observable. On peut constater qu'en plus du renommage appliqué précédemment, il convient de remplacer l'événement non observable InfGest par l'action interne τ afin de traduire correctement l'effet de l'environnement (les calculs sont illustrés sur la figure 39). D'où la définition :

Règle empirique 2 : renommer en τ les émissions non observables de la spécification.

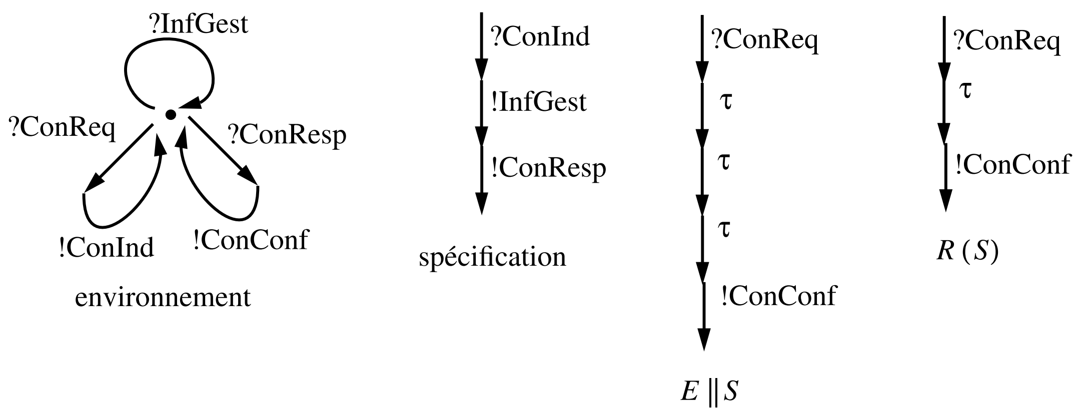


Figure 39 : événements non observables (règle empirique)

On remarque sur cet exemple que nous devons ici utiliser à la fois les règles empiriques 1 et 2 pour modéliser l'effet de l'environnement (il est difficile de trouver un exemple où la règle 1 ne s'applique pas, puisque par hypothèse nous supposons que les actions n'ont pas le même nom à l'interface entre le testeur et l'environnement et à l'interface entre l'environnement et l'implantation. Donc nous devons toujours introduire un renommage, sauf bien évidemment pour un test local, architecture dans laquelle il n'y a pas d'environnement).

De plus, cet exemple montre que la notion d'action interne est relative à l'interface observable (donc l'interface de test) d'un système : les actions liées au temporisateur sont, du point de vue du test, traitées comme celles liées au module de gestion.

4.3 Événements non contrôlables

Certains environnements de test ne donnent pas au testeur une maîtrise complète des événements déclenchés aux interfaces de la spécification à tester . Par exemple, reprenons l'architecture de test à travers un réseau de la figure 32. En plus du mécanisme d'établissement de connexion qu'il véhicule, on suppose que le réseau peut envoyer de façon spontanée une indication de déconnexion (DisInd) à l'implantation (ceci modélise une possible rupture de la liaison, détectée par le réseau). La spécification doit alors confirmer cette déconnexion, par le message DisResp, transformé par le réseau en DisConf (voir figure 40).

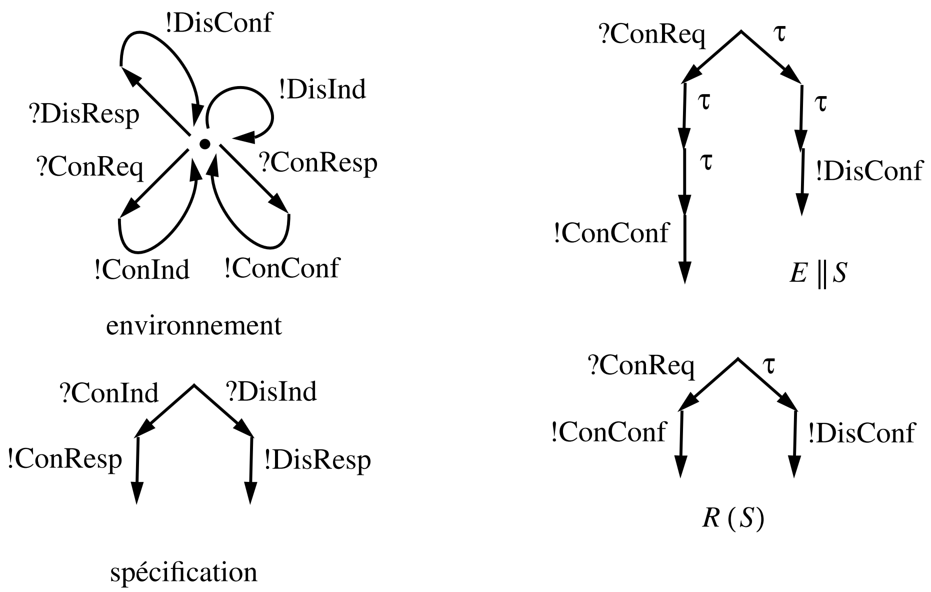


Figure 40 : événements non contrôlables

En plus du renommage, il convient ici de remplacer l'événement non contrôlable DisInd par l'action interne τ afin de traduire correctement l'effet de l'environnement (les résultats des calculs sont illustrés sur la figure 40). D'où la définition :

Règle empirique 3 : renommer en τ les réceptions non contrôlées de la spécification.

Sur cet exemple on remarque que la notion d'événement spontané est relative elle aussi à l'interface contrôlable, c'est à dire l'interface de test, d'un système. En particulier les événements non contrôlables sont traités exactement comme les échéances de temporisateur , qui sont des événements spontanés internes de la spécification.

4.4 Événements non testables

Certaines configurations de test ne permettent pas de déclencher toutes les transitions prévues par la spécification. Reprenons encore une fois (on ne s'en lasse pas !) l'exemple du test à travers le réseau de la figure 32. Maintenant supposons que la spécification, en plus du mécanisme d'établissement de connexion déjà décrit, prévoie que si on reçoit une interaction erronée (message Invalid) alors on signale l'erreur (message Erreur). Le réseau à travers lequel se fait le test ne permet pas d'envoyer d'interaction erronée à l'implantation. Il est donc impossible de tester le mécanisme de signalisation d'erreur (voir figure 37). Pour prendre en compte ce phénomène, nous introduisons la règle :

Règle empirique 4 : supprimer les transitions de la spécification correspondant à des événements non testables.

On rappelle que nous ne considérons que la composante connexe du graphe de transitions de l'automate contenant l'état initial. Donc lorsqu'on supprime une transition dans la spécification, cela peut entraîner de fait la suppression d'autres transitions. C'est le cas dans l'exemple ci-dessus, où l'élimination de la transition étiquetée par ?Invalid entraîne celle de la transition étiquetée par !Erreur .

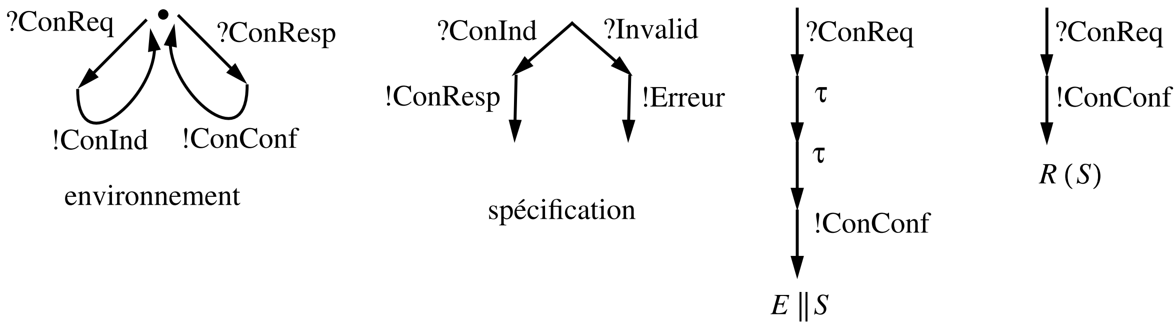


Figure 41 : événements non testables

Cet exemple est caractéristique des difficultés parfois rencontrées pour mettre en œuvre le test de robustesse des implantations à travers un environnement normal. Le test de robustesse consiste à tester les fonctions prévues pour lutter contre les réceptions d'interactions erronées (ici le message Invalid). Mais en général, un environnement correct protège l'implantation contre de telles interactions, c'est à dire qu'il ne peut pas émettre ces interactions erronées. On est donc dans ce cas obligé de faire appel, pour tester ces fonctions, à des environnements dédiés au test, qui permettent de créer des erreurs à la demande.

4.5 Désynchronisation

Dans tous les exemples précédents nous avons vu que l'utilisation des règles empiriques permet d'obtenir exactement le même testeur que le calcul exact exigé par la théorie. Nous étudions enfin un cas un peu plus complexe, où la règle empirique ne permet d'obtenir qu'une approximation du testeur . Cependant, comme nous allons le voir , cette approximation est suffisante pour nos besoins de test, en un certain sens que nous nous contenterons ici de définir de façon informelle.

Reprenons l'exemple présenté par la figure 36, où l'environnement de test est constitué d'une double file d'attente à une case. Nous avons expliqué que la présence de cet environnement permet à des messages en direction et en provenance de l'implantation de se croiser : il introduit une désynchronisation entre le comportement du testeur et celui de l'implantation, qui rend invalide le testeur canonique calculé directement à partir de la spécification.

Un tel effet de l'environnement est modélisé dans TVEDA de la façon suivante. Nous définissons les *états instables* comme les états de la spécification qui comportent à la fois au moins une transition étiquetée par une entrée et transition étiquetée par une sortie. Ces états correspondent à des situations où des réceptions sont en concurrence avec des événements spontanés produits par la spécification, d'où leur appellation «instable». La règle empirique utilisée (règle 5) consiste à transformer les états instables comme indiqué sur la figure 42.

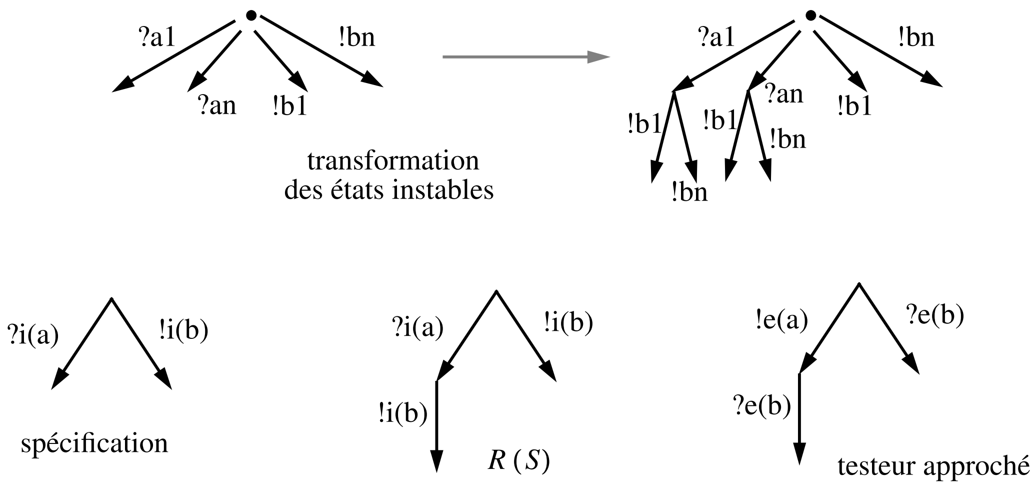


Figure 42 : désynchronisation

Le testeur canonique ainsi calculé est différent de celui calculé selon la théorie et que nous avons représenté sur la figure 36. Cependant, on peut remarquer que dans ce testeur théorique les branches des extrémités (étiquetées par des !e(a) et !e(b)) ainsi que la branche débutant par !e(b) servent en fait à tester le comportement de la double file d'attente, c'est à dire à vérifier qu'elle ne contient qu'un message au plus (la spécification ne prévoit qu'une réception de ?i(a), donc ce qui se passe ensuite ne nous intéresse pas). La partie vraiment utile du testeur est celle qui vérifie que l'on peut envoyer !e(a) vers la spécification, et que ce message peut éventuellement se croiser , sans que l'on puisse contrôler ce phénomène, avec l'émission de !i(b) par l'implantation. Or on constate que cette partie du testeur est correctement obtenue par application de la règle empirique. Contrairement au calcul direct, la règle empirique permet donc d'obtenir un testeur qui ne signale pas d'erreur à tort, ce qui est la condition minimale à respecter . L'approximation obtenue est donc satisfaisante. Elle constitue une sorte d'approximation «par défaut», puisque le testeur détecte moins d'erreurs que le testeur exact.

Une telle définition de l'approximation est très informelle, mais aller plus loin dans la formalisation nous obligerait à des développements complexes dans d'autres directions que celles étudiées dans cette thèse. Il faudrait en effet introduire un moyen permettant de décrire ce que signifie une vérification approchée de propriétés d'un système. Ce sujet constitue une vaste étude en soi, pour laquelle l'article [Brinksma 93] développe quelques idées nouvelles.

4.6 Une autre approche de la modélisation de l'environnement

Nous avons montré, sur des exemples simples, que l'on peut dans certains cas particuliers retrouver le testeur prévu par la théorie au moyen de transformations simples sur la spécification initiale, en évitant ainsi la modélisation explicite de l'environnement de test, et le calcul, parfois complexe, de sa composition parallèle avec la spécification à tester . Quand les règles empiriques ne fournissent pas le résultat exact attendu, du moins obtenons nous une approximation acceptable (au sens discuté à la section précédente) du testeur .

Il ne nous reste plus qu'à extrapoler à partir de ces quelques exemples et à affirmer que cette démarche est générale. C'est à dire qu'il est toujours possible de prendre en compte l'effet d'un environnement de test, même complexe, au moyen de quelques règles empiriques simples. Le test résultant, même s'il n'est pas exactement celui que fournirait la théorie, est toujours une solution acceptable.

Nous justifions ce point de vue par l'expérience acquise au CNET à travers l'étude de plusieurs exemples significatifs de génération de tests. Nous avons produit des tests pour le protocole D (niveaux 2 [CCITT 88b] et 3 [CCITT 88c]), pour le protocole de messagerie P1 [CCITT 88d], pour le protocole MAP de la norme GSM [CCITT 88e] (le système de communications avec les mobiles Itinéris), pour l'exemple INRES étudié dans le cadre du groupe de travail ISO-CCITT sur les méthodes formelles pour le test de conformité [Hogrefe 92]. Dans tous les cas, les architectures de test utilisées, qui étaient imposées par le contexte de l'étude, ont pu être modélisées par l'utilisation des règles empiriques 1 à 5 définies ci-dessus.

Nous pouvons même inverser le point de vue ci-dessus : si les automates à entrées et sorties constituent un formalisme adapté à la modélisation de la spécification, les règles empiriques constituent un formalisme adapté à la modélisation de l'environnement. L 'expérience pratique dont nous faisons état au paragraphe précédent nous amène à conclure que modéliser l'environnement en utilisant un tel formalisme est relativement aisé, car il est bien adapté au niveau d'abstraction auquel il est pertinent de décrire l'environnement.

Nous obtenons donc une méthode à la fois *simple* et *théoriquement exacte* pour calculer le testeur canonique d'une spécification en présence d'un environnement de test.

5 Méthodologie de conception des tests ISO

La norme ISO 9646 [ISO 91], dans sa partie 2, définit un processus de production des suites de tests. Lors du développement de ces suites, l'influence de l'architecture de test est prise en compte de la façon suivante :

1. après avoir choisi les buts des tests, on construit d'abord des cas de tests *génériques* , c'est à dire les cas de tests correspondant à l'architecture appelée *conceptuelle* dans la norme. Comme le montre [T retmans 92b], cette architecture correspond à la situation idéale où le testeur et l'implantation sont en contact direct, représentée sur la figure 27, et étudiée dans les deux chapitres précédents.
2. on choisit ensuite une méthode de test parmi celles qui sont normalisées, ce qui définit une architecture de test particulière. Il convient alors de transformer les cas de tests génériques pour obtenir les cas de tests dits *abstraits* , qui correspondent au test en présence d'un environnement, comme étudié dans ce chapitre.

La relation entre les cas de tests génériques et abstraits est résumée de façon très informelle dans une annexe informative de la norme ISO 9646, où l'on trouve également, toujours décrite de manière très informelle, une méthode pour calculer les tests abstraits à partir des tests génériques.

À la lumière des résultats de ce chapitre, il apparaît qu'en *théorie* l'étape des tests génériques n'est pas forcément un préliminaire pertinent pour la conception des tests abstraits. En effet, le test abstrait $TCA(E \| S)$ ne s'exprime pas comme une transformation du test générique $TCA(S)$. Ce résultat plaide pour une conception directe des tests abstraits, pour une spécification et une architecture de tests données.

Cependant, la méthode approchée par utilisation de règles empiriques, telle que nous l'avons exposée à la section précédente, permet de donner un sens formel, et une utilité retrouvée, aux tests génériques. En effet, n'oublions pas que le testeur canonique asynchrone est obtenu par une simple inversion des entrées et des sorties sur l'automate de traces de la spécification. Donc au lieu d'appliquer d'abord les règles empiriques à la spécification, puis de calculer le testeur de l'automate ainsi obtenu, nous pouvons d'abord calculer le testeur de la spécification originale (c'est à dire le test générique), puis appliquer les règles empiriques modélisant l'effet de l'environnement. Il ne reste plus qu'à prendre enfin l'automate de traces de l'automate ainsi obtenu. On permute donc l'application des règles empiriques avec le calcul du testeur . Bien que l'on puisse imaginer des règles empiriques pour lesquelles cette permutation ne conserve pas le résultat final, on constate que pour les règles 1 à 5 ci-dessus ce n'est pas le cas. Nous pouvons donc interpréter ces règles comme une formalisation de la transformation existant entre les tests génériques et les tests abstraits. Cette interprétation de la norme 9646 est résumée sur le tableau ci-dessous :

concepts ISO 9646	spécification	test générique	environnement	test abstrait
interprétation formelle	IOSM S	$TCA(S)$	règles empiriques R	$TM(R(TCA(S)))$

Remarquons que nous aurions pu présenter l'utilisation des règles empiriques en suivant directement l'approche ci-dessus. Cependant, il nous a paru plus logique, en tout cas plus cohérent avec les résultats théoriques de la section 3, d'interpréter d'abord ces règles comme une transformation sur la spécification, et non comme une transformation sur le test.

CHAPITRE 5

Hypothèse de régularité

1 Introduction

Après avoir étudié, trois chapitres durant, le pouvoir du test, nous abordons maintenant la deuxième partie de cette thèse, visant à répondre à la question : comment sélectionner un nombre raisonnable de tests ?

Les testeurs sont les objets mathématiques que nous utilisons pour modéliser les suites de tests. Leur taille reflète directement la longueur des suites de tests ainsi définies, autrement dit l'effort de test, forcément limité en pratique pour des raisons de coût, de rentabilité de l'activité de test. Or les testeurs que nous avons définis jusqu'à présent peuvent être très gros, allant même jusqu'à représenter des tests de longueur infinie. Cela provient du fait que nous voulons tester une relation d'implantation assez forte, portant sur l'ensemble des traces du système sous test, alors que nous ne savons rien a priori de l'implantation testée, si ce n'est qu'elle est modélisée par un automate à entrées et sorties.

La première partie de la thèse nous conduit à penser que les relations d'implantation qui y sont étudiées correspondent au type de conformité que l'on cherche à vérifier en pratique. Dès lors que nous renonçons à envisager des relations d'implantation plus faibles, nous n'avons d'autre solution, pour réduire la taille des testeurs, que de restreindre l'espace des implantations potentielles. C'est ce que nous appelons faire des *hypothèses de test*, un concept (et un terme) que nous empruntons à [Bernot 91]. Les cinq chapitres à venir vont décliner cinq types différents, couramment utilisés en pratique, de telles hypothèses.

Nous commençons, dans ce chapitre, par étudier une hypothèse de *régularité*, qui consiste à supposer une limite au nombre d'états des automates modélisant les implantations. Cette première hypothèse nous permet, dans tous les cas, de nous contenter de vérifier une conformité portant sur un sous-ensemble fini du comportement de la spécification. De plus, nous montrons que c'est sur cette hypothèse que sont basées la plupart des méthodes usuelles de test d'automates finis. En nous inspirant des développements les plus récents autour de ces méthodes [Luo 93], nous montrons comment réduire, sous une telle hypothèse, la conformité à vérifier dans le cas des automates à entrées et sorties.

2 Sélection des tests

2.1 Effort de test et taille des testeurs

Sélectionner un nombre raisonnable de tests est une obligation imposée par les moyens nécessairement limités qui sont affectés au processus de test d'un système. Il est difficile d'établir une limite universelle, tant est grande la variété des systèmes à tester, et donc la complexité des tests individuels (qui dépend de ce qui est considéré comme un test «élémentaire») et les moyens disponibles pour une campagne de test donnée (degré d'automatisation de l'exécution des tests, ressources humaines disponibles). Cependant, l'observation de nombreuses expériences de test au sein du CNET nous conduit à penser qu'au delà de quelques milliers de tests simples, consistant chacun en quelques échanges d'interactions entre testeur et implantation sous test, le coût du test devient en général trop élevé.

Nous avons choisi de modéliser au moyen d'un testeur toutes les exécutions nécessaires de tous les tests d'une suite complète de tests (chapitre 2, section 4.1). Le test lui-même est représenté par la composition parallèle du testeur et de l'implantation sous test, le verdict étant obtenu, pour les exemples étudiés dans les chapitres précédents, par l'observation des traces de cette composition parallèle. Au chapitre 9 nous étudierons d'autres testeurs auxquels est associée une définition différente du verdict de test, mais portant également sur les traces de la composition parallèle avec l'implantation. Compte tenu de cette modélisation, il apparait donc naturel de mesurer la longueur totale de l'exécution d'une suite complète de tests par le nombre de transitions de la trace (ou des traces) de $T \| I$ qui permet de donner le verdict de test. Ce nombre pouvant varier en fonction de la trace effectivement exécutée (du fait du non déterminisme des implantations, le testeur ne peut pas contrôler complètement les choix de l'implantation), nous considérerons ici que l'effort de test est mesuré par la *longueur maximale des traces* de $T \| I$, ce qui correspond au pire cas d'exécution du test.

Une telle définition peut conduire, bien évidemment, à des tests de longueur infinie. Considérons les exemples représentés sur la figure 43, qui représentent des testeurs canoniques asynchrones tels que nous les avons définis au chapitre 2. La spécification S_1 décrit un comportement fini, c'est à dire ayant un nombre fini de traces. Le testeur correspondant, T_1 est donc lui aussi fini. Dans ce cas, que les implantations considérées soient finies comme I_1 ou infinies comme I_2, l'exécution du test est toujours finie et en conséquence l'effort de test également.

Par contre, si nous considérons une spécification admettant des comportements infinis comme la spécification S_2, alors le testeur correspondant est lui aussi infini, et les tra-

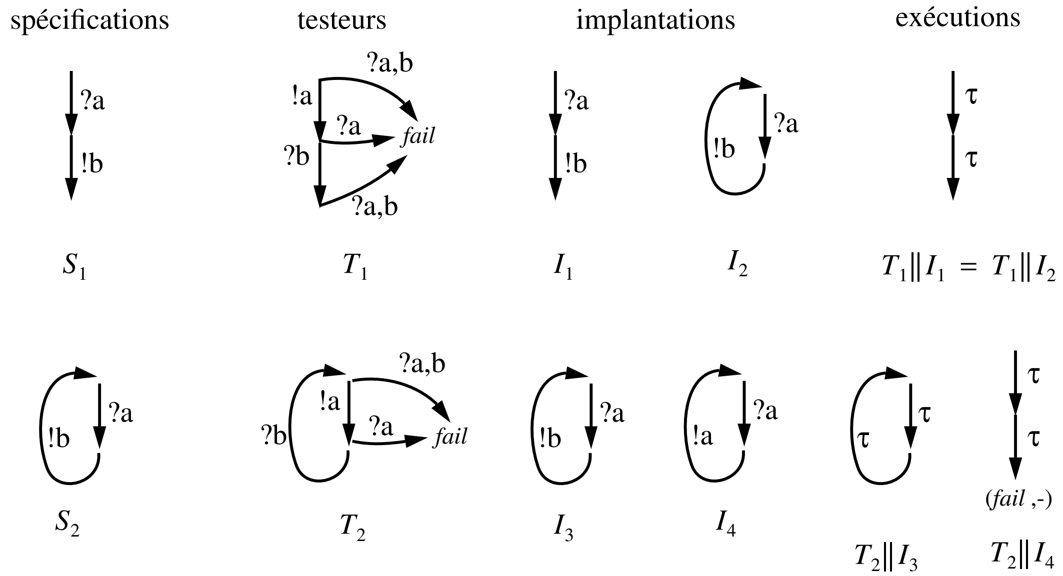


Figure 43 : testeurs et effort de test

ces de l’exécution peuvent être finies ou infinies, selon l’implantation considérée. Remarquons que le fait d’avoir une implantation finie est évidemment une condition suffisante, mais non nécessaire, pour que l’effort de test soit fini : l’implantation I_4, quoiqu’infinie, est déclarée non conforme en deux transitions. Cela est dû à la conception du testeur canonique asynchrone, qui vérifie l’absence de sorties illégales dans les implantations : dès qu’une telle sortie est détectée, on peut conclure à l’erreur , sans aller plus loin. Par contre, pour conclure à la conformité, il faut obligatoirement explorer tout le comportement de l’implantation, ce qui ne peut pas se faire en un temps fini quand l’implantation a un comportement infini.

Dans la pratique, la plupart des spécifications intéressantes ont bien sûr un comportement infini (bien qu’elles puissent toujours, comme nous l’avons expliqué au chapitre 2, être modélisées par des automates finis). Cela signifie tout simplement qu’un protocole ou un équipement de communication doit pouvoir servir plusieurs fois et possède donc un mécanisme permettant de le réinitialiser après usage. Les testeurs canoniques asynchrones, calculés selon la théorie du chapitre 2, sont donc souvent infinis, et sans hypothèse supplémentaire sur les implantations, nous devons malheureusement nous attendre, dans la plupart des cas, à passer un temps infini à les tester , ce qui requiert une patience largement supérieure à celle du testeur moyen travaillant au CNET .

Il est donc nécessaire d’étudier comment modifier la théorie que nous avons développée jusqu’ici afin de diminuer l’effort de test : c’est ce que nous appelons la *sélection des tests* . D’ordinaire, ce terme évoque un choix permettant d’extraire un nombre limité de tests à partir d’une longue suite de tests; c’est en ce sens en tout cas qu’il est utilisé dans les travaux modélisant explicitement une suite de tests comme un ensemble de tests individuels [T retmans 92b]. L ’approche suivie dans cette thèse consiste au contraire à modéliser sous forme d’un testeur toutes les exécutions de tous les tests d’une suite de tests (chapitre 2, section 4.1). Dans ce cas, nous appelons sélection des tests les moyens utilisés pour réduire l'effort de test, c'est à dire garantir que les traces d'exécution du testeur en parallèle avec n'importe quelle implantation ont une longueur réduite. Remarquons que ce sens donné à la sélection des tests dans le milieu de la recherche sur le test est différent de celui de la norme ISO 9646 [ISO 91], couramment utilisé dans les centres de tests, selon lequel la sélection des tests consiste à choisir , en fonction des *PICS* (terme barbare, défini dans la norme ISO 9646, désignant tout simplement les options d'un protocole mises en œuvre dans une implantation donnée), les tests qu'il est pertinent de passer parmi tous ceux d'une suite donnée.

Dans le cas qui nous préoccupe, une condition suffisante (et nécessaire, sans autre supposition sur les implantations) pour avoir un test de longueur finie est d'utiliser un testeur fini, c'est à dire un testeur pour lequel on connaît une borne à la longueur maximale des séquence d'exécution. Cette borne ne doit dépendre que du testeur lui-même, et doit bien sûr être indépendante de l'implantation testée. Ce n'est pas le cas, en général, des testeurs canoniques asynchrones. Ce sera par contre le cas des testeurs dits «exécutables» qui nous définirons au chapitre 9.

2.2 Hypothèses de test et modèles de fautes

Dans [T retmans 93] quatre techniques de sélection des tests sont présentées. La première consiste à *sélectionner des buts de tests* , lorsque les tests individuels sont exprimés en fonction de buts précis. C'est l'approche préconisée dans la norme ISO 9646 pour réduire la taille des suites de tests. Elle n'est pas applicable à notre modèle de testeur où les buts de tests ne peuvent être identifiés individuellement.

La deuxième consiste à *élargir la spécification* . En considérant une spécification plus souple, qui admet plus d'implantations conformes, on doit tester moins de propriétés, donc on a besoin d'un effort de test réduit. Cependant, cette approche ne nous parait pas très naturelle, car la spécification à tester est une donnée du problème et il ne nous parait pas opportun de s'autoriser à la modifier .

La troisième technique de sélection consiste à *affaiblir la relation d'implantation* testée. En effet, plus la relation d'implantation est faible (au sens de la relation d'ordre de la définition 2.20), plus la classe des implantations conformes est large, donc moins on doit tester de propriétés, ce qui signifie que l'effort de test sera réduit. Considérons la relation d'implantation suivante :

$$R_{11}(I, S) \text{ ssi } (\forall \sigma \in Tr(S), |\sigma| \leq 1)\ (\sigma \in Tr(I) \Rightarrow O(\sigma, I) \subseteq O(\sigma, S))$$

Cette relation signifie qu'au tout début, en l'occurrence ici à l'instant initial ou après un seul échange de message, le comportement de l'implantation doit être autorisé par la spécification. C'est une relation d'implantation très faible, en ce sens qu'elle contraint très peu le comportement des implantations, qui doivent rester sages seulement un tout petit moment. En conséquence de quoi, elle est évidemment facile à tester , puisque le testeur fini représenté sur la figure 44 est suffisant pour la caractériser .

Cependant, deux raisons nous conduisent à rejeter cette façon de procéder à une sélection des tests :

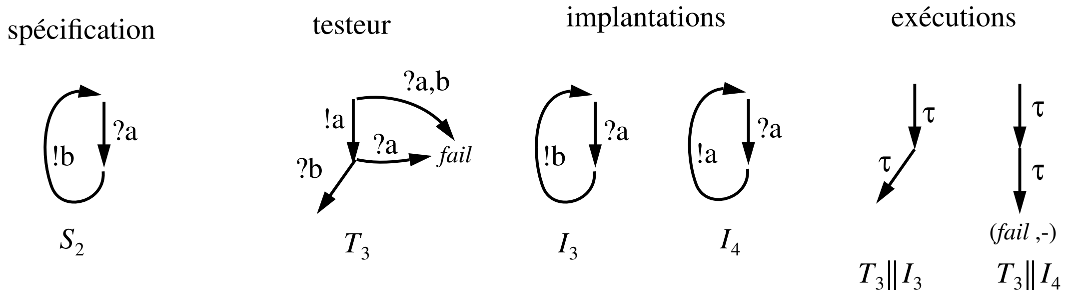


Figure 44 : affaiblir la relation d'implantation

1. la relation d'implantation doit être choisie au moment où la spécification est conçue, car comme nous l'avons indiqué au chapitre 2, section 3.1, une spécification seule ne signifie rien si elle n'est pas accompagnée d'une caractérisation précise des implantations qui lui correspondent. Il n'est donc pas souhaitable de la modifier au moment du test pour des raisons purement liées aux contraintes du test.
2. l'expérience que nous avons pu acquérir à travers l'étude de divers exemples au CNET nous conduit à penser que des relations d'implantation trop faibles, comme celle définie ci-dessus, ne sont d'aucun intérêt en pratique. Une relation d'implantation finie, portant sur un nombre fini de traces de l'implantation, ne contraint le comportement de celle-ci que pendant un temps fini et lui laisse entière liberté par la suite. Or les équipements testés doivent continuer à fonctionner bien après que le test ait été terminé. L'étude menée dans les chapitres 2 à 4 nous conduit au contraire à penser que le réduction modulo l'environnement de test (ou la quasi-équivalence modulo l'environnement de test) est *la* relation d'implantation pertinente, une fois pour toutes.

Ceci nous conduit donc à la quatrième technique de sélection des tests recensée dans [Tretmans 93], à savoir *renforcer l'hypothèse de test*. L'hypothèse de test est un concept introduit initialement dans le cadre de théories du test de programmes séquentiels [Bernot 91] [Gaudel 92], maintenant adopté de façon plus générale comme un concept de base dans la formalisation du test [Tretmans 93]. Faire une hypothèse de test consiste à supposer que les implantations testées, qui sont en réalité des objets physiques, peuvent être correctement modélisées par un ensemble d'objets mathématiques donné.

Dans le chapitre 2, nous avons déjà fait, sans le dire explicitement, une hypothèse de test, lorsque nous avons décidé que les implantations seraient modélisées par des automates à entrées et sorties. Cette hypothèse est faible, en ce sens qu'elle définit un vaste ensemble d'implantations. Nous dirons qu'une hypothèse A est *plus forte* qu'une hypothèse B si l'ensemble des modèles des implantations selon l'hypothèse A est inclus dans celui qui découle de l'hypothèse B.

Renforcer l'hypothèse de test signifie donc faire plus de suppositions a priori sur les implantations testées. Si ces suppositions sont correctes, alors le test se fait en connaissant déjà un certain nombre de propriétés de l'implantation, donc on a besoin de moins d'effort pour vérifier la conformité. Plus la classe d'implantations potentielles est réduite, plus l'effort de test sera réduit. Cette démarche correspond selon nous à ce qui est fait par les experts chargés de concevoir manuellement les suites de tests. Dans ce chapitre et dans les quatre chapitres suivants, nous allons étudier quatre types d'hypothèses de test utilisées en pratique, que nous nommons hypothèses de régularité, d'indépendance, d'uniformité et d'équité bornée. Nous montrerons alors de façon rigoureuse comment utiliser ces hypothèses pour concevoir des testeurs réduits.

Pour donner ici un exemple très simple, supposons que l'on fasse l'hypothèse de test suivante :

> «on suppose que les implantations sont telles que si elles ont un comportement correct pour toutes les traces de longueur inférieure à un alors leur comportement sera toujours correct»

Bien évidemment une telle supposition est peu vraisemblable, mais il ne s'agit ici que d'un cas d'école. Moyennant cette hypothèse le testeur fini représenté sur la figure 44 suffit à tester la relation d'implantation infinie R_1. Cet exemple montre qu'il existe une sorte de dualité entre l'affaiblissement de la relation d'implantation testée et le renforcement de l'hypothèse de test. Ceci est illustré sur la figure 45 ci-dessous.

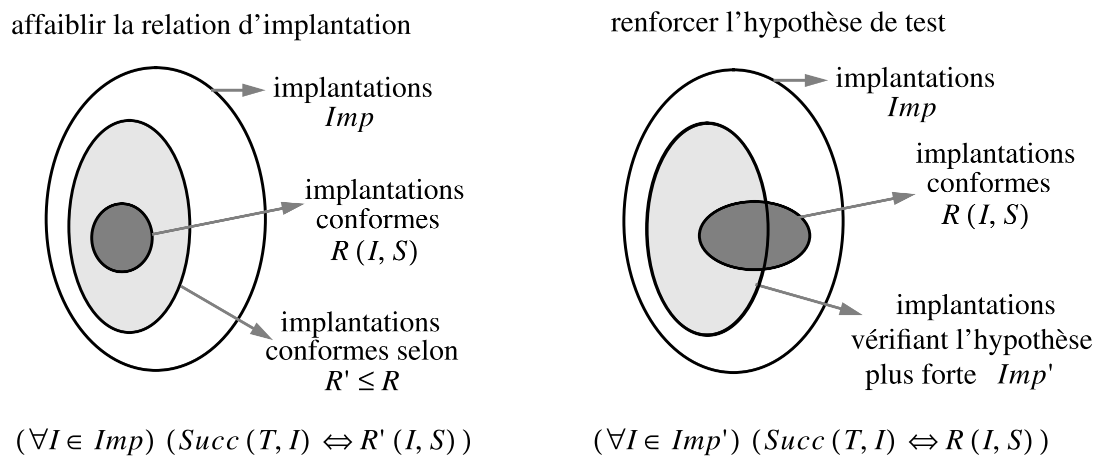


Figure 45 : deux techniques de sélection des tests

Notons toutefois une différence, visible sur la figure ci-dessus : si par définition une relation d'implantation plus faible ne peut qu'élargir l'ensemble des implantations considérées comme conformes, il n'est a priori pas garanti que toutes les implantations conformes satisfont une hypothèse de test donnée. Lorsque c'est le cas, la suite de tests obtenue au moyen de l'hypothèse de tests est susceptible de fournir des résultats faux (par exemple déclarer non conforme une implantation réellement conforme). Nous donnerons au chapitre 6 un exemple d'un tel phénomène. C'est afin de se prémunir contre ce genre de propriétés indésirables que certains auteurs se limitent à l'étude d'hypothèses de test vérifiées par toutes les implantations conformes [Tretmans 93]. Cette restriction nous semble exclure des hypothèses de test fort utiles en pratique, comme le montrera le chapitre suivant.

Le concept d'hypothèse de test est très proche de celui de *modèle de fautes* également utilisé dans le domaine du test, non seulement pour la sélection des tests [Petrenko 92] mais aussi pour définir des mesures de couverture des suites de tests (voir la section suivante). Définir un modèle de fautes consiste à définir explicitement quelles sont les fautes que les implantations peuvent comporter, généralement exprimées comme des déviations à partir d'une implantation correcte. Par exemple, pour des implantations modélisées par des automates à entrées et sorties, et en partant d'une implantation correcte égale à la spécification, on peut définir les types de fautes classiques suivants :

1. *erreur de sortie* : on remplace l'action correcte d'une transition de sortie par une action incorrecte.
2. *erreur de transfert* : on remplace l'état d'arrivée d'une transition par un état incorrect.

Un modèle de fautes consiste à établir la liste des types de fautes possibles dans l'implantation (ce qui inclut bien sûr le nombre d'occurrences permises de chaque type de faute). C'est donc une définition en extension de l'ensemble qui modélise les implantations, tandis que les hypothèses de test constituent en général une définition en compréhension d'un tel ensemble.

2.3 Couverture des tests

Qui dit sélection de tests suppose une réduction de l'effort de test, donc que le test que l'on effectue est en quelque sorte incomplet en comparaison de ce qui pourrait être nécessaire. Par exemple, en nous référant au paragraphe précédent, on caractérise la conformité sous réserve que l'implantation testée satisfasse l'hypothèse de test choisie, mais le test n'est pas complet vis à vis d'hypothèses plus faibles. Cette perspective d'incomplétude, ainsi que la notion sous-jacente de *couverture* obtenue au moyen d'une sélection de tests donnée, semble hanter les praticiens des laboratoires de tests, qui réclament désespérément à la théorie de leur en fournir une *mesure*, comme si chiffrer une telle incertitude était un moyen efficace de la maîtriser, en la ramenant à cette chose familière et rassurante qu'est un nombre entre zéro et un.

La couverture des tests est donc un sujet fortement lié à la sélection des tests. Bien que nous ne cherchions pas dans cette thèse à approfondir ce sujet, il nous parait opportun, dans cette section sur la sélection des tests, de donner un bref aperçu des théories reconnues liant ces deux concepts.

Une analyse des besoins des utilisateurs, c'est à dire des centres de test, montre que deux critères doivent être pris en compte lorsque l'on définit une mesure de la couverture d'une suite de tests :

1. prendre en compte la distribution réelle des fautes dans les implantations. Il est irréaliste par exemple de considérer que les implantations réelles sont uniformément distribuées sur le domaine défini par un modèle de fautes, ou une hypothèse de test donnée. Il faut prendre en compte le fait que la distribution réelle des fautes a peu de chances d'être uniforme et que certaines implantations peuvent exhiber des fautes non prévues.

2. prendre en compte la notion de gravité, ou de coût, des différentes fautes. Dans la pratique certaines fautes ont peu d'importance, tandis que d'autres sont au contraire considérées comme très graves par les experts. Il parait donc logique d'attendre d'une mesure de couverture qu'elle pondère les erreurs détectées par la gravité de ces erreurs.

Deux théories assez proches l'une de l'autre proposent des formalisations de la notion de couverture qui prennent en considération les critères ci-dessus. La première, développée à l'université de Montréal [Bochmann 91] considère que l'on connaît, sur l'ensemble $\Im$ de toutes les implantations possibles, deux fonctions :

1. une loi de probabilité $P_{\Im}$ qui définit la répartition des implantations sous test dans l'ensemble $\Im$ (rappelons que cet ensemble contient par hypothèse *toutes* les implantations imaginables, et n'est pas restreint pour le moment à un modèle de fautes ou à une hypothèse de test donnés).
2. une fonction décrivant le coût de non-détection d'une implantation erronée. En effet, c'est le fait de laisser passer des erreurs lors de la phase de test qui peut provoquer des coûts plus tard, lors de l'utilisation opérationnelle du système. Cette fonction a évidemment une valeur non nulle uniquement sur le sous ensemble $\Im_F \subseteq \Im$ des implantations non conformes. Elle est notée $C_{uf}(I)$.

De plus, les implantations pouvant être non déterministes, le résultat de l'application d'un test à une implantation est donné sous la forme d'une probabilité de succès ou d'échec $P_{FAIL}(T, I)$ ou $P_{PASS}(T, I)$. On définit alors deux notions distinctes :

1. la *couverture effective* d'une suite de tests T qui mesure en quelque sorte la probabilité de détecter les erreurs :

$$EC_T = \frac{\sum_{I \in \Im_F} P_{FAIL}(T, I) \times P_{\Im}}{\sum_{I \in \Im_F} P_{\Im}}$$

2. le *coût de non détection d'erreurs* résultant de l'application de la suite de tests T :

$$C_{uf} = \sum_{I \in \Im_F} P_{PASS}(T, I) \times C_{uf}(I) \times P_{\Im}$$

Ces deux notions étant définies, la sélection des tests est présentée comme un problème d'optimisation, visant à minimiser la somme $C_{uf} + C_T$, où C_T représente le coût global de la suite de tests T. Il faut noter que dans cette approche la notion de couverture n'est pas directement reliée à la sélection des tests.

Dans la pratique, l'ensemble de toutes les implantations possibles $\Im$ n'est pas directement manipulable. On a donc recours à un modèle de fautes afin de décrire une approximation de cet ensemble. Ce modèle de fautes définit en extension l'ensemble $\Im_F$ à partir de l'ensemble des implantations conformes. Sous réserve de connaître la loi de probabilité $P_{\Im}$ et la fonction de coût $C_{uf}(I)$, il est alors possible de calculer effectivement la couverture et le coût de non détection d'erreurs d'une suite de tests donnée. La limite de cette approche tient alors aux difficultés de mise en œuvre pratique : il faut être capable d'énumérer toutes les implantations incorrectes et d'appliquer la suite de tests à chacune d'entre elles, ce qui sur des exemples réalistes de modèles de fautes peut s'avérer coûteux en ressources de calcul. En outre une telle approximation présente l'inconvénient de ne rien prévoir dans le cas où les implantations ne satisfont pas le modèle de fautes.

Dans [Phalippou 93], qui reprend certains des résultats de cette thèse, nous montrons comment définir des testeurs complets vis à vis de certaines hypothèses de test. C'est ce que nous ferons dans la suite de ce chapitre et au cours des quatre chapitres suivants. Cela signifie que les testeurs que nous définissons garantissent la détection de toutes les implantations non conformes pourvu qu'elles satisfassent les hypothèses de test choisies. Nous avons donc proposé de définir la couverture de la suite de test (ici, du testeur) comme la probabilité que les hypothèses de test choisies soient vérifiées par les implantations testées. Le coût de non détection d'erreurs peut s'obtenir , comme dans l'approche précédente, en sommant les coûts des erreurs non détectées, c'est à dire celles qui correspondent aux implantations qui ne vérifient pas l'hypothèse de test.

Contrairement à l'approche précédente, où le modèle de fautes définit l'espace approché de toutes les implantations possibles, ici l'hypothèse de test définit les implantations qui sont correctement évaluées par la suite de tests. L'espace des implantations est toujours l'ensemble théorique maximal $\Im$. La question qui se pose est donc comment déterminer la probabilité que les implantations réelles satisfont les hypothèses de test. Selon nous, on peut envisager d'obtenir cette donnée par une analyse statistique des fautes réellement détectées dans les laboratoires de test. Afin de pouvoir faire des prédictions sur des nouveaux systèmes, pour lesquels on ne dispose pas encore de statistiques, il serait intéressant d'étudier les corrélations existant entre les fautes détectées et les types de spécifications, ou entre les fautes détectées et le processus de développement ayant conduit aux implantations.

Enfin, signalons qu'une seconde théorie de la sélection des tests, voisine de celle-ci, également basée sur les notions de coût et de couverture a été développée à l'université de Twente [Brinksma 91] [Tretmans 92b] [Brinksma 93].

3 Hypothèse de régularité

Nous étudions maintenant un premier exemple d'hypothèse de test, dite hypothèse de régularité. Nous commençons par celle-ci car cette hypothèse est à la base de la plupart des méthodes classiques de génération de tests à partir de modèles d'automates finis, que nous passerons en revue à la section suivante. De plus, l'utilisation de cette hypothèse à des fins de génération de tests est fondée sur de vieux résultats de la théorie des automates [Moore 56] [Hennie 64] [Hsieh 71]; nous respectons donc la chronologie en l'étudiant en premier lieu.

Les résultats théoriques tout comme les méthodes de test issus de l'hypothèse de régularité sont donc classiques dans le domaine du test, bien qu'ils ne soient pas appelés ainsi d'habitude. Nous nous contentons dans cette thèse de reformuler ces résultats dans le cadre d'utilisation d'hypothèses de test (cette section), de faire une étude de l'état de l'art sur ces méthodes (section 4), puis de montrer comment adapter ces résultats au modèle des automates à entrées et sorties (section 5).

Comme nous l'expliquons à la fin de cette section, les hypothèses de régularité peuvent prendre des formes très variées, ce qui rend difficile la formulation d'une définition unique. Nous nous contentons donc ici de donner la définition d'un cas particulier d'hypothèse de régularité, qui correspond au cas le plus fréquemment utilisé en pratique pour le test basé sur les automates finis.

Définition 5.1 : soit N un entier quelconque. Nous faisons une *hypothèse de régularité* $R(N)$ en supposant que les implantations sont modélisées par des automates à entrées et sorties ayant au plus N états.

Moyennant une telle hypothèse, il est possible de ramener la vérification d'une relation d'implantation portant sur un comportement infini (ce qui est le cas avec les relations R_1 ou R_5 définies au chapitre 2, section 3.2, quand la spécification a un comportement infini) à celle d'une relation d'implantation finie. C'est sur ce principe que reposent les méthodes classiques de génération de tests pour les automates. Plus formellement :

Définition 5.2 : soit $\Sigma \subseteq (\{!,?\} \times L)^*$ un ensemble *fini* de séquences d'événements. Posons : $R_\Sigma(I,S)$ ssi $(\forall\sigma \in \Sigma)\ (\sigma \in Tr(I) \cap Tr(S) \Rightarrow (O(\sigma,S) = O(\sigma,I)))$

Nous disons que de telles relations d'implantation sont des *relations d'implantation finies* car elles ne relient qu'un sous-ensemble fini des traces de l'implantation et de la spécification. Notons Imp l'ensemble des IOSM ayant au plus N états. Les méthodes classiques de test d'automates, adaptées au modèle des IOSM, nous indiquent comment calculer en fonction de la spécification S et de N un ensemble Σ tel que :

$$(\forall I \in Imp)\ (R_\Sigma(I,S) \Leftrightarrow R_5(I,S))$$

La dénomination «hypothèse de régularité» que nous employons ici est empruntée à des travaux formalisant le test de programmes séquentiels spécifiés par des types abstraits de données [Bernot 91] [Gaudel 92]. Dans cette théorie, les spécifications sont exprimées sous la forme d'un couple $SP = \langle\Sigma, Ax\rangle$ où Σ est une signature (noms de sortes et d'opérations avec leur domaine et codomaine) et Ax un ensemble d'axiomes définissant la sémantique des opérations (on pourra se reporter à un ouvrage de référence sur les types abstraits algébriques, comme [Ehrig 85] ou [Gaudel 91]).

Un programme P est une implantation conforme de la spécification SP si P implante les opérations définies dans la signature Σ et elles seules, et si tous les axiomes définissant la sémantique des opérations sont vérifiés par P. Un ensemble exhaustif de tests est alors constitué de toutes les instances closes bien typées des axiomes, puisque selon la sémantique choisie pour les spécifications algébriques (sémantique d'algèbre initiale) si on les vérifie toutes on a montré la conformité de l'implantation.

Cet ensemble présente l'inconvénient d'être très gros, souvent même infini. Pour le réduire, on fera donc une hypothèse dite de *régularité*. Un exemple d'une telle hypothèse est : si P satisfait un axiome Φ pour toutes les instanciations closes des variables de profondeur inférieure à un entier donné k, alors il satisfait toutes les instances closes de Φ. Soit en hiéroglyphes mathématiques décoratifs :

$$(\forall t \in T_\Sigma)\ (|t| \leq k \Rightarrow Succ\,(P, \Phi\,(t)))\ \Rightarrow\ (\forall t \in T_\Sigma)\ (Succ\,(P, \Phi\,(t)))$$

Pour les lecteurs encore sceptiques quant à la similitude entre l'hypothèse ci-dessus et celle formulée par la définition 5.1, qui justifie de notre point de vue de les appeler de la même manière, nous proposerons l'explication suivante : comme son nom l'indique, cette hypothèse exprime que le comportement de l'implantation a une certaine régularité. Si son comportement est correct lorsqu'on le teste au moyen de tests d'une longueur suffisante, mais cependant finie, alors le comportement face à des tests de n'importe quelle longueur sera correct. C'est ce qui est exprimé par les deux équations ci-dessus, pour chacune des deux théories. Remarquons enfin que dans le cas des automates on retrouve le concept de langage régulier accepté par les automates finis.

Bien entendu, la définition informelle ci-dessus du concept de régularité est plus large que la définition 5.1. D'autres types d'hypothèses de régularité que $R\,(N)$ peuvent être envisagés : on en trouvera des exemples à la section 4.3, ainsi que dans le chapitre 10.

4 Méthodes classiques de test d'automates finis

Cette section est une étude bibliographique des principales théories du test basées sur les automates d'états finis. Comme en témoigne l'abondance des méthodes, de leurs variantes, améliorations et optimisations en fonction des propriétés de l'automate à tester (déterministe ou non, complet ou incomplet, avec ou sans reset), illustrée par une riche bibliographie, nous avons ici affaire à l'un des sujets de prédilection des chercheurs travaillant sur ce domaine. Nous nous limitons ici à l'étude des techniques définies pour des modèles asynchrones (avec communication par entrées et sorties), mais il faut signaler qu'il existe aussi des variantes de ces techniques adaptées aux systèmes synchrones (étudiés au chapitre 3) [Cavalli 92] [Cavalli 93b].

4.1 Automates minimaux

Au chapitre 2 nous avons étudié les automates du point de vue de leurs traces, car les relations d'implantation que nous considérons relient les traces de l'implantation à celles de la spécification. Dans ce chapitre, les méthodes de test que nous allons passer en revue sont toutes plus ou moins fondées sur un principe d'identification des états d'un automate au moyen d'une certaine séquence d'événements caractéristique. La minimalité des automates est une question clé pour définir cette séquence.

Dans cette section bibliographique, nous abandonnons temporairement notre modèle d'automates à entrées et sorties au profit du modèle de machines de Mealy [Hopcroft 79] pour lequel sont définies les méthodes classiques. Nous adoptons les notations de [Luo 93] car cette étude, récente, présente les résultats les plus généraux.

Définition 5.3 : une *machine à états finis partiellement spécifiée et non déterministe* (PNFSM) est un quintuplet $\langle St, Li, Lo, h, S_0\rangle$ où :

1. St est un ensemble fini d'états $St = \{S_0, S_1, \ldots, S_{n-1}\}$.
2. Li est un ensemble fini d'entrées.
3. Lo est un ensemble fini de sorties.
4. h est une fonction de comportement : $h : D \Rightarrow \wp(St \times Lo) - \{\emptyset\}$ où $D \subseteq St \times Li$ (si $D = St \times Li$ l'automate est dit complètement spécifié). Rappelons que $\wp(A)$ désigne l'ensemble des parties de l'ensemble A et que la différence d'ensembles est définie par $A - B = \{x \mid \mathrm{x} \in A \wedge x \notin B\}$.
5. $S_0 \in St$ est l'état initial.

Nous utiliserons les notations suivantes :

L	$Li \times Lo$, ensemble de paires entrée/sortie. Un élément de L est noté u.
ε	La séquence vide.
L^*	Ensemble des séquences d'éléments de L. Un élément de L^* est noté x.
$P - a/b \rightarrow Q$	Est la transition de l'état P à l'état Q étiquetée par a/b. C'est une autre notation pour $(Q, b) \in h(P, a)$.
$P = \varepsilon \Rightarrow Q$	$P = Q$
$P = a/b \Rightarrow Q$	$P - a/b \rightarrow Q$
$P = x \Rightarrow Q$	$(\exists P_0 \ldots P_{k-1} \in St)\ (P = P_0 = u_1 \Rightarrow P_1 \ldots P_{k-1} = u_k \Rightarrow P_k = Q)$ où $x = u_1 \ldots u_k$
$P = x \Rightarrow$	$(\exists Q \in St)\ (P = x \Rightarrow Q)$
$Tr(P)$	$\{x \mid P = x \Rightarrow\ \}$
x^{in}	Pour $x \in L^*$, x^{in} est la séquence d'entrées obtenue en enlevant les sorties.
V^{in}	Pour $V \subseteq L^*$, $V^{in} = \{x^{in} \mid x \in V\}$
$Tr^{in}(P)$	$\{x^{in} \mid P = x \Rightarrow\ \}$ ($Tr^{in}(P) = Li^*$ pour tout automate complètement spécifié).

Une PNFSM est *observable* (en abrégé OPNFSM) si et seulement si pour chaque état $S \in St$ et pour chaque couple d'entrée-sortie a/b il y a au plus une transition étiquetée par a/b partant de S. Nous donnons maintenant la définition des automates minimaux.

Définition 5.4 : deux états S_i et S_j sont *distingables* , ce que l'on note $S_i \not\simeq S_j$ ssi :

$$(\exists x \in Tr(S_i) \oplus Tr(Sj))\ (x^{in} \in Tr^{in}(S_i) \cap Tr^{in}(S_j))$$

où $Tr(S_i) \oplus Tr(Sj) = (Tr(S_i) \cup Tr(S_j)) - (Tr(S_i) \cap Tr(S_j))$. Si deux états ne sont pas distingables nous disons qu'ils sont *indistingables*.

Les états distingables peuvent être différentiés à l'aide d'ensembles de caractérisation :

Définition 5.5 : on dit qu'un ensemble $W \subseteq Li^*$ est un *ensemble de caractérisation* pour une OPNFSM $\langle St, Li, Lo, h, S_0 \rangle$ si :

$$(\forall S_i, S_j \in St)\ (S_i \not\equiv S_j \Rightarrow (\exists x \in Tr(S_i) \oplus Tr(S_j))\ (x^{in} \in Tr^{in}(S_i) \cap Tr^{in}(S_j) \cap W))$$

Un algorithme pour construire des ensembles de caractérisations peut être trouvé dans [Luo 93].

Définition 5.6 : on dit qu'une PNFSM $\langle St, Li, Lo, h, S_0 \rangle$ est *minimale* si et seulement si $(\forall S_i, S_j \in St)\ (i \neq j \Rightarrow S_i \not\equiv S_j)$. On dit qu'elle est *réduite* si et seulement si $(\forall S_i, S_j \in St)\ (i \neq j \Rightarrow Tr(S_i) \neq Tr(S_j))$.

Une machine est minimale si tout couple d'états est distingable. Cette propriété va permettre de concevoir des suites de tests, par construction de séquences qui, sous une hypothèse de régularité adéquate, vont distinguer tous les états de l'implantation sous test. Un résultat classique de la théorie des automates dit que l'on peut minimiser tout automate complet, c'est à dire calculer un automate minimal ayant les mêmes traces que l'automate considéré (voir le chapitre 3 de [Gill 62]). Dans le cas où la spécification est modélisée par un automate complet, la relation de conformité choisie est l'équivalence de traces (notée ici $I \equiv S$) : le test va donc consister à montrer que l'implantation sous test est isomorphe à la spécification minimisée, en caractérisant ses états au moyen des séquences distingantes. C'est sur ce principe que sont fondées les méthodes présentées dans la section 4.2.

Par contre il n'est pas possible dans le cas général de minimiser un automate incomplet en préservant ses traces. Nous pouvons seulement supposer que la spécification est toujours modélisée par un automate réduit, obtenu par déterminisation si elle ne l'était pas au départ (construction semblable à celle de l'automate de traces, définition 2.8). Dans ce cas, comme l'implantation est toujours complète par hypothèse, la conformité testée est la quasi-équivalence :

Définition 5.7 : l'implantation I est *quasi-équivalente* à la spécification S, ce que l'on note $S \leq_{quasi} I$ ssi $Tr(S) \subseteq Tr(I)$ et $(\forall x \in Tr(I))\ (x^{in} \in Tr^{in}(S) \Rightarrow x \in Tr(S))$.

Dans un tel cas, une implantation peut être conforme à la spécification réduite et avoir moins d'états que celle-ci. Les méthodes de test adaptées à ce cas ne peuvent donc pas être basées sur l'identification des états de l'implantation à des états correspondants de la spécification. Les techniques de preuve de certaines méthodes évoquées dans la section 4.4 seront donc différentes de celles mentionnées ci-dessus.

Pour bien comprendre l'influence de la non complétude sur les possibilités de minimisation, étudions quelques exemples. Sur la figure 46 nous représentons un automate complet et l'automate minimal correspondant (cet exemple est tiré de [Gill 62]). On peut constater que l'opération de minimisation préserve les traces. Si cet automate est la spécification, le test consistera à établir un isomorphisme entre l'implantation testée (supposée modélisée par un automate minimal) et la minimisation de la spécification.

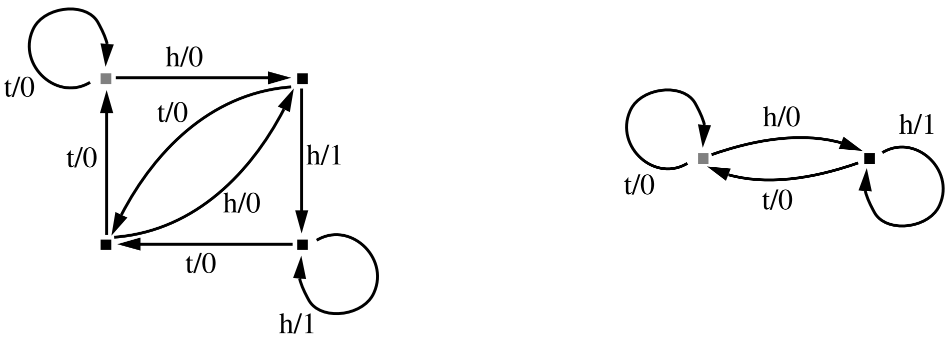


Figure 46 : automate complet et sa minimisation

Sur la figure 47 au contraire nous représentons un automate incomplet. Quoique réduit (les deux états ont des traces différentes, commençant par $a/0$ pour le premier et par $b/0$ pour le deuxième), cet automate n'est pas minimal puisque les deux états ne sont

Figure 47 : automate incomplet réduit et une implantation conforme

pas distingables, car il n'existe aucune séquence d'entrées acceptée simultanément par les deux états. L'implantation, qui a un seul état, est quasi-équivalente à la spécification, donc conforme. En effet, $Tr(S) = (a/0 \cdot b/0)^* \subseteq (a/0 + b/0)^* = Tr(I)$ et pour tout $x \in Tr(I)$ tel que $x^{in} \in Tr(S)$ alors $x \in Tr(S)$. Il n'est donc pas possible dans un tel cas de concevoir le test comme la recherche d'un isomorphisme entre la spécification et l'implantation, ni même d'un homomorphisme des implantations vers la spécification, comme nous l'avions espéré à tort dans [Phalippou 93] !

4.2 Méthodes W, DS, Wp, UIOv

Historiquement, les premières études concernaient des automates complets et *déterministes*, c'est à dire pour lesquels la fonction de comportement h est définie sur $St \times Li$ et n'a pour images que des singletons. On suppose que la spécification et les implantations potentielles satisfont ce modèle. Cette supposition sur les implantations est une première hypothèse de test, que nous notons FSM. La plupart de ces études supposent en outre que la spécification possède un reset, c'est à dire une entrée $r \in Li$ telle que pour tout état il existe une transition déclenchée par l'entrée r et aboutissant à l'état initial. Ce reset est supposé bien implanté, ce qui signifie que la même entrée r est un reset de l'implantation. Remarquons que ceci constitue une hypothèse de test supplémentaire, que nous notons ici RI.

La spécification étant complète, nous pouvons considérer qu'elle est modélisée par un automate minimal. Une conséquence immédiate des définitions 5.4 et 5.6 est qu'il existe un ensemble de caractérisation fini $W \subseteq Li^*$ qui permet de distinguer tous les couples d'états, c'est à dire :

$$(\forall S_i, S_j \in St)\ (\exists x \in Tr(S_i) \oplus Tr(Sj))\ (x^{in} \in W)$$

La *méthode W* est présentée dans l'article [Chow 78], qui reprend des résultats de [Vasilevsky 73]. Soit n le nombre d'états de la spécification sous forme minimale. Pour tout entier $m \geq n$ on construit un ensemble de séquences d'entrées appelé *ensemble de distinction* :

$$Z = W \cup Li \cdot W \cup \ldots \cup Li^{m-n} \cdot W$$

où l'opérateur $\cdot$ représente la concaténation de séquences. Le graphe de la spécification étant supposé connexe, il existe un ensemble $P \subseteq Li^*$, appelé *ensemble couvrant les transitions* de la spécification tel que pour toute transition $S_i - a/b \rightarrow S_j$ il existe $p \in P$ et $y \in Lo^*$ tels que $pa \in P$ et $S_0 = p/y \Rightarrow S_i$. La méthode W consiste à utiliser l'ensemble d'entrées $P \cdot Z$ comme suite de tests. Toujours en utilisant les notations de [Luo 93], on définit une suite de tests comme un ensemble de séquences d'entrées, et le succès d'une suite de tests sur une implantation est défini comme l'équivalence de la spécification et de l'implantation vis à vis de cette séquence d'entrées :

Définition 5.8 : soient S et I deux PNFSM.

1. l'*inclusion de traces* entre deux états P et Q vis à vis de l'ensemble d'entrées $\Pi \subseteq Li^*$ est la relation : $P \leq_\Pi Q$ ssi $(V \cap Tr(P)) \subseteq Tr(Q)$ où :

   $$V = \{x \mid x \in L^* \wedge x^{in} \in \Pi \cap Tr^{in}(P) \cap Tr^{in}(Q)\}$$

   On note $S \leq_\Pi I$ ssi $S_0 \leq_\Pi I_0$.

2. l'*équivalence* entre deux états P et Q vis à vis de l'ensemble d'entrées $\Pi \subseteq Li^*$ est la relation : $P =_\Pi Q$ ssi $P \leq_\Pi Q \wedge Q \leq_\Pi P$. On note $S =_\Pi I$ ssi $S_0 =_\Pi I_0$.

Remarquons que cette notion d'équivalence vis à vis d'un ensemble fini d'entrées Π joue le même rôle que la relation d'implantation finie R_Π dans le modèle des IOSM. Le résultat établissant le principe de la méthode W peut donc s'exprimer formellement :

Propriété 5.1 : soit S une FSM possédant un reset, et I un automate satisfaisant les hypothèses FSM, RI et $R(m)$. Alors : $S \equiv I \Leftrightarrow S =_{P \cdot Z} I$

On appelle *séquence de distinction* DS un ensemble de caractérisation réduit à un élément. La *méthode DS* (du moins l'une de ses versions, telle qu'elle est présentée dans [Fujiwara 91a] par exemple) est donc un cas particulier de la méthode W lorsque la structure de la spécification entraîne l'existence d'une telle séquence, ce qui n'est bien sûr pas toujours le cas !

La méthode W présente l'inconvénient de définir de très grosses suites de tests. On a donc cherché comment réduire la longueur de ces suites. La méthode Wp, présentée dans [Fujiwara 91a] est une optimisation de la méthode W, qui élimine certaines redondances de la suite de tests issue de cette méthode. En effet, l'ensemble Z, qui permet de caractériser les états de l'implantation, est appliqué après la vérification de chaque transition, c'est à dire après chaque élément de P. La méthode Wp consiste à utiliser cet ensemble Z pour caractériser les états de l'implantation, c'est à dire après chaque élément d'un ensemble $Q \subseteq Li^*$ *couvrant les états* de la spécification (pour tout $S_i \in St$ il existe $q \in Q$ et $y \in Lo^*$ tels que $S_0 = q/y \Rightarrow S_i$: un tel ensemble existe pour toute PNFSM connexe). Le reste du test consiste à vérifier les transitions, mais pour cela au lieu d'utiliser l'ensemble Z on peut se contenter d'utiliser les ensembles réduits Z_i définis ci-dessous.

Un ensemble $W_i \subseteq Li^*$ est un *ensemble d'identification pour l'état* s'il permet de distinguer S_i de tous les autres états de la spécification :

$$(\forall S_j \in St)\ (\exists x \in Tr(S_i) \oplus Tr(Sj))\ (x^{in} \in W_i)$$

De tels ensembles existent toujours par définition d'un automate minimal, et ils sont en général plus petits que l'ensemble W (au pire, aussi longs). On pose alors $Z_i = W_i \cup Li \cdot W_i \cup \ldots \cup Li^{m-n} \cdot W_i$ et $\Pi = Q \cdot Z \cup \bigcup_{p \in P-Q} \{p\} \cdot Z_{s(p)}$, où $s(p)$ désigne l'état de la spécification atteint après la séquence p. Le résultat établissant la méthode Wp s'exprime donc par la propriété suivante :

Propriété 5.2 : soit S une FSM possédant un reset, et I un automate satisfaisant les hypothèses FSM, RI et $R(m)$. Alors : $S \equiv I \Leftrightarrow S =_{\Pi} I$

On appelle *séquence d'uniques entrées-sorties pour l'état* S_i (UIO) un ensemble d'identification pour cet état réduit à un seul élément. Lorsque la structure de la spécification est telle qu'une telle séquence existe pour chaque état, alors le cas particulier de la méthode Wp qui est obtenu est appelé *méthode UIOv*. Cette méthode est décrite pour la première fois dans [Vuong 89], mais bien sûr elle n'apparait comme un cas particulier de la méthode Wp que dans l'article, plus récent, décrivant celle-ci [Fujiwara 91a].

4.3 Méthodes UIO et TT

L'utilisation des séquences d'uniques entrées-sorties pour définir des suites de tests a été proposée par [Sabnani 88], donc avant la publication de la méthode UIOv évoquée ci-dessus. La *méthode UIO* consiste à concaténer un ensemble couvrant les transitions P et des ensembles d'uniques entrées-sorties ($\{w_i\}$ pour l'état S_i) afin d'obtenir une suite de tests. Formellement une telle suite s'exprime par :

$$\Pi = \bigcup_{p \in P} \{p\} \cdot \{w_{s(p)}\}$$

Malheureusement, il n'est pas possible de prouver une propriété semblable aux propriétés 5.1 ou 5.2 dans ce cas. L'hypothèse de régularité $R(m)$ n'est pas suffisante pour conclure à l'équivalence entre implantation et spécification à partir du succès du test ainsi défini. Un contre exemple est décrit dans [Vuong 89].

La méthode de *tour de transitions* (TT) consiste à choisir comme suite de tests, pour une spécification fortement connexe, une séquence d'entrées qui permette de passer dans toutes les transitions de la spécification (dans le cas où la spécification admet un reset, elle est nécessairement fortement connexe). Là non plus, il n'est pas possible de prouver une propriété concluant à l'équivalence entre implantation et spécification à partir du succès du test, sous une hypothèse de régularité de type $R(m)$.

Il faut cependant se garder de conclure hâtivement que les méthodes UIO et TT ne sont pas utilisables. En effet, les séquences de tests qu'elles définissent sont plus courtes, parfois significativement, que celles résultant des méthodes présentées à la section précédente. De plus, le fait que les suites de tests ne caractérisent pas la conformité sous les hypothèses FSM, RI et $R(m)$ ne signifie pas qu'elles ne détectent pas d'erreurs. Il est possible, en utilisant l'approche décrite dans la section 2.3, de mesurer le taux de couverture de ces suites de tests vis à vis du modèle de fautes défini implicitement par ces hypothèses. Une autre manière de redonner une certaine légitimité à ces méthodes est de leur tailler des hypothèses de tests sur mesure. Par exemple, si l'on note RTT l'hypothèse suivante :

> «On suppose que si l'implantation satisfait la suite de tests résultant de la méthode TT alors elle est conforme à la spécification.»

Il est évident que moyennant une telle hypothèse la méthode des tours de transition caractérise toutes les implantations conformes. Loin d'être complètement saugrenue, une telle hypothèse peut, selon moi, prétendre encore à l'appellation de régularité, pour les mêmes raisons que celles exposées à la fin de la section 3 au sujet de l'hypothèse plus conventionnelle $R(m)$.

4.4 Affaiblir les hypothèses de test

Les hypothèses FSM et RI faites dans cette section sont souvent l'objet de débats animés dans le petit monde des spécialistes du test d'automates. Le jeu consiste à trouver des méthodes de plus en plus générales, en affaiblissant autant que possible les hypothèses en question. Dès qu'une méthode est publiée permettant de se passer de l'une d'entre elles, un peu de rhétorique nous apprend qu'il est incompréhensible que pendant si longtemps on ait cru sérieusement qu'elle était réaliste.

On peut se passer de la présence d'un reset si la spécification est fortement connexe, c'est à dire si à partir de tout état il existe une séquence de transitions permettant d'atteindre n'importe quel autre état. Dans le cas le plus général, [Hennie 64] montre comment définir , à partir d'un ensemble de caractérisation, une suite de test qui montre l'équivalence entre la spécification et l'implantation. Dans le cas particulier où il existe une séquence de distinction, on retrouve la façon dont fut initialement formulée la méthode DS [Gonenc 70]. Enfin, plus récemment, la même méthode a été adaptée au cas intermédiaire où la spécification possède des séquences d'uniques entrées-sorties : c'est la méthode UIOG de [Y ao 93].

Dans chacun de ces trois cas, la suite de tests obtenue se présente sous la forme d'une seule séquence d'entrées à appliquer à l'implantation : c'est pourquoi ces méthodes sont dites «à séquence unique». Elles caractérisent la conformité moyennant les hypothèses de test FSM et $R(m)$. Cependant, force est de constater que les séquences de test obtenues ont trop souvent, dès que la spécification est un tant soit peu complexe, une longueur prohibitive qui, compte tenu de la chance somme toute raisonnable pour que l'on dispose d'un reset bien implanté, amoindrit significativement l'intérêt pratique de ces théories.

Affaiblir l'hypothèse FSM est en revanche d'un tout autre intérêt. Nous avons exposé, au chapitre 2, les raisons de notre choix en faveur d'un modèle d'automates non déterministes et incomplets. De plus, l'étude effectuée au chapitre 4 montre que la présence d'un environnement de test limitant le contrôle et l'observation du testeur sur l'implantation ne peut qu'accroître le non déterminisme et l'incomplétude apparents de l'implantation sous test. De telles idées, quoique récentes, sont maintenant bien acceptées, comme en témoigne l'abondance actuelle des publications sur le sujet [Petrenko 93].

Le cas où les spécifications et les implantations peuvent être non déterministes (hypothèse de test $NFSM$) est étudié dans [Kloosterman 92] et [T ripathy 92]. Cependant, ces études généralisent la méthode UIO à cette classe plus large de modèles, et tout comme dans la méthode originale il n'est pas possible d'établir de propriété de caractérisation sous l'hypothèse de test $R(m)$. Les automates déterministes mais incomplètement spécifiés (hypothèse $PFSM$) ont été étudiés dans [Evtushenko 89] et [Petrenko 91]. Cependant on préférera à ces travaux l'étude plus récente [Luo 93], qui définit la *méthode HSI*, qui s'applique à la classe la plus large des automates incomplets et non déterministes. Cette méthode montre comment construire des séquences de tests qui sous les hypothèses $NFSM$, RI et $R(m)$ caractérisent la quasi-équivalence entre une implantation et sa spécification. C'est donc cette méthode que, dans la section suivante, nous adapterons aux automates à entrées et sorties.

Quelques mots, pour conclure cette section, sur cette dernière approche. Pour prendre en compte les spécifications incomplètes et le non déterminisme, la relation d'implantation de quasi-équivalence (définition 5.7) est introduite. Comme nous l'avons expliqué à la section 4.1, la preuve du pouvoir de détection d'erreurs de la méthode HSI ne peut pas s'appuyer sur la construction d'un isomorphisme, comme celles des méthodes présentées à la section 4.2. Un autre type de démonstration est mis en œuvre, basé sur le pouvoir de séparation d'états d'une séquence donnée.

5 Application aux automates à entrées et sorties

Après avoir passé en revue, dans la section précédente, les principales méthodes résultant de l'état de l'art en matière d'hypothèse de régularité sur les machines de Mealy , nous étudions comment adapter ces méthodes à notre modèle d'automates à entrées et sorties.

5.1 Principes généraux

Un certain nombre de méthodes définies pour les PNFSM peuvent être appliquées telles quelles, au prix d'une légère modification de leur formulation seulement. C'est le cas de toutes les méthodes utilisant une hypothèse de régularité ad hoc, comme par exemple la méthode de tour de transitions. Dans le cas le plus général, comme indiqué à la section 3, on se donne une relation d'implantation finie R_{Σ} (dans l'exemple du tour de transitions, Σ contient une séquence unique passant par toutes les transitions de l'automate), et on fait l'hypothèse de régularité que pour les implantations considérées R_5 est satisfaite si et seulement si R_{Σ} est satisfaite.

Cependant, la validité des autres méthodes, fondées sur l'hypothèse de régularité $R(m)$, résulte de théorèmes parfois complexes utilisant les propriétés particulières des automates manipulés. Ces théorèmes exploitent la structure propre aux PNFSM, qui regroupe réception et émission de messages dans une transition. Leur transposition au monde des IOSM, ou des systèmes de transitions étiquetées, n'est donc pas immédiate, et doit être soigneusement démontrée.

S'il peut être tentant, pour une méthode de test donnée, d'en transposer la démonstration d'un modèle d'automates à l'autre, cette façon de procéder présente l'inconvénient d'exiger de refaire les démonstrations de toutes les méthodes que nous voudrions adapter . Nous préférons adopter ici une démarche plus générale qui consiste à décrire formellement une transformation entre les deux modèles d'automates IOSM et PNFSM. À l'aide de cette transformation, nous importons alors les résultats déjà démontrés pour les PNFSM dans le monde des IOSM. Nous avons utilisé une démarche similaire au chapitre 3, pour relier la théorie du test des IOSM à celle du test des systèmes de transitions étiquetées. Comme dans ce chapitre, compte tenu des résultats de la première partie de cette thèse, nous considérons dans la suite de cette section les IOSM modulo leurs automates de traces.

Dans cette section, nous nous limitons à montrer comment exploiter cette transformation pour transposer la méthode HSI présentée à la section précédente, car comme nous l'avons vu cette méthode est actuellement la plus générale qui a été publiée. Mais bien entendu ceci ne constitue qu'un exemple : d'autres méthodes pourraient être transposées au moyen de la même transformation. Plus précisément, nous allons suivre les étapes suivantes, illustrées sur la figure 48 :

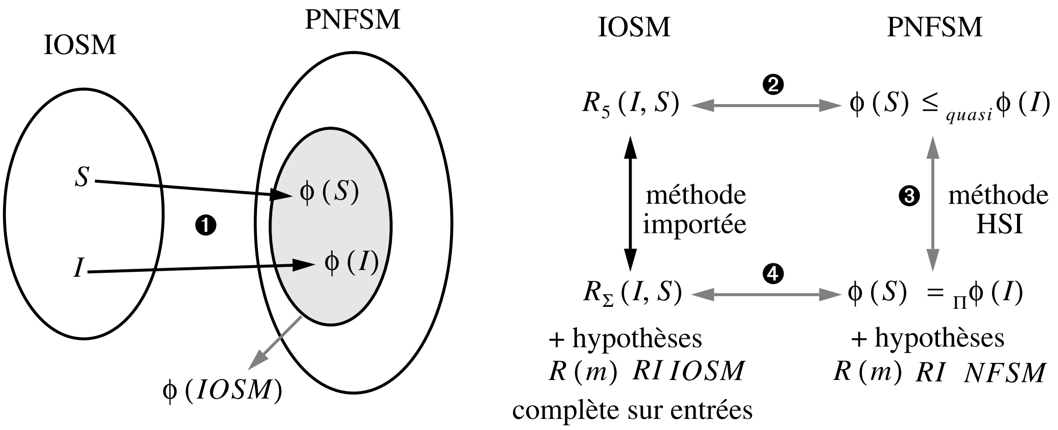


Figure 48 : comment importer une méthode

1. on définit une fonction ϕ qui transforme toute IOSM en une PNFSM. Cette fonction définit une bijection induite entre les traces d'une IOSM S et les traces de son image $\phi(S)$ qui sera fort utilisée par la suite. En outre, elle transforme les hypothèses de test RI et $R(m)$ formulées sur les IOSM en des hypothèses similaires formulées sur les PNFSM.
2. on montre que la relation d'implantation $R_5(I, S)$ sur des IOSM est équivalente à la relation de quasi-équivalence sur leurs images $\phi(S) \le_{quasi} \phi(I)$.
3. pour tester $R_5(I, S)$, ce qui est notre objectif, nous devons donc tester la quasi-équivalence des images. Pour cela, nous utilisons le résultat de la méthode HSI, qui indique comment calculer une suite de tests finie Π telle que, sous les hypothèses de test $PNFSM$, RI et $R(m)$, on ait $\phi(S) \le_{quasi} \phi(I)$ si et seulement si $\phi(S) =_\Pi \phi(I)$.
4. on montre alors comment calculer en fonction de Π un ensemble Σ tel que $\phi(S) =_\Pi \phi(I)$ soit équivalent à $R_\Sigma(I, S)$. Nous obtenons donc une méthode de test qui, sous les hypothèses de test $PNFSM$, RI et $R(m)$ (formulées sur les IOSM), permet de ramener le test de $R_5(I, S)$ au test de la relation d'implantation finie $R_\Sigma(I, S)$.

Pour pouvoir établir les démonstrations des points 2 et 4 ci-dessus nous devons nous placer dans le cas où les blocages de sortie des automates sont explicitement modélisés, comme indiqué au chapitre 2 (section 2.5). Ceci n'est pas une simple restriction ad hoc pour faciliter les démonstrations, mais reflète le fait que dans le modèle des PNFSM les blocages de sortie sont *toujours* explicitement décrits puisque toute transition est étiquetée par un couple entrée-sortie. Il est donc très difficile de relier les modèles sans faire cette hypothèse supplémentaire. Cependant ceci ne doit pas être considéré comme pénalisant, attendu que l'on peut transformer un automate pour faire apparaître explicitement les actions $!\delta$ et qu'au chapitre 9 nous verrons que pour définir des algorithmes de test cette hypothèse est de toutes façons nécessaire.

5.2 Définition de la transformation entre les modèles

Conformément à l'esprit du modèle d'automates à entrées et sorties, où chaque transition représente un événement indépendant, les entrées n'étant pas liées aux sorties, nous interprétons une entrée d'une IOSM comme une transition d'une PNFSM ayant la même entrée et une sortie vide, et une sortie d'une IOSM comme une transition d'une PNFSM déclenchée par une entrée vide et ayant cette sortie. Entrée vide et sortie vide sont représentées par deux interactions particulières dans le modèle PNFSM, notées respectivement ?e et !e.

Définition 5.9 : soit $S = \langle S_S, L_S, T, s_0 \rangle$ une IOSM. Son image par la *fonction* ϕ est une PNFSM $\phi(S) = S_M = \langle St, Li, Lo, h, S_0 \rangle$ où :

1. $St = S_S$
2. $Li = \{?\} \times (L_S \cup \{e\})$
3. $Lo = \{!\} \times (L_S \cup \{e\})$
4. $S_0 = s_0$
5. h est la fonction de comportement définie par :
   a) pour toute transition $(s, ?a, s') \in T$: $(s', !e) \in h(s, ?a)$
   b) pour toute transition $(s, !a, s') \in T$: $(s', !a) \in h(s, ?e)$

Cette transformation est illustrée sur la figure 49. Remarquons que les images des IOSM exploitent bien la richesse du modèle PNFSM, puisque sur l'exemple présenté l'automate $\phi(S)$ est effectivement incomplet et non déterministe.

Nous définissons une transformation, également notée ϕ, entre les séquences d'entrées-sorties utilisées dans les deux modèles de la façon suivante :

1. pour $a \in L_S$, $\phi(?a) = ?a/!e \in L$ (rappelons que $L = Li \times Lo$).
2. pour $a \in L_S$, $\phi(!a) = ?e/!a \in L$
3. pour $\sigma = \mu_1 \ldots \mu_n \in L_{IOSM}^*$, $\phi(\sigma) = \phi(\mu_1) \ldots \phi(\mu_n) \in L^*$ (où $L_{IOSM} = \{?, !\} \times L_S$)

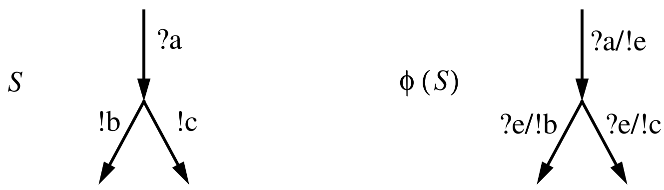


Figure 49 : transformation entre IOSM et PNFSM

Il découle immédiatement de cette définition que la fonction ϕ est injective, donc qu'elle définit une bijection entre $L_{IOSM}{}^*$ et $\phi(L_{IOSM}{}^*) \subseteq L^*$. Nous pouvons donc utiliser la fonction réciproque ϕ^{-1} définie sur $\phi(L_{IOSM}{}^*)$. De plus, cette fonction sur les séquences d'entrées-sorties est reliée à la fonction de transformation d'automates par la propriété suivante :

Propriété 5.3 : soit S une IOSM. Alors $Tr(\phi(S)) = \phi(Tr(S))$

Démonstration : nous montrons que si (s_0, σ, s) est une dérivation de S alors $s_0 = \phi(\sigma) \Rightarrow s$ est une dérivation de $\phi(S)$, par induction sur la longueur de σ. Le cas où $\sigma = \varepsilon$ étant trivial, supposons la propriété vraie pour toute séquence de longueur inférieure à n, et considérons une séquence σ de longueur $n+1$.

Si on est dans le cas où $\sigma = \sigma'!a$, alors (s_0, σ', s') est une dérivation de S et $(s', !a, s) \in T$. Donc par hypothèse d'induction $s_0 = \phi(\sigma') \Rightarrow s'$, et par définition de ϕ on a $s' = ?e/!a \Rightarrow s$, ce qui entraîne $s_0 = \phi(\sigma) \Rightarrow s$. Si on est dans le cas où $\sigma = \sigma'?a$, alors (s_0, σ', s') est une dérivation de S et $(s', ?a, s) \in T$. Donc par hypothèse d'induction $s_0 = \phi(\sigma') \Rightarrow s'$, et par définition de ϕ on a $s' = ?a/!e \Rightarrow s$, ce qui entraîne $s_0 = \phi(\sigma) \Rightarrow s$.

On montre de façon similaire que si $s_0 = \phi(\sigma) \Rightarrow s$ est une dérivation de $\phi(S)$ alors (s_0, σ, s) est une dérivation de S. ❑

Il est important d'étudier comment la fonction ϕ sur les automates transforme les sous-ensembles d'automates correspondant aux hypothèses de test. Par définition, l'hypothèse $IOSM$ définit l'ensemble de tous les automates à entrées et sorties, ensemble dont l'image est $\phi(IOSM) \subseteq PNFSM$. De plus, si une IOSM est complète pour les entrées, et si les blocages de sortie sont explicitement modélisés, alors son image par ϕ est complète, donc satisfait l'hypothèse $NFSM$. Par définition de la fonction ϕ, un reset de l'IOSM S est transformé en un reset de la PNFSM $\phi(S)$: le sous-ensemble RI des IOSM correspond donc au sous-ensemble RI de $\phi(IOSM)$. Enfin, les ensembles d'états de S et de

$\phi(S)$ étant les mêmes, le sous-ensemble $R(m)$ de $IOSM$ est transformé en le sous-ensemble $R(m)$ de $PNFSM$.

5.3 Lien entre les relations de conformité infinies

La transformation ϕ nous permet de ramener la vérification de la conformité entre deux automates à entrées et sorties à celle de la conformité entre deux PNFSM. Nous utilisons dans la démonstration la propriété suivante de la relation d'implantation R_5 :

Propriété 5.4 : si $R_5(I, S)$ alors $Tr(S) \subseteq Tr(I)$

Démonstration : supposons que $Tr(S) \not\subseteq Tr(I)$. C'est donc qu'il existe $\sigma \in Tr(S)$ telle que $\sigma \notin Tr(I)$. Soit σ' le plus petit préfixe de σ qui n'appartient pas à $Tr(I)$. Si $\sigma' = \varepsilon$ on a une contradiction puisque $\varepsilon \in Tr(I)$. Si $\sigma' = \sigma''?a$ on a également une contradiction car toute implantation est complète vis à vis des entrées. Enfin, si $\sigma' = \sigma''!a$, alors $\sigma'' \in Tr(I)$ (puisque σ' est le plus petit préfixe de σ qui n'appartient pas à $Tr(I)$). Donc $a \in O(\sigma'', S)$ (car $\sigma' \in Tr(S)$) et $a \notin O(\sigma'', I)$ (car $\sigma' \notin Tr(I)$), ce qui contredit la relation $R_5(I, S)$. ❑

Propriété 5.5 : soient S et I deux IOSM. Alors : $R_5(I, S) \Leftrightarrow \phi(S) \leq_{quasi} \phi(I)$

Démonstration : posons $S_M = \phi(S)$ et $I_M = \phi(I)$.

1. nous montrons d'abord l'implication $R_5(I, S) \Rightarrow \phi(S) \leq_{quasi} \phi(I)$. D'après la propriété 5.4, $R_5(I, S)$ implique $Tr(S) \subseteq Tr(I)$, donc $\phi(Tr(S)) \subseteq \phi(Tr(I))$, donc $Tr(S_M) \subseteq Tr(I_M)$ d'après la propriété 5.3.

   Soit $x \in Tr(I_M)$ tel que $x^{in} \in Tr^{in}(S_M)$. D'après la propriété 5.3 $\phi^{-1}(x) \in Tr(I)$. Supposons que $\phi^{-1}(x) \notin Tr(S)$. Soit σ le plus petit préfixe de $\phi^{-1}(x)$ qui n'appartient pas à $Tr(S)$.

   a) si $\sigma = \sigma'?a$, $\sigma' \in Tr(S)$ (car σ est le plus petit préfixe), donc $\phi(\sigma') \in Tr(S_M)$ (propriété 5.3). Comme $\sigma \notin Tr(S)$, $\phi(\sigma) \notin Tr(S_M)$ (encore la propriété 5.3). Comme $\phi(\sigma) = \phi(\sigma')?a!e$, on en déduit $\phi(\sigma)^{in} \notin Tr^{in}(S_M)$, car S_M appartenant à $\phi(IOSM)$ ne peut pas émettre autre chose que $!e$ si une transition d'entrée $?a$ existe. Ceci contredit $x^{in} \in Tr^{in}(S_M)$.

   b) si $\sigma = \sigma'!a$ alors $\sigma' \in Tr(S)$, $\sigma \in Tr(I)$ et $R_5(I, S)$ impliquent $\sigma \in Tr(S)$, ce qui contredit la définition de σ.

   Donc $\phi^{-1}(x) \in Tr(S)$, et donc d'après la propriété 5.3 $x \in Tr(S_M)$. Nous avons donc montré que $(\forall x \in Tr(I_M))\ (x^{in} \in Tr^{in}(S_M) \Rightarrow x \in Tr(S_M))$. Donc en fin de compte $\phi(S) \leq_{quasi} \phi(I)$.

2. nous montrons ensuite l'implication $\phi(S) \leq_{quasi} \phi(I) \Rightarrow R_5(I, S)$. Puisque $\phi(S) \leq_{quasi} \phi(I)$, on a $Tr(\phi(S)) \subseteq Tr(\phi(I))$, donc $Tr(S) \subseteq Tr(I)$. Considérons $\sigma \in Tr(S) \cap Tr(I)$. Pour montrer $R_5(I, S)$, il reste donc à prouver que $O(\sigma, S) = O(\sigma, I)$.

   a) soit $a \in O(\sigma, S)$. Cela signifie $\sigma!a \in Tr(S)$, donc d'après la propriété 5.3 $\phi(\sigma!a) \in Tr(S_M)$. Comme $\phi(S) \leq_{quasi} \phi(I)$ il en résulte $\phi(\sigma!a) \in Tr(I_M)$, donc $\sigma!a \in Tr(I)$, soit $a \in O(\sigma, I)$.

   b) réciproquement soit $a \in O(\sigma, I)$. Puisque $\sigma!a \in Tr(I)$ alors $\phi(\sigma!a) \in Tr(I_M)$. De plus, comme $\sigma \in Tr(S)$ on a $\phi(\sigma) \in Tr(S_M)$, donc $\phi(\sigma)^{in}?e \in Tr^{in}(S_M)$ puisque les éléments de $\phi(IOSM)$ admettent l'entrée $?e$ dans tout état (c'est une conséquence de la modélisation explicite des blocages de sortie et de la définition de ϕ). Par définition de la quasi-équivalence, il en résulte que $\phi(\sigma!a) \in Tr(S_M)$, donc que $\sigma!a \in Tr(S)$, soit $a \in O(\sigma, S)$.

   Nous avons donc montré $R_5(I, S)$. ❑

Il faut remarquer que la modélisation explicite des blocages de sortie est une hypothèse nécessaire pour garantir la validité de la propriété ci-dessus. Le contre-exemple présenté sur la figure 50 en témoigne : dans la modélisation à base d'IOSM, le fait que la spécification n'admette pas de sortie après avoir reçu ?a est pris en compte par la relation R_5, d'où la non conformité de l'implantation. Par contre, dans l'image obtenue par la transformation ϕ cette absence de sortie correspond à une partie non spécifiée du comportement, donc non prise en compte par la relation de quasi-équivalence.

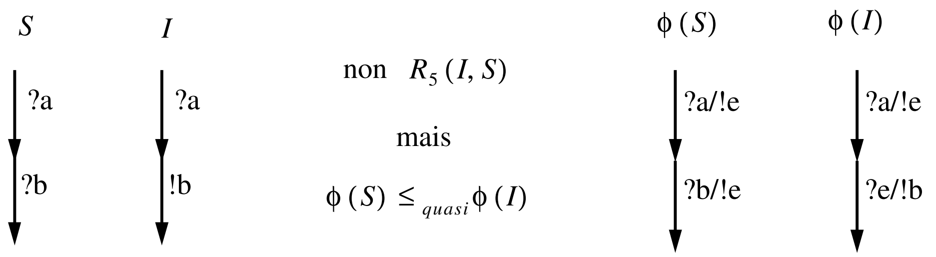


Figure 50 : nécessité de blocages de sortie explicites

5.4 Application de la méthode HSI

Nous rappelons ici brièvement les résultats exposés dans [Luo 93]. Afin de distinguer , dans la mesure du possible, les états d'un automate incomplet, on introduit les ensembles d'identification d'états harmonisés, qui sont une variante des ensembles d'identification d'états définis à la section 4.2.

Définition 5.10 : soit une OPNFSM ayant n états et W un ensemble de caractérisation pour cet automate. On dit que $\{D_0, \ldots, D_{n-1}\}$ est un n-uplet d'*ensembles d'identification d'états harmonisés* si pour tout $i \in [0, n-1]$, D_i est un ensemble minimal tel que :

1. $D_i \subseteq Tr^{in}(S_i) \cap pref(W)$ (où $pref(W)$ est l'ensemble des préfixes non vides des éléments de W)
2. $(\forall j \in [0, n-1])\ (S_i \not\equiv S_j \Rightarrow (\exists x \in Tr(S_i) \oplus Tr(S_j))\ (x^{in} \in pref(D_i) \cap pref(D_j)))$

Le degré d'incomplétude des automates est en quelque sorte mesuré par leur degré de flou.

Définition 5.11 : soit S_i un état d'une OPNFSM, on dit que $f(S_i) \subseteq St$ est un *ensemble maximal d'états distingables deux à deux pour l'état* S_i si :

1. $S_i \in f(S_i)$
2. $(\forall S_k, S_j \in f(S_i))\ (k \neq j \Rightarrow S_k \not\equiv S_j)$
3. il n'y a pas d'ensemble $B \subseteq St$ tel que
   a) $S_i \in B$
   b) $(\forall S_k, S_j \in B)\ (k \neq j \Rightarrow S_k \not\equiv S_j)$
   c) $|B| > |f(S_i)|$ ou $|B| = |f(S_i)|$ et les indices (ordonnés) des éléments de B précèdent ceux de $f(S_i)$ dans l'ordre lexicographique.

Le *degré de flou* Δ d'une OPNFSM est alors défini par $\Delta = |\{f(S_i) \mid S_i \in St\}|$.

Soit donc une OPNFSM S possédant un reset et ayant n états, Δ son degré de flou et m un entier tel que $\delta m \geq n$. Soit Q un ensemble couvrant les états de S et $\{D_0, \ldots, D_{n-1}\}$ un n-uplet d'ensembles d'identification d'états harmonisés pour S. On considère l'ensemble suivant :

$$\Pi = \bigcup_{\substack{S_0 = x \Rightarrow S_i \\ x^{in} \in Q \cdot (\{\varepsilon\} \cup Li \cup \ldots \cup Li^{\Delta m - n + 1})}} \{x^{in}\} \cdot D_i$$

Cet ensemble satisfait la propriété suivante, base de la méthode HSI :

Propriété 5.6 : soit I un automate satisfaisant les hypothèses $NFSM$, RI et $R(m)$. Alors $S \leq_{quasi} I \Leftrightarrow S =_{\Pi} I$.

Remarquons que si nous partons d'une spécification sous la forme d'un automate de traces d'IOSM, alors son image $\phi(S)$ est observable, donc on peut appliquer la méthode ci-dessus.

L'ensemble de séquences Π ci-dessus constitue la suite de tests définie par la méthode HSI, telle qu'elle est présentée dans [Luo 93]. Cependant, pour pouvoir ramener cette suite de tests dans le monde des IOSM, il nous faut tout d'abord la compléter un peu, pour des raisons techniques, dont la nécessité apparaîtra lors de la démonstration de la section suivante. Soit Π' la clôture par préfixes de l'ensemble Π. Ce nouvel ensemble vérifie :

Propriété 5.7 : soit I un automate satisfaisant les hypothèses $NFSM$, RI et $R(m)$. Alors $S \leq_{quasi} I \Leftrightarrow S =_{\Pi'} I$.

Démonstration : cette propriété est déduite de la propriété 5.6 au moyen des deux lemmes suivants :

1. soient Π et Π' deux ensembles de séquences. Si $\Pi \subseteq \Pi'$ alors $S =_{\Pi'} I \Rightarrow S =_{\Pi} I$. En effet, si l'on note V_{Π} et $V_{\Pi'}$ les ensembles intervenant dans la définition 5.8, on a immédiatement $V_{\Pi} \subseteq V_{\Pi'}$, donc si $V_{\Pi'} \cap Tr(I) \subseteq Tr(S)$ alors a fortiori on aura $V_{\Pi} \cap Tr(I) \subseteq Tr(S)$.
2. si $S \leq_{quasi} I$ alors $S =_{\Pi} I$ pour tout ensemble de séquences Π. En effet, si $S \leq_{quasi} I$ alors $Tr(S) \subseteq Tr(I)$ donc $V \cap Tr(S) \subseteq Tr(I)$ ce qui signifie $S \leq_{\Pi} I$. Réciproquement, si $x \in V \cap Tr(I)$ alors $x \in Tr(I)$ et $x^{in} \in Tr^{in}(S)$ (définition de V), donc puisque $S \leq_{quasi} I$ on en déduit que $x \in Tr(S)$. Donc $V \cap Tr(I) \subseteq Tr(S)$, soit $I \leq_{\Pi} S$. ❑

5.5 Lien entre les relations de conformité finies

La section précédente nous indique comment calculer une suite de tests Π' à appliquer à l'implantation. Il nous reste maintenant à utiliser à nouveau la transformation ϕ pour ramener l'application de cette suite de tests à une vérification de conformité dans le monde des IOSM. Posons donc :

$$\Sigma = \{\phi^{-1}(x) \mid x \in \phi(L_{IOSM}{}^{*}) \wedge x^{in}?e \in \Pi'\}$$

Propriété 5.8 : soient S et I deux IOSM. Soit Π' la suite de tests calculée en appliquant la méthode décrite à la section précédente à $\phi(S)$ et Σ l'ensemble défini ci-dessus.
Alors $R_{\Sigma}(I, S) \Leftrightarrow \phi(I) =_{\Pi'} \phi(S)$.

Démonstration : posons $S_M = \phi(S)$ et $I_M = \phi(I)$.

1. nous montrons d'abord l'implication $R_{\Sigma}(I, S) \Rightarrow \phi(I) =_{\Pi'} \phi(S)$ par sa contraposée. Si $\phi(I) \neq_{\Pi'} \phi(S)$ c'est que $\phi(I) \leq_{\Pi'} \phi(S)$ est faux ou que $\phi(S) \leq_{\Pi'} \phi(I)$ est faux. Supposons que nous sommes dans le premier cas. Alors il existe $x \in V \cap Tr(I_M)$ tel que $x \notin Tr(S_M)$ (avec $V = \{x \mid x \in L^{*} \wedge x^{in} \in \Pi' \cap Tr^{in}(I_M) \cap Tr^{in}(S_M)\}$). On

peut choisir x pour que tous ses préfixes appartiennent à $Tr(S_M)$ (c'est là la raison pour laquelle on exige que Π' soit clos par préfixe !). De la propriété 5.3 on déduit $\phi^{-1}(x) \in Tr(I)$ et $\phi^{-1}(x) \notin Tr(S)$. Puisque x est une trace appartenant à l'image par ϕ d'un automate, x ne peut avoir que deux formes possibles :

a) soit $x = x_1 ?e/!a$. On a $x_1 \in Tr(S_M)$ (préfixe de x), donc $\phi^{-1}(x_1) \in Tr(S)$. Comme $\phi^{-1}(x_1) \in Tr(I)$, on a $\phi^{-1}(x_1) \in \Sigma \cap Tr(I) \cap Tr(S)$ et $\phi^{-1}(x_1)!a = \phi^{-1}(x) \in Tr(I)$ et $\phi^{-1}(x_1)!a = \phi^{-1}(x) \notin Tr(S)$, donc $R_\Sigma(I, S)$ est faux.

b) soit $x = x_1 ?a/!e$. Comme dans le cas précédent on a $x_1 \in Tr(S_M)$. De plus, $x^{in} = x_1^{in} ?a \in Tr^{in}(S_M)$, donc nécessairement $x \in Tr(S_M)$ à cause de la forme particulière des éléments de $\phi(IOSM)$, ce qui constitue une contradiction.

On traite de la même façon le cas où c'est $\phi(S) \leq_{\Pi'} \phi(I)$ qui est faux.

2. nous montrons maintenant $\phi(I) =_{\Pi} \phi(S) \Rightarrow R_\Sigma(I, S)$ par sa contraposée. Si $R_\Sigma(I, S)$ est faux alors il existe $\sigma \in \Sigma$ telle que $\sigma \in Tr(I) \cap Tr(S)$ et $O(\sigma, I) \neq O(\sigma, S)$. Supposons par exemple que $a \in O(\sigma, I)$ et $a \notin O(\sigma, S)$. Par définition de Σ on a $\phi(\sigma)^{in} ?e \in \Pi'$. De plus, $\phi(\sigma)^{in} \in Tr^{in}(I_M)$, donc $\phi(\sigma)^{in} ?e \in Tr^{in}(I_M)$ car la modélisation explicite des blocages de sortie sur les IOSM fait que tout élément de l'image par ϕ d'une IOSM peut recevoir $?e$ dans tous les états. Pour la même raison $\phi(\sigma)^{in} ?e \in Tr^{in}(S_M)$. Donc $\phi(\sigma!a) \in V$. Or $\sigma!a \in Tr(I)$ donc $\phi(\sigma!a) \in Tr(I_M)$ et $\sigma!a \notin Tr(S)$ donc $\phi(\sigma!a) \notin Tr(S_M)$. Donc $\phi(I) \neq_{\Pi} \phi(S)$. On traite de même le cas symétrique où $a \notin O(\sigma, I)$ et $a \in O(\sigma, S)$. ❑

5.6 Exemple d'application

Pour conclure ce chapitre, nous présentons un exemple très simple d'utilisation de la méthode «importée». Cet exemple n'est pas destiné à montrer comment fonctionne la méthode HSI (comment sont détectées les erreurs, comment l'incomplétude des spécifications ou leur degré de flou influe sur la longueur de la suite de tests), mais il sert plutôt à illustrer les différentes étapes liées à l'utilisation de la fonction ϕ. Nous considérons donc la spécification représentée sur la figure 51 :

Remarquons que la modélisation explicite des blocages de sortie impose la présence de la transition étiquetée par $!\delta$ dans l'état S_0. L'automate transformé $\phi(S)$ est une PNFSM qui a les caractéristiques suivantes :

1. $Li = \{?e, ?a\}$
2. $Lo = \{!b, !c, !e, !\delta\}$
3. $Q = \{\varepsilon, ?a\}$ est un ensemble couvrant les états.

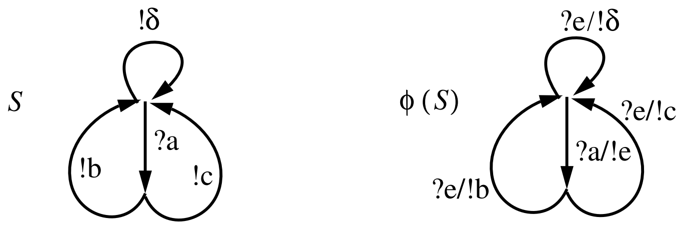


Figure 51 : exemple d'utilisation de la méthode importée

4. $D_0 = D_1 = \{?e\}$ sont des ensembles d'identification d'états harmonisés (en fait, dans ce cas particulier , nous avons affaire à une séquence de distinction).
5. le degré de flou est $\Delta = 1$ (l'ensemble $St = \{S_0, S_1\}$ est lui-même un ensemble d'états distingables).

Sous une hypothèse de régularité $R(2)$, la méthode HSI conduit donc à la suite de tests suivante :

$$\Pi = \bigcup_{\substack{S_0 = x \Rightarrow S_i \\ x^{in} \in Q \cdot (\{\varepsilon\} \cup Li \cup \ldots \cup Li^{\Delta m - n + 1})}} \{x^{in}\} \cdot D_i$$

$$= \bigcup_{\substack{S_0 = x \Rightarrow S_i \\ x^{in} \in \{\varepsilon, ?a\} \cdot \{\varepsilon, ?e, ?a\}}} \{x^{in}\} \cdot D_i$$

$$= \{\varepsilon \cdot D_0, ?a \cdot D_1, ?e \cdot D_0, ?a?e \cdot D_0\}$$

$$= \{?e, ?a?e, ?e?e, ?a?e?e\}$$

La clôture par préfixe conduit à l'ensemble suivant :

$$\Pi' = \{\varepsilon, ?e, ?a, ?a?e, ?e?e, ?a?e?e\}$$

L'ensemble Σ s'obtient alors en employant la formule définie à la section précédente :

$$\Sigma = \{\phi^{-1}(x) \mid x \in \phi(L_{IOSM}{}^{*}) \wedge x^{in}?e \in \Pi'\}$$

$$= \{\phi^{-1}(x) \mid x \in \phi(L_{IOSM}{}^{*}) \wedge x^{in} \in \{\varepsilon, ?a, ?e, ?a?e\}\}$$

$$= \{\phi^{-1}(x) \mid x \in \{\varepsilon, ?a/!e, ?e/!b, ?e/!c, ?e/!\delta, ?a/!e?e/!b, ?a/!e?e/!c, ?a/!e?e/!\delta\}\}$$

$$= \{?a, !b, !c, !\delta, ?a!b, ?a!c, ?a!\delta\}$$

Cet exemple illustre donc les différentes étapes de calcul nécessaires. On remarquera que l'ensemble final comprend sept séquences (plus la séquence vide ε, conséquence de la clôture par préfixe), mais que parmi celles-ci trois ne servent à rien, puisque ne faisant pas partie des traces de la spécification elles ne sont pas prises en compte dans la relation R_Σ. En les supprimant, nous obtenons quatre séquences, ce qui correspond au nombre de cas de tests défini par la méthode HSI.

Ceci met en lumière le fait que l'ensemble Σ que nous avons défini n'est en aucun cas optimisé. Notre but dans cette section était simplement d'illustrer la possibilité de transposer des résultats d'un modèle d'automates dans un autre. Cependant, ces résultats restent difficiles à appliquer, car les suites de tests fournies par de telles méthodes sont beaucoup trop longues pour être applicables dans des cas réels. Nous verrons, au chapitre 10, au prix de quels compromis les belles théories sur la régularité exposées dans ce chapitre sont utilisées, concrètement, pour la définition de suites de tests opérationelles.

CHAPITRE 6

Hypothèse d'indépendance

1 Introduction

Au chapitre précédent nous avons vu comment l'utilisation d'une hypothèse de régularité permet de ramener la vérification d'une relation de conformité infinie entre une implantation et sa spécification à celle d'une relation de conformité finie, la seule que nous serons capables de vérifier en pratique au moyen d'un algorithme de test au chapitre 9.

Cependant, compte tenu des algorithmes qui seront utilisés au chapitre 9, la longueur du test à effectuer est proportionnelle à la taille de l'ensemble fini de séquences Σ définissant la relation d'implantation finie R_Σ à vérifier. Or l'une des conclusions du chapitre 5 est que la taille des ensembles calculés au moyen de la méthode HSI version importation, sous hypothèse de régularité, est beaucoup trop grande pour que cette méthode soit directement applicable en pratique. Nous allons donc maintenant consacrer nos efforts, et trois chapitres, à étudier comment et sous quelles hypothèses il est possible de réduire la taille de ces ensembles Σ.

Pour cela, nous introduisons une méthode dite de *test abstrait*, qui consiste à tirer parti de la connaissance d'une structure particulière sur l'automate représentant la spécification. Cette structure nous permet de calculer un automate plus petit représentant une vue abstraite du système complet. Nous faisons alors l'hypothèse que pour vérifier la conformité entre l'implantation et la spécification complète il suffit d'établir une certaine relation entre les abstractions respectives de ces deux automates. Nous montrons enfin comment cette dernière relation peut être caractérisée par une relation de conformité finie entre l'implantation et la spécification, faisant appel à des ensembles de séquences beaucoup plus courts que ceux résultant d'une application directe de la méthode exposée au chapitre précédent.

Ce chapitre fournit une première illustration de cette méthode. Nous mettons ici en œuvre la pratique bien connue qui consiste à diviser un gros problème en plusieurs petits sous-problèmes plus faciles à résoudre individuellement. Concrètement, cela se fait en montrant que l'automate global que l'on teste s'exprime comme le produit d'automates plus petits. On modélise ainsi le fait que la spécification est constituée par la composition de plusieurs mécanismes indépendants, chacun d'eux étant représenté par un automate.

Mettant à profit la connaissance de cette structure, il est possible de remplacer le test de la conformité à l'automate global par les tests de conformité aux diverses composantes du produit. Faisant alors l'hypothèse que ces composantes sont effectivement indépendantes dans l'implantation testée, nous montrons que l'on peut déduire la conformité globale de ces diverses conformités partielles.

Quelques exemples d'application nous permettent alors de débattre de la pertinence d'une telle hypothèse, couramment faite en pratique, et pourtant ô combien dangereuse, car facilement prise en défaut si on ne la manipule pas avec précaution !

2 Méthode du test abstrait

La modélisation de systèmes de télécommunications un tant soit peu réalistes sous forme d'automates à entrées et sorties conduit à des automates ayant un très grand nombre d'états et de transitions (nous donnerons des exemples plus quantifiés au chapitre 10). Il n'est donc pas réaliste de tenter une application directe des méthodes issues du chapitre précédent, qui conduiraient à des tests de longueur excessive.

Donnons quelques chiffres, qui, bien qu'approximatifs, aident à situer l'ampleur du problème. L 'ensemble Σ défini au chapitre 5, section 5.5, a une taille comparable à l'ensemble Π à partir duquel il est calculé (la formule ne permet pas une comparaison précise des cardinaux de ces deux ensembles, mais ceci est vrai pour les ordres de grandeur , comme on peut le deviner à partir du petit exemple présenté). Cet ensemble Π résulte de l'application de la méthode HSI à l'automate transformé $\phi(S)$. Bien que l'article [Luo 93] ne donne pas de chiffre exact, une étude qualitative faite dans cet article montre que la longueur des suites de tests produites par la méthode HSI est comparable à celles issues de la méthode Wp, c'est à dire de taille inférieure, mais du même ordre de grandeur , que celles produites par la méthode W .

Tout ceci nous permet de nous raccrocher à l'article [Chow 78], qui chiffre la longueur de la suite de tests obtenue par la méthode W à $n^2 \times k$, où n est le nombre d'états de l'automate et k le nombre d'interactions. Encore ce chiffre concerne-t-il les automates déterministes et complets étudiés dans l'article ci-dessus, et l'étude de la méthode HSI montre que le non déterminisme (par l'augmentation du nombre des transitions) et surtout l'incomplétude des automates (à travers le degré de flou) accroissent significativement la longueur des suites de tests à employer . Considérons donc le chiffre ci-dessus comme une évaluation grossièrement sous-estimée de l'effort de test nécessaire à l'application de la méthode HSI dans le cas général. $\phi(S)$ ayant les mêmes états et à peu près les mêmes interactions que S, ce chiffre est valable pour l'application de notre méthode importée aux automates à entrées et sorties. Enfin le chapitre 9 nous montrera que dans le meilleur des cas, c'est à dire quand le non déterminisme ne fait pas des siennes, la longueur du test à effectuer pour vérifier une relation d'implantation finie, mesuré, comme indiqué dans la section 2.1 du chapitre 5, par le nombre d'interactions à échanger entre le testeur et l'implantation, est égale au cardinal de l'ensemble Σ définissant cette relation d'implantation, donc ici encore $n^2 \times k$.

Reportons nous alors au chapitre 10, où nous faisons état de la modélisation de quelques protocoles réels. Le LAPD par exemple [CCITT 88b], protocole de la couche liaison de l'interface usager-réseau du RNIS, serait représenté par un automate à entrées et sorties ayant plus de 10^6 états et 10^4 interactions (dans une certaine configuration du protocole; ces chiffres sont tirés de [Phalippou 91]). L'application directe de la méthode présentée au chapitre 5 conduirait donc à un test d'une longueur supérieure à 10^{16}, ce qui n'est évidemment pas acceptable d'un point de vue pratique : au rythme pourtant optimiste de dix échanges de messages par seconde, ce test durerait plus d'un million d'années, échéance à laquelle il est peu probable que le réseau Numéris soit encore en service !

Les concepteurs de suites de tests ne peuvent donc pas utiliser des modélisations des spécifications sous forme d'automates finis. Une analyse des suites de tests conçues manuellement par des experts nous enseigne que ceux-ci raisonnent à partir de *vues abstraites* des systèmes à tester, qui, ne représentant qu'une partie du problème, peuvent être modélisées par des automates de taille raisonnable. Par exemple :

1. on décompose un système complexe en plusieurs sous-systèmes, ou fonctions, que l'on déclare indépendants. Chacun de ces sous-systèmes est bien évidemment beaucoup moins complexe que le système global.
2. au lieu de considérer un protocole complet, on ne s'intéresse qu'à l'automate de contrôle, c'est à dire que l'on néglige les aspects liés aux données (cette définition est bien imprécise, mais n'oublions pas qu'il s'agit ici de faire état d'une pratique empirique d'experts, pour laquelle n'existe pas forcément de formulation mathématique précise).
3. lors de l'étude d'un système, on ne s'intéresse qu'aux interactions valides (les messages bien formés) qui ne représentent qu'un infime pourcentage des séquences de bits de longueur acceptable que l'on est susceptible de recevoir. Le comportement de la spécification face aux messages invalides, ce que l'on a coutume d'appeler sa robustesse, est ainsi ignoré, au bénéfice d'une réduction considérable de la taille du modèle.
4. on conçoit les tests à partir d'un tableau où sont représentés les réceptions de tous les types de messages possibles dans tous les états principaux du système. Suivant cette méthode, très souvent mise en œuvre en pratique, les *types* de messages et les états *principaux* sont des abstractions respectives des messages et des états de l'automate fini qui décrirait le système.
5. on conçoit les tests à partir des différentes fonctions qu'un système offre à ses utilisateurs. On essaie d'être assez exhaustif, c'est à dire que l'on prend les paragra-

phes de la norme un par un, chacun décrivant un cas de fonctionnement (établissement d'une liaison, rupture de la liaison à l'initiative du demandeur , rupture à l'initiative du demandé, etc.) et on fait un ou plusieurs tests pour chacun de ces cas. Là encore, chaque cas est une abstraction, représentant un grand nombre de séquences de messages possibles au niveau de l'automate fini.

Remarquons que tous ces types d'abstractions sont utilisés en pratique, souvent de façon combinée. Par exemple, il est courant pour le test d'un système d'isoler chaque protocole (point 1 ci-dessus), de ne tester que le comportement valide de chaque protocole (point 3) et d'utiliser soit la méthode décrite au point 4, soit celle décrite au point 5 afin d'énumérer les cas de tests (la première semble mieux convenir pour les protocoles des couches basses, tels le LAPD, le protocole D niveau 3, pour lesquels on s'intéresse surtout au mécanisme de l'automate mis en œuvre, la deuxième pour ceux des couches hautes, comme les protocoles de messagerie X.400, où la notion de service rendu à un utilisateur est de première importance [Boullier 93]).

Nous appelons *méthodes de test abstrait* ces méthodes de conception des tests faisant appel à des abstractions des systèmes à tester . Nous pouvons formaliser cette démarche en trois étapes :

1. trouver une *abstraction* adéquate de la spécification à tester . Dans le cadre de cette thèse, nous nous limitons aux abstractions que l'on peut modéliser par des automates à entrées et sorties, mais bien sûr dans le cas général d'autres formalismes pourraient être envisagés pour représenter ces abstractions. Pour être utiles, ces abstractions doivent être de taille significativement réduite par rapport à la spécification complète (voir la figure 52).

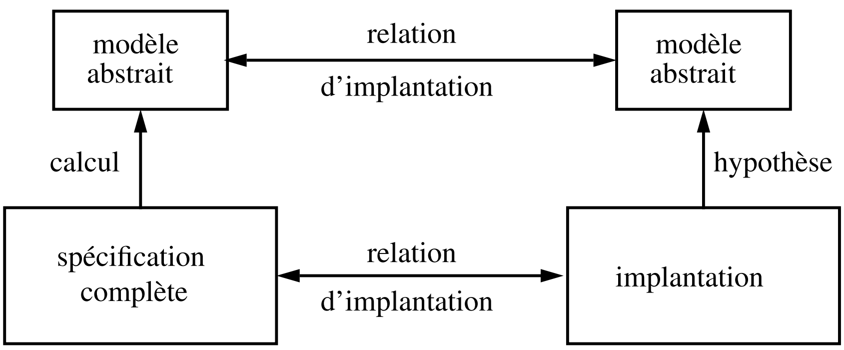


Figure 52 : automates détaillés et leurs abstractions

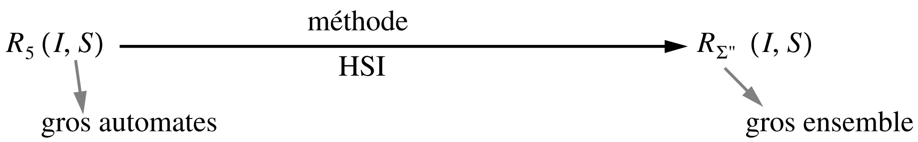
2. formuler précisément sous quelles *hypothèses* la conformité de l'implantation à la spécification complète peut résulter d'une relation entre les modèles abstraits de ces deux automates. Là encore dans le cas général, on peut imaginer des relations très variées. Cependant, dans cette thèse, nous nous limiterons à étudier le cas où la relation d'implantation $R_5(I, S)$ entre l'implantation et la spécification se ré-

duit, sous des hypothèses de test convenables, à la relation $R_5(I_A, S_A)$ ou à une relation d'implantation finie $R_\Sigma(I_A, S_A)$ entre les abstractions.

3. il reste enfin à étudier comment tester en pratique cette relation entre des abstractions. En effet, le test *concret*, en fin de compte, dialogue avec la véritable implantation, pas avec un modèle abstrait quelconque. On commence donc par se ramener, dans tous les cas, à la vérification d'une relation d'implantation finie R_Σ entre les abstractions, en appliquant la méthode décrite au chapitre précédent lorsque le point 2 exige la vérification de la relation R_5 entre les abstractions. Puis on montre comment ramener $R_\Sigma(I_A, S_A)$ à une relation entre les automates concrets $R_{\Sigma'}(I, S)$. Bien entendu, la taille de ce nouvel ensemble Σ' doit être comparable à celle de l'ensemble Σ, afin de préserver le gain sur l'effort de test résultant de l'utilisation d'un modèle abstrait réduit lors des étapes précédentes.

La méthode complète est résumée sur la figure 53, où elle est comparée avec la méthode décrite au chapitre précédent :

❶ méthode directe HSI



❷ méthode de test abstrait

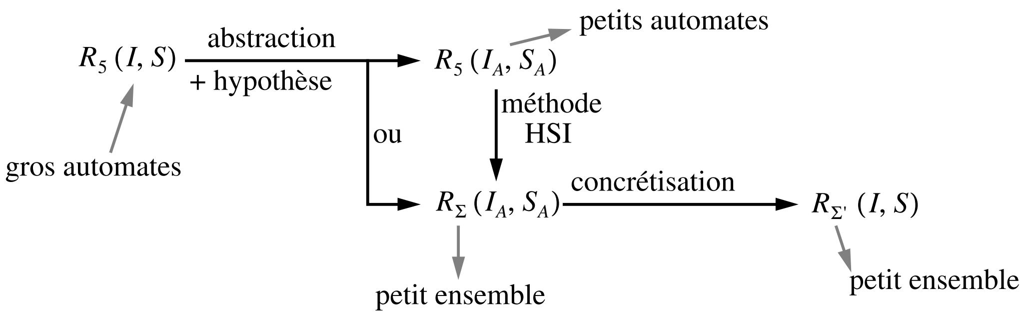


Figure 53 : méthode de test abstrait

Nous présentons dans la suite de ce chapitre une première application de cette méthode, qui illustre le point 1 de la page 113 (abstractions modélisant des sous-systèmes indépendants). D'autres suivront dans les deux chapitres suivants, présentant les abstractions liées à l'omission des données et aux interactions valides (points 2 et 3, traités au chapitre 7), puis celles liées à l'exploitation de structures abstraites sur l'automate, ce qui correspond aux points 4 et 5, étudiés au chapitre 8.

3 Produit d'automates indépendants

Nous présentons dans cette section les concepts mathématiques qui seront utilisés pour formuler l'hypothèse d'indépendance à la section suivante.

3.1 Indépendance et structure d'automates

Nous voulons modéliser le fait que la spécification étudiée est composée de plusieurs modules, fonctions ou mécanismes indépendants. Nous utilisons pour cela la notion de produit indépendant d'automates.

Définition 6.1 : soient $S_1 = \langle S_{s1}, L_1, T_1, s_{01} \rangle$ et $S_2 = \langle S_{s2}, L_2, T_2, s_{02} \rangle$ deux IOSM tels que $L_1 \cap L_2 = \emptyset$. Le *produit indépendant* $S = S_1 \times S_2 = \langle S_s, L, T, s_0 \rangle$ de ces deux automates est défini par :

1. $S_s = S_1 \times S_2$
2. $L = L_1 \cup L_2$
3. $s_0 = (s_{01}, s_{02})$
4. les transitions de S sont obtenues par les règles suivantes :
   $(s_1, \mu_1, s'_1) \in T_1 \Rightarrow (\forall s_2 \in S_{s2})\ (((s_1, s_2), \mu_1, (s'_1, s_2)) \in T)$
   $(s_2, \mu_2, s'_2) \in T_2 \Rightarrow (\forall s_1 \in S_{s1})\ (((s_1, s_2), \mu_2, (s_1, s'_2)) \in T)$

Le produit indépendant nous permet de composer des automates entre eux. Remarquons que d'autres choix auraient été possibles, par exemple le *produit libre* d'automates défini dans [Arnold 92]. Cependant, pour ce dernier, une transition du système composé est le produit de transitions de chacun des composants, alors que pour représenter des modules indépendants tels qu'ils sont étudiés dans ce chapitre il est plus commode, comme ci-dessus, de laisser chaque composant évoluer librement.

Nous utiliserons aussi la notion duale de projection, qui nous permet d'extraire d'un automate un autre automate plus petit représentant une partie seulement du comportement global. Pour cela, nous prenons un sous-ensemble des actions de l'automate global. La projection consiste à extraire de l'automate global un sous-automate qui ne comporte que les actions du sous-ensemble choisi. Les états de la projection sont définis comme des classes d'équivalences sur les états de l'automate global, qui regroupent des états que l'on doit considérer comme similaires du point de vue de la projection choisie.

Définition 6.2 : soit $S = \langle S_s, L, T, s_0 \rangle$ une IOSM et $L_1 \subset L$. Notons $L_2 = L - L_1$. Nous définissons une relation $\approx_{L_2} \subseteq S_s \times S_s$ en deux temps :

1. $s \leftrightarrow_{L_2} s'$ ssi $(\exists \sigma \in (\{!, ?\} \times L_2)^*)\ ((s, \sigma, s') \vee (s', \sigma, s))$
2. $\approx_{L_2}$ est la fermeture transitive de la relation $\leftrightarrow_{L_2}$: $s \approx_{L_2} s'$ si et seulement si $s \leftrightarrow_{L_2} s'$ ou $(\exists s_1, \ldots, s_{n-1} \in S_s)\ (s \leftrightarrow_{L_2} s_1 \wedge \ldots \wedge s_{n-1} \leftrightarrow_{L_2} s')$

Propriété 6.1 : la relation $\approx_{L_2}$ est une relation d'équivalence sur S_s.

Démonstration : la relation $\leftrightarrow_{L_2}$ est réflexive puisque si l'on note ε la séquence vide on a pour tout état (s, ε, s). Elle est symétrique puisque s'il existe une séquence qui justifie que $s \leftrightarrow_{L_2} s'$ alors la même séquence justifie que $s' \leftrightarrow_{L_2} s$. La relation $\approx_{L_2}$ est donc elle aussi réflexive et symétrique puisque $s \approx_{L_2} s'$ (car $s \leftrightarrow_{L_2} s$) et si l'on déduit $s \approx_{L_2} s'$ de $s \leftrightarrow_{L_2} s_1 \wedge \ldots \wedge s_{n-1} \leftrightarrow_{L_2} s'$ alors on a $s' \leftrightarrow_{L_2} s_{n-1} \wedge \ldots \wedge s_1 \leftrightarrow_{L_2} s$ d'après la symétrie de $\leftrightarrow_{L_2}$, donc $s' \approx_{L_2} s$. Enfin, $\approx_{L_2}$ est transitive par construction : si $s \approx_{L_2} s'$ et $s' \approx_{L_2} s''$ c'est que $s \leftrightarrow_{L_2} s_1 \wedge \ldots \wedge s_{n-1} \leftrightarrow_{L_2} s'$ et $s' \leftrightarrow_{L_2} s_{n+1} \wedge \ldots \wedge s_m \leftrightarrow_{L_2} s''$, donc $s \approx_{L_2} s''$. ❑

Cette relation d'équivalence nous sert à définir les états de l'automate projection.

Définition 6.3 : soit $S = \langle S_s, L, T, s_0 \rangle$ une IOSM, $L_1 \subset L$ et $L_2 = L - L_1$. La *projection* de S sur L_1 notée $Pr(S, L_1)$ est une IOSM $\langle S_{s1}, L_1, T_1, s_{01} \rangle$ telle que :

1. $S_{s1} = S_s / \approx_{L_2}$
2. $s_{01} = \tilde{s}_0$ (la classe d'équivalence de s_0)
3. les transitions de $Pr(S, L_1)$ sont obtenues par la règle suivante :

   $(s, \mu, s') \in T \wedge \mu \in L_1 \Rightarrow (\tilde{s}, \mu, \tilde{s}') \in T_1$

Dans ce chapitre nous utiliserons la notion de projection afin de retrouver les composantes d'un produit indépendant lorsque l'automate étudié s'exprime effectivement sous la forme d'un tel produit. Cela nous permet de définir ce que nous entendons par «événements indépendants» au sein d'un automate.

Définition 6.4 : soit $S = \langle S_s, L, T, s_0 \rangle$ une IOSM, $L_1 \subset L$ et $L_2 = L - L_1$. On dit que L_1 et L_2 sont des *ensembles d'actions indépendantes* si $S = Pr(S, L_1) \times Pr(S, L_2)$.

La cohérence des définitions que nous avons données pour le produit indépendant et pour la projection peut s'exprimer par la propriété suivante :

Propriété 6.2 : soient $S_1 = \langle S_{s1}, L_1, T_1, s_{01} \rangle$ et $S_2 = \langle S_{s2}, L_2, T_2, s_{02} \rangle$ deux IOSM tels que $L_1 \cap L_2 = \emptyset$. Soit $S = S_1 \times S_2 = \langle S_s, L, T, s_0 \rangle$. Alors $(Pr(S, L_1) = S_1)$ et $(Pr(S, L_2) = S_2)$.

Démonstration : nous allons montrer que $Pr(S, L_1) = S_1$. Dans un premier temps, nous caractérisons les classes d'équivalence des états de S pour la relation $\approx_{L_2}$, en montrant que des états équivalents ont une forme particulière.

1. Soit $s_2 \in S_2$. Tous les états d'une IOSM étant accessibles à partir de l'état initial, il existe $\sigma = \mu_1 \ldots \mu_n$ séquence d'éléments de $\{!, ?\} \times L_2$ telle que (s_{02}, σ, s_2). Soit $s_1 \in S_1$. Par définition du produit indépendant, on a donc $((s_1, s_{02}), \sigma, (s_1, s_2))$,

donc d'après la définition 6.2 $(s_1, s_{02}) \leftrightarrow_{L_2} (s_1, s_2)$. De même, quel que soit $s'_2 \in S_2$ on a une relation similaire : $(s_1, s_{02}) \leftrightarrow_{L_2} (s_1, s'_2)$, d'où il résulte par définition de la relation $\approx_{L_2}$, que $(\forall s_1 \in S_1)\ (\forall s_2, s'_2 \in S_2)\ ((s_1, s_2) \approx_{L_2} (s_1, s'_2))$.

2. De plus, considérons deux états de S équivalents pour la relation $\approx_{L_2}$: $(s_1, s_2) \approx_{L_2} (s'_1, s'_2)$. Par définition de l'équivalence, soit $(s_1, s_2) \leftrightarrow_{L_2} (s'_1, s'_2)$, soit il existe une suite d'états de l'automate composé (s_{1i}, s_{2i}), pour $0 \le i \le n$ (en notant $(s_1, s_2) = (s_{10}, s_{20})$ et $(s'_1, s'_2) = (s_{1n}, s_{2n})$) tels que pour tout i on ait : $(s_{1i}, s_{2i}) \leftrightarrow_{L_2} (s_{1(i+1)}, s_{2(i+1)})$. Or si $(s_a, s_b) \leftrightarrow_{L_2} (s'_a, s'_b)$ c'est qu'il existe $\sigma \in (\{!, ?\} \times L_2)^*$ telle que $((s_a, s_b), \sigma, (s'_a, s'_b))$ ou $((s'_a, s'_b), \sigma, (s_a, s_b))$. Donc dans tous les cas, par définition du produit, $s_a = s'_a$. On en déduit que quelle que soit la raison pour laquelle $(s_1, s_2) \approx_{L_2} (s'_1, s'_2)$, on a $s_1 = s'_1$.

Les classes d'équivalence des états de S pour la relation $\approx_{L_2}$ sont donc les ensembles $\ddot{s}_1 = \{(s_1, s_2), s_2 \in S_2\}$, pour tout $s_1 \in S_1$. L'application qui à s_1 associe $\ddot{s}_1$ est une bijection de S_{s1} sur $S_s/\approx_{L_2}$. Nous allons maintenant montrer que S_1 et $Pr(S, L_1)$ sont égaux à un isomorphisme près, l'isomorphisme étant défini par cette bijection. Les ensembles d'états sont donc en correspondance, de même que les états initiaux. De plus par définition ces deux automates ont le même ensemble d'interactions L_1. Pour terminer notre démonstration, il nous reste donc à montrer que $(s_1, \mu_1, s'_1) \in T_1$ si et seulement si $(\ddot{s}_1, \mu_1, \ddot{s}'_1) \in T_{Pr(S, L_1)}$. Nous aurons ainsi prouvé l'isomorphisme.

1. si $(s_1, \mu_1, s'_1) \in T_1$ alors $((s_1, s_{02}), \mu_1, (s'_1, s_{02})) \in T$ par définition du produit indépendant. Donc $(\ddot{s}_1, \mu_1, \ddot{s}'_1) \in T_{Pr(S, L_1)}$ par définition de la projection.

2. réciproquement, si $(\ddot{s}_1, \mu_1, \ddot{s}'_1) \in T_{Pr(S, L_1)}$ c'est qu'il existe $(s, \mu_1, s') \in T$ telle que $\tilde{s} = \ddot{s}_1$ et $\tilde{s}' = \ddot{s}'_1$ (par définition de la projection). Donc il existe $s_2 \in S_2$ tel que $s = (s_1, s_2)$ (par la caractérisation de la classe d'équivalence $\ddot{s}_1$ donnée ci-dessus). Par définition du produit indépendant, et puisque $((s_1, s_2), \mu_1, s') \in T$, c'est qu'il existe $(s_1, \mu_1, s''_1) \in T_1$ avec $s' = (s''_1, s_2)$. Donc par définition des classes d'équivalence $\tilde{s}' = \{(s''_1, s_2), s_2 \in S_2\} = \ddot{s}'_1 = \{(s'_1, s_2), s_2 \in S_2\}$ ce qui signifie que $s'_1 = s''_1$, donc que $(s_1, \mu_1, s'_1) \in T_1$.

Nous avons donc montré que $(Pr(S, L_1) = S_1)$. La démonstration de $(Pr(S, L_2) = S_2)$ s'obtient bien évidemment de façon semblable. ❑

Il est grand temps d'illustrer tout ce que nous venons de dire au moyen de quelques exemples simples. La figure 54 montre comment obtenir le produit de deux automates simples :

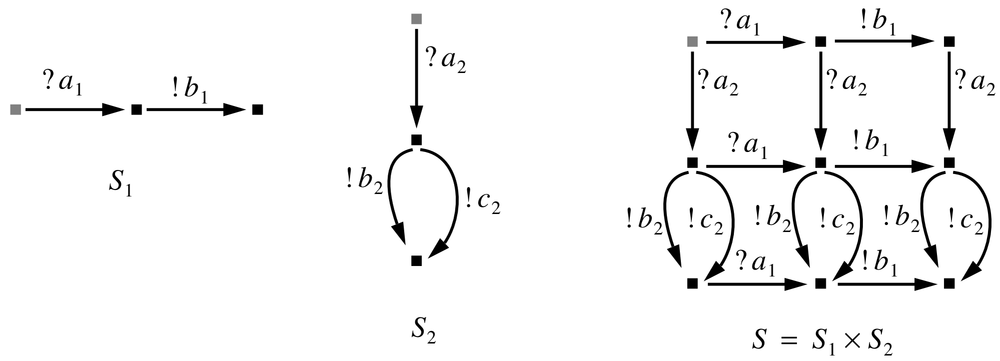


Figure 54 : produit indépendant d'automates

Par contre, la figure 55 illustre le cas où deux ensembles d'actions ne sont pas indépendants. Puisque les ensembles d'actions ne sont pas indépendants, l'automate global n'est pas égal au produit de ses projections sur ces ensembles d'actions, ce qui montre que le résultat exprimé par la propriété 6.2 peut être mis en défaut si l'automate global n'est pas un produit d'automates. Sur cet exemple, les ensembles $\{a_1, b_1\}$ et $\{a_2, b_2, c_2\}$ ne sont pas indépendants car les actions du premier ($?a_1$) ont une influence sur les actions possibles du deuxième (exécution de $!b_2$ ou de $!c_2$). Il y a donc une corrélation, au sein de l'automate global, entre les éléments de ces deux ensembles.

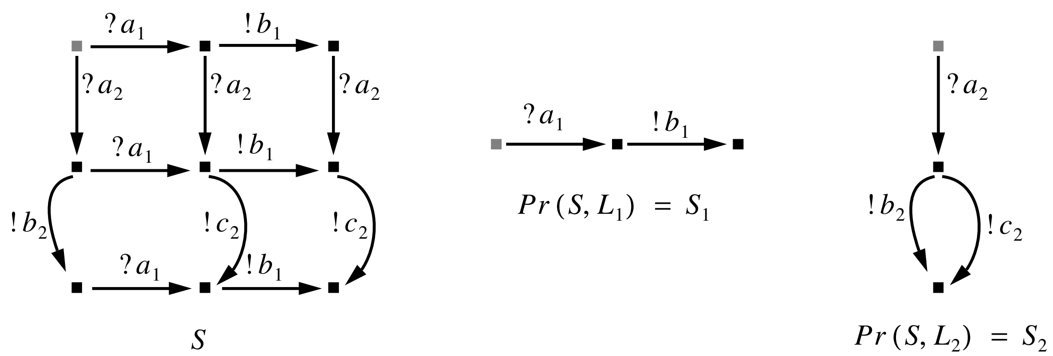


Figure 55 : ensembles d'actions non indépendantes

3.2 Actions indépendantes et traces

Les notions de produit indépendant et de projection ont été définies à partir de la structure des automates. Comme les notions de conformité que nous manipulons (c'est à dire les relations d'implantation) sont basées sur la notion de trace, il est intéressant d'étudier comment se traduit le produit d'automates en terme d'ensembles de traces. La propriété principale pour notre étude est que lorsqu'un automate s'écrit sous forme d'un produit, ses traces sont entièrement déterminées par les traces de ses composantes. Nous montrons cette propriété ci-dessous.

La notion de projection est étendue aux séquences d'événements de la façon suivante : soit $L_1 \subset L$ et $\sigma = \mu_1 \ldots \mu_n$ où $\mu_i \in \{!, ?\} \times L$. La projection d'une séquence est définie de façon inductive : $Pr(\varepsilon, L_1) = \varepsilon$, et si $|\mu_n| \in L_1$ alors $Pr(\sigma, L_1) = Pr(\mu_1 \ldots \mu_{n-1}, L_1) \cdot \mu_n$, sinon $Pr(\sigma, L_1) = Pr(\mu_1 \ldots \mu_{n-1}, L_1)$. En clair , la projection d'une séquence sur un ensemble d'actions est la sous-séquence ne comportant que les actions de cet ensemble. En termes savants, on parlera de morphisme effaçant.

Propriété 6.3 : soient $S_1 = \langle S_{s1}, L_1, T_1, s_{01} \rangle$ et $S_2 = \langle S_{s2}, L_2, T_2, s_{02} \rangle$ deux IOSM telles que $L_1 \cap L_2 = \varnothing$. Soit $S = S_1 \times S_2 = \langle S_s, L, T, s_0 \rangle$. Alors :

$$(\forall \sigma \in \{!, ?\} \times L)\ (Pr(\sigma, L_1) \in Tr(S_1) \wedge Pr(\sigma, L_2) \in Tr(S_2) \Leftrightarrow \sigma \in Tr(S))$$

Démonstration : soit $\sigma \in \{!, ?\} \times L$.

1. Montrons d'abord que $Pr(\sigma, L_1) \in Tr(S_1) \wedge Pr(\sigma, L_2) \in Tr(S_2) \Rightarrow \sigma \in Tr(S)$. Posons $Pr(\sigma, L_1) = \mu_{11} \ldots \mu_{n1}$. Puisque $Pr(\sigma, L_1) \in Tr(S_1)$ il existe $s_{01}, \ldots, s_{n1} \in S_{s1}$ tels que quel que soit $i \leq n$, $(s_{(i-1)1}, \mu_{i1}, s_{i1}) \in T_1$ (1). De même posons $Pr(\sigma, L_2) = \mu_{12} \ldots \mu_{m2}$. Puisque $Pr(\sigma, L_2) \in Tr(S_2)$ il existe $s_{02}, \ldots, s_{m2} \in S_2$ tels que pour tout $i \leq m$, $(s_{(i-1)2}, \mu_{i2}, s_{i2}) \in T_2$ (2). Comme $L_1 \cap L_2 = \varnothing$, nous pouvons poser $\sigma = \mu_1 \ldots \mu_{n+m}$. Nous montrons que $\sigma \in Tr(S)$ par itération pour i allant de 1 à $n+m$ sur la propriété :

$$(\exists j \in [0, n])\ (\exists k \in [0, m])\ ((s_{01}, s_{02}), \mu_1 \ldots \mu_i, (s_{j1}, s_{k2}))$$

   a) par définition $((s_{01}, s_{02}), \varepsilon, (s_{01}, s_{02}))$.
   b) supposons que $((s_{01}, s_{02}), \mu_1 \ldots \mu_i, (s_{j1}, s_{k2}))$. Si $|\mu_{i+1}| \in L_1$ alors par définition de la projection $Pr(\sigma, L_1)$, d'après (1) et par définition du produit, $((s_{01}, s_{02}), \mu_1 \ldots \mu_{i+1}, (s_{(j+1)1}, s_{k2})) \in T$. De même, si $|\mu_{i+1}| \in L_2$, en utilisant (2), on conclut $((s_{01}, s_{02}), \mu_1 \ldots \mu_{i+1}, (s_{j1}, s_{(k+1)2})) \in T$.

2. Montrons maintenant que $\sigma \in Tr(S) \Rightarrow Pr(\sigma, L_1) \in Tr(S_1) \wedge Pr(\sigma, L_2) \in Tr(S_2)$. Puisque $S = S_1 \times S_2$ on sait d'après la propriété 6.2 que $S_1 = Pr(S, L_1)$. Soit $\sigma = \mu_1 \ldots \mu_n \in Tr(S)$. Il existe $s_0, \ldots, s_n \in S_s$ tels que $(\forall i \leq n)\ ((s_{i-1}, \mu_i, s_i) \in T)$. Si $|\mu_i| \in L_1$ alors $(\tilde{s}_{i-1}, \mu_i, \tilde{s}_i) \in T_1$ par définition de la projection. Si $|\mu_i| \in L_2$ alors $\tilde{s}_{i-1} = \tilde{s}_i$ par définition de la relation d'équivalence $\approx_{L_2}$. De la définition de la projection sur les séquences d'événements, on déduit donc, par itération sur i allant de 1 à n que $Pr(\sigma, L_1) \in Tr(S_1)$. On montre de façon similaire que $Pr(\sigma, L_2) \in Tr(S_2)$. ❑

4 Test de conformité et hypothèse d'indépendance

4.1 Hypothèse d'indépendance

Nous considérons une spécification S qui peut s'exprimer comme un produit indépendant de deux automates : $S = S_1 \times S_2$ ayant respectivement pour ensembles d'événements L_1 et L_2. Nous nous proposons de tester la conformité d'une implantation I à cette spécification. Nous supposons bien sûr, comme nous l'avons fait dans les chapitres précédents, que les événements de l'implantation sont les mêmes que ceux de la spécification, c'est à dire $L = L_1 \cup L_2$. Pour pouvoir utiliser les résultats de la section précédente, donc pour tirer parti de la structure particulière de S, nous devons faire une hypothèse de test supplémentaire sur I.

Définition 6.5 : soit $I = \langle S, L, T, s_0 \rangle$ une IOSM, $L_1 \subset L$ et $L_2 = L - L_1$. Nous faisons une *hypothèse d'indépendance* sur I en supposant que L_1 et L_2 sont des ensembles d'actions indépendantes pour I. Nous notons IN cette hypothèse de test.

Nous discuterons à la section 5, par l'étude de divers exemples, de l'opportunité de faire en pratique des hypothèses d'indépendance. En effet, une telle hypothèse est souvent contestable, et peut être la cause de la non détection d'erreurs. Mais gardons nous pour l'instant de ces polémiques, et analysons seulement ici les conséquences de cette hypothèse quant à la définition des tests.

4.2 Test abstrait

Les deux projections $S_1 = Pr(S, L_1)$ et $S_2 = Pr(S, L_2)$ constituent des abstractions de la spécification S, au sens défini à la section 2. En effet, ils représentent des vues partielles de la spécification globale, et ce sont des automates beaucoup plus petits que celle-ci : si S_1 a n_1 états et S_2 a n_2 états, alors S a $n_1 \times n_2$ états, ce qui est beaucoup plus, dès que ces nombres sont grands, que la somme $n_1 + n_2$ des états des deux abstractions. Notons $I_1 = Pr(I, L_1)$ et $I_2 = Pr(I, L_2)$. L'hypothèse d'indépendance s'exprime donc par $I = I_1 \times I_2$. Sous l'hypothèse d'indépendance, la conformité de I à S est caractérisée par les conformités des projections de I, comme l'expriment les deux propriétés suivantes (c'est l'étape 2 de la méthode du test abstrait).

Propriété 6.4 : soient $S_1 = \langle S_{s1}, L_1, T_1, s_{01} \rangle$ et $S_2 = \langle S_{s2}, L_2, T_2, s_{02} \rangle$ deux IOSM tels que $L_1 \cap L_2 = \emptyset$. Soit $S = S_1 \times S_2 = \langle S_s, L, T, s_0 \rangle$. Soit I une IOSM ayant pour ensemble d'événements L. Posons $I_1 = Pr(I, L_1)$ et $I_2 = Pr(I, L_2)$. Alors :

$$R_5(I_1, S_1) \wedge R_5(I_2, S_2) \wedge (I = I_1 \times I_2) \Rightarrow R_5(I, S)$$

Démonstration: $R_5(I_1, S_1)$ signifie $(\forall \sigma \in Tr(S_1))\ (\sigma \in Tr(I_1) \wedge (O(\sigma, S_1) = O(\sigma, I_1)))$. Soit $\sigma \in Tr(S)$. D'après la propriété 6.3, puisque $S = S_1 \times S_2$, $Pr(\sigma, L_1) \in Tr(S_1)$. Donc

$Pr(\sigma, L_1) \in Tr(I_1)$ (en vertu de la conformité des projections $R_5(I_1, S_1)$). On montre de même que $Pr(\sigma, L_2) \in Tr(I_2)$. Par hypothèse d'indépendance $I = I_1 \times I_2$, donc nous pouvons appliquer la propriété 6.3 à l'automate I, ce qui permet de conclure que $\sigma \in Tr(I)$.

Soit $a \in O(\sigma, S)$. Cela signifie que $\sigma!a \in Tr(S)$, donc $\sigma!a \in Tr(I)$ (d'après ce que nous venons de montrer au paragraphe précédent), ce qui signifie $a \in O(\sigma, I)$. Donc $O(\sigma, S) \subseteq O(\sigma, I)$. Réciproquement, soit $a \in O(\sigma, I)$. Cela signifie que $\sigma!a \in Tr(I)$. Supposons que $a \in L_1$. D'après la propriété 6.3, $Pr(\sigma!a, L_1) = Pr(\sigma, L_1) \cdot !a \in Tr(I_1)$, donc $a \in O(Pr(\sigma, L_1), I_1)$. Comme $Pr(\sigma, L_1) \in Tr(S_1)$, on déduit de la conformité $R_5(I_1, S_1)$ que $a \in O(Pr(\sigma, L_1), S_1)$, donc que $Pr(\sigma!a, L_1) \in Tr(S_1)$.

De plus, $Pr(\sigma!a, L_2) = Pr(\sigma, L_2) \in Tr(S_2)$. Donc d'après la propriété 6.3, $\sigma!a \in Tr(S)$, ce qui signifie $a \in O(\sigma, S)$. Donc $O(\sigma, I) \subseteq O(\sigma, S)$. Nous avons donc montré $(\forall \sigma \in Tr(S))\ (\sigma \in Tr(I) \wedge (O(\sigma, S) = O(\sigma, I)))$, c'est à dire $R_5(I, S)$. ❑

La propriété précédente nous permet de conclure, sous l'hypothèse d'indépendance, à la conformité globale à partir des conformités des projections. Réciproquement, si une non-conformité est constatée sur une des projections, et en faisant à nouveau l'hypothèse d'indépendance, nous pouvons conclure à une non-conformité globale, comme l'exprime la propriété suivante.

Propriété 6.5 : soient $S_1 = \langle S_{s1}, L_1, T_1, s_{01}\rangle$ et $S_2 = \langle S_{s2}, L_2, T_2, s_{02}\rangle$ deux IOSM tels que $L_1 \cap L_2 = \emptyset$. Soit $S = S_1 \times S_2 = \langle S_s, L, T, s_0\rangle$. Soit I une IOSM ayant pour ensemble d'événements L. Soient $I_1 = Pr(I, L_1)$ et $I_2 = Pr(I, L_2)$. Alors :

$$R_5(I, S) \wedge (I = I_1 \times I_2) \Rightarrow R_5(I_1, S_1) \wedge R_5(I_2, S_2)$$

Démonstration : nous supposons $R_5(I, S)$ et nous allons montrer $R_5(I_1, S_1)$. Soit $\sigma = \mu_1 \dots \mu_n \in Tr(S_1)$. Il existe $s_{01}, \dots, s_{n1} \in S_{s1}$ tels que $(\forall i \le n)\ ((s_{(i-1)1}, \mu_i, s_{i1}) \in T_1)$. Par définition du produit indépendant, $(\forall i \le n)\ (((s_{(i-1)1}, s_{02}), \mu_i, (s_{i1}, s_{02})) \in T)$. Donc $\sigma \in Tr(S)$. Donc $\sigma \in Tr(I)$ puisque $R_5(I, S)$. Donc il existe $s_{0I}, \dots, s_{nI} \in S_i$ tels que pour tout $i \le n$, $(s_{(i-1)I}, \mu_i, s_{iI}) \in T_i$. Comme $(\forall i \le n)\ (|\mu_i| \in L_1)$, par définition de la projection $(\forall i \le n)\ ((\tilde{s}_{(i-1)I}, \mu_i, \tilde{s}_{iI}) \in T_{Pr(I, L_1)})$, donc $\sigma \in Tr(I_1)$.

Soit $a \in O(\sigma, S_1)$. Cela signifie que $\sigma!a \in Tr(S_1)$, donc $\sigma!a \in Tr(I_1)$ d'après le résultat du paragraphe précédent, c'est à dire $a \in O(\sigma, I_1)$. Donc $O(\sigma, S_1) \subseteq O(\sigma, I_1)$. Réciproquement soit $a \in O(\sigma, I_1)$. Cela signifie que $\sigma!a \in Tr(I_1)$, donc $\sigma!a \in Tr(I)$ puisque par hypothèse d'indépendance $I = I_1 \times I_2$. De plus comme $\sigma \in Tr(S_1)$ par hypothèse on a $\sigma \in Tr(S)$, car $S = S_1 \times S_2$. Comme $R_5(I, S)$ on en déduit que $\sigma!a \in Tr(S)$. Comme $\sigma!a \in Tr(I_1)$ on sait que $\sigma!a \in (\{!, ?\} \times L_1)^*$, et donc par définition de la projection $\sigma!a \in Tr(S_1)$. Donc $O(\sigma, I_1) \subseteq O(\sigma, S_1)$, ce qui achève de montrer que $R_5(I_1, S_1)$.

La preuve de $R_5(I_2, S_2)$ est similaire. ❏

4.3 T est concret

Nous avons montré à la section précédente comment ramener la vérification de la conformité globale à celle des conformités $R_5(I_1, S_1)$ et $R_5(I_2, S_2)$ entre les projections respectives de l'implantation et de la spécification. Nous pouvons alors appliquer les résultats du chapitre 5, afin de ramener la vérification de ces relations de conformité à des relations finies : nous savons construire un ensemble $\Sigma_1 \subseteq (\{?, !\} \times L_1)^*$ tel que pour tout automate I_1 satisfaisant les hypothèses de test « $IOSM$ complète sur les entrées», RI et $R(m)$, on ait :

$$R_5(I_1, S_1) \Leftrightarrow R_{\Sigma_1}(I_1, S_1)$$

(la même démarche est appliquée pour vérifier $R_5(I_2, S_2)$). Cependant, si l'on admet que l'automate global I est une modélisation fidèle de l'implantation, la projection $I_1 = Pr(I, L_1)$ n'est qu'une *abstraction*. Le test concret, comme expliqué à la section 2, doit porter sur l'implantation I. Nous montrons donc dans cette section comment ramener la vérification d'une relation abstraite comme $R_{\Sigma_1}(I_1, S_1)$ à celle d'une relation concrète : c'est la troisième étape de la méthode du test abstrait. Pour cela, nous devons raffiner les relations d'implantation finies que nous avons définies au chapitre précédent.

Définition 6.6 : soit $\Sigma \subseteq (\{!, ?\} \times L)^*$ un ensemble *fini* de séquences d'événements et $A \subseteq L$ un ensemble d'actions. On définit la relation :

$$R_{\Sigma, A}(I, S) \text{ ssi } (\forall \sigma \in \Sigma)\ (\sigma \in Tr(I) \cap Tr(S) \Rightarrow (O(\sigma, S) \cap A = O(\sigma, I) \cap A))$$

Remarquons que cette définition généralise celle du chapitre précédent. En effet il est immédiat de constater que $R_\Sigma(I, S) = R_{\Sigma, L}(I, S)$. Nous sommes maintenant en mesure de définir le test concret qui servira à tester la conformité des projections :

Propriété 6.6 : soient $S_1 = \langle S_{s1}, L_1, T_1, s_{01}\rangle$ et $S_2 = \langle S_{s2}, L_2, T_2, s_{02}\rangle$ deux IOSM telles que $L_1 \cap L_2 = \varnothing$. Soit $S = S_1 \times S_2 = \langle S_s, L, T, s_0\rangle$. Soit I une IOSM ayant pour ensemble d'événements L. Soient $I_1 = Pr(I, L_1)$ et $I_2 = Pr(I, L_2)$. On suppose $I = I_1 \times I_2$ (hypothèse *IN*). Soit $\Sigma_1 \subseteq (\{!, ?\} \times L_1)^*$ un ensemble fini de séquences. Alors :

$$R_{\Sigma_1}(I_1, S_1) \Leftrightarrow R_{\Sigma_1, L_1}(I, S)$$

Démonstration : nous montrons d'abord l'implication $R_{\Sigma_1}(I_1, S_1) \Rightarrow R_{\Sigma_1, L_1}(I, S)$. Soit $\sigma \in \Sigma_1 \cap Tr(I) \cap Tr(S)$. Comme $\Sigma_1 \subseteq (\{!, ?\} \times L_1)^*$, $\sigma \in \Sigma_1 \cap Tr(I_1) \cap Tr(S_1)$. Soit donc $a \in O(\sigma, I) \cap L_1$. Cela signifie que $\sigma!a \in Tr(I)$, donc que $\sigma!a \in Tr(I_1)$ puisque cette séquence n'est composée que d'actions de L_1. Comme $R_{\Sigma_1}(I_1, S_1)$, on en déduit que

$\sigma!a \in Tr(S_1)$, donc $\sigma!a \in Tr(S)$ (car $S = S_1 \times S_2$), et donc que $a \in O(\sigma, S) \cap L_1$. On montre de même que si $a \in O(\sigma, S) \cap L_1$ alors $a \in O(\sigma, I) \cap L_1$, ce qui en fin de compte permet de conclure que $R_{\Sigma_1, L_1}(I, S)$.

Réciproquement, montrons que $R_{\Sigma_1, L_1}(I, S) \Rightarrow R_{\Sigma_1}(I_1, S_1)$. Soit $\sigma \in \Sigma_1 \cap Tr(I_1) \cap Tr(S_1)$. Alors $\sigma \in \Sigma_1 \cap Tr(I) \cap Tr(S)$, car $S = S_1 \times S_2$ et $I = I_1 \times I_2$ (ce qui entraîne, par définition du produit indépendant, $Tr(I_1) \subseteq Tr(I)$ et $Tr(S_1) \subseteq Tr(S)$). De plus, considérons $a \in O(\sigma, I_1)$. Cela signifie $\sigma!a \in Tr(I_1)$, donc $\sigma!a \in Tr(I)$, donc $a \in O(\sigma, I) \cap L_1$. Comme $R_{\Sigma_1, L_1}(I, S)$, il s'ensuit $a \in O(\sigma, S) \cap L_1$, donc $\sigma!a \in Tr(S_1)$ car cette séquence n'est composée que d'actions de L_1, donc $a \in O(\sigma, S_1)$. On montre de même que si $a \in O(\sigma, S_1)$ alors $a \in O(\sigma, I_1)$, ce qui prouve donc $R_{\Sigma_1}(I_1, S_1)$. ❑

En conclusion, nous avons ramené la vérification de la conformité $R_5(I, S)$ à celle de deux relations d'implantation finies $R_{\Sigma_1, L_1}(I, S)$ et $R_{\Sigma_2, L_2}(I, S)$, sous les hypothèses de test suivantes :

1. L_1 et L_2 sont des ensembles d'actions indépendantes pour I.
2. $I_1 = Pr(I, L_1)$ est une IOSM complète sur les entrées, ayant un reset bien implanté et un nombre d'états dont on connaît une borne.
3. $I_2 = Pr(I, L_2)$ satisfait le même type d'hypothèses que I_1.

Ce faisant, nous avons bien réduit de façon significative l'effort de test. Esquissons une évaluation grossière, dans la veine de ce qui est fait à la section 2. On supposera donc ici aussi que l'effort de test résultant de l'application de la méthode HSI est de l'ordre de $n^2 \times k$, où n est le nombre d'états et k le nombre d'interactions de l'automate. Prenons l'exemple d'un gros automate ayant 10^4 états et 10^2 interactions, qui s'exprime comme le produit indépendant de deux petits automates ayant chacun 10^2 états et 50 interactions :

1. le test de l'automate global par application directe de la méthode HSI demande un effort de 10^{10}.
2. le test de chacun des petits automates abstraits par la méthode HSI demande un effort de $10^4 \times 50$. Comme le test concret se fait sur les mêmes ensembles de séquences Σ_1 et Σ_2 que le test abstrait, l'effort total de test en utilisant l'hypothèse d'indépendance est 10^6.

Ce qui, tout de même, constitue un gain d'un facteur 10^4 !

Il est possible d'exprimer une relation de la forme $R_{\Sigma_1, L_1}(I, S)$ (problème auquel nous nous sommes ramenés) comme une composition finie (assemblage de conjonctions ou de disjonctions) de propriétés élémentaires à vérifier sur l'implantation. Si nous posons :

$$R_{\sigma}(I) \text{ ssi } \sigma \in Tr(I)$$

$$R_{\neg\sigma}(I) \text{ ssi } \sigma \notin Tr(I)$$

alors on obtient immédiatement (en notant $O_1 = O(\sigma, S) \cap L_1$ et $O_2 = L_1 - O(\sigma, S)$) :

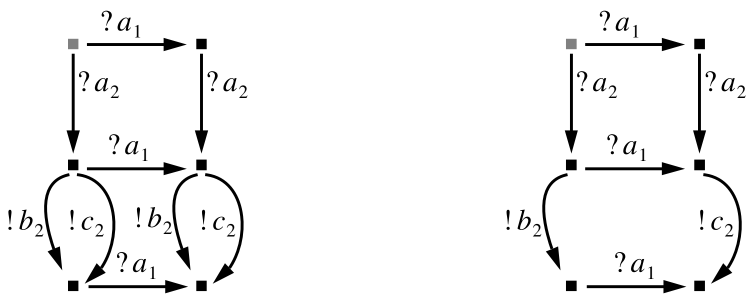
$$R_{\Sigma_1, L_1}(I, S) \text{ ssi}$$
$$(\forall \sigma \in \Sigma_1 \cap Tr(S))\ (R_{\neg\sigma}(I) \vee ((\forall a \in O_1)\ (R_{\sigma!a}(I)) \wedge (\forall a \in O_2)\ (R_{\neg\sigma!a}(I))))$$

Notons que les quantificateurs universels sur des ensembles finis équivalent à des conjonctions finies. Nous montrerons au chapitre 9 comment vérifier ce type de propriété au moyen d'un algorithme de test.

4.4 Importance de l'hypothèse d'indépendance

L'hypothèse d'indépendance $I = I_1 \times I_2$ est utilisée de façon cruciale dans la méthode de test par projection décrite ci-dessus, car elle est indispensable pour que les conformités entre les projections soient reliées à la conformité entre les automates globaux. Dans cette section nous illustrons par des contre-exemples comment les résultats démontrés sous l'hypothèse d'indépendance sont faux si cette hypothèse ne tient plus.

Si l'hypothèse d'indépendance n'est pas vérifiée il peut y avoir conformité entre les projections et non conformité entre les automates globaux. Ceci est illustré sur les figures 56 et 57 :



S $\qquad\qquad$ I

Figure 56 : un automate et une implantation non conforme

I n'est pas conforme à S puisque $?a_2!c_2 \in Tr(S)$ et $?a_2!c_2 \notin Tr(I)$ donc $\neg R_5(I, S)$. Cependant, calculons les projections sur les alphabets $L_1 = \{a_1\}$ et $L_2 = \{a_2, b_2, c_2\}$. Nous notons $S_1 = Pr(S, L_1)$, $S_2 = Pr(S, L_2)$, $I_1 = Pr(I, L_1)$ et $I_2 = Pr(I, L_2)$.

Comme $S_1 = I_1$ et $S_2 = I_2$ il y a conformité entre les projections : $R_5(I_1, S_1)$ et $R_5(I_2, S_2)$. Cependant les résultats de la section 4.2 ne sont pas applicables car l'hypo-

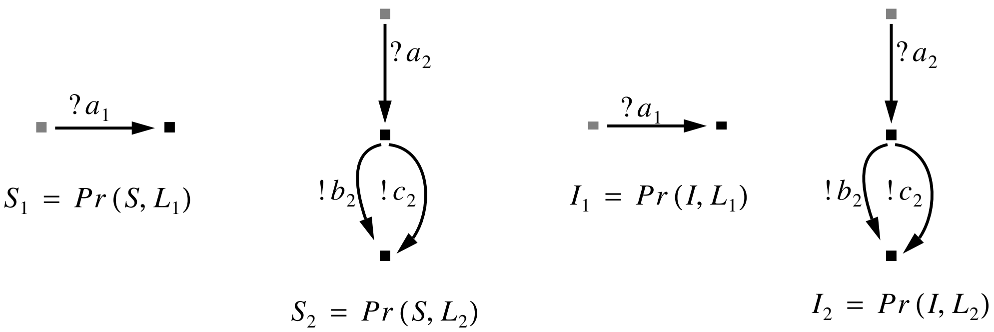


Figure 57 : les projections sont respectivement conformes

thèse d’indépendance n’est pas vérifiée : si nous avons bien $S = S_1 \times S_2$, par contre $I \neq I_1 \times I_2$.

Cet exemple est basé sur le fait que lorsque les ensembles d’actions considérés ne sont pas indépendants, le non-déterminisme peut être «plus grand» sur les projections que sur l’automate global.

Nous montrons maintenant que réciproquement si l’hypothèse d’indépendance n’est pas vérifiée il peut y avoir conformité entre les automates globaux mais pas conformité entre les projections. Considérons les automates ci-dessous :

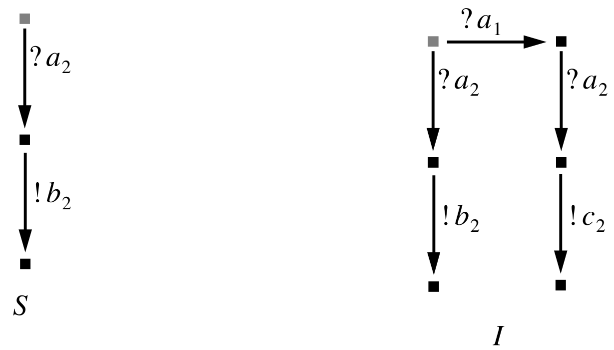


Figure 58 : un automate et une implantation conforme

I est conforme à S car $(\forall \sigma \in Tr(S))\ (\sigma \in Tr(I) \wedge (O(\sigma, S) = O(\sigma, I)))$, soit $R_5(I, S)$. Cependant, calculons les projections sur les alphabets $L_1 = \{a_1\}$ et $L_2 = \{a_2, b_2, c_2\}$. Nous notons $S_1 = Pr(S, L_1)$, $S_2 = Pr(S, L_2)$, $I_1 = Pr(I, L_1)$ et $I_2 = Pr(I, L_2)$.

$c_2 \in O(?a_2, I_2)$ et $c_2 \notin O(?a_2, S_2)$ donc $\neg R_5(I_2, S_2)$: I_2 n’est pas conforme à S_2. Comme dans l’exemple précédent, nous avons bien $S = S_1 \times S_2$, mais l’hypothèse d’indépendance n’est pas vérifiée car $I \neq I_1 \times I_2$. Cet exemple exploite le fait que les spécifications que l’on considère peuvent être incomplètes. Il semble que si l’on se restreint à des spécifications complètes pour les entrées, la conformité d’une implantation à une spécification

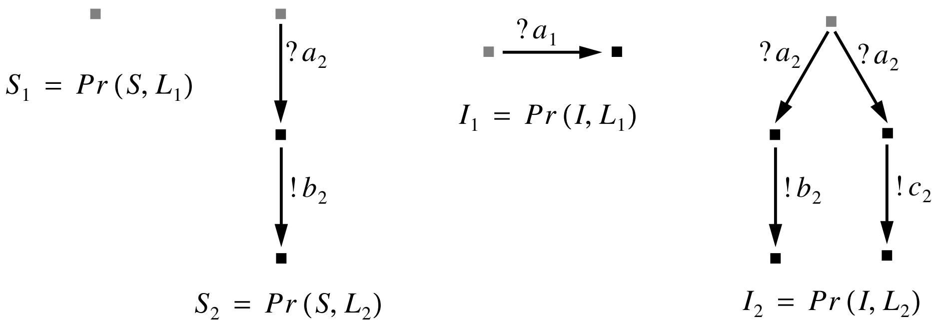


Figure 59 : la deuxième projection n'est pas conforme

où L_1 et L_2 sont des ensembles d'actions indépendants implique que ces ensembles d'actions sont indépendants sur I. Nous laissons de côté dans cette thèse l'étude de cette conjecture.

5 Applications

Dans les sections précédentes, nous avons consacré nos efforts à présenter une formalisation, adaptée à notre théorie du test, de la notion de fonctions indépendantes à l'intérieur d'un système donné. Nous avons montré comment il est possible d'exploiter l'hypothèse d'indépendance afin de réduire significativement la taille des suites de tests à appliquer pour vérifier la conformité, et comment formuler rigoureusement les limites de cette approche.

Il nous faut maintenant convaincre le lecteur que l'on est fondé à utiliser une telle hypothèse en pratique. En effet, le sujet prête à polémique. Avec un peu de mauvaise foi, ou de réalisme, il est facile de prétendre (et de montrer, exemples à l'appui) que *jamais*, dans un système complexe, deux fonctions ne sont totalement indépendantes. Pour s'en convaincre, il suffit de se reporter aux exemples étudiés dans la suite de cette section.

Débattre trop longtemps sur le sujet risque donc de mener tout droit à la conclusion que tout ce qui est fait dans ce chapitre peut être jeté aux oubliettes. Or nous n'avons pas le choix : confrontés à l'explosion combinatoire de la complexité des systèmes composés, l'hypothèse d'indépendance est le seul moyen efficace à notre disposition pour nous ramener à l'étude de problèmes de taille décente.

Il nous reste donc, et c'est l'objet de la présente section, à étudier comment utiliser en pratique les résultats de ce chapitre tout en sachant que les hypothèses qui sont nécessaires pour les obtenir sont rarement totalement justifiées. Plus précisément, nous allons montrer que dans certains cas nous avons intérêt à faire comme si elles étaient valides, car le risque est faible, et le gain important. Par contre, dans d'autres situations, il est déconseillé, car trop dangereux, de négliger imprudemment les dépendances entre automates.

5.1 Plusieurs instances d’une même fonction

Il est fréquent qu’un système fournisse des services à un ou plusieurs utilisateurs qui peuvent activer simultanément plusieurs instances d’une même fonction. Par exemple, dans le modèle OSI [CCITT 84], une entité de la couche $N+1$ peut avoir recours aux services de plusieurs entités de la couche N à travers différents points d’accès au service (SAP). Ceci est illustré sur la figure 60. En pratique, chacun de ces accès peut être

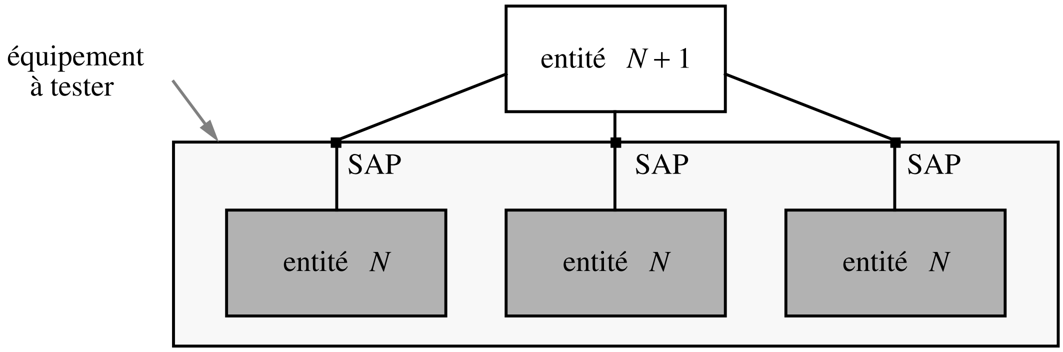


Figure 60 : utilisation de plusieurs entités de la couche N

implanté comme l’activation d’une même fonction, fournie par l’équipement qui implante la couche N. Si l’on considère l’automate global modélisant le comportement de cet équipement, on est donc confronté à un très gros automate, qui contient au moins le produit de tous les automates élémentaires qui représentent chacun une activation de la fonction.

Un exemple similaire est celui d’un commutateur fournissant un service téléphonique de base à un grand nombre d’abonnés. Chaque appel passé par un abonné peut être représenté sous la forme d’un petit automate, ayant, grosso-modo, une dizaine d’états (exemple aussi commun dans les exposés sur la commutation que le bit alterné peut l’être lorsqu’on parle de protocoles : on en trouvera une description dans [Boullier 93]). Cependant, comme illustré par la figure 61, le fonctionnement du commutateur contient le produit de tous ces automates élémentaires, et même si chacun d’eux n’a que dix états, lorsque l’on a 20 000 abonnés, cela fait tout de même un automate à 10^{20000} états, ce qui est beaucoup.

Dans chacun de ces deux cas, l’explosion combinatoire du nombre d’états de l’équipement à tester impose le recours à l’hypothèse d’indépendance. Celle-ci est ici relativement réaliste, puisque les instances indépendantes d’une même fonction, représentées sur les figures ci-dessus, vont probablement être implantées par des appels indépendants au même module de code.

Il faut néanmoins être conscient des pièges possibles : dans le premier cas, des ressources communes (partage de la mémoire disponible entre les différentes fonctions par exemple) peuvent cacher des dépendances entre les entités de protocole, sources d’erreurs. Dans le deuxième cas, des collisions d’appel peuvent être à l’origine d’une violation de l’hypothèse d’indépendance. Cependant, dans de tels cas, même si des erreurs

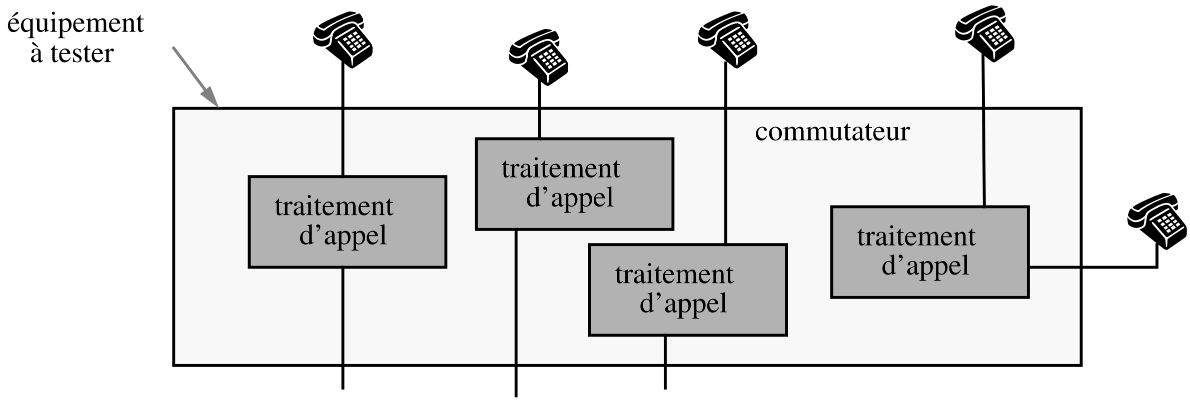


Figure 61 : les petits abonnés font les gros commutateurs

sont masquées, leur probabilité semble suffisamment faible pour ne pas justifier le surcoût, énorme, qui résulterait d'un abandon de cette hypothèse de test.

5.2 Le protocole INRES et son codeur

Le protocole INRES est un cas d'école destiné à illustrer , de la façon la plus simple possible, quelques-uns des mécanismes mis en œuvre dans les protocoles réels [Hogrefe 92]. Nous n'allons pas ici décrire encore une fois et en détail ce fameux protocole de transfert de données orienté connexion, mais nous nous intéresserons cependant à son architecture. En effet, une entité de protocole (que ce soit du côté appelant ou du côté appelé) est constituée de deux modules qui interagissent (voir la figure 62).

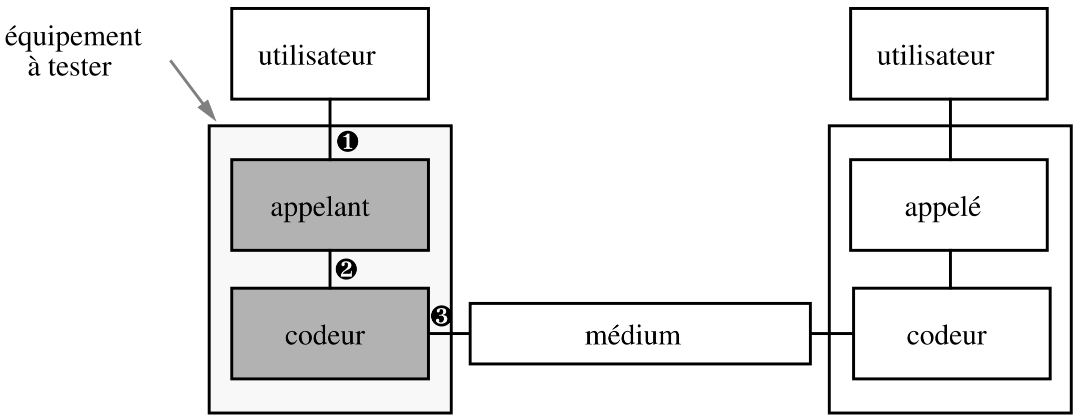


Figure 62 : architecture du protocole INRES

Le module appelant constitue le véritable fonctionnement du protocole; le module codeur , quant à lui, ne sert qu'à construire des trames de données acceptables par la couche inférieure, c'est à dire transmissibles à travers le médium : il ne réalise donc qu'une simple fonction de traduction. Si l'on modélise directement l'équipement à tester au moyen d'automates à entrées et sorties, et même dans le cas où l'on respecte l'architecture représentée sur la figure ci-dessus (c'est à dire si l'on modélise l'appelant et le codeur par deux automates distincts), on a toutes les chances d'aboutir à une solution comme celle de la figure 63. Dans ce modèle, la communication entre ces deux automa-

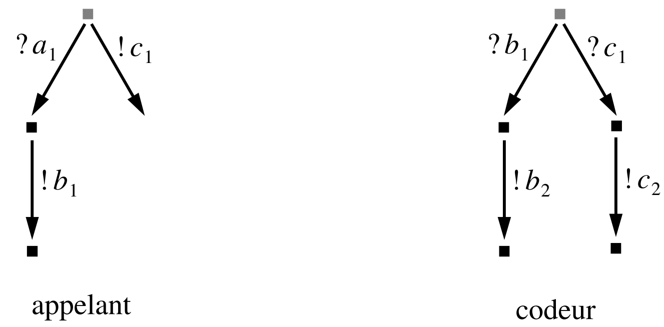


Figure 63 : modélisation directe du protocole INRES (vue partielle)

tes est directement représentée par l'utilisation d'actions communes (b_1 et c_1). Il n'est donc pas possible d'utiliser l'hypothèse d'indépendance. Or , au niveau de l'implantation, il est possible (sinon probable) que ces deux automates soient implantés par des modules logiciels différents ou même par des circuits intégrés différents. Un autre modèle, différent de celui de la figure 63, utilise des actions distinctes pour l'appelant et le codeur , le lien entre ces deux modules étant décrit par ailleurs, par exemple sous la forme d'une file d'attente qui transforme les b_1 en b_2 et les c_1 en c_2. On obtient alors la solution présentée sur la figure 64. Cette fois, l'automate global est bien le produit

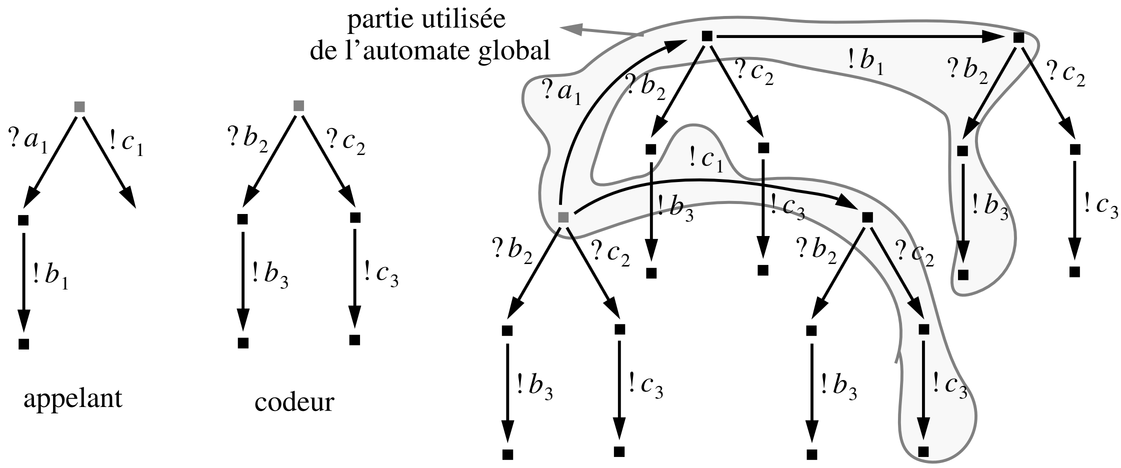


Figure 64 : une autre modélisation du protocole INRES (vue partielle)

indépendant des automates appelant et codeur : cela traduit l'indépendance intrinsèque de ces deux mécanismes. C'est seulement lors de leur utilisation conjointe que l'on se limite à utiliser une partie seulement du comportement de cet automate global.

Nous justifierons donc le test séparé des modules appelant et codeur en invoquant, ici aussi, l'hypothèse d'indépendance. Certes, sa justification, tout au moins dans le cadre de la formalisation faite dans ce chapitre, est plus délicate : nous avons eu bien du mal à montrer qu'il existe une modélisation de notre système, de surcroît pas la plus naturelle, qui se prête à cette hypothèse. De plus, le point d'observation intermédiaire ❷ de la figure 62, nécessaire pour pouvoir tester la conformité de chaque composant, n'est généralement pas accessible à l'expérience, en raison de l'architecture de test utilisée : nous sommes donc obligés de tester chacun des modules codeur et appelant à travers un environnement de test constitué de l'autre module (voir le chapitre 4). De ce fait, nous ne pouvons vérifier qu'une relation d'implantation affaiblie pour chaque composant, et certaines erreurs peuvent passer inaperçues (par exemple une double permutation des émissions de b et c dans les modules appelant et codeur ci-dessus, qui fait que le comportement observé à travers l'environnement semble normal).

Pourtant, sur cet exemple, les éventuels problèmes dus à l'interdépendance entre les fonctions testées ne sont pas primordiaux : le cœur du problème est le fonctionnement indépendant de ces fonctions. Il serait par contre probablement intéressant d'étudier comment enrichir les suites de tests conçues sous hypothèse d'indépendance avec de nouveaux tests destinés à traquer des erreurs additionnelles dues à l'interdépendance. Cette question revêt une grande importance en pratique, car par la force des choses on est bien souvent obligé de faire l'hypothèse d'indépendance, tout en sachant que c'est peut être là que se situe le défaut de la cuirasse, le talon d'Achille de la suite de tests obtenue. Cependant, l'approche retenue dans cette thèse ne nous semble pas adaptée pour calculer judicieusement ces tests additionnels : nous laisserons donc ce sujet pour des études ultérieures.

5.3 Interfonctionnement entre services

Enfin, il est des cas où il n'est pas recommandé de faire l'hypothèse d'indépendance, par exemple lorsque l'aspect critique du système sous test est précisément sa réaction à l'activation simultanée de plusieurs composants. Un exemple bien connu est celui des «services confort» offerts aux usagers du téléphone, comme les fameux compléments de service du RNIS. Ceux-ci ont les caractéristiques suivantes :

1. chaque service est décrit de façon séparée des autres dans les documents de spécifications informelles.
2. il est donc possible d'en faire des modèles sous forme d'automates indépendants, comme dans le cas de l'exemple précédent.
3. les différents services ont toutes les chances d'être implantés par des modules distincts.

L'hypothèse d'indépendance est donc bien tentante : elle conduirait à concevoir des tests séparés, chacun activant un service particulier. C'est, de plus, ce qui est souvent fait dans la pratique. Cependant, il est notoirement connu que des problèmes cruciaux peuvent être liés à l'activation simultanée de plusieurs de ces services. Par exemple, la combinaison de services de renvois d'appel et de services de filtrage d'appel peut aboutir à des situations non désirées [Combes 93]. De plus, comme indiqué dans cet article, les problèmes d'interactions entre services peuvent n'apparaître que lors de la phase d’implantation du système, et donc dans ce cas être indétectables au niveau de la spécification (en reprenant les termes de ce chapitre, cela signifie que la spécification est constituée de modules indépendants, mais que l’hypothèse d’indépendance est violée par l’implantation considérée). Il est donc recommandé dans ce cas, de concevoir les tests sans utiliser l’hypothèse d’indépendance.

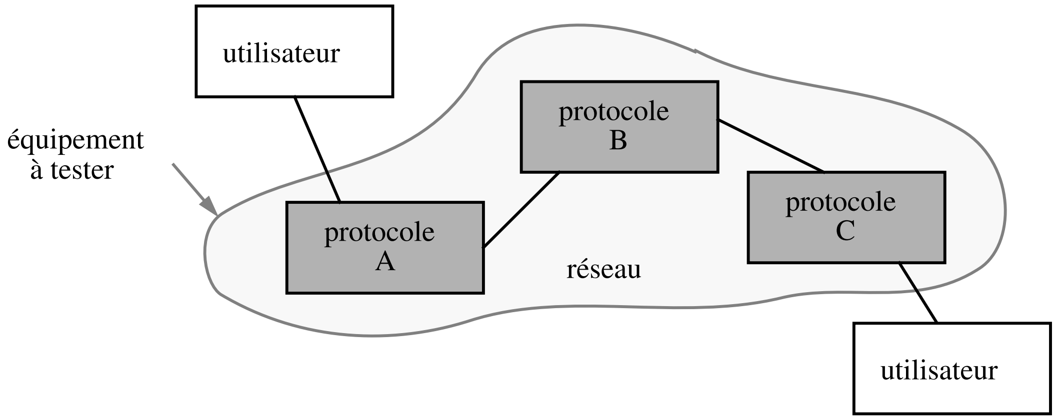


Figure 65 : test d’interfonctionnement de bout en bout

De même, comme son nom l’indique, le test d’ *interfonctionnement* est exactement à l’opposé du test fait sous hypothèse d’indépendance. Considérons l’exemple schématisé sur la figure 65. Chacun des protocoles A, B et C utilisés pour assurer la communication à travers le réseau entre les deux utilisateurs peut être considéré comme une entité à part. C’est le test dit «unitaire» des divers éléments de réseau correspondants. On fait alors l’hypothèse d’indépendance, ce qui évite d’avoir à recourir à une modélisation explicite du fonctionnement global du réseau. Le test d’interfonctionnement, quant à lui, s’intéresse au fonctionnement de bout en bout. Il est généralement conçu à partir de l’automate global composé [Rafiq 90] [Arakawa 92] [Castanet 93], ce que confirment les résultats de ce chapitre. Ces résultats fournissent également une explication possible (parmi d’autres) de la nécessité des tests d’interfonctionnement : si l’indépendance des systèmes n’est pas rigoureusement vérifiée, il peut y avoir conformité locale de chacun des sous-systèmes sans pour autant que la conformité globale soit garantie, comme indiqué à la section 4.4.

CHAPITRE 7

Hypothèse d'uniformité : congruences

1 Introduction

Au chapitre précédent nous avons introduit la méthode de test abstrait, et étudié un premier exemple d'application de cette méthode : comment diviser , afin de le résoudre plus facilement, un gros problème en plusieurs problèmes plus petits. Dans ce chapitre nous montrons que l'on peut parfois avoir intérêt à faire le contraire : nous montrons comment grouper ensemble des problèmes similaires afin de les résoudre d'un seul coup.

Rappelons la démarche qui a été suivie. Nous avons mis à profit la structure d'une spécification dans laquelle nous avons identifié des ensembles d'actions indépendantes. Nous avons montré que la conformité entre une implantation et cette spécification pouvait alors se déduire de la conformité entre des abstractions (en l'occurrence les projections), moyennant une hypothèse de test (en l'occurrence l'hypothèse d'indépendance) sur les implantations considérées. Nous avons également montré comment vérifier cette conformité entre des abstractions au moyen d'un test concret, appliqué, lui, à l'implantation bien réelle dont nous disposons.

Le *test par partitionnement* , que nous étudions dans ce chapitre, relève de la même démarche. Nous mettons à profit un autre type de structure (à savoir une congruence) sur les spécifications. Puis nous montrons que moyennant une hypothèse sur les implantations testées (dite hypothèse d'uniformité), la conformité entre une implantation et la spécification se déduit de la conformité entre des abstractions (les quotients) de ces automates. Enfin, nous montrons comment vérifier cette conformité entre des abstractions au moyen d'un test concret appliqué à l'implantation réelle. Cette méthode nous sert en particulier à montrer comment choisir les tests de robustesse des systèmes, ou comment faire abstraction des données véhiculées de façon transparente par un système de communication.

2 Congruences sur un automate

2.1 La congruence vue comme une structure

Nous voulons modéliser le fait que certains comportements dans un automate sont similaires. Pour cela, nous regroupons, au moyen de relations d'équivalence, les états et les événements que nous considérons comme équivalents. En conséquence, la structure que nous utilisons est basée sur une double partition de l'ensemble des états et de l'ensemble des événements d'un automate. Ce qui nous fait tout naturellement penser à la formulation anglaise *partition testing* : le *test par partitionnement* est une technique couramment employée dans le domaine du test de programmes séquentiels [Weyuker 91]. Ce que nous présentons dans ce chapitre peut être considérée comme une généralisation de cette méthode, de la même façon que nous présenterons les programmes séquentiels comme un cas particulier d'automates étendus, au chapitre suivant.

Définition 7.1 : soit $S = \langle S_s, L, T, s_0 \rangle$ une IOSM. Soit $\approx_S \subseteq S_s \times S_s$ une relation entre états et $\approx_L \subseteq L \times L$ une relation entre événements. Nous disons que $(\approx_S, \approx_L)$ est une *congruence* pour S si $\approx_S$ et $\approx_L$ sont des relations d'équivalence, et si elles sont compatible avec la structure de l'automate, c'est-à dire si :

$$(\forall s_1, s_2, s_3 \in S_s)\ (\forall \mu_1 \in \{?,!\} \times L)$$
$$(s_1 \approx_S s_2 \wedge (s_1, \mu_1, s_3) \in T) \Rightarrow$$
$$(\exists s_4 \in S_s)\ (\exists \mu_2 \in \{?,!\} \times L)\ (\ (s_2, \mu_2, s_4) \in T \wedge s_3 \approx_S s_4 \wedge \mu_1 \approx_L \mu_2)$$

La notion que nous voulons modéliser au moyen du concept de congruence est en quelque sorte une généralisation de celle de fonction calculée par un programme. Une fonction est un moyen d'associer à tout élément d'un domaine un élément d'un domaine image. Ici, les domaines de départ et les domaines image sont les classes d'équivalence d'états. Sur une classe d'équivalence d'états, une fonction correspond à toutes les transitions associées à une classe d'équivalence d'événements. La condition définissant une congruence signifie que sur un domaine ces fonctions sont totales. Attention : on autorise une sorte de non-déterminisme entre les fonctions au départ d'une classe d'équivalence d'états (ce qui correspondra au fait que les automates quotients définis un peu plus loin peuvent être non-déterministes). Ces considérations intuitives sont illustrées par la figure 66 ci-dessous.

Il est possible de définir la congruence en faisant appel à des concepts connus dans le domaine des systèmes de transitions étiquetés ou plus généralement de l'algèbre : $(\approx_S, \approx_L)$ est une congruence si et seulement si $\approx_S$ est une *bisimulation* sur $\langle S_s, L/\approx_L, T/\approx_L, s_0 \rangle$, ou bien $(\approx_S, \approx_L)$ est une congruence si et seulement si le passage au quotient par $(\approx_S, \approx_L)$ - voir ci-dessous - est un *homomorphisme* entre l'automate initial et l'automate quotient (c'est à dire s'il préserve la structure des transitions de l'automate). Cependant, nous garderons ici l'appellation «congruence» qui permet de désigner ce concept dans le cas précis étudié ici, i.e. le test par partitionnement des IOSM.

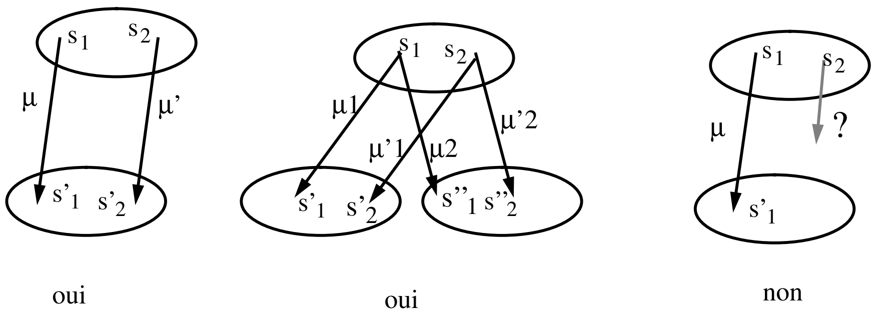


Figure 66 : congruence sur un automate

Une congruence est donc une *structure* sur un automate. Cette structure constitue une information supplémentaire par rapport à la donnée de l'automate initial. Elle ne permet pas de reconstituer l'automate : en effet, connaissant la congruence, nous connaissons seulement les domaines des «fonctions» mentionnées ci-dessus, mais pas comment celles-ci sont définies sur leur domaine; seule, la connaissance des transitions de l'automate élémentaire sous-jacent nous fournit cette information.

Nous montrons maintenant comment construire, à partir d'un automate et d'une congruence sur cet automate, une structure d'automate plus abstraite qui «factorise» le comportement de l'automate. Pour $s \in S_s$ et $a \in L$ nous notons $\tilde{s}$ et $\tilde{a}$ les classes d'équivalence de ces éléments. Pour $\mu = ?a$ (resp. $\mu = !a$) nous notons $\tilde{\mu}$ l'événement $?\tilde{a}$ (resp. $!\tilde{a}$). Cette notation est étendue aux séquences (si $\sigma = \mu_1 \ldots \mu_n$ alors $\tilde{\sigma} = \tilde{\mu}_1 \ldots \tilde{\mu}_n$).

Définition 7.2 : soit $S = \langle S_s, L, T, s_0 \rangle$ une IOSM et soit $(\approx_S, \approx_L) \subseteq (S_s \times S_s, L \times L)$ un couple de relations d'équivalence. L'automate *quotient* $S/(\approx_S, \approx_L) = \langle S_q, L_q, T_q, s_{0q} \rangle$ de S par $(\approx_S, \approx_L)$ est défini par :

1. $S_q = S_s / \approx_S$
2. $L_q = L / \approx_L$
3. $s_{0q} = \tilde{s}_0$
4. les transitions sont définies par la règle suivante : $(s, \mu, s_1) \in T \Rightarrow (\tilde{s}, \tilde{\mu}, \tilde{s}_1) \in T_q$

Bien que la définition ci-dessus puisse être appliquée à tout couple $(\approx_S, \approx_L)$ de relations d'équivalence, ce quotient n'est vraiment intéressant que pour les congruences. En effet c'est dans ce cas seulement que chaque transition de l'automate quotient peut être considérée comme une fonction, dont le domaine est l'état de départ et le codomaine est l'état d'arrivée de la transition. L'automate quotient représente bien dans ce cas la structure fonctionnelle de l'automate original.

2.2 Ensembles d'actions équivalentes

Nous avons introduit la notion de congruence comme une structure sur un automate qui permet de calculer un automate plus abstrait, le quotient. Cependant, dans notre approche du test, seuls les événements sont visibles de l'extérieur . Les états d'un automate n'interviennent pas dans la définition de la conformité. Nous montrons ci-dessous comment on peut définir une congruence uniquement à partir des événements d'un automate, à condition que les traces de cet automate vérifient une certaine propriété. Notons que de cette façon on obtient des congruences d'un type particulier , pour lesquelles les automates quotients sont déterministes : il n'est donc pas possible de définir ainsi toute congruence. Les exemples présentés dans ce chapitre (section 4) sont tous de ce type particulier; cependant nous étudierons au chapitre suivant des exemples qui ne le sont pas.

Soit $\approx_L \subseteq L \times L$ une relation d'équivalence. Nous étendons cette relation aux événements : pour $\mu_1 = ?a_1$ et $\mu_2 = ?a_2$ on a $\mu_1 \approx_L \mu_2$ ssi $a_1 \approx_L a_2$ et de même pour $\mu_1 = !a_1$ et $\mu_2 = !a_2$ on a $\mu_1 \approx_L \mu_2$ ssi $a_1 \approx_L a_2$. Puis pour les séquences $\sigma = \mu_1 \ldots \mu_n \in (\{!, ?\} \times L)^*$ et $\sigma' = \mu'_1 \ldots \mu'_n \in (\{!, ?\} \times L)^*$, on définit $\sigma \approx_L \sigma'$ ssi $\mu'_1 \approx_L \mu_1 \wedge \ldots \wedge \mu'_n \approx_L \mu_n$.

Définition 7.3 : soit $S = \langle S_s, L, T, s_0 \rangle$ une IOSM et $\approx_L \subseteq L \times L$ une relation d'équivalence. Nous disons que $\approx_L$ détermine des *ensembles d'actions équivalentes* pour S si et seulement si $(\forall \sigma, \sigma' \in Tr(S))\ (\forall \mu \in \{!, ?\} \times L)$
$(\sigma\mu \in Tr(S) \wedge \sigma \approx_L \sigma') \Rightarrow (\exists \mu' \in \{!, ?\} \times L)\ (\mu' \approx_L \mu \wedge \sigma'\mu' \in Tr(S))$

À partir d'une relation d'équivalence sur les événements qui détermine des ensembles d'actions équivalentes, nous pouvons construire une congruence.

Propriété 7.1 : soit $S = \langle S_s, L, T, s_0 \rangle$ une IOSM et $\approx_L \subseteq L \times L$ une relation d'équivalence qui détermine des ensembles d'actions équivalentes pour S. Soit $\approx_S \subseteq S_s \times S_s$ la clôture transitive de la relation $\sim_S$ définie par : $s \sim_S s'$ ssi
$(\exists \sigma, \sigma' \in Tr(S))\ (\sigma \approx_L \sigma' \wedge (s_{0s}, \sigma, s) \wedge (s_{0s}, \sigma', s'))$. Alors :

1. $\approx_S$ est une relation d'équivalence.
2. $(\approx_S, \approx_L)$ est une congruence pour S.

Démonstration : $\approx_S$ est réflexive puisque par hypothèse tout état d'une IOSM est accessible à partir de l'état initial. Elle est symétrique par la forme de la définition. Enfin elle est transitive par définition. Pour prouver le point 2, nous devons montrer que $(\approx_S, \approx_L)$ satisfait la propriété de compatibilité de la définition 7.1.

Soient $s_1, s_2, s_3 \in S_s$ et $\mu_1 \in \{!, ?\} \times L$. Si $s_1 \sim_S s_2$ alors il existe $\sigma_1, \sigma_2 \in Tr(S)$ telles que $\sigma_1 \approx_L \sigma_2 \wedge (s_{0s}, \sigma_1, s_1) \wedge (s_{0s}, \sigma_2, s_2)$ (par définition de la relation $\sim_S$). Si $(s_1, \mu_1, s_3) \in T$ nous déduisons donc $(s_{0s}, \sigma_1\mu_1, s_3)$. Comme $\approx_L$ détermine des ensembles d'actions équivalentes sur S, il s'ensuit qu'il existe $\mu_2 \in \{!, ?\} \times L$ tel que $\mu_1 \approx_L \mu_2$ et $\sigma_2\mu_2 \in Tr(S)$. Donc il existe $s_4 \in S_s$ tel que $(s_{0s}, \sigma_2\mu_2, s_4)$, ce qui entraîne $(s_2, \mu_2, s_4) \in T$, car les automates de traces sont déterministes au sens des automates. Or $\sigma_1\mu_1 \approx_L \sigma_2\mu_2$, donc nous savons aussi que $s_3 \sim_S s_4$.

Soient maintenant $s_1, s_2, s_3 \in S_s$ et $\mu_1 \in \{!, ?\} \times L$. Si $s_1 \approx_S s_2$, alors par définition il existe une suite d'états tels que $s_1 \sim_S s'_1$, $s'_1 \sim_S s'_2, \ldots, s'_n \sim_S s_2$. Si $(s_1, \mu_1, s_3) \in T$, en appliquant n fois le résultat obtenu ci-dessus, on en conclut qu'il existe $\mu_1 \approx_L \mu'_1$, $\mu'_1 \approx_L \mu'_2, \ldots$, $\mu'_n \approx_L \mu_2$ et $s_3 \sim_S s''_1$, $s''_1 \sim_S s''_2, \ldots, s''_n \sim_S s_4$ qui permettent de conclure $\mu_1 \approx_L \mu_2$ et $s_3 \approx_S s_4$, ce qui démontre la propriété. ❑

Dans la suite, nous parlons parfois par abus de notation de la congruence $\approx_L$ lorsque $\approx_L$ est une relation d'équivalence qui détermine des ensembles d'actions équivalentes pour S. De même, nous notons l'automate quotient $S/\approx_L$ au lieu de $S/(\approx_S, \approx_L)$.

2.3 Congruence et propriétés des traces

Les notions de conformité que nous manipulons sont toutes basées sur la notion de trace. Il est donc important d'étudier comment la structure de congruence sur un automate se traduit en propriétés permettant de relier les traces d'un automate à celles de l'automate quotient. D'où les deux propriétés ci-dessous. La première signifie qu'à toute trace de l'automate correspond une trace du quotient. La deuxième propriété montre que toute trace du quotient peut être *instanciée* (opération que nous définissons ci-dessous) par une trace de l'automate. Cette deuxième propriété sera utilisée dans la suite de ce chapitre pour construire des tests concrets qui simulent sur l'automate réel le fonctionnement de tests abstraits exprimés au niveau des automates quotients.

Propriété 7.2 : soit $S = \langle S_s, L, T, s_0 \rangle$ une IOSM et $(\approx_S, \approx_L)$ est une congruence pour S. Alors : $(\forall \sigma \in (\{!, ?\} \times L)^*)\ (\sigma \in Tr(S) \Rightarrow \tilde{\sigma} \in Tr(S/(\approx_S, \approx_L)))$

Démonstration : posons $\sigma = \mu_1 \ldots \mu_n$. Si $\sigma \in Tr(S)$ il existe $s_1, \ldots, s_n \in S_s$ tels que pour tout $i \in [1, n]$, $(s_{i-1}, \mu_i, s_i) \in T$. Par définition de l'automate quotient, $(\tilde{s}_{i-1}, \tilde{\mu}_i, \tilde{s}_i) \in T_q$, donc $\tilde{\sigma} \in Tr(S/(\approx_S, \approx_L))$. ❑

Propriété 7.3 : soit $S = \langle S_s, L, T, s_0 \rangle$ une IOSM et $(\approx_S, \approx_L)$ est une congruence pour S. Alors : $(\forall \sigma_q \in (\{!, ?\} \times L/\approx_L)^*)\ (\sigma_q \in Tr(S/(\approx_S, \approx_L)) \Rightarrow (\exists \sigma \in Tr(S))\ (\tilde{\sigma} = \sigma_q))$

Démonstration : soit $\sigma_q = \mu_{q1} \ldots \mu_{qn}$. Il existe $s_{q1} \ldots s_{qn} \in S_q$ tels que $(s_{q(i-1)}, \mu_{qi}, s_{qi}) \in T_q$ pour tout $i \in [1, n]$. Nous construisons par itération sur $i \in [1, n]$ les états $s_1 \ldots s_n \in S_s$

et les événements $\mu_1 \ldots \mu_n \in \{!, ?\} \times L$ tels que $(s_{i-1}, \mu_i, s_i) \in T$ pour tout $i \in [1, n]$ avec $\tilde{s}_i = s_{qi}$ et $\tilde{\mu}_i = \mu_{qi}$.

1. Puisque $(s_{q0}, \mu_{q1}, s_{q1}) \in T_q$, par définition du quotient il existe $s_1 \in S_s$ et $\mu_1 \in \{!, ?\} \times L$ tels que $(s_0, \mu_1, s_1) \in T$ et $\tilde{\mu}_1 = \mu_{q1}$ et $\tilde{s}_1 = s_{q1}$.
2. Supposons que nous avons construit $s_1 \ldots s_j \in S_s$ et $\mu_1 \ldots \mu_j \in \{!, ?\} \times L$ $(j < n)$ qui vérifient les propriétés énoncées plus haut. Puisque $(s_{qj}, \mu_{q(j+1)}, s_{q(j+1)}) \in T_q$, par définition du quotient il existe $s'_j, s'_{j+1} \in S_s$ et $\mu'_{j+1} \in \{!, ?\} \times L$ tels que $(s'_j, \mu'_{j+1}, s'_{j+1}) \in T$ et $\tilde{\mu'}_{j+1} = \mu_{q(j+1)}$ et $\tilde{s'}_j = s_{qj}$ et $\tilde{s'}_{j+1} = s_{q(j+1)}$. Puisque $(\approx_S, \approx_L)$ est une congruence pour S, (définition 7.1), il existe $\mu_{j+1} \approx_L \mu'_{j+1}$ et $s_{j+1} \approx_L s'_{j+1}$ tels que $(s_j, \mu_{j+1}, s_{j+1}) \in T$. ❑

Nous appelons *instanciation* d'une trace σ_q de l'automate quotient toute trace σ telle que $\tilde{\sigma} = \sigma_q$ (une telle trace existe toujours d'après la propriété précédente). Cette notion d'instanciation peut bien sûr être étendue à tout ensemble de traces Σ_q de l'automate quotient. Remarquons que les instanciations possibles d'une trace du quotient sont en général multiples, comme le montre l'exemple ci-dessous. Par contre, lorsque nous parlons d'une instanciation, c'est que nous avons fait le choix (arbitraire) de l'une d'entre elles.

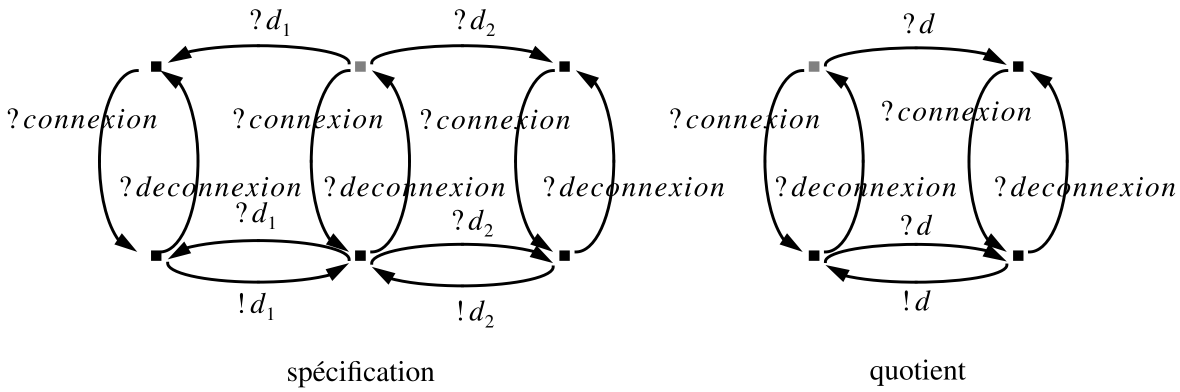


Figure 67 : un automate et son quotient

Il nous reste à illustrer les concepts introduits jusqu'ici au moyen d'un petit exemple. Considérons l'automate représenté sur la figure 67. Il représente une version très simplifiée d'un protocole de transmission de données orienté connexion. Cet automate a deux états principaux, repos et connecté, et les événements $connexion$ et $deconnexion$ font passer d'un état à l'autre. À n'importe quel moment, un message de données peut être envoyé à l'automate, qui le met dans une file d'attente de longueur 1, pour le réémettre dès que l'état connecté est atteint. Sur la figure, le message de données ne peut avoir que deux valeurs (actions d_1 et d_2) mais on peut en imaginer un nombre quelconque. La partition $\{\{connexion\}, \{deconnexion\}, \{d_1, d_2\}\}$ détermine des ensembles d'actions équivalentes selon la définition 7.3. Nous notons d la classe d'équivalence $\{d_1, d_2\}$. L'automate quotient est représenté sur la figure 67.

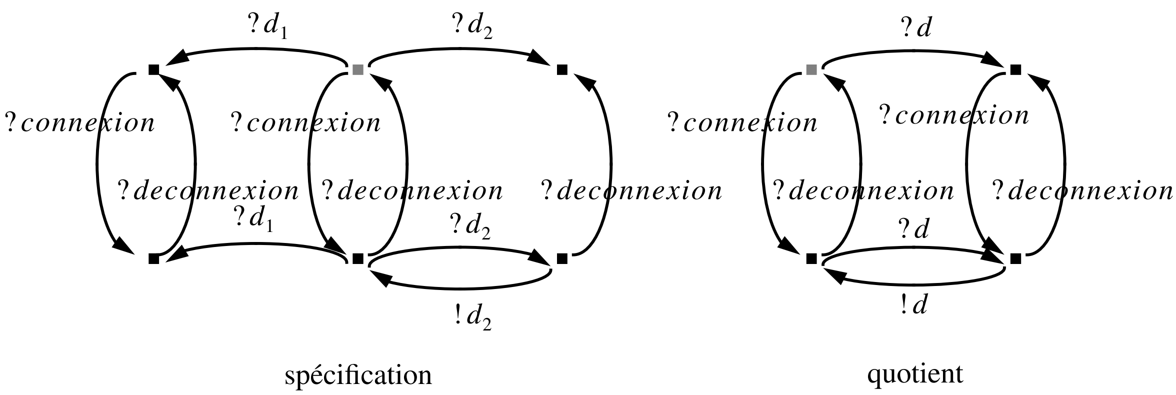


Figure 68 : exemple où il n'y a pas congruence

La trace $?d?connexion!d$ est la trace quotient correspondant à la trace $?d_1?connexion!d_1$ ou à la trace $?d_2?connexion!d_2$ de la spécification. Réciproquement, chacune de ces deux traces est une instanciation de la trace quotient ci-dessus. Enfin, il faut remarquer que l'hypothèse selon laquelle $\approx_L$ détermine des ensembles d'actions équivalentes pour S est capitale pour valider la propriété 7.3, comme le montre le contre-exemple de la figure 68 : on reprend les mêmes relations d'équivalences entre états et entre événements que celles de l'exemple précédent. En appliquant la définition 7.2, on obtient le même quotient que celui calculé précédemment. Par contre, sur ce nouvel exemple, le couple de relations d'équivalences n'est pas une congruence. La trace de l'automate quotient $?d?connexion!d$ n'a pas d'instanciation dans la spécification.

Ceci explique l'importance de la notion de congruence pour l'application de la méthode de test par partitionnement qui est présentée dans la suite de ce chapitre.

3 Test de conformité et hypothèse d'uniformité

3.1 Hypothèse d'uniformité

La méthode de test par partitionnement a été définie pour être appliquée au test de programmes séquentiels [Rapps 85] [Weyuker 91]. Elle consiste à diviser le domaine d'entrée d'un programme (à le partitionner) et ensuite à effectuer un seul test dans chacun des sous-ensembles ainsi définis. L'hypothèse de test sous-jacente est que sur l'un de ces sous-ensembles, le programme à tester est soit tout juste, soit tout faux.

Ce type d'hypothèse est repris et formalisé dans [Gaudel 92] sous le nom d' *hypothèse d'uniformité* . Cet article s'intéresse au test de spécifications exprimées sous la forme de types abstraits de données (voir le chapitre 5, section 3). Nous préférons employer ici la terminologie définie dans cet article, de préférence à «test par partitionnement», car elle situe mieux cette méthode de test dans le cadre plus général de la sélection basée sur les hypothèses de test, que nous avons adopté dans cette thèse. Pour compléter la présentation sommaire de cette théorie faite au chapitre 5, nous donnons ici la formulation mathématique de l'hypothèse d'uniformité (qui n'est autre qu'une reformulation de la description informelle du paragraphe précédent). Une hypothèse d'uniformité sur un domaine D est exprimée par la formule :

$$(\forall t_0 \in D)\ [Succ(P, \Phi(t_0)) \Rightarrow (\forall t \in D)\ (Succ(P, \Phi(t)))]$$

Les opérations définies sur des types abstraits de données décrivent des programmes séquentiels, qui consistent à calculer le résultat d'une fonction sur un certain domaine. Une vue intuitive du modèle d'un programme séquentiel et du test par partitionnement appliqué aux programmes séquentiels est illustrée sur la figure 69 ci-dessous.

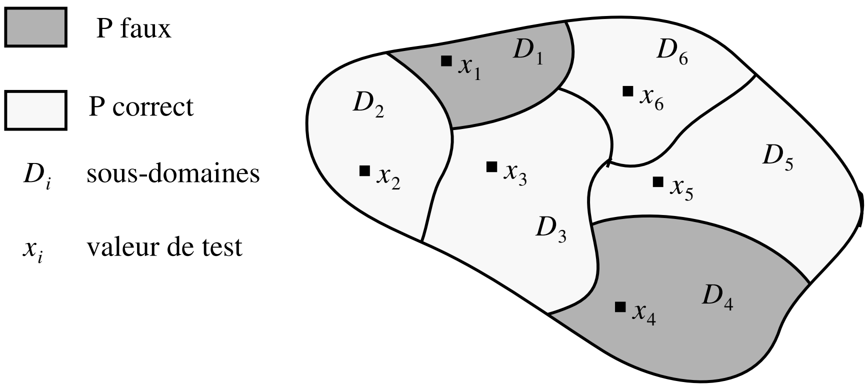


Figure 69 : test par partitionnement de programmes séquentiels

Les spécifications que nous considérons dans cette thèse sont des automates à entrées et sorties, qui relèvent d'un modèle différent de programme. Nous nous intéressons à l'enchaînement dans le temps d'interactions entre un système et son environnement. C'est pourquoi les relations d'implantation que nous étudions sont basées sur les traces d'événements observables.

Nous pouvons cependant considérer un IOSM comme une succession de fonctions, qui à chaque événement observé associent l'état de l'IOSM après cet événement. De la sorte, nous pouvons, grâce à la notion d'ensemble d'actions équivalentes qui a été introduite à la section 2.2, partitionner les domaines de ces fonctions successives, et appliquer au test des IOSM une approche similaire à celle du test par partitionnement des programmes séquentiels. Il se trouve, comme nous le montrerons à la section 4, que cette approche formalise une pratique courante pour tester certaines parties de tels automates.

Dans le cas de programmes séquentiels, le test par partitionnement consiste à vérifier une valeur d'entrée sur chacun des sous-domaines définis par le partitionnement. A vec l'analogie faite ci-dessus, cela revient, dans le cas d'un IOSM sur lequel on s'est fixé un partitionnement (une congruence), à tester une transition concrète pour chacune des transitions de l'automate quotient.

Définition 7.4 : soit $S = \langle S_s, L, T, s_0 \rangle$ un IOSM et $(\approx_S, \approx_L)$ une congruence pour S. $\Sigma_q \subseteq Tr(S / (\approx_S, \approx_L))$ est un ensemble de séquences *couvrant les transitions de l'automate quotient*, ssi :

$$(\forall (s, \mu, s') \in T_Q) \ (\exists \sigma \in \Sigma_q) \ ((s_{0q}, \sigma, s) \wedge \sigma\mu \in \Sigma_q)$$

Nous savons qu'un tel ensemble existe toujours puisque l'automate quotient est un IOSM, donc tous ses états sont accessibles à partir de l'état initial. Soit Σ une instanciation (au sens de la section 2.3) de cet ensemble Σ_q. Cette instanciation existe d'après la propriété 7.3. «Vérifier une transition concrète pour chaque transition de l'automate quotient» peut se formaliser par $R_\Sigma(I, S)$, c'est à dire, selon la définition 5.2, $(\forall \sigma \in \Sigma) \ (\sigma \in Tr(I) \cap Tr(S) \Rightarrow (O(\sigma, S) = O(\sigma, I)))$ (en réalité, on en fait un peu trop, puisqu'en testant toutes les sorties à l'issue des transitions de Σ on essaie deux fois certaines transitions. Mais ce n'est pas très important, car ça ne change pas significativement la taille du test, et de toutes façons ce qu'on fait est une hypothèse, comme indiqué ci-dessous).

Nous sommes maintenant en mesure de proposer une première formalisation de l'hypothèse d'uniformité dans le cas des automates à entrées et sorties :

Définition 7.5 : soit $S = \langle S_s, L, T, s_0 \rangle$ une IOSM et $(\approx_S, \approx_L)$ une congruence pour S. Soit Σ l'instanciation d'un ensemble de séquences couvrant les transitions de l'automate quotient $S / (\approx_S, \approx_L)$. Nous faisons une *hypothèse d'uniformité forte* (notée UF) sur une IOSM I en supposant que $R_\Sigma(I, S) \Rightarrow R_5(I, S)$.

Cependant cette hypothèse peut parfois sembler trop forte pour être réaliste. Vérifier R_Σ n'est pas toujours suffisant pour conclure avec une confiance raisonnable à la conformité de l'implantation selon R_5. En comparaison des programmes séquentiels, où il y a seulement un domaine d'entrée et une fonction sur ce domaine, nous avons en plus, dans le cas des automates, la structure de l'automate, c'est à dire l'enchaînement des transitions d'état en état. Nous aimerions tester que cette structure est la même pour l'implantation et pour la spécification. En effet, nous pouvons trouver des implantations qui sont conformes à leur spécification au sens de la relation R_Σ ci-dessus, mais qui ne sont pas conformes au sens R_5 car :

1. soit les actions qui sont équivalentes pour S ne le sont plus pour I.
2. soit la structure fonctionnelle des automates est différente, ce que nous traduirons par : les automates quotients n'ont pas la même structure.

Afin d'illustrer ce deuxième cas, considérons par exemple la figure ci-dessous. La spécification est celle déjà représentée sur la figure 67. L'implantation n'est pas conforme à

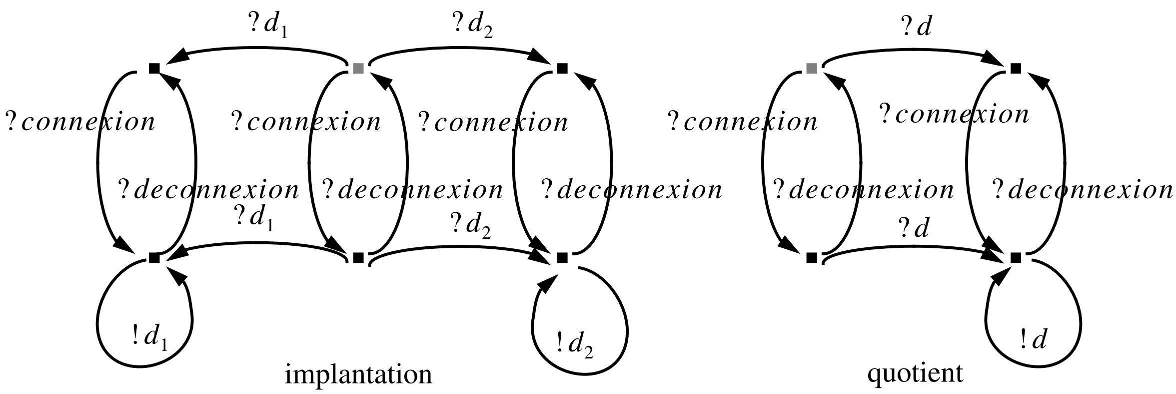


Figure 70 : contre-exemple à l'hypothèse d'uniformité forte

la spécification puisque $d_1 \in O(?d_1?connexion!d_1, I)$ et $d_1 \notin O(?d_1?connexion!d_1, S)$ (conformité au sens R_5). Pourtant, sur cet exemple, la partition utilisée sur la spécification ($\{\{connexion\}, \{deconnexion\}, \{d_1, d_2\}\}$) détermine des ensembles d'actions équivalentes pour l'implantation. De plus,

$$\Sigma = \{?connexion?deconnexion, ?connexion?d_1!d_1, ?d_1?connexion?deconnexion\}$$

constitue un ensemble d'instanciations de séquences couvrant les transitions de l'automate quotient, et on peut vérifier $R_\Sigma(I, S)$. Nous allons donc proposer une version plus faible de l'hypothèse d'uniformité, qui conduira à un test plus complet.

Une hypothèse d'uniformité ne se justifie que si les événements équivalents sont traités de façon équivalente dans l'implantation (voir discussion à la section 4). C'est ce qui est supposé dans le cas de programmes séquentiels. Ceci peut s'exprimer formellement par le fait qu'il existe une congruence $(\approx_{SI}, \approx_L)$ sur I dans laquelle $\approx_L$ est la relation d'équivalence sur les actions. Dans ce cas, cela a un sens de considérer l'automate quotient $I/(\approx_{SI}, \approx_L)$, et nous pouvons formaliser la notion de «similarité de structure» entre I et S par la conformité de leurs automates quotients : $R_5(I/(\approx_{SI}, \approx_L), S/(\approx_S, \approx_L))$. La version faible de l'hypothèse d'uniformité est alors : si nous avons une similarité de structure entre l'implantation et la spécification et si nous avons vérifié une transition concrète pour chaque transition de l'automate quotient, alors nous concluons à la conformité globale entre les automates. Soit :

Définition 7.6 : soit $S = \langle S_s, L, T, s_0 \rangle$ une IOSM et $(\approx_S, \approx_L)$ est une congruence pour S. Soit Σ l'instanciation d'un ensemble de séquences couvrant les transitions de l'automate quotient $S/(\approx_S, \approx_L)$. Nous faisons une *hypothèse d'uniformité faible* (notée Uf) sur une IOSM I en supposant que :

1. il existe une congruence $(\approx_{SI}, \approx_L)$ sur I.
2. $R_5(I/(\approx_{SI}, \approx_L), S/(\approx_S, \approx_L)) \wedge R_\Sigma(I, S) \Leftrightarrow R_5(I, S)$

Nous obtenons donc deux variantes de l'hypothèse d'uniformité. Elles correspondent à la mise en œuvre de tests différents.

Sous l'hypothèse d'uniformité forte, il nous suffit de vérifier $R_{\Sigma}(I, S)$ pour conclure à la conformité entre spécification et implantation. Nous montrerons au chapitre 9 comment et sous quelles hypothèses on peut définir des algorithmes de test pour vérifier une telle relation d'implantation finie.

Sous l'hypothèse d'uniformité faible, en plus de la vérification de la relation $R_{\Sigma}(I, S)$, il nous faut vérifier $R_5(I/(\approx_{SI}, \approx_L), S/(\approx_S, \approx_L))$ pour conclure. Dans ce cas, nous appliquons, comme au chapitre précédent, la méthode de test abstrait, mais cette fois-ci les abstractions de la spécification et de l'implantation sont les quotients de ces automates. En appliquant les résultats du chapitre 5, et donc en faisant une hypothèse de régularité, nous savons ramener la vérification de cette dernière relation à celle d'une relation finie entre les automates abstraits : nous pouvons construire un ensemble $\Sigma_q \subseteq (\{?, !\} \times L/\approx_L)^*$ tel que pour tout automate I_q (ayant ses actions dans $L/\approx_L$) satisfaisant les hypothèses de test $IOSM$ complète sur les entrées, RI et $R(m)$, on ait (notons $S_q = S/(\approx_S, \approx_L)$) :

$$R_5(I_q, S_q) \Leftrightarrow R_{\Sigma_q}(I_q, S_q)$$

3.2 Test concret

Il nous reste maintenant à montrer comment ramener la vérification de cette conformité abstraite à celle d'une vérification concrète portant sur l'implantation I. Pour cela, exprimons tout simplement ce que signifie la relation d'implantation abstraite à vérifier : $R_{\Sigma_q}(I_q, S_q)$ signifie

$$(\forall \sigma_q \in \Sigma_q)\ (\sigma_q \in Tr(I_q) \cap Tr(S_q) \Rightarrow (O(\sigma_q, I_q) = O(\sigma_q, S_q)))$$

Il est possible de ramener cette vérification abstraite à une vérification portant sur les traces de l'implantation réelle. Mais une vérification trop naïve et directe de la relation ci-dessus (basée sur une exploration exhaustive de toutes les instanciations de chaque séquence quotient, par exemple) conduirait à un test beaucoup trop long, qui nous ferait perdre tout l'avantage du recours à la méthode de test abstrait. L'idée à exploiter est de considérer que pour les interactions à envoyer vers l'implantation sous test le testeur peut faire un choix arbitraire, tandis que le testeur doit être prêt à recevoir n'importe quelle action concrète faisant partie d'une classe d'équivalence donnée, lorsque le test abstrait stipule que l'on doit recevoir telle action abstraite. En profitant des propriétés de la congruence, nous pouvons donc restreindre notre recherche : soit une séquence σ_q d'éléments de $L/\approx_L$, nous définissons l'ensemble de séquences $C(\sigma_q)$ par induction sur la structure de σ_q. Pour $a_q \in L/\approx_L$, notons $i(a_q) \in a_q$ un représentant quelconque (arbitraire) de cette classe d'équivalence. Alors :

$$C(\varepsilon) = \{\varepsilon\}$$

$$C(\sigma_q!a_q) = C(\sigma_q) \cdot \{!a \mid a \in a_q\}$$

$$C(\sigma_q?a_q) = C(\sigma_q) \cdot \{?i(a_q)\}$$

Au moyen de tels ensembles, il est possible d'exprimer la relation $R_{\Sigma_q}(I_q, S_q)$ en fonction des traces de l'implantation, comme indiqué par les deux propriétés suivantes.

Propriété 7.4 : soit $(\approx_{SI}, \approx_L)$, une congruence sur une IOSM $I = \langle S_I, L, T_I, s_{0I}\rangle$. Soit σ_q une séquence d'éléments de $L/\approx_L$. Alors : $\sigma_q \in Tr(I_q)$ ssi $(\exists\sigma \in C(\sigma_q))\ (\sigma \in Tr(I))$

Démonstration : la partie «si» de cette propriété est une reformulation de la propriété 7.2. Il nous reste donc ici à montrer que $\sigma_q \in Tr(I_q) \Rightarrow (\exists\sigma \in C(\sigma_q))\ (\sigma \in Tr(I))$. La démonstration de cette propriété est similaire à celle de la propriété 7.3. Soit $\sigma_q = \mu_{q1}\ldots\mu_{qn}$. Puisque $\sigma_q \in Tr(I_q)$ il existe $s_{q1}\ldots s_{qn} \in S_{Iq}$ tels que $(s_{q(i-1)}, \mu_{qi}, s_{qi}) \in T_{Iq}$ pour tout $i \in [1, n]$. Nous construisons par itération sur $i \in [1, n]$ les états $s_1\ldots s_n \in S_I$ et les événements $\mu_1\ldots\mu_n \in \{!, ?\} \times L$ tels que $(s_{i-1}, \mu_i, s_i) \in T_I$ pour tout $i \in [1, n]$ avec $\tilde{s}_i = s_{qi}$ et $\tilde{\mu}_i = \mu_{qi}$.

1. On pose $s_0 = s_{0I}$, ce qui correspond à la séquence $\sigma_q = \varepsilon$, qui vérifie la propriété ci-dessus.
2. Supposons que nous avons construit $s_1\ldots s_j \in S_I$ et $\mu_1\ldots\mu_j \in \{!, ?\} \times L$ $(j < n)$ qui vérifient les propriétés énoncées plus haut. Puisque $(s_{qj}, \mu_{q(j+1)}, s_{q(j+1)}) \in T_{Iq}$, par définition du quotient il existe $s'_j, s'_{j+1} \in S_I$ et $\mu'_{j+1} \in \{!, ?\} \times L$ tels que $(s'_j, \mu'_{j+1}, s'_{j+1}) \in T_I$ et $\tilde{\mu}'_{j+1} = \mu_{q(j+1)}$ et $\tilde{s}'_j = s_{qj}$ et $\tilde{s}'_{j+1} = s_{q(j+1)}$. On distingue alors deux cas :
   a) si $\mu_{q(j+1)} = !a_{q(j+1)}$, on a $\mu'_{j+1} = !a'$. Puisque $(\approx_{SI}, \approx_L)$ est une congruence pour I, (définition 7.1), il existe $\mu_{j+1} \approx_L \mu'_{j+1}$ et $s_{j+1} \approx_L s'_{j+1}$ tels que $(s_j, \mu_{j+1}, s_{j+1}) \in T$. De plus, par définition de l'ensemble C, on a $\mu_1\ldots\mu_j\mu_{j+1} \in C(\mu_{q1}\ldots\mu_{qj}\mu_{q(j+1)})$.
   b) si $\mu_{q(j+1)} = ?a_{q(j+1)}$, comme l'implantation est complète pour les entrées, il existe s_{j+1} tel que $(s_j, ?i(a_{q(j+1)}), s_{j+1}) \in T_I$ et donc $\mu_1\ldots\mu_j?i(a_{q(j+1)}) \in C(\mu_{q1}\ldots\mu_{qj}\mu_{q(j+1)})$ par définition de l'ensemble C. ❑

Propriété 7.5 : soit $(\approx_{SI}, \approx_L)$, une congruence sur une IOSM $I = \langle S_I, L, T_I, s_{0I}\rangle$, et $(\approx_S, \approx_L)$ une congruence sur une IOSM $S = \langle S_S, L, T_S, s_{0S}\rangle$. Soit σ_q une séquence d'éléments de $L/\approx_L$. Alors $O(\sigma_q, I_q) = O(\sigma_q, S_q)$ si et seulement si les deux propriétés suivantes sont vraies :

1. $(\forall a_q \in O(\sigma_q, S_q))\ (\exists\sigma \in C(\sigma_q!a_q))\ (\sigma \in Tr(I))$
2. $(\forall a_q \in L/\approx_L - O(\sigma_q, S_q))\ (\forall\sigma \in C(\sigma_q!a_q))\ (\sigma \notin Tr(I))$

Démonstration : $O(\sigma_q, I_q) \supseteq O(\sigma_q, S_q)$ si et seulement si pour toute action quotient $a_q \in O(\sigma_q, S_q)$ on a $a_q \in O(\sigma_q, I_q)$, soit $\sigma_q ! a_q \in Tr(I_q)$, ce qui d'après la propriété précédente 7.4 s'écrit $(\exists \sigma \in C(\sigma_q ! a_q))\ (\sigma \in Tr(I))$. L'inclusion inverse $O(\sigma_q, I_q) \subseteq O(\sigma_q, S_q)$ signifie que pour tout $a_q \notin O(\sigma_q, S_q)$ (c'est à dire pour $a_q \in L/\approx_L - O(\sigma_q, S_q)$) on a $a_q \notin O(\sigma_q, I_q)$, soit $\sigma_q ! a_q \notin Tr(I_q)$, ce qui d'après la propriété 7.4 s'écrit $(\forall \sigma \in C(\sigma_q ! a_q))\ (\sigma \notin Tr(I))$. ❑

Comme au chapitre précédent, pour faciliter la définition d'un algorithme de test au chapitre 9, nous exprimons la relation $R_{\Sigma_q}(I_q, S_q)$ comme une composition finie (assemblage de conjonctions ou de disjonctions) de propriétés élémentaires à vérifier sur l'implantation. Si nous posons:

$$R_{C(\sigma_q)}(I) \text{ ssi } (\exists \sigma \in C(\sigma_q))\ (\sigma \in Tr(I))$$

$$R_{\neg C(\sigma_q)}(I) \text{ ssi } (\forall \sigma \in C(\sigma_q))\ (\sigma \notin Tr(I))$$

alors on obtient immédiatement (en notant $O_1 = O(\sigma_q, S_q)$ et $O_2 = L/\approx_L - O(\sigma_q, S_q)$) :

$$R_{\Sigma_q}(I_q, S_q) \text{ ssi}$$
$$(\forall \sigma_q \in \Sigma_q \cap Tr(S_q))\ (R_{\neg C(\sigma_q)}(I) \vee [\,(\forall a_q \in O_1)\ (R_{C(\sigma_q ! a_q)}(I)) \wedge (\forall a_q \in O_2)\ (R_{\neg C(\sigma_q ! a_q)}(I))\,])$$

Notons que les quantificateurs universels sur des ensembles finis équivalent à des conjonctions finies. Nous montrerons au chapitre 9 comment définir un algorithme de test qui permette la vérification des relations ci-dessus. La principale qualité de cet algorithme est que la longueur du test nécessaire à la vérification de $R_{C(\sigma_q)}(I)$ est la même que celle qui serait nécessaire à la vérification de $\sigma_q \in Tr(I_q)$ si on avait un accès direct aux actions quotient et à l'automate quotient (et de même pour $R_{\neg C(\sigma_q)}(I)$ et $\sigma_q \notin Tr(I_q)$). On en conclut que l'effort de test résultant du test concret ci-dessus est identique à l'effort de test dont on a besoin pour vérifier , dans l'espace quotient, la relation R_{Σ_q}. En reprenant l'évaluation faite au chapitre précédent du coût de la méthode HSI, la vérification de cette relation demande donc un effort de l'ordre de $n_q^2 \times k_q$, où n_q est le nombre d'états de l'automate quotient et k_q le nombre d'événements de cet automate. Si nous prenons l'exemple d'un automate ayant 10^4 états et 10^2 interactions, tous deux groupés par paquets de 10 par les relations d'équivalences :

1. le test de l'automate par application directe de la méthode HSI demande un effort de 10^{10}.
2. le test par application de l'hypothèse d'uniformité faible demande un effort de l'ordre de 10^7 (en effet, l'automate quotient a au plus 10^4 transitions, ce qui donne une limite supérieure à l'effort requis pour tester la relation R_Σ, où Σ est une instanciation d'un ensemble de séquences couvrant les transitions de l'automate quotient).

3. le test par application de l'hypothèse d'uniformité forte demande un effort de 10^4.

4 Exemples d'application de la méthode

Comme dans le chapitre précédent, nous consacrons enfin une dernière section aux applications pratiques. Comme l'hypothèse d'indépendance, celles d'uniformité sont, en toute rigueur , souvent prises en défaut en pratique. Cependant, dans certains cas, comme ceux exposés ci-dessous, les risques sont suffisamment faibles pour justifier l'utilisation de la théorie exposée dans ce chapitre. Les hypothèses d'uniformité s'avèrent en effet particulièrement utiles pour rendre compte de la pratique courante du test des interactions *invalides* (ou test de *robustesse*) et de celui des *données transmises de façon transparente* .

Il faut noter que ce sont deux points dont l'originalité n'est pas prise en compte dans les théories classiques du test (aussi bien celles basées sur les systèmes de transitions étiquetés, décrites au chapitre 3, que celles basées sur les automates finis, décrites au chapitre 5). Or , l'observation de la pratique des centres de tests enseigne que ces types de tests reçoivent un traitement particulier , rendu nécessaire à cause de la complexité des systèmes réels et du besoin d'appliquer une sélection plus ou moins sévère selon que l'on teste des parties plus ou moins critiques des systèmes.

Ceci est généralement négligé : l'application des théories classiques du test à des systèmes réels conduit à ne pas modéliser les parties des spécifications correspondant à ces deux types de tests, ceci afin de ne pas s'encombrer de tests en trop grand nombre, mais avec l'inconvénient de passer de tout à rien. L 'approche originale choisie dans cette thèse permet au contraire de tenir explicitement compte du comportement en question, mais avec ses spécificités propres et le degré de sélection des tests approprié.

4.1 T ests de robustesse

Lors de la modélisation d'un système de télécommunications sous forme d'un automate à entrées et sorties, nous devons définir l'ensemble des interactions que peut recevoir ou émettre cet automate.

En général, la spécification en langage naturel (spécification CCITT par exemple) d'un protocole ou d'un système définit un certain nombre de messages permettant au système de dialoguer avec son environnement. Ces messages sont souvent présentés sous une forme structurée, regroupant des champs, eux-mêmes subdivisés en d'autres champs et ainsi de suite, de longueur fixe ou variable, obligatoires ou optionnels. Les techniques de description formelles normalisées par les instances internationales (ISO ou CCITT) comme ASN.1 [ISO 92a] ou Estelle [ISO 89a] se prêtent à de telles descriptions structurées des messages. Par exemple, en Estelle, un message comporte un nom de message et divers paramètres, chacun étant décrit par un type de données Pascal.

Cependant, ce qui est effectivement envoyé ou reçu, en fin de compte, par une implantation du protocole, ce sont des séquences de bits correspondant à des valeurs particulières du message (la structure n'apparait plus à ce niveau de modélisation). La correspondance entre la description structurée des messages et les séquences binaires apparaît dans la norme informelle, et le langage ASN.1, par exemple, va jusqu'à standardiser certaines règles de codage [ISO 92b].

On peut choisir comme ensemble L d'interactions du modèle l'ensemble de toutes les séquences de bits correspondant à toutes les valeurs autorisées des messages. Cet ensemble constitue les interactions *syntaxiquement valides* du système considéré. Mais si on limite la modélisation à cet ensemble, on ne représente pas ce qui se passe lorsqu'une implantation reçoit une interaction ne faisant pas partie de cet ensemble. Or nous avons fait l'hypothèse qu'il est toujours possible d'envoyer n'importe quel message vers une implantation (c'est pour cela que les implantations sont représentées par des automates complets sur les entrées). Il est donc important de contrôler (c'est à dire définir puis tester) les réactions du système dans les cas autres que ceux correspondant aux interactions syntaxiquement valides. Ces autres interactions sont appelées *syntaxiquement invalides*.

Le test du comportement en réponse à ces interactions constitue ce que l'on a coutume d'appeler le test de *robustesse*, par opposition au test de *conformité*, qui concerne le test d'une implantation en réponse aux interactions syntaxiquement valides. De notre point de vue, il s'agit dans *tous les cas* de tester la conformité de l'implantation, mais à des parties différentes de la spécification.

Remarquons que nous limitons ici notre définition du test de robustesse au test des réactions de l'implantation aux interactions invalides. Il est d'usage dans les milieux du test de classer les réceptions d'interactions valides entre événements «vraiment» valides (réceptions de bons messages aux bons moments) et événements dits *inopportuns* (réceptions de bons messages mais aux mauvais moments). Le test de ces derniers est souvent qualifié de test de robustesse. Selon notre définition ci-dessus, on a affaire plutôt à de la conformité. Tout ceci démontre une fois de plus qu'il est bien difficile de se mettre d'accord sur certains termes.

Si on ne borne pas leur longueur, l'ensemble de toutes les séquences de bits possibles est infini, donc nous ne pouvons pas le choisir comme ensemble L (l'ensemble des interactions d'un automate à entrées et sorties est fini). En pratique nous nous contenterons de fixer une borne n à la longueur des séquences de bits qu'il est intéressant de prendre en compte dans le modèle. Nous pouvons alors définir $L = \{0, 1\}^n$. Cet ensemble L contient 2^n éléments, ce qui est très gros lorsque l'on modélise des protocoles réels (pour le protocole de messagerie P1 par exemple [CCITT 88d], on peut avoir des interactions valides ayant 10^9 bits, ce qui oblige à prendre $n \geq 10^9$). Le sous-ensemble $L_v \subseteq L$ des interactions syntaxiquement valides est généralement beaucoup plus petit, de plusieurs ordres de grandeur.

Or le comportement d'un système en réponse à des interactions invalides présente souvent un caractère uniforme : le mécanisme de défense mis en œuvre est le même pour une grande partie de ces interactions. En pratique, le test de robustesse cherche à vérifier que cette réponse systématique est bien renvoyée, ceci pour les principaux «types d'erreurs» possibles. Cela suggère d'utiliser le test par partitionnement pour réduire de façon significative l'effort de test : on définit une relation d'équivalence $\approx_L$ sur l'ensemble L :

1. chaque interaction valide définit une classe d'équivalence.
2. les interactions invalides sont regroupées en un nombre réduit de classes d'équivalences qui représentent les types de fautes mentionnés ci-dessus.

Il est réaliste de supposer que la relation $\approx_L$ définit des ensembles d'actions équivalentes pour la spécification. Ceci résulte du caractère systématique, mentionné ci-dessus, de la réponse à des interactions invalides. Dans les cas où plusieurs types de réponses sont prévus, les événements correspondants sont bien sûr groupés dans plusieurs classes différentes.

Considérons par exemple le protocole LAPD [CCITT 88b] qui définit l'interface usager - réseau de la couche liaison pour l'accès au RNIS. La structure des trames utilisées par ce protocole est représentée sur la figure 71.

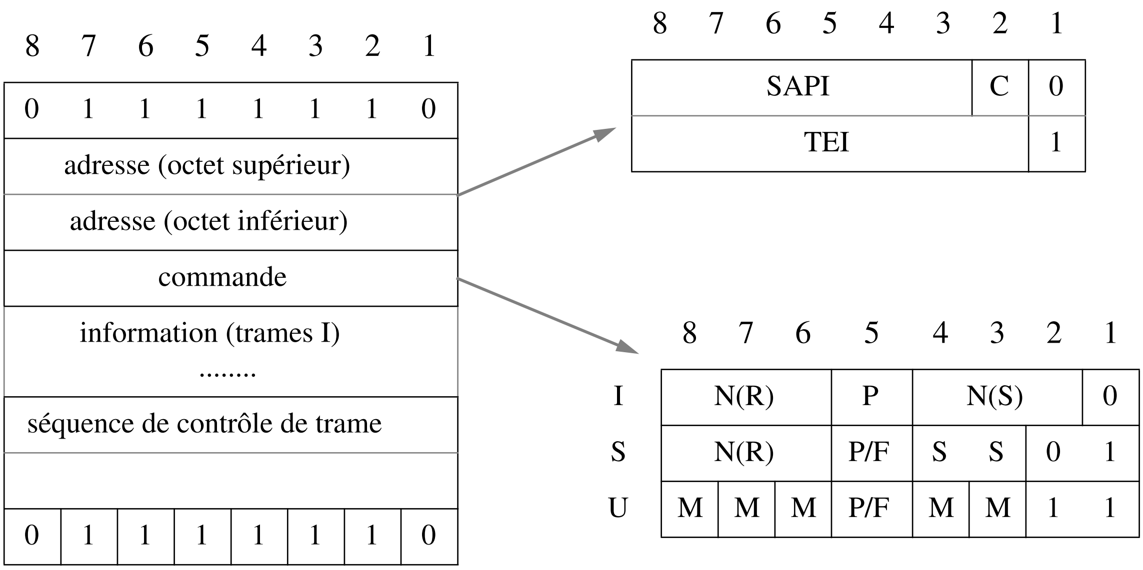


Figure 71 : structure des trames du LAPD

On se limite au cas où les commandes du protocoles sont codées sur un seul octet (dit «fonctionnement modulo 8»). La trame comporte 3 ou 4 parties :

1. *adresse* : bien que la norme définisse la structure de l'adresse et précise les valeurs réservées à certains champs (SAPI, TEI), nous considérons comme valide toute adresse ayant 0 comme bit de poids faible sur le premier octet et 1 comme bit de poids faible sur le deuxième octet. Ce qui fait 2^{14} adresses valides et 3×2^{14} adresses invalides.

2. *commande* : la définition précise des commandes possibles, non reprise sur la figure, est donnée par la norme. On trouve la répartition suivante : trames I (2^7 trames, toutes valides), trames S (48 valides et 16 indéfinies), trames U (16 valides et 48 indéfinies). Au total on a bien $2^8 = 256$ possibilités.
3. *information* : uniquement dans le cas de trames I, ce champ peut comporter 128 octets, ce qui fait $2^{128 \times 8}$ valeurs, toutes valides.
4. *séquence de contrôle de trame* : calculée par un algorithme en fonction du contenu de la trame. Une seule valeur est la bonne, les autres sont fausses, soit 1 séquence valide pour $2^{16} - 1$ séquences invalides.

La trame la plus longue est la trame I, qui peut donc atteindre 133 octets. De plus, lors du test du protocole, on est intéressé à tester le cas où les séquences reçues sont trop longues. Nous supposerons donc que dans notre modèle nous devons prendre en compte toutes les séquences de bits de longueur inférieure ou égale à 134 octets, soit $2^{134 \times 8}$ interactions possibles.

Les tests standardisés pour le protocole D (définis par des experts humains et normalisés par le CCITT) répartissent ce grand nombre d'interactions en cinq classes d'interactions invalides, plus les interactions valides. En se basant sur l'analyse ci-dessus, on peut évaluer la taille de ces classes (on suppose ici que l'on vérifie d'abord le contrôle de trame, puis la longueur de la séquence, enfin la validité de l'adresse et du champ de commande) :

1. *Erreur de séquence de contrôle de trame* : $2^{132 \times 8} \times (2^{16} - 1)$ c'est à dire la très grande majorité des trames (99,998 %).
2. *Trames I trop longues* : $1/2 \times (2^{132 \times 8} - 2^{131 \times 8})$ (256 fois plus que de trames I de longueur acceptable, dans notre modèle).
3. *Trames U trop longues* : $1/4 \times (2^{132 \times 8} - 2^{3 \times 8})$
4. *Trames S trop longues* : $1/4 \times (2^{132 \times 8} - 2^{3 \times 8})$
5. *Trames indéfinies* (c'est à dire de longueur correcte, mais invalides) : on ajoute celles d'adresse incorrectes $(3 \times 2^{14}) \times 2^8$ à celles d'adresse correcte mais de format de commande incorrect $2^{14} \times 56$
6. Enfin, parmi les trames *valides* , on distingue :

   a) les trames I : $2^{14} \times 2^7 \times 2^{128 \times 8}$

   b) les trames U et S : $2^{14} \times 64$

Dans la spécification quotient $S/\approx_L$ il n'y a donc que cinq interactions modélisant les interactions syntaxiquement invalides. Informellement, cette spécification est à peine plus grosse qu'une spécification où on n'aurait modélisé que les comportement syntaxiquement valides, en utilisant L_v comme ensemble d'interactions. L'effort de test requis pour le test de robustesse devient raisonnable (c'est le nombre de trames valides qui est encore inquiétant : nous verrons comment réduire celui-ci à la section suivante).

Le fait que l'implantation sous test satisfait l'hypothèse d'uniformité (faible, dans ce cas, car on ne fait pas confiance à la structure de l'implantation quotient) se justifie puisque les différents types d'erreurs (correspondant aux classes d'invalidité) sont gérées par des mécanismes indépendants. Par exemple, un circuit spécifique vérifie la séquence de contrôle de trame des interactions reçues : il est donc raisonnable de supposer que sur indication d'erreur une même action sera entreprise, quel que soit le contenu du reste de la trame.

Enfin, il ne faut pas avoir peur au vu des chiffres énormes ci-dessus obtenus en dénombrant les interactions. Les techniques de description formelle comme Estelle ou LDS permettent de décrire directement l'automate quotient $S/\approx_L$, et l'automate complet ne joue que le rôle d'une abstraction, utile pour raisonner sur les problèmes de test, mais qui n'est jamais manipulée en pratique. Une interaction au niveau de la spécification formelle est un élément de $L/\approx_L$. Nous avons réalisé un codage en Estelle du LAPD, par exemple, où un message Estelle *I-too-long* représente la classe d'équivalence des trames I trop longues.

4.2 Données transmises de façon transparente

Une des finalités d'un système de communication est de permettre la transmission d'information entre des utilisateurs distants sans altérer cette information. Pour cela, les protocoles comportent généralement des champs de données qu'ils transmettent sans même en analyser le contenu. Pour le protocole LAPD étudié à la section précédente, c'est le cas de la partie information des trames I : la seule vérification qui est faite est que cette partie a une longueur acceptable.

Le contenu de ce champ information n'a aucune influence sur le fonctionnement du protocole, tel qu'on le trouve décrit par la spécification informelle. Dans le modèle du protocole sous la forme d'un automate à entrées et sorties, une partition des interactions qui regroupe les trames ne différant que par leur champ information va donc définir des ensembles d'actions équivalentes. On peut se reporter à l'exemple décrit sur la figure 67, qui est un cas simplifié de protocole véhiculant des données de façon transparente. Pour se convaincre que la définition 7.3 est bien respectée, on peut raisonner comme suit. Soit μ l'événement mentionné dans la définition 7.3. D'après ce qui est dit ci-dessus, deux séquences σ et σ' sont équivalentes si elles ne diffèrent que par les valeurs des données reçues ou émises. Donc :

1. si μ n'est pas une émission ou réception de donnée, alors cet événement μ doit pouvoir se produire aussi bien après σ que σ', puisque l'état du protocole après ces traces ne diffère que par la valeur des données précédemment manipulées, qui n'ont, par hypothèse, aucune influence sur le fonctionnement du protocole.
2. si μ est une réception de données, alors cette réception doit pouvoir également se produire après σ', le protocole se trouvant à un stade de son fonctionnement qui autorise la réception de données.

3. si μ est une émission de données, le protocole retransmet par cet événement une valeur qu'il a reçue précédemment, sans la modifier , puisque la transmission est transparente par hypothèse. Dans ce cas, après σ' le protocole se trouve dans un état équivalent, mais où une autre valeur a été mémorisée. Donc il doit pouvoir faire une émission ne différant que par cette valeur , c'est à dire une action équivalente à μ.

Dans l'implantation, ce champ de données est manipulé en bloc : écriture de son contenu en mémoire lors de la réception, en vue d'une réémission ultérieure. Si les opérations de lecture et écriture sont correctement implantées (ce que l'on peut raisonnablement supposer), il est probable que la partition mentionnée ci-dessus détermine des actions équivalentes pour l'implantation. Les mécanismes du protocole proprement dit étant implantés séparément des opérations de manipulation de données, l'hypothèse d'uniformité faible est justifiée.

Pour établir notre modèle du LAPD, nous n'avons considéré que la partie de la spécification informelle concernant le transfert de données (la partie chargée de l'«affectation des TEI» étant vue comme une fonction indépendante). Le champ adresse des trames décrites sur la figure 71 n'a aucune influence sur cette partie du protocole (il n'y est même pas analysé). Donc, là encore, nous pouvons regrouper toutes les trames qui ne diffèrent que par leur champ adresse dans une même classe d'équivalence.

En conséquence, dans l'automate quotient finalement obtenu après ces divers regroupements, on trouve les trames valides suivantes :

1. les trames I modulo la partie information et la partie adresse : $2^7 = 128$ valeurs différentes.
2. les trames U et S modulo la partie adresse : 64 valeurs différentes.

Nous nous sommes donc ramenés à des chiffres presque raisonnables. Partant d'un automate à $2^{134 \times 8}$ interactions, nous obtenons en fin de compte un automate à $128 + 64 + 5$ interactions. Pour ce faire, nous avons utilisé notre connaissance de la structure de la spécification, et formulé des hypothèses raisonnables sur la façon dont se comporte l'implantation sous test. Nous montrerons au chapitre suivant comment on peut encore aller plus loin en exploitant une autre forme d'hypothèse d'uniformité.

Comme dans le cas des interactions invalides, les techniques de description formelle permettent de décrire directement l'automate quotient. Par exemple, on peut tout simplement choisir de ne pas modéliser en Estelle les données transmises de façon transparente. C'est ce que nous avons fait pour notre spécification du LAPD, où les champs adresse et données ne sont pas modélisés. On obtient ainsi une spécification plus abstraite (et moins complète) que la spécification détaillée, suffisante cependant pour certains objectifs. Cette approche a été expliquée et utilisée à des fins de vérification dans [Phalippou 88]. Ici, pour la définition des tests, une telle modélisation évite d'avoir à manipuler l'automate complet très gros et d'avoir à calculer explicitement le quotient.

Enfin, contrairement à ce que pourraient faire croire les exemples cités jusqu'ici, l'hypothèse d'uniformité forte n'a pas le monopole de l'utilisation pratique. Nous avons étudié jusqu'ici le LAPD, qui est un protocole de basse couche (du modèle OSI), dont la finalité principale est le fonctionnement correct de l'automate : d'où l'importance de la structure de celui-ci. Mais comme nous l'avons déjà mentionné, dans le cas des protocoles de couches hautes (application par exemple), ce sont au contraire les fonctions de bout en bout offertes à l'utilisateur qui sont primordiales. Parfois, comme dans le cas du protocole de messagerie P1 [CCITT 88d], ces fonctions s'expriment comme un simple calcul séquentiel : réception d'une interaction, puis émission du résultat calculé. Dans un tel cas, la structure de l'automate a peu d'importance; c'est donc l'hypothèse d'uniformité faible que nous utiliserons pour nous abstraire des données véhiculées de façon transparente par ce protocole, champ de données, qui, soit dit en passant, peut contenir jusqu'à 100 M octets, autorisant donc par le biais de la congruence une réduction de l'ensemble des événements par un facteur $2^{1000000000}$ environ.

CHAPITRE 8

Hypothèse d'uniformité : automates étendus

1 Introduction

Le test par partitionnement, étudié au chapitre précédent, met à profit la connaissance d'une *structure* sur la spécification du produit à tester : une congruence sur cet automate. En supposant que cette structure est correctement reproduite dans l'implantation sous test, on réduit fortement l'effort de test nécessaire. La principale difficulté pour mettre en œuvre cette méthode consiste à trouver la fameuse congruence. Hormis pour le traitement des événements invalides ou des données, les systèmes n'ont pas l'habitude de se répéter , chaque interaction a un rôle bien précis dans l'accomplissement du grand œuvre, et ne se laisse donc pas embrigader dans une quelconque classe d'équivalence. Pour ne rien nous épargner , même dans les cas où des congruences existent, elles sautent rarement aux yeux, pourtant experts, du spécifieur de protocoles, et n'apparaissent donc pas de façon explicite dans la spécification.

Nous étudions dans ce chapitre un autre type de structures sur les automates, que nous appelons automates à états finis étendus. Ces structures sont plus générales que les congruences, car elles peuvent être (presque) arbitraires, et permettent de décrire des situations ou des morceaux de comportement d'un automate sont similaires, mais sans exiger l'improbable régularité d'une congruence parfaite. Et en effet, nous verrons que beaucoup de systèmes et protocoles réels peuvent être facilement structurés au moyen d'automates étendus.

En adoptant une démarche similaire à celle du chapitre précédent, nous montrons comment mettre à profit la connaissance de cette structure pour sélectionner un nombre raisonnable de tests. Nous introduisons deux nouvelles hypothèses de test directement applicables aux automates étendus, les hypothèses d'uniformité étendue forte et faible, très proches de leurs quasi-homonymes déjà étudiées. Nous les appliquons respectivement au test de programmes séquentiels et à celui de la partie valide des protocoles.

2 Automates à états finis étendus

2.1 Définition

Dans la même optique que pour les congruences, nous définissons ici une structure sur les automates à entrées et sorties. Comme nous allons le voir , une telle structure est une information supplémentaire, indépendante, qui ne saurait se substituer à l'automate lui-même : on ne peut pas en dériver l'automate, pas plus que la donnée de l'automate ne détermine une structure d'automate étendu unique.

Avant même d'aller plus loin, justifions le terme employé, qui ne manquera pas de soulever des objections. En effet, on parle beaucoup des «automates étendus» dans le petit monde des spécifieurs formels, et bien qu'à ma connaissance ce terme ne soit pas unanimement défini (chacun ayant sa version précise de la chose), il désigne en général un modèle mathématique d'automates plus ou moins enrichis par des données, c'est à dire des variables et des opérations manipulant celles-ci.

Dans cette thèse, nous nous intéressons uniquement à l'aspect structurant de tels automates. Par une méthode classique appelée *analyse d'accessibilité* , il est possible de construire un automate d'états finis qui développe le comportement d'un automate étendu : celui-ci est donc une représentation compacte d'un gros automate fini sous-jacent. Soulagés de reconnaître des objets aussi familiers que les automates finis, des chercheurs ont proposé d'utiliser , à partir de ce développement des automates étendus, les méthodes classiques de génération de tests, présentées au chapitre 5.

Ce faisant, l'information de structuration est perdue, dissoute dans la masse uniforme des transitions et des états élémentaires. Or de notre point de vue (et pas seulement du notre, d'ailleurs [Castanet 86]) cette information est capitale pour la génération de tests: c'est elle, comme nous le verrons dans ce chapitre, qui donne la clé d'une sélection des tests efficace.

Nous avons donc choisi d'inverser le point de vue ci-dessus, afin de souligner l'aspect primordial de la structure. Nous partons de l'automate fini, celui des états et des événements élémentaires. Il est représenté par un automate à entrées et sorties, et constitue la spécification du système à tester . Nous définissons alors un automate étendu comme une pure structure sur cet automate. Puisque nous connaissons déjà le fonctionnement détaillé de l'automate développé, nous n'avons pas besoin de rajouter dans l'automate étendu l'information nécessaire pour le calculer .

Nous obtenons donc une définition inhabituelle des automates étendus. Cependant, nous avons choisi de garder cette appellation pour bien montrer que nous parlons du concept habituel, même si c'est avec un point de vue différent. Sur ce, place aux formules.

Définition 8.1 : soit $S = \langle S_s, L, T, s_{0s} \rangle$ une IOSM. On appelle *transition étendue* sur S un sous-ensemble non vide $t \subseteq T$. On appelle *automate à états finis étendu* (EFSM pour

Extended Finite State Machine) sur S un ensemble fini de transitions étendues sur S : $X = \{t_i\}_{1 \le i \le n}$.

Chaque transition étendue de l'EFSM X regroupe des transitions de l'automate S. C'est donc une structure qui factorise une partie du comportement de l'automate S. La partie du comportement de S représentée par l'EFSM est appelée son expansion :

Définition 8.2 : soit $S = \langle S_s, L, T, s_{0s} \rangle$ une IOSM et $X = \{t_i\}_{1 \le i \le n}$ une EFSM sur S. L'*expansion* de l'EFSM X est le sous-ensemble de T : $E(X) = \bigcup_{1 \le i \le n} t_i$.

Il peut y avoir recouvrement dans les parties factorisées par des transitions étendues différentes, c'est à dire qu'une même transition élémentaire peut appartenir aux expansions de plusieurs transitions étendues. Une attention particulière est portée aux EFSM qui structurent la totalité du comportement de S.

Définition 8.3 : une EFSM X sur une IOSM $S = \langle S_s, L, T, s_{0s} \rangle$ est *complète* si $T = E(X)$.

La relation de transition d'un automate à entrées et sorties met en correspondance des couples (état,événement) avec l'état d'arrivée de la transition correspondante (ou les états d'arrivée en cas de non-déterminisme au sens des automates). Chaque transition étendue est une relation partielle. Nous aurons besoin, dans la suite de ce chapitre, de porter attention à la façon dont ces relations partielles mettent en correspondance des ensembles d'états ou d'événements. D'où les définitions suivantes :

Définition 8.4 : soit $S = \langle S_s, L, T, s_{0s} \rangle$ une IOSM et $t \subseteq T$ une transition étendue.

1. soit $D \subseteq S_s \times (\{!, ?\} \times L)$. L'*image* du domaine D par la transition étendue t est le sous-ensemble de S_s défini par :

$$im(D, t) = \{s' \in S_s \mid (\exists (s, \mu) \in D)\ ((s, \mu, s') \in t)\}$$

2. soit $S_1 \subseteq S_s$. L'*image réciproque* du domaine S_1 par la transition étendue t est le sous-ensemble de $S_s \times (\{!, ?\} \times L)$ défini par :

$$im^{-1}(S_1, t) = \{(s, \mu) \in S_s \times (\{!, ?\} \times L) \mid (\exists s' \in S_1)\ ((s, \mu, s') \in t)\}$$

Nous noterons en abrégé $im(t) = im(S_s \times (\{!, ?\} \times L), t)$ et $im^{-1}(t) = im^{-1}(S_s, t)$.

2.2 Exemples

L'intérêt des automates étendus réside dans le fait qu'il est très facile, avec un peu d'imagination, d'en découvrir partout, ce qui n'est pas étonnant, car leur définition est aussi simple que générale. En particulier les techniques de description formelles comme Estelle ou LDS ou les langages de programmation structurés comme Pascal ou C définissent implicitement des automates étendus sur les automates à états finis qu'ils spécifient.

Considérons l'exemple présenté sur la figure 72. À une spécification écrite en Estelle, nous faisons correspondre un automate d'états finis, et une structure d'EFSM sur cet automate fini (le lecteur non familier d'Estelle peut se reporter à la norme [ISO 89a]).

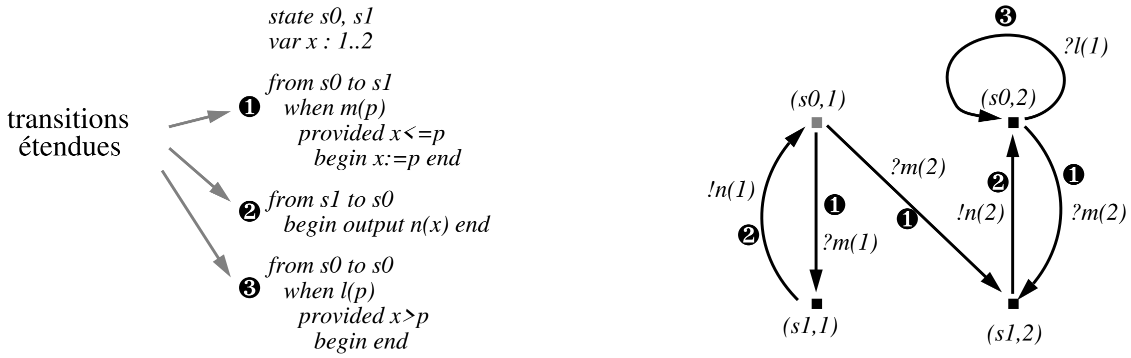


Figure 72 : description en Estelle et automate étendu

L'automate fini est obtenu par la construction du graphe d'accessibilité à partir de la description en Estelle. Cet automate constitue, au sens traditionnel, la *sémantique* du programme Estelle, c'est à dire la signification du programme. L'automate étendu est obtenu quant à lui à partir de la *syntaxe* du programme Estelle : à chaque transition (au sens Estelle) correspond une transition étendue, qui regroupe des transitions de l'automate fini sous-jacent, comme indiqué sur la figure. C'est précisément parce qu'il exploite la structure des transitions étendues pour sélectionner les tests que l'outil TVEDA est réputé «basé sur un principe syntaxique» (voir le chapitre 10). En fait, comme nous allons le voir dans ce chapitre, cette formulation est trompeuse : si les hypothèses de tests, donc la sélection des tests, sont issues de la structure d'automate étendu, les tests proprement dits sont calculés à partir de l'automate fini.

Bien sûr, le cas présenté ci-dessus n'est qu'un exemple. Dans le cas général, la traduction d'Estelle vers les automates finis et étendus n'est pas toujours aussi simple. Nous étudierons au chapitre 10 comment elle est réalisée dans TVEDA.

La figure 73 montre comment on peut faire correspondre à un programme séquentiel (ici en Pascal) un automate fini et une structure d'automate étendu. Les instructions de lecture sont modélisées par des entrées de l'automate fini, les instructions d'écriture par des sorties, et les affectations par des transitions internes τ, non visibles de l'environnement, qui représentent les calculs internes de l'automate. Les transitions étendues correspondent aux instructions du langage qui manipulent les données (lectures, écritures, affectations). On notera que l'automate fini obtenu est déterministe (au sens du test) et que les instructions de décision ne correspondent pas à des transitions; par contre, elles sont bien sûr utilisées pour calculer la structure de l'automate. Là encore,

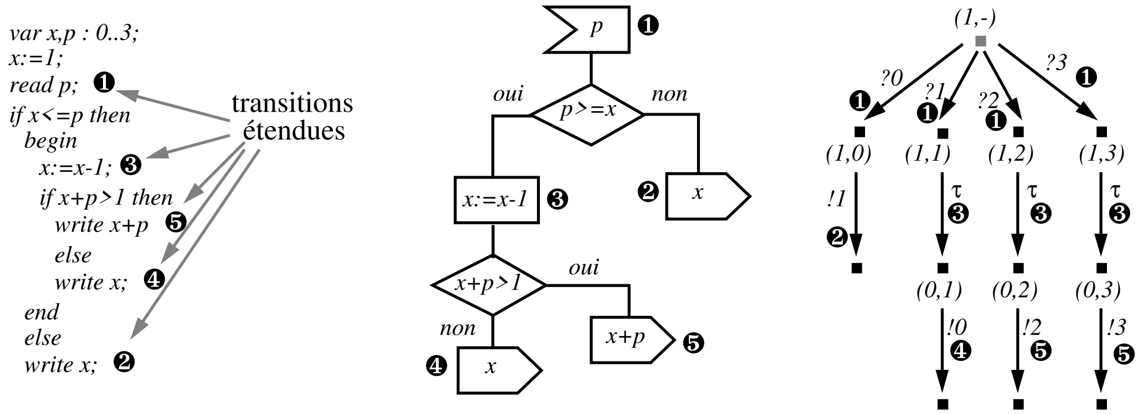


Figure 73 : description en Pascal, organigramme et automate étendu

l'automate fini représente la sémantique du programme, tandis que l'automate étendu reflète sa syntaxe.

Il apparaît donc qu'un large spectre de systèmes, aussi bien ceux décrits par des spécifications en Estelle ou LDS (traditionnellement les systèmes communicants) que ceux décrits par des programmes Pascal ou C (les systèmes dits séquentiels) se prêtent bien à une structuration sous la forme d'un automate étendu. Au cours de ce chapitre, nous examinerons deux exemples, qui correspondent respectivement à ces deux types de systèmes : le protocole LAPD (déjà étudié au chapitre précédent) et le protocole de messagerie P1 [CCITT 88d].

On peut même aller plus loin, et reconnaître des automates étendus dans la structure syntaxique d'un document en langage naturel. Il ne faut pas oublier que c'est sous cette forme que se trouvent la plupart des normes internationales ou des spécifications émises par France Télécom (les descriptions en langages formels, malheureusement, sont encore l'exception). Ces documents sont généralement structurés en une hiérarchie de sections qui décrivent les différents mécanismes ou les différentes fonctions faisant partie de la spécification. La traduction automatique d'une spécification en langage naturel vers un automate fini relevant encore du domaine de la science-fiction, nous supposerons donné l'automate fini implantant cette spécification (codé manuellement par un expert par exemple). Il est alors facile de faire correspondre à chaque division élémentaire de la spécification une ou plusieurs transitions de l'automate fini. Cette structure est donc bien une EFSM. On notera, sur l'exemple présenté par la figure 74, qu'une même transition élémentaire peut appartenir à plusieurs transitions étendues.

2.3 Automates de traces et transitions étendues

Comme on peut le constater sur l'exemple présenté par la figure 73, dans le cas où la structure d'automate étendu représente la structure syntaxique d'un programme séquentiel, les instructions du programme qui ne sont ni des entrées ni des sorties correspondent à des transitions élémentaires étiquetées par l'action interne τ. Or d'un point de vue du test, le chapitre 2 de cette thèse nous a appris que les automates finis

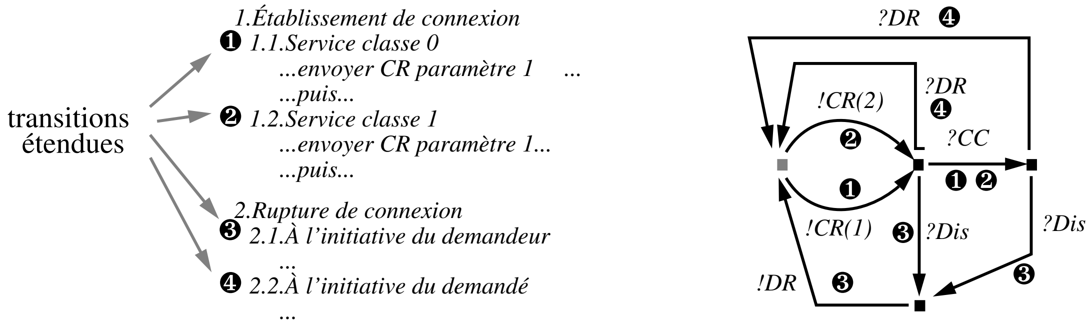


Figure 74 : description structurée en langage naturel et automate étendu

pouvaient être distingués jusqu’à l’équivalence de traces. C’est pourquoi jusqu’ici nous avons défini les tests à partir des automates de traces des spécifications. Mais si nous prenons l’automate de traces de l’automate représentant un programme séquentiel, nous perdons l’information de structuration issue de toutes les transitions étendues qui ne sont ni des entrées ni des sorties. Or cette structuration est généralement mise à profit pour choisir les tests des programmes séquentiels, comme nous allons l’expliquer dans la section 3.

Nous définissons donc ci-dessous une procédure permettant de préserver l’information de structure (l’automate étendu) lors du passage à l’automate de traces : à partir d’un automate étendu sur un automate quelconque nous construisons un automate étendu sur son automate de traces. La procédure de définition de l’automate de traces (définition 2.8 du chapitre 2) construit celui-ci en regroupant des transitions de l’automate initial. L ’idée mise en œuvre ci-dessous est qu’une transition de l’automate de traces appartient à toutes les transitions étendues auxquelles appartiennent ses composantes.

Soit $S = \langle S_s, L, T, s_{0s} \rangle$ une IOSM. Par définition on note (s, ε, s') s’il existe une séquence de transitions internes reliant s à s', c’est à dire des états $s_1 \ldots s_n$ tels que $(s, \tau, s_1) \ldots (s_n, \tau, s')$. A priori il peut exister m séquences de transitions internes reliant deux états donnés, passant respectivement par les états $s_{1i} \ldots s_{ni}$ pour $1 \le i \le m$. Nous notons :

$$trans(s, \varepsilon, s') = \bigcup_{1 \le i \le m} \{ (s, \tau, s_{1i}), \ldots, (s_{ni}, \tau, s') \}$$

l’ensemble des transitions élémentaires correspondant à ces différents chemins. De même, puisque $(s, \vec{\mu}, s')$ si il existe s_1 et s_2 tels que (s, ε, s_1), (s_1, μ, s_2) et (s_2, ε, s'), on note :

$$trans(s, \vec{\mu}, s') = trans(s, \varepsilon, s_1) \cup \{ (s_1, \mu, s_2) \} \cup trans(s_2, \varepsilon, s')$$

D’après la définition 2.8, un état de l’automate de traces est un ensemble d’états de l’automate initial. On étend donc la définition de $trans$ aux transitions de l’automate de traces de la façon suivante :

$$trans(s, \mu, s') = \bigcup_{\substack{s_i \in s \\ s_j \in s'}} trans(s_i, \vec{\mu}, s_j)$$

La fonction $trans$ permet de définir l'ensemble des transitions de l'automate initial qui correspondent à une transition de l'automate de traces. Elle sert à construire un automate étendu sur cet automate de traces. Soit $X = \{t_i\}_{1 \le i \le n}$ une EFSM sur l'automate S. Nous définissons une EFSM $X' = \{t'_i\}_{1 \le i \le n}$ sur l'automate $TM(S)$ par :

$$(s, \mu, s') \in t'_i \text{ ssi } (\exists t \in trans(s, \mu, s'))\ (t \in t_i)$$

Remarquons qu'avec cette définition, toutes les transitions étendues de l'automate initial se retrouvent sur l'automate de traces, sauf celles comportant exclusivement des transitions internes appartenant à $trans(s_{0s}, \varepsilon, s)$ puisque ces transitions n'apparaissent pas dans la définition des transitions de l'automate de traces. La figure 75 illustre la façon dont la structure d'automate étendu est transformée par le passage à l'automate de traces.

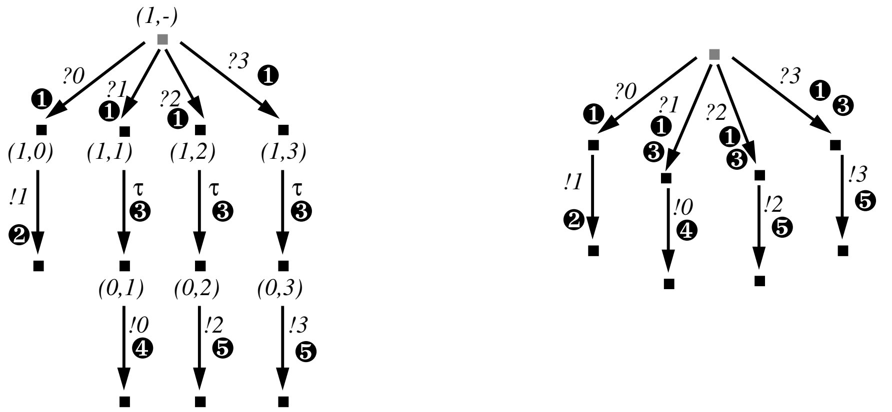


Figure 75 : automate de traces et automate étendu

Comme nous l'avons vu, pour un programme séquentiel, une transition étendue correspond à une instruction du programme. L'automate de traces d'un programme séquentiel déterministe est toujours semblable à ce qui est représenté sur la figure 75 : à partir de l'état initial, des transitions correspondant à toutes les entrées acceptées par le programme, suivies chacune de la transition étiquetée par la sortie correspondante (le résultat du calcul pour l'entrée données). La structure d'automate étendu sur l'automate de traces indique, pour chaque entrée, quelles sont les instructions qui sont exécutées au cours du calcul correspondant à cette entrée.

Dans la suite de ce chapitre nous supposons que tous les automates manipulés sont des automates de traces, sur lesquels une structure d'EFSM a été définie en appliquant éventuellement la transformation ci-dessus.

3 Uniformité étendue forte

Nous étudions maintenant comment tirer parti de la structure d'automate étendu afin de sélectionner des tests. Comme les congruences étudiées au chapitre précédent, les automates étendus regroupent des parties du comportement de la spécification à tester. Les techniques de sélection des tests correspondantes vont donc, ici aussi, être des variations sur le thème du test par partitionnement, mettant à profit cette structure.

3.1 Instanciation de transitions étendues

Les relations d'implantation que nous utilisons dans cette thèse sont basées sur les traces observables des automates à entrées et sorties. Au chapitre précédent nous avons étudié comment tirer parti des traces de l'automate quotient pour définir des tests, et comment nous pouvions observer ces traces d'action quotient à travers des instanciations. Dans cette section, nous définissons des concepts semblables pour les automates étendus : les instanciations de transitions étendues. Soit $S = \langle S_s, L, T, s_{0s} \rangle$ une IOSM et $t = (s, \mu, s') \in T$ une transition. On note $dep(t) = s$ l'état de départ de la transition et $arr(t) = s'$ l'état d'arrivée de la transition.

Définition 8.5 : soit $t_e \subseteq T$ une transition étendue. Une *instanciation* de t_e est une séquence de transitions élémentaires consécutives appartenant à t_e : $t_1 \ldots t_n$ telles que pour tout $i \in [1, n]$, $t_i \in t_e$ et pour $i \in [1, n-1]$, $arr(t_i) = dep(t_{i+1})$. Une instanciation est *maximale* si ses transitions sont toutes différentes et s'il n'existe aucune autre instanciation de t_e avec des transitions toutes différentes dont elle soit une sous-séquence propre.

On notera qu'une transition étendue peut regrouper des transitions s'exécutant en alternative (par exemple c'est le cas de la transition étendue ❶ de la figure 72), mais aussi des transitions pouvant s'exécuter consécutivement (c'est le cas de la transition étendue ❶ de la figure 74). C'est à cause de ce deuxième cas de figure que nous sommes conduits à considérer qu'une instanciation est composée d'un enchaînement de transitions élémentaires, et non pas d'une seule de ces transitions. Dans ce cas, seule une instanciation maximale peut prétendre représenter un exemple complet du comportement regroupé par la transition étendue : c'est pourquoi nous utilisons ci-dessous des instanciations maximales. Remarquons que comme par hypothèse une transition étendue est un ensemble non vide de transitions élémentaires, toute transition étendue admet au moins une instanciation. De plus, comme le nombre de transitions élémentaires est fini, toute transition étendue admet au moins une instanciation maximale.

3.2 Hypothèse d'uniformité étendue forte

Comme au chapitre 7, nous faisons un test par partitionnement en utilisant les sous-domaines du comportement de l'automate définis par les transitions étendues : nous allons tester une instanciation de chacune des transitions étendues de l'EFSM. Cette démarche ne cherchant pas du tout à vérifier la *structure* présente sur l'implantation, nous obtenons une hypothèse d'uniformité forte, dans la veine de celle déjà étudiée au chapitre précédent.

Soit $S = \langle S_s, L, T, s_{0s} \rangle$ une IOSM et $X = \{t_i\}_{1 \le i \le n}$ une EFSM sur l'automate S. Pour une transition étendue t_i on se donne une instanciation maximale $t_{i1} \ldots t_{in}$. Soit $s_i = dep(t_{i1})$. Comme tout état d'un automate à entrées et sorties est accessible à partir de l'état initial, il existe une séquence π_i d'événements telle que (s_{0S}, π_i, s_i) : cette séquence, qui permet d'amener l'automate dans l'état de départ de l'instanciation de la transition étendue t_i, est appelée le *préambule* menant à cet état. Il peut exister plusieurs préambules menant à un état; mais dès lors que les automates considérés sont des automates de traces (donc déterministes au sens des automates), si l'on exécute l'un de ces préambules, on est sûr que l'on a bien atteint l'état visé, car l'état de l'automate atteint après une séquence donnée est unique. Notons σ_i la séquence d'événements étiquetant les transitions $t_{i1} \ldots t_{in}$ de l'instanciation. Nous dirons que l'ensemble de séquences

$$\bigcup_{t_i \in X} \{\pi_i \sigma_i\}$$

est un ensemble de séquences *couvrant les transitions étendues* de l'automate étendu X. Notons que la manière ci-dessus de construire cet ensemble est systématique mais n'est pas optimale. En effet, certaines séquences de l'ensemble peuvent être des préfixes d'autres séquences. Or , ce qui importe, pour formuler une hypothèse d'uniformité, c'est de passer par toutes les transitions étendues au moins une fois : dans cette optique, nous nous autoriserons à optimiser l'ensemble obtenu ci-dessus en supprimant les séquences qui sont des préfixes d'autres séquences.

Cet ensemble joue ici le rôle d'échantillon du comportement de l'automate fini couvrant la structure définie par l'automate étendu. On l'utilise donc naturellement pour formuler l'hypothèse d'uniformité :

Définition 8.6 : soit $S = \langle S_s, L, T, s_{0s} \rangle$ une IOSM, $X = \{t_i\}_{1 \le i \le n}$ une EFSM sur l'automate S et Σ un ensemble de séquences couvrant les transitions étendues de X. Nous faisons une *hypothèse d'uniformité étendue forte* (notée UEF) sur une IOSM I en supposant que $R_\Sigma(I, S) \Rightarrow R_5(I, S)$.

Remarquons que cette formulation est très générale, en particulier elle n'oblige pas l'automate étendu à être complet, i.e. à structurer la totalité de l'automate fini sous-jacent. Cependant, dans tous les cas que nous étudierons en exemple, et dans la plupart des applications pratiques, on considère toujours des automates étendus complets.

3.3 Applications

L'hypothèse d'uniformité étendue forte modélise bien ce qui est fait en pratique pour le test de programmes séquentiels ou dans le cas de tests fonctionnels conçus pour les protocoles ou services de la couche application.

Dans le cas de programmes séquentiels, le calcul de l'automate de traces à partir de l'automate étendu identifie les valeurs d'entrée du programme qui permettent de passer par les différentes instructions. L'hypothèse d'uniformité forte correspond donc à un cas particulier classique de test par partitionnement, celui qui consiste à passer par toutes les instructions du programme (appelé *«all nodes* » dans [Rapps 85]). On s'aperçoit donc que la théorie présentée ici permet d'interpréter de façon unifiée les approches classiques de test de programmes séquentiels par partitionnement avec les méthodes usuelles du monde des automates.

Reprenons l'exemple présenté sur la figure 73. L'automate de traces de l'automate étendu dérivé du programme Pascal est représenté sur la figure 75. L'ensemble

$$\Sigma = \{?0!1, ?1!0, ?2!2\}$$

est un ensemble de séquences couvrant les transitions étendues de l'EFSM, optimisé comme indiqué à la section précédente (la séquence ?0!1 couvre les transitions étendues ❶ et ❷, la séquence ?1!0 les transitions étendues ❶, ❸ et ❹, et la séquence ?2!2 les transitions étendues ❶, ❸ et ❺). Cet ensemble sert donc d'ensemble de données de test, modulo une hypothèse d'uniformité étendue forte. Il permet effectivement de passer par toutes les instructions du programme.

Il existe plusieurs autres critères de partitionnement pour le test de programmes séquentiels (voir toujours [Rapps 85]) : du plus complet (*«all paths»* , i.e. passer par tous les chemins possibles dans le graphe représentant la syntaxe du programme, c'est à dire le graphe des transitions étendues), au moins complet (toutes les instructions, celui utilisé ci-dessus) en passant par des stratégies intermédiaires explorant les mille et une façons dont les valeurs sont associées aux variables, et la manière dont ces associations peuvent affecter l'exécution du programme. Dans l'article cité ci-dessus, on distingue trois catégories d'occurrences de variables :

1. les définitions (c'est à dire les entrées, qui affectent des valeurs aux variables pour la première fois).
2. les utilisations de calcul (c'est à dire les affectations et les sorties) qui peuvent modifier la valeur des variables.
3. les utilisations dans des prédicats (branchements conditionnels par exemple) qui utilisent la valeur des variables mais sans modifier celle-ci.

Il est alors possible de définir tout un ensemble de critères de sélection des chemins à tester , selon que l'on veut passer par toutes les définitions, toutes les utilisations dans des prédicats, toutes les utilisations de calcul, ou des versions combinées de ces catégories. Bien que l'article cité ne donne pas de précisions quant à la pertinence de ces différents critères en pratique (à-t-on vraiment besoin de tout ça pour trouver des erreurs ?), il pourrait être intéressant d'étudier s'il est possible de les retrouver par la méthode exposée dans ce chapitre, c'est à dire en associant au programme séquentiel des automates étendus adéquats.

L'hypothèse d'uniformité étendue forte est aussi utilisée dans le cas des tests fonctionnels conçus à partir des descriptions structurées en langage naturel, comme par exemple celle représentée sur la figure 74. Dans ce cas en effet, chacune des sections élémentaires de la spécification décrit un des mécanismes du protocole ou du système considéré. Chaque transition étendue regroupe donc la partie du comportement de l'automate fini correspondant à un mécanisme complet. Par exemple, sur l'exemple proposé, une transition étendue regroupe tout ce qui concerne l'établissement de la connexion de classe 0, une autre l'établissement de la connexion de classe 1, une troisième la rupture de la liaison à l'initiative du demandeur , etc.

Une instanciation maximale d'une transition étendue permet donc d'activer de façon relativement complète un des mécanismes du système sous test (le caractère «relatif» vient du fait qu'on s'interdit de boucler plusieurs fois sur une transition élémentaire dans une instanciation maximale : sinon, il serait plus compliqué de définir de façon systématique quand arrêter le test. Donc si le mécanisme à tester comporte des boucles, on ne passe qu'une fois dans la boucle avec cette stratégie de sélection de tests). Pour l'exemple proposé, l'ensemble

$$\Sigma = \{ !CR(1)?CC?DIS!DR, !CR(2)?CC?DR \}$$

est un ensemble de séquences couvrant les transitions étendues de l'EFSM, optimisé comme indiqué à la section précédente (la séquence $!CR(1)?CC?DIS!DR$ couvre les transitions étendues ❶ et ❸, la séquence $!CR(2)?CC?DR$ les transitions étendues ❷ et ❹. Cet ensemble sert donc d'ensemble de données de test, modulo une hypothèse d'uniformité étendue forte. Il permet effectivement d'activer les quatre mécanismes décrits dans la spécification.

En pratique, une telle stratégie de sélection des tests est souvent utilisée pour concevoir les suites de tests des protocoles de la couche application ou des systèmes offrant des services aux utilisateurs [Boullier 93]. Pour de tels systèmes en effet, ce sont les fonctions offertes aux usagers qui importent et qui doivent être systématiquement essayées.

4 Uniformité étendue faible

Sous l'hypothèse d'uniformité étendue forte, toutes les transitions étendues sont essayées, mais rien n'est fait pour vérifier une quelconque structure sur l'implantation. Cela s'avère parfois insuffisant pour certains types de systèmes. Par exemple, reprenons la spécification représentée sur la figure 72 et considérons l'implantation suivante :

L'implantation est complétée par des transitions ne modifiant pas l'état de l'automate, qui ne sont pas représentées sur la figure pour des raisons de lisibilité. Cette implantation n'est pas conforme au sens de la relation d'implantation R_5 : en effet $n(2) \in O(?m(1)!n(1)?m(2), S)$ et $n(2) \notin O(?m(1)!n(1)?m(2), I)$. Pourtant, si l'on considère l'ensemble de séquences

$$\Sigma = \{ ?m(2)!n(2)?l(1) \}$$

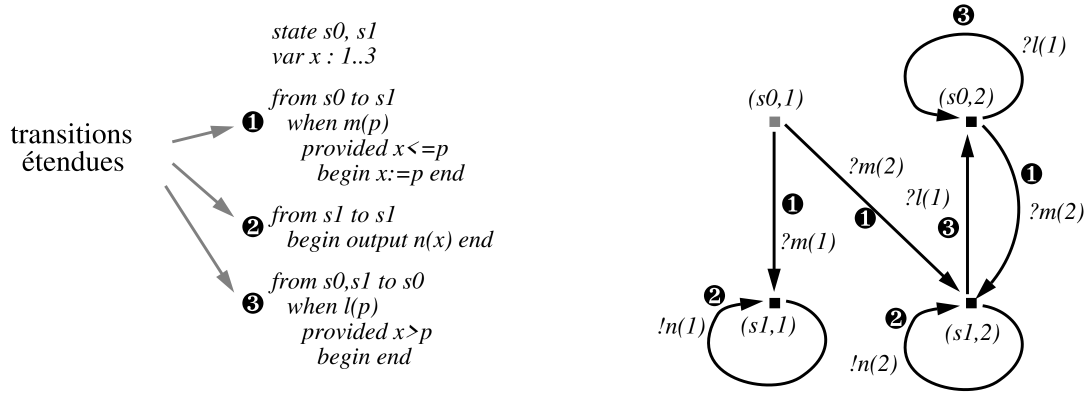


Figure 76 : structure d'automate étendu incorrecte

on a affaire à un ensemble de séquences couvrant les transitions de l'automate étendu et on a bien $R_{\Sigma}(I, S)$. Pour ce contre-exemple, l'implantation correspond à la description Estelle de la figure 72 : elle résulte d'une mauvaise structure d'automate étendu (l'état d'arrivée de la transition Estelle ❷ est incorrect, ainsi que l'état de départ de la transition Estelle ❸). Ceci souligne donc l'importance, dans certains cas, de vérifier une certaine structure sur l'implantation.

4.1 Approximations supérieures

Contrairement à ce que nous avons fait au chapitre précédent avec les automates quotients, il n'est pas facile dans le cas des automates étendus d'exhiber un automate fini qui représente exactement la structure de la spécification. Si nous voulons malgré tout tenter de vérifier une certaine conformité de structure entre la spécification et l'implantation, nous devons représenter l'automate étendu de façon *approchée* seulement par un automate fini qui modélise *imparfaitement* cette structure.

Nous considérons des automates à entrées et sorties ayant mêmes ensembles d'interactions et mêmes états initiaux : $S_1 = \langle S_{S1}, L, T_1, s_0 \rangle$ et $S_2 = \langle S_{S2}, L, T_2, s_0 \rangle$. Nous définissons une relation sur ces automates par :

$$S_1 \subseteq S_2 \text{ ssi } T_1 \subseteq T_2 \wedge S_{S1} \subseteq S_{S2}$$

Cette relation est une relation d'ordre sur les automates puisque l'inclusion est une relation d'ordre sur les ensembles. Elle nous permet de parler d'automates «plus grands», c'est à dire d'automates dont le comportement englobe celui d'un automate donné. À l'aide de tels automates plus grands nous allons définir ce qu'est l'approximation supérieure d'un automate étendu sur une IOSM : il s'agit d'une structure de congruence sur un automate plus grand que cette IOSM initiale qui soit d'une certaine façon compatible avec la structure définie par l'automate étendu :

Définition 8.7 : soit S une IOSM et $X = \{t_i\}_{1 \le i \le n}$ une EFSM sur l'automate S. Une IOSM S' munie d'une congruence $(\approx_S, \approx_L)$ est une *approximation supérieure* de l'automate étendu X si :

1. $S \subseteq S'$
2. $(\forall t_i \in X)\ (\exists (s_q, \mu_q, s'_q) \in T_q)\ (\forall t \in t_i)\ (dep(t) \in s_q \wedge arr(t) \in s'_q \wedge lab(t) \in \mu_q)$ (dans cette formule T_q désigne l'ensemble des transitions de l'automate quotient $S' / (\approx_S, \approx_L)$ et $lab(s, \mu, s') = \mu$ désigne l'événement étiquetant une transition).

La condition 2 ci-dessus signifie que toute transition étendue t_i est contenue dans une transition de l'automate quotient, en ce sens que toutes les transitions élémentaires appartenant à t_i ont leur quotient dans la même transition quotient. Par contre, d'autres transitions élémentaires de l'automate «plus grand» S' ont également leur quotient dans cette transition quotient : c'est en cela que la congruence sur un automate plus grand est une approximation supérieure de l'automate étendu sur l'automate plus petit S. Nous autorisons l'automate S' à avoir des états que n'a pas S (et non pas seulement des transitions en plus), car dans la pratique c'est ce à quoi on aboutit (voir section ci-dessous).

Au vu de cette définition, il est évident que l'on ne peut pas définir d'approximation supérieure pour n'importe quel automate étendu : en particulier , si une transition étendue regroupe à la fois des émissions et des réceptions (c'est le cas pour l'exemple de la figure 74), elle ne pourra pas correspondre à une seule transition quotient qui, elle, regroupe soit des émissions, soit des réceptions. La figure 77 donne un exemple d'approximation supérieure de l'automate déjà présenté sur la figure 72, ainsi que l'automate quotient associé à cette approximation. Sur le quotient, on indique quelles transitions étendues de l'automate initial sont contenues dans une transition du quotient. Les partitions sur les ensembles sont $\{\{(s0, 1), (s0, 2)\}, \{(s1, 1), (s1, 2)\}\}$ pour les états et $\{\{m(1), m(2)\}, \{n(1), n(2)\}, \{l(1), l(2)\}\}$ pour les événements.

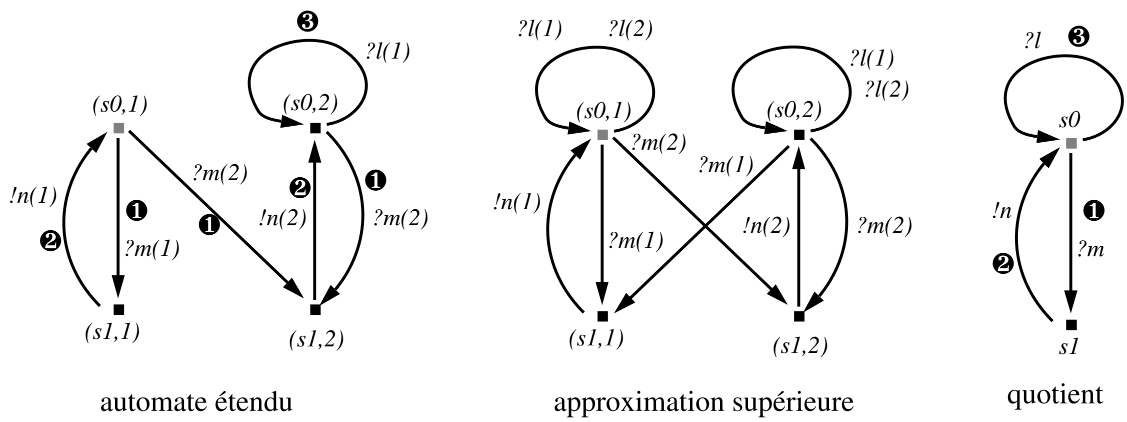


Figure 77 : approximation supérieure d'un automate étendu

4.2 Hypothèse d’uniformité étendue faible

L’approximation supérieure nous permet de représenter la structure d’automate étendu, de façon approchée, par l’automate quotient correspondant à cette approximation. Nous pouvons donc essayer de vérifier que cette structure approchée se retrouve dans l’implantation, afin d’obtenir un test plus complet que celui qui résulte de l’hypothèse d’uniformité étendue forte.

Pour cela, nous devons supposer que l’implantation possède une structure d’automate étendu (ce qui est toujours vrai, car une telle structure est totalement arbitraire), et, ce qui est moins évident, que celle-ci admet une approximation supérieure ayant les mêmes actions quotient que l’approximation de la spécification. Sous cette hypothèse, en notant I'_q et S'_q les quotients respectifs des approximations de l’implantation et de la spécification, nous formalisons la similarité des structures approchées entre spécification et implantation par $R_5(I'_q, S'_q)$. Comme au chapitre précédent, la version faible de l’hypothèse d’uniformité impose de vérifier à la fois la similarité entre les structures et d’essayer au moins une séquence concrète pour chaque transition étendue. Soit :

Définition 8.8 : soit S une IOSM, $X = \{t_i\}_{1 \le i \le n}$ une EFSM sur l’automate S et Σ un ensemble de séquences couvrant les transitions étendues de X. Soit S'_q un automate quotient correspondant à une approximation supérieure de X. Nous faisons une hypothèse d’ *uniformité étendue faible* (notée UEf) sur une IOSM I en supposant que :

1. on peut définir sur I une structure d’EFSM admettant une approximation supérieure dont le quotient I'_q a les mêmes événements que S'_q.
2. $R_5(I'_q, S'_q) \wedge R_\Sigma(I, S) \Leftrightarrow R_5(I, S)$

Pour mettre en application cette deuxième version de l’hypothèse d’uniformité étendue, nous devons vérifier , en plus de la relation d’implantation finie $R_\Sigma(I, S)$, la relation abstraite $R_5(I'_q, S'_q)$. Il s’agit encore une fois de l’application de la méthode de test abstrait, les abstractions étant ici les automates quotients des approximations supérieures. En appliquant les résultats du chapitre 5, nous savons ramener la vérification de cette dernière relation à celle d’une relation finie entre les automates abstraits : à partir de l’automate de traces $TM(S'_q)$ nous pouvons construire un ensemble Σ_q de séquences d’événements quotients tel que, pour tout automate I'_q ayant les mêmes actions que S'_q et satisfaisant les hypothèses de test $IOSM$ complète sur les entrées, RI et $R(m)$ on ait :

$$R_5(I'_q, S'_q) \Leftrightarrow R_{\Sigma_q}(I'_q, S'_q)$$

(ces relations portant sur les traces, on peut les exprimer indifféremment sur les automates initiaux ou sur leurs automates de traces !). Malheureusement, il n’est pas possible dans la pratique d’avoir accès aux traces de I', qui est une approximation supérieure de la véritable implantation I. Nous étudions dans la section suivante comment et sous quelles conditions il est possible de définir un testeur concret afin de vérifier *approximativement* cette relation d'implantation finie abstraite.

4.3 Existence d'un test concret

Au chapitre précédent nous avons établi (section 3.2) que pour vérifier une relation d'implantation finie portant sur des automates abstraits quotients nous devions être capables de vérifier l'existence ou la non existence de traces sur l'implantation réelle. En appliquant les résultats de ce chapitre au cas qui nous préoccupe maintenant, nous devons être capables de vérifier :

$$(\forall \sigma_q \in \Sigma_q \cap Tr(S'_q))\ (\sigma_q \in Tr(I'_q) \Rightarrow (O(\sigma_q, I'_q) = O(\sigma_q, S'_q)))$$

ce qui se traduit en :

$$\begin{gathered}(\forall \sigma_q \in \Sigma_q \cap Tr(S'_q))\ ((\exists \sigma \in C(\sigma_q))\ (\sigma \in Tr(I')) \Rightarrow P_1 \wedge P_2) \text{ où} \\ P_1 \text{ est } (\forall a_q \in O(\sigma_q, S'_q))\ (\exists \sigma \in C(\sigma_q ! a_q))\ (\sigma \in Tr(I')) \text{ et} \\ P_2 \text{ est } (\forall a_q \in L_q - O(\sigma_q, S'_q)) \neg (\exists \sigma \in C(\sigma_q ! a_q))\ (\sigma \in Tr(I'))\end{gathered}$$

Informellement, pour vérifier la conformité entre les automates abstraits, et à la suite de l'application de l'une des hypothèses de régularité étudiées au chapitre 5, nous devons vérifier :

1. que l'implantation I' possède un certain comportement minimal qu'elle doit satisfaire (ce qui est exprimé par la proposition P_1 ci-dessus).
2. que l'implantation I' ne dépasse pas un certain comportement : certaines traces ne doivent pas faire partie du comportement de l'implantation, ce qui est exprimé par la proposition P_2 ci-dessus).

La difficulté réside ici dans le fait que l'automate I' est seulement une abstraction de l'implantation réelle I que nous testons (une approximation supérieure). De plus, celle abstraction est introduite, de façon purement arbitraire, pour pouvoir formuler l'hypothèse d'uniformité étendue faible : elle n'a aucune existence réelle. Nous ne savons pas comment relier exactement les traces de cette approximation à celles de l'implantation disponible, la seule sur laquelle nous puissions faire des expériences de test. Tout ce que nous pouvons vérifier par le test, c'est, pour une séquences d'actions quotients σ_q :

$$(\exists \sigma \in C(\sigma_q))\ (\sigma \in Tr(I))$$

Il est possible cependant d'exploiter la méthode décrite ci-dessus de manière approchée. Tout d'abord, puisque I' est une approximation supérieure de I, c'est que $S_I \subseteq S_{I'}$ et $T_I \subseteq T_{I'}$, donc $Tr(I) \subseteq Tr(I')$. Donc si on montre au moyen d'un test que

$$(\exists \sigma \in C(\sigma_q))\ (\sigma \in Tr(I))$$

on en conclut immédiatement que

$$(\exists \sigma \in C(\sigma_q))\ (\sigma \in Tr(I'))$$

Réciproquement, si l'on suppose que l'implantation I est complète sur les entrées, alors la conformité $R_5(I, S)$ entraîne $Tr(S) \subseteq Tr(I)$ (propriété 5.4 du chapitre 5). Donc étant donné un ensemble de séquences quotient $C(\sigma_q)$, si on montre au moyen d'un test que

$$\neg(\exists \sigma \in C(\sigma_q))\ (\sigma \in Tr(I))$$

et que l'on vérifie, par un examen de l'automate (connu) S

$$(\exists \sigma \in C(\sigma_q))\ (\sigma \in Tr(S))$$

on conclut immédiatement que I n'est pas conforme à S. Enfin, nous dirons qu'une séquence d'actions quotient σ_q est *exécutable* si $C(\sigma_q) \cap Tr(S) \neq \emptyset$ (si besoin est et si c'est possible, on choisit les instanciations $i(a_q)$ intervenant dans la construction de l'ensemble $C(\sigma_q)$ pour garantir cette propriété). Notons que bien sûr toutes les séquences d'actions quotient ne sont pas exécutables en général.

Si les séquences de $\Sigma_q \cap Tr(S'_q)$ sont toutes exécutables, et si $\forall a_q \in O(\sigma_q, S'_q)$, $\sigma_q!a_q$ est exécutable, alors nous pouvons appliquer la méthode d'uniformité étendue faible de façon approchée, en utilisant la procédure suivante :

1. on vérifie que pour tout $\sigma_q \in \Sigma_q \cap Tr(S'_q)$ on a $(\exists \sigma \in C(\sigma_q))\ (\sigma \in Tr(I))$ (sinon, puisque les séquences σ_q sont exécutables on aurait $\neg R_5(I, S)$).
2. on vérifie $(\forall a_q \in O(\sigma_q, S'_q))\ (\exists \sigma \in C(\sigma_q!a_q))\ (\sigma \in Tr(I))$: si c'est vrai alors on a montré $(\forall a_q \in O(\sigma_q, S'_q))\ (\exists \sigma \in C(\sigma_q!a_q))\ (\sigma \in Tr(I'))$ d'après ce qui est dit plus haut. Si c'est faux, comme toutes les séquences $\sigma_q!a_q$ sont exécutables on conclut $\neg R_5(I, S)$.
3. on vérifie $(\forall a_q \in L_q - O(\sigma_q, S'_q))\ \neg(\exists \sigma \in C(\sigma_q!a_q))\ (\sigma \in Tr(I))$: si c'est faux alors la propriété $(\forall a_q \in L_q - O(\sigma_q, S'_q))\ \neg(\exists \sigma \in C(\sigma_q!a_q))\ (\sigma \in Tr(I'))$ est fausse, donc on conclut à la non conformité dans le cadre de notre hypothèse d'uniformité. Si c'est vrai, cela signifie que l'implantation ne dépasse pas le comportement que ne doit pas dépasser son approximation supérieure : on conclut à la conformité, sous notre hypothèse d'uniformité.

Il faut être conscient que la procédure ci-dessus n'est pas une façon de prouver la conformité entre les abstractions, requise par l'hypothèse d'uniformité faible telle qu'elle a été définie précédemment, pour la bonne raison que nous ne savons pas tester cette conformité entre abstractions. Cependant, il ne faut pas perdre de vue que nous ne faisons ici que des hypothèses de test. Il est donc tout à fait légitime de modifier la définition 8.8 pour adopter le point de vue suivant : nous faisons une hypothèse d'uniformité étendue faible en supposant que si une implantation satisfait la relation d'implantation finie $R_\Sigma(I, S)$ (de la définition 8.8) et la procédure de test décrite ci-dessus alors elle est conforme à sa spécification au sens $R_5(I, S)$.

Dans le cas où les séquences requises sont exécutables, nous avons donc défini une procédure permettant de vérifier , de façon approchée, une certaine structure sur l'implantation sous test, dérivée de la structure d'automate étendue sur la spécification. Mais les séquences issues de l'application d'une des hypothèses de régularité au quotient de l'approximation supérieure peuvent ne pas être exécutables. Cela provient du fait que nous considérons justement une approximation *supérieure* , qui contient donc plus de traces que la spécification. Ce problème est bien connu dans en pratique : nous montrerons à la section suivante qu'il correspond par exemple à celui de l'exécutabilité des séquences de tests calculées à partir de l'automate de contrôle d'une spécification Estelle [Sarikaya 87].

Dans le cas contraire, nous avons deux attitudes possibles :

1. essayer de trouver des algorithmes qui permettent de trouver des séquences de tests sous hypothèse de régularité qui satisfont à la contrainte supplémentaire d'être exécutables. Cela consiste à réfléchir au problème suivant : peut-on, sous une hypothèse de régularité $R(m)$, caractériser la conformité R_5 par une relation de conformité finie R_Σ où l'on restreint le choix de l'ensemble Σ à un sous-ensemble arbitraire de séquences d'événements ? Est-ce possible pour des types particuliers de telles restrictions ?
2. supprimer toutes les vérifications concernant des séquences non exécutables. Formuler une hypothèse d'uniformité, plus forte que notre hypothèse d'uniformité étendue faible (mais cependant plus faible que notre hypothèse d'uniformité étendue forte), qui consiste à se satisfaire des tests restants.

La première de ces attitudes ouvre un problème de recherche non résolu pour l'instant, à ma connaissance, qui fera probablement la joie des nombreuses personnes à l'affût d'améliorations potentiellement publiables des méthodes actuelles basées sur l'hypothèse de régularité $R(m)$ (voir le chapitre 5 pour les épisodes précédents de cette grande aventure). Je suis pourtant personnellement enclin à penser que la deuxième approche a plus de chances de satisfaire à moindre coût les besoins réels de conception de suites de tests : comme quoi la solution de facilité est parfois la meilleure. C'est d'ailleurs celle-ci qui est adoptée dans [Sarikaya 87].

4.4 Application

L'hypothèse d'uniformité étendue faible modélise ce qui est fait en pratique lorsque l'on essaie d'appliquer les méthodes traditionnelles de test d'automates finis (c'est à dire une hypothèse de régularité $R(m)$) à l'automate de *contrôle* d'un automate étendu. Nous basons notre exemple ci-dessous sur l'utilisation du langage Estelle. Nous obtenons l'automate de contrôle en faisant abstraction des données : nous supprimons la clause *provided* de la garde des transitions, et les manipulations de données (y compris les valeurs des paramètres des messages émis) dans le corps des transitions.

Cette transformation est illustrée sur la figure 78. On s'aperçoit que l'automate de contrôle décrit directement l'automate quotient d'une approximation supérieure de l'autospécification complète

```
state s0, s1
var x : 1..3

from s0 to s1
❶ when m(p)
    provided x<=p
      begin x:=p end

❷ from s1 to s0
    begin output n(x) end

   from s0 to s0
❸ when m(p)
    provided x>p
      begin end
```

automate de contrôle

```
state s0, s1

   from s0 to s1
❶ when m
      begin end

❷ from s1 to s0
    begin output n end

❸ from s0 to s0
    when m
      begin end
```

Figure 78 : automate de contrôle d'une description en Estelle

mate étendu déduit de la spécification Estelle, que nous avons déjà représenté sur la figure 77.

Ce résultat est général et non pas spécifique à l'exemple particulier que nous avons choisi. Considérons l'automate fini obtenu par construction du graphe d'accessibilité à partir d'une spécification Estelle intermédiaire où seules les clauses *provided* sont supprimées (nous supposons ici que les messages sont déclarés avec leurs paramètres, et que les variables locales sont toujours là). Sur cet automate, les relations d'équivalence qui regroupent d'une part tous les états élémentaires ayant même valeur d'état principal Estelle et d'autre part toutes les interactions élémentaires ayant même identificateur de message, est une congruence. En effet :

1. si deux états sont équivalents (i.e ont même état principal) et s'il existe une transition de réception à partir de l'un d'entre eux, alors la même transition existe à partir de l'autre, puisque la transition Estelle qui la décrit n'a pas de *provided*.
2. si deux états sont équivalents et s'il existe une transition d'émission à partir de l'un d'entre eux, alors il existe une transition d'émission équivalente à partir de l'autre, puisque la transition Estelle qui décrit cette émission n'a pas de *provided* : donc l'instruction *output* existe à partir du deuxième état, éventuellement avec une valeur différente de paramètre.

L'automate muni de cette congruence est une approximation supérieure de l'automate étendu initial. En effet :

1. il contient l'automate initial car la transformation rajoute des transitions et éventuellement des états (rappelons que nous ne prenons en compte que l'ensemble des états d'un automate fini accessibles à partir de l'état initial).
2. toute transition étendue de l'automate étendu (i.e. toute transition étendue de la spécification Estelle initiale) est contenue dans une transition Estelle de l'automate intermédiaire, qui correspond à une transition quotient de son graphe d'accessibilité.

L'approximation supérieure représentée sur la figure 77 est obtenue de cette façon. Le graphe de contrôle, qui est dérivé de la spécification intermédiaire en supprimant toutes les données, représente bien directement le graphe quotient. Sur ce graphe quotient, on peut vérifier que l'on est dans l'état s_0 en vérifiant qu'il n'y a pas d'émission possible après réception de $?l$. Donc une séquence destinée à tester la transition quotient ❷ est par exemple $\sigma_q = ?m!n?l$ (une telle séquence pourrait être obtenue par l'une de méthodes exploitant l'hypothèse de régularité présentées au chapitre 5). En testant $R_{\{\sigma_q\}}$ on distingue l'automate quotient de sa variante où la transition quotient ❷ boucle sur l'état s_1. Cette séquence est exécutable, puisque $\sigma = ?m(2)!n(2)?l(1)$ appartient à $Tr(S) \cap C(\sigma_q)$. En utilisant la procédure décrite à la section 4.3, on détecte que l'implantation présentée sur la figure 76 est incorrecte, car $\sigma!n(2) \in Tr(I)$ et $n \in L_q - O(\sigma_q, S'_q)$ (le point 3 est mis en défaut).

CHAPITRE 9

Hypothèse d'équité

1 Introduction

À l'issue des chapitres 5 à 8, au cours desquels nous avons étudié comment, et sous quelles hypothèses, réduire le problème du test de conformité (c'est à dire implicitement sélectionner un nombre raisonnable de tests), nous avons ramené la vérification de la conformité la plus générale à celle de certaines propriétés de l'implantation sous test, qui s'expriment sous la forme d'un nombre fini de conjonctions ou de disjonctions reliant les propriétés élémentaires $R_{\sigma}(I)$, $R_{\neg\sigma}(I)$, $R_{C(\sigma_q)}(I)$ et $R_{\neg C(\sigma_q)}(I)$. Nous aurons également besoin, au chapitre suivant, pour définir une méthode complète et applicable de génération de tests, de vérifier deux nouveaux types de propriétés élémentaires, que nous noterons $R_{CC(\sigma_{qq})}(I)$ et $R_{\neg CC(\sigma_{qq})}(I)$.

Il nous reste maintenant, pour conclure notre étude, à montrer comment on peut tester ces propriétés sur l'implantation par une procédure effective.

En effet, une analyse des conditions de mise en œuvre du test nous montre que des testeurs semblables à ceux utilisés dans les chapitres 2 à 4 ne peuvent pas être transformés facilement en algorithmes de test. Nous définissons donc dans un premier temps quels critères un testeur doit vérifier pour être *exécutable*, c'est à dire traduisible en une procédure de test.

Puis nous montrons comment construire des testeurs exécutables qui caractérisent les propriétés élémentaires mentionnées ci-dessus. Il apparaît que du fait du non déterminisme des automates à entrées et sorties, il n'est possible de tester effectivement ces propriétés que moyennant une nouvelle hypothèse de test, exprimant le fait que le non déterminisme des implantations n'est pas complètement sauvage, mais obéit à certaines règles d'équité. Pour formuler précisément cette hypothèse, nous sommes conduits à enrichir le modèle d'automates utilisé pour représenter les implantations.

Ce chapitre se termine alors par la présentation de l'algorithme de test issu de ces testeurs.

2 Analyse des problèmes à résoudre

2.1 Quels critères pour exécuter les tests ?

Dans la section 4.1 du chapitre 2 nous avons expliqué pourquoi on peut modéliser l'activité de test comme la composition parallèle de l'automate représentant l'implantation avec l'automate représentant le test. Moyennant la prise en compte de l'architecture de test, comme indiqué au chapitre 4, cette composition parallèle est un modèle réaliste de l'activité de test.

Implanter un automate à entrées et sorties ne pose pas de problème, car les ensembles d'états et d'interactions sont finis. Mais pour pouvoir utiliser un tel automate comme programme de test, il nous faut également définir une procédure opérationnelle pour déterminer le verdict d'un test.

Nous rappelons que dans notre approche l'ensemble des échanges de messages ayant lieu entre le testeur et l'implantation représente toutes les exécutions nécessaires de tous les tests d'une suite de tests. C'est à dire que le test est terminé après une séquence d'exécution de $T \| I$, séquence qui doit donc aboutir dans tous les cas à un verdict en un temps fini. Nous dirons donc :

Définition 9.1 : un testeur est *exécutable* s'il satisfait les deux conditions suivantes :

1. observation d'une *trace finie* : à partir de l'état initial du testeur, et quel que soit l'enchaînement de transitions du testeur suivi lors de son exécution en parallèle avec l'implantation, le testeur aboutira dans un état terminal après au plus N transitions, où N est un entier connu à l'avance (qui dépend uniquement du testeur).
2. émission d'un *verdict* à partir de l'observation d'*une* trace : le succès ou l'échec du test doit être déterminé comme une propriété de l'état terminal atteint par le testeur. Nous appelons ceci un verdict *local*.

Aucun de ces deux critères n'est rempli par le testeur canonique asynchrone présenté au chapitre 2. En effet, comme le montre l'exemple de la figure 79, l'exécution en parallèle du testeur et de l'implantation peut comporter des traces infinies. Dans un tel cas le test ne termine jamais (il n'y a pas de notion de temps explicite dans notre modèle d'automates, mais nous assimilons la durée d'exécution d'une séquence au nombre d'événements de cette séquence). Dans l'exemple présenté, lors du test de l'implantation 1, une suite infinie de transitions de $T \| I$ résulte de synchronisations successives sur les actions !a et ?b du testeur, sans qu'un état terminal soit atteint. Ici nous interprétons un état terminal comme un état où l'automate $T \| I$ ne peut plus faire de transitions. C'est le cas lorsque le testeur a atteint l'état $fail$. Remarquons que dans le cas ci-dessus la nécessité de séquences de test infinies s'explique par le fait que l'on cherche à vérifier

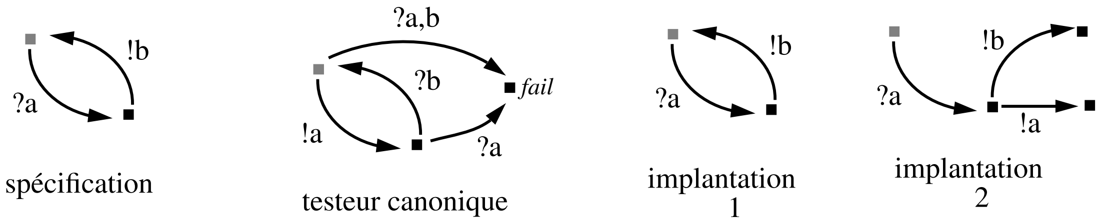


Figure 79 : testeur canonique à traces d'éxécution infinies

directement la relation d'implantation R_1, qui n'est pas une relation d'implantation finie selon la définition donnée au chapitre 5.

De plus, la définition du verdict (définition 2.19) :

$$Fail\,(T, I) \text{ ssi } (\exists\sigma \in Tr\,(T))\ (\bar{\sigma} \in Tr\,(I) \wedge (T, \sigma, fail))$$
$$Succ\,(T, I) \text{ ssi } \neg\,(Fail\,(T, I))$$

ne correspond pas à un verdict local. En effet, lorsque l'on teste l'implantation 2, qui n'est pas conforme à la spécification au sens de la relation d'implantation R_1, l'automate composé peut exécuter deux transitions résultant de synchronisations sur les actions ?a et !b de l'implantation, auquel cas le testeur aboutit à un état terminal qui n'est pas l'état $fail$, et il est impossible de conclure à la non conformité uniquement d'après l'observation de cette exécution.

Avec la définition du verdict ci-dessus, pour fournir un verdict dans le cas général il faut une connaissance globale du graphe de $T \| I$, d'où l'appellation verdict *global* mentionnée au chapitre 2. Cela est du au fait que seul l'échec peut être décidé localement, par observation d'une trace incorrecte, tandis que le succès est défini par une propriété globale, i.e. l'absence de traces incorrectes menant à $fail$ dans le graphe de $T \| I$. Les testeurs que nous définissons dans ce chapitre voient au contraire leur verdict exprimé par :

$$Fail\,(T,\ (I, X\,(I))) \text{ ssi } (\exists\sigma \in X\,(I))\ ((T, \bar{\sigma}, fail))$$
$$Succ\,(T,\ (I, X\,(I))) \text{ ssi } (\exists\sigma \in X\,(I))\ ((T, \bar{\sigma}, pass))$$

Les définitions précises seront données plus loin, mais ce qui est important ici c'est que comme nous montrerons que quelle que soit l'implantation l'automate $T \| I$ ne peut se bloquer que si le testeur est dans un des états $pass$ ou $fail$, il en résulte que l'on peut toujours associer un verdict à *une* trace d'exécution de test : il s'agit bien d'un verdict local.

Nous allons maintenant analyser les trois principaux problèmes qui se posent pour concevoir des testeurs exécutables pour les propriétés mentionnées dans l'introduction de ce chapitre.

2.2 Test de plusieurs séquences

Même s'il est possible, comme nous l'avons mentionné au chapitre 5, de ramener, sous la seule hypothèse de régularité, le test de conformité selon R_5 à celui d'une seule séquence $R_{\{\sigma\}}$, nous avons vu que pour réduire fortement la longueur totale des séquences à vérifier, nous avons plutôt intérêt à tester une relation R_Σ où l'ensemble Σ comporte plusieurs séquences. C'est également une telle situation qui résulte de l'utilisation des autres hypothèses de test (indépendance, uniformité).

En outre, indépendamment des questions de longueur, le fait de baser le test sur plusieurs séquences permet en pratique d'identifier les séquences qui ne sont pas reproduites correctement dans l'implantation, donc d'effectuer un *diagnostic* sur les fautes de l'implantation, ce qui est très important dans le cas où l'on teste du logiciel. Les méthodes de test basées sur une seule séquence sont au contraire issues du test de circuits, où l'on décide de jeter un composant à la moindre faute, ce qui rend un tel diagnostic inutile.

Dans notre cas, nous devons vérifier la présence (ou l'absence) parmi les traces de l'implantation de plusieurs séquences d'événements, toutes devant débuter dans l'état initial de l'implantation. Nous devons donc disposer d'un moyen pour ramener, lors de l'exécution du test, l'implantation dans son état initial. Une solution à ce problème est de supposer que la spécification comprend un reset, et que ce reset est bien implanté. Un reset est une interaction qui permet exclusivement de ramener l'automate dans son état initial, et ce à partir de n'importe quel état. Dans notre modèle, cela s'exprime par:

Définition 9.2 : soit $S = \langle S_S, L_S, T_S, s_{0S}\rangle$ et $I = \langle S_I, L_I, T_I, s_{0I}\rangle$ deux IOSM représentant une spécification et une implantation. La spécification a un *reset* s'il existe un événement $r \in L_S$ tel que $(\forall s \in S_S)\ ((s, ?\mathrm{r}, s_{0S}) \in T_S)$ et $(\forall (s, ?\mathrm{r}, s') \in T_S)\ (s' = s_{0S})$. Le reset est *bien implanté* si $r \in L_I$, $(\forall s \in S_I)\ ((s, ?\mathrm{r}, s_{0I}) \in T_I)$ et $(\forall (s, ?\mathrm{r}, s') \in T_I)\ (s' = s_{0I})$.

Dans la suite de ce chapitre, nous considérerons que les spécifications ont toujours un reset, mais nous ne représenterons pas, en règle générale, les transitions correspondantes sur les figures, pour ne pas alourdir les dessins. Le fait que le reset soit bien implanté est une supposition a priori, c'est à dire une hypothèse de test, faite sur les implantations. Nous noterons cette hypothèse RI. Notre expérience pratique du test nous amène à penser que l'existence d'un reset et l'hypothèse que celui-ci est bien implanté sont des suppositions réalistes dans de nombreux cas. Cependant, on peut aussi parfois avoir besoin de se passer de ces hypothèses [Yao 93]. Dans ce cas, nous sommes réduits pour l'instant à l'utilisation de la seule hypothèse de régularité et des méthodes comportant une seule séquence de test, évoquées au chapitre 5. Il est peut être possible de trouver des solutions pour utiliser les hypothèses d'indépendance et d'uniformité avec une seule séquence de test, mais ceci n'est pas étudié dans cette thèse.

2.3 Non déterminisme des implantations

Dans notre modèle, les implantations, tout comme les spécifications, peuvent avoir du non déterminisme (au sens du test, comme expliqué au chapitre 2). Il n'est pas possible de contrôler de façon externe les choix qui sont faits par l'implantation. Ainsi, on n'est jamais sûr , sans hypothèse supplémentaire, que l'on a bien observé la totalité des comportements possibles d'une implantation. Considérons l'exemple illustré par la figure 80. Quelle que soit la longueur du test effectué, il est possible que l'implantation choi-

Figure 80 : implantation non déterministe

sisse toujours d'émettre des !b, et dans ce cas on n'observera jamais de !c, donc on pourra ne pas détecter la non conformité vis à vis de la propriété $R_{\neg\{?a!c\}}(I)$ (la séquence ?a!c est une trace de l'implantation). Pour résoudre ce problème, nous devons faire l'hypothèse que l'ensemble des comportements de l'implantation est exhibé lors de l'exécution du test. Ceci est formalisé à la section 3 par l'hypothèse d'équité bornée.

Là aussi, une conséquence de la propriété d'équité bornée est que l'on peut avoir besoin de plusieurs tentatives avant de pouvoir conclure qu'une séquence fait ou non partie des traces de l'implantation. Donc se pose à nouveau le problème de savoir ramener l'implantation, de façon fiable, dans son état initial. L 'hypothèse de reset bien implanté nous fournit, dans ce chapitre, un mécanisme approprié. Savoir s'il est possible de se passer de cette hypothèse pour tester des implantations non déterministes est une question intéressante, mais que nous n'étudions pas dans cette thèse.

2.4 Détection des blocages de sortie

Enfin, comme indiqué à la section 2.1, un testeur , pour être exécutable, doit toujours fournir un verdict de test au bout d'un temps fini. Considérons la situation décrite sur la figure 81. Supposons que l'on veuille tester $R_{\{?a!b\}}$. La séquence ?a!b faisant partie des traces de la spécification, nous voulons nous assurer qu'elle fait aussi partie des traces de l'implantation. Pour cela, après avoir envoyé !a, il nous faut recevoir ?b pour conclure au succès. Si une autre interaction est reçue (?c par exemple), cela peut être dû au non déterminisme de l'implantation, donc, sous l'hypothèse d'équité bornée, il suffit de recommencer le test un nombre de fois raisonnable. Mais si rien n'est reçu, il faut bien se décider un jour à conclure, pour pouvoir différencier une implantation correcte d'une implantation incorrecte, présentant un blocage de sortie.

L'analyse ci-dessus montre que pour pouvoir définir un testeur exécutable, nous devons être en mesure de détecter les blocages de sortie des implantations. En accord avec le choix de modélisation fait au chapitre 2 (section 2.5), nous considérerons donc dans ce chapitre que les blocages de sortie sont représentés par l'action observable δ. Cette action sera traitée comme les autres pour la définition du testeur . Cependant,

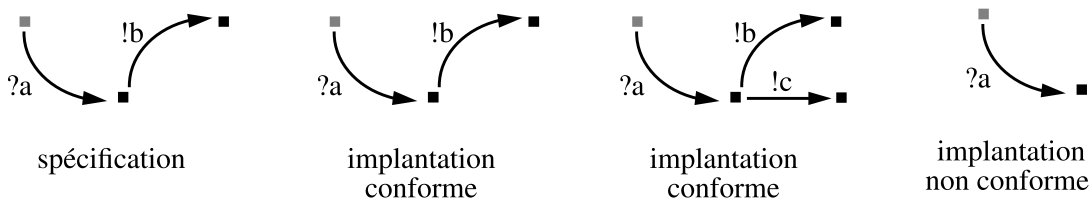


Figure 81 : blocage de sortie

elle est nécessairement implantée par un mécanisme spécial, comme nous le verrons à la section 5.

2.5 État de l'art des solutions à ces problèmes

En règle générale, la question de l'exécution des tests n'est pas traitée de manière très approfondie dans les principales méthodes connues de génération de tests. Peut être est-ce dû à un certain snobisme vis à vis d'une question souvent considérée comme relevant des problèmes d'implantation, donc indigne de la recherche. Nous nous limitons ici à passer en revue les méthodes applicables au même type de systèmes (automates) que ceux considérés dans cette thèse.

Dans l'article initial [Brinksma 88], après la présentation des propriétés du testeur canonique, les deux problèmes ci-dessus sont abordés. Le testeur canonique synchrone étant une structure comportant des choix non déterministes, il est remplacé, pour être exécuté, par une suite de tests, c'est-à dire un ensemble de tests déterministes dont le pouvoir global de test est égal à celui du testeur (corollaire 3.7 de l'article cité). Le lecteur peut supposer implicitement (bien que ce ne soit pas précisé) que tester consiste à exécuter successivement ces tests. Cela suppose, toujours implicitement, l'existence d'un mécanisme de reset, pour revenir à l'état initial après chaque test, ce reset étant supposé bien implanté car là aussi c'est $T \| I$ qui doit retrouver son aspect initial. L 'ennui c'est que si les traces du testeur sont infinies (ce qui est possible dans cette approche) et les choix assez nombreux on risque d'avoir un nombre infini de tests infinis à faire consécutivement, ce qui a peu de chance d'aboutir rapidement.

Le passage d'un test infini à un test fini, qui correspond dans cette thèse à l'hypothèse de régularité, n'est pas étudié dans cette théorie. Le prix à payer pour retrouver une suite de test finie (donc utilisable) semble être, comme nous le faisons dans cette thèse, d'affaiblir la relation à tester , par le biais d'hypothèses limitatives sur les implantations possibles.

Le problème du non-déterminisme des implantations est évoqué mais seulement de façon informelle. La conclusion est que «plus on augmente le nombre d'essais, plus la qualité des tests est grande», c'est-à dire plus l'implantation a de chances de révéler toutes les facettes de son talent.

Dans les versions plus récentes, donc plus élaborées, de cette théorie [T retmans 92b], le test est directement défini sous la forme d'une suite de tests. On suppose, sans détailler ce point, que l'on peut obtenir tous les ensembles d'observations, notés $Obs(t, I)$ que ces tests permettent sur les implantations. Sous cette hypothèse se masquent les deux problèmes évoqués plus haut : comment obtenir en pratique l'ensemble des observations correspondant à un test lorsque l'implantation est non déterministe ? Comment obtenir successivement les ensembles d'observations correspondant à tous les tests ? Ces questions ne sont pas étudiées dans la référence ci-dessus. De plus, la manière de se ramener à un nombre fini de tests, quand le comportement à vérifier est infini, n'est pas non plus étudié ici.

Les méthodes classiques de génération de tests à partir d'automates finis, que nous avons brièvement présentées au chapitre 5, abordent les deux problèmes étudiés ici. Le problème du non déterminisme des implantations est évacué dans la plupart des études relevant de cette approche car le modèle d'automates utilisé pour représenter les spécifications et les implantations est déterministe : c'est le cas des méthodes passées en revue dans [Fujiwara 91a]. Cependant quelques publications récentes étendent ces méthodes à des modèles d'automates non déterministes. C'est le cas par exemple de [Fujiwara 91b] ou [Luo 93]. Dans ces articles, le fait que l'on observe bien tous les comportements possibles des implantations est appelé hypothèse de test complet (*complete testing assumption*), qui exprime une «certaine équité entre les différents comportements autorisés par le non-déterminisme». C'est la même démarche que nous adoptons dans cette thèse, en l'élaborant plus explicitement dans les deux sections suivantes.

Quant à la question du reset, elle est également prise en compte. Certaines méthodes font l'hypothèse du reset bien implanté [Fujiwara 91a], d'autres supposent seulement la connexité de la spécification et définissent une seule séquence de test [Hennie 64] [Gonenc 70] [Hsieh 71] [Yao 93]. Mais cette séquence unique est obtenue au prix d'un allongement considérable de la longueur totale du test à effectuer.

3 Hypothèse d'équité bornée

Pour résoudre le problème posé à la section 2.3, nous devons formaliser l'existence d'une certaine équité entre les choix qui peuvent être faits par l'implantation. Comme nous voulons être sûrs d'observer tous les comportements de l'implantation au bout d'un nombre fini, calculable à l'avance, de tentatives, la notion adéquate à utiliser est une équité bornée, comme celle introduite par exemple par [Silva 92] sur le modèle des réseaux de Petri.

Dans le modèle d'automates à entrées et sorties que nous utilisons, les automates sont décrits par leur structure en états et transitions. Les traces d'un automate ont été définies comme les enchaînements d'actions observables permis par cette structure, à partir de l'état initial de l'automate. Jusqu'à présent, nous avons basé toute notre étude sur les seules propriétés de ces traces, supposant implicitement que toutes les traces d'un automate peuvent être observées lors d'une «exécution» de cet automate.

Un tel concept d'exécution ne permet pas d'exprimer la notion d'équité mentionnée ci-dessus. En effet, si l'on considère par exemple une implantation non déterministe comme celle représentée sur la figure 79, les exécutions étant assimilées aux traces, donc déterminées par la structure de l'automate, une séquence infinie (?a!b)*, non équitable vis à vis de la transition étiquetée par !c, fait partie des exécutions. Nous avons donc besoin de modéliser un autre concept d'exécution, différent de celui de trace.

Définition 9.3 : les *exécutions* d'un automate à entrées et sorties S sont un sous ensemble de son ensemble de traces, $X \subseteq Tr(S)$.

Nous considérons ici que toutes les traces ne peuvent pas être exécutées par une IOSM, mais seulement celles qui font partie de l'ensemble des exécutions. Nous supposons que la donnée de cet ensemble est arbitraire. Nous appelons :

Définition 9.4 : un *automate exécutable* est un couple $(I, X(I))$ constitué d'un automate à entrées et sorties I et de son ensemble d'exécutions $X(I)$.

Bien entendu, l'ensemble d'exécutions doit posséder un certain nombre de propriétés minimales pour que notre modèle soit réaliste. Par exemple, tout préfixe d'une exécution doit aussi être une exécution. Nous n'étudions pas plus en détail comment caractériser les ensembles d'exécutions «réalistes», car ce n'est pas nécessaire pour ce chapitre. Cependant, nous exigeons les deux propriétés ci-dessous, qui sont nécessaires pour pouvoir prouver que les testeurs que nous allons définir fonctionnent bien, et qui nous paraissent suffisamment réalistes :

1. si $\sigma \in X(I)$ et $\sigma?a \in Tr(I)$ alors $\sigma?a \in X(I)$, autrement dit les exécutions ne peuvent pas restreindre les réceptions (propriété «exécution 1»).
2. comme nous supposons que les blocages de sortie sont modélisés explicitement, tous les états de l'implantation admettent au moins une émission. Nous supposons que si $\sigma \in X(I)$, alors il existe au moins une séquence $\sigma!a \in X(I)$. Autrement dit, les exécutions peuvent restreindre les choix possibles (le non déterminisme des implantations), mais ne peuvent pas supprimer tout choix, provoquant un blocage (propriété «exécution 2»).

Nous voulons modéliser le fait que lorsqu'on essaie d'observer une séquence d'actions sur un automate, comme les choix non déterministes de cet automate sont à peu près équitables, au bout d'un nombre raisonnable de tentatives on verra cette séquence dans les exécutions, sauf si cette séquence ne fait pas partie des traces de l'automate. Nous définissons d'abord les tentatives ratées pour observer une séquence :

Définition 9.5 : soit L un ensemble d'interactions et $\sigma \in (\{?, !\} \times L)^*$. On dit que $\sigma'!a$ est *un essai inconcluant* de σ si σ' est un préfixe strict de σ et que $\sigma'!a$ n'est pas un préfixe de σ. Nous notons $Inconc(\sigma)$ l'ensemble des essais inconcluants de σ.

L'équité bornée exprime que les tentatives ratées sont en nombre borné. Comme indiqué à la section 2, nous supposons que l'automate possède un reset, ce qui nous permet de concaténer plusieurs tentatives.

Définition 9.6 : un automate exécutable $(I, X(I))$ ayant un reset r a une *équité bornée* si :

$$(\forall n \in \mathbb{N})\ (\exists p \in \mathbb{N})\ (\forall \sigma \in (\{?, !\} \times L_I)^n)\ (\forall \sigma_1 \ldots \sigma_p \in Inconc(\sigma))$$
$$(\sigma_1 ?r \sigma_2 ?r \ldots \sigma_p \in X(I) \Rightarrow \sigma \notin Tr(I))$$

Cette formule sera aussi notée $(\forall n \in \mathbb{N})\ (\exists p \in \mathbb{N})\ E_{n,p}$ en abrégé. Notons que si l'automate est effectivement non déterministe l'équité bornée n'est possible que si les exécutions ne couvrent pas toutes les traces. En effet, si une trace de l'automate admet un essai inconcluant qui est lui aussi une trace de l'automate (ce qui est le cas pour les automates non déterministes), alors par application du reset et concaténation il est clair qu'un nombre quelconque d'essais inconcluants consécutifs fait partie des traces de l'automate.

Il pourrait paraître plus raisonnable de définir l'équité bornée en remplaçant $\sigma \notin Tr(I)$ par $\sigma \notin X(I)$ dans l'expression $E_{n,p}$. Cela cadre mieux si l'on interprète les exécutions comme la partie du comportement effectivement observable de l'implantation, donc celle qu'il faut prendre en compte lors du test de conformité. Mais cette solution nous obligerait à redéfinir les relations d'implantation testées au moyen du concept d'exécutions, et nous ne l'avons pas approfondie dans cette thèse.

Pour illustrer cette définition, considérons à nouveau l'implantation non déterministe représentée sur la figure 79. Supposons que nous représentons cette implantation par un automate exécutable à équité bornée, et que pour $n = 2$ la valeur de p soit 3. Cela signifie, puisque la séquence ?a!c (de longueur 2) fait partie des traces de l'automate, que l'on ne peut pas observer 3 essais inconcluants consécutifs de cette trace, donc que ?a!b?r?a!b?r?a!b $\notin X(I)$. Inversement, si nous faisons l' *hypothèse d'équité* suivante :

> «on suppose que les implantations possibles ont une équité bornée et que pour des traces de longueur 2 alors le nombre maximal d'essais inconcluants consécutifs est 2»

Alors si nous observons la séquence d'exécution ?a!b?r?a!b?r?a!b nous pouvons conclure que la séquence ?a!c ne fait pas partie des traces de l'implantation testée. Soit φ une fonction sur l'ensemble des entiers naturels. Dans la suite de ce chapitre nous notons $E_\varphi = E_{n,\varphi(n)}$ l'hypothèse d'équité bornée pour laquelle les traces de longueur inférieure ou égale à n admettent au plus $\varphi(n) - 1$ essais inconcluants consécutifs. Sous une telle hypothèse de test, nous avons deux moyens de faire des conclusions au sujet des traces d'une implantation sous test :

1. si nous observons une trace, nous savons qu'elle fait partie des traces de l'implantation.
2. si nous observons $\varphi(n)$ essais inconcluants consécutifs d'une séquence de longueur inférieure ou égale à n nous savons que cette séquence ne fait pas partie des traces de l'implantation.

Dans la section 4 nous définissons des testeurs exécutables basés sur ces principes. Pour conclure cette section, il nous reste à justifier le choix de l'hypothèse d'équité bornée comme solution au problème du test d'implantations non déterministes. Il paraîtrait peut être plus intuitif, au lieu d'imposer une limite stricte au nombre de fois où un même choix non déterministe peut être fait, de donner des probabilités à chacune des transitions possibles. Cependant, une telle modélisation ne pourrait que fournir une conclusion probable au sujet de l'implantation testée (d'autant plus fiable que le test serait long). Elle n'est pas bien adaptée à l'approche algébrique que nous avons adoptée dans cette thèse, qui vise à établir des propriétés sur les implantations sous test.

On aurait pu songer également à borner le nombre de fois consécutives où une transition est choisie dans un état donné, au lieu d'imposer une borne globale, sur une séquence entière. Cependant une telle borne locale ne permet aucune conclusion au sujet de séquences testées. Considérons par exemple les automates de la figure 82. En suppo-

Figure 82 : comment modéliser l'équité ?

sant seulement que les implantations ne peuvent pas faire deux fois consécutives le même choix entre deux transitions en non déterminisme, on ne peut pas savoir , si on observe ?a!b!d?a!c!e si l'on a affaire à I1 ou à I2, donc si l'implantation testée possède ou non la trace ?a!b!e.

Enfin, l'hypothèse d'équité telle que nous l'avons formulée semble correspondre à ce qui est supposé implicitement lors de l'exécution des tests dans la pratique. En effet, lorsque l'on veut vérifier la présence d'un comportement donné, mais que du fait du non déterminisme de l'implantation on observe un autre comportement licite (ce que les praticiens du test appellent un test «inconcluant»), on réessaie le même test un certain nombre de fois. Si on n'observe toujours pas ce qui est attendu, on conclut que ce comportement n'est pas correctement implanté.

4 Testeurs exécutables

4.1 Testeur d'appartenance d'une séquence

La définition d'un testeur exécutable vérifiant une propriété composée va se faire par composition de testeurs plus élémentaires vérifiant des propriétés élémentaires. Soit L un ensemble d'interactions et $\sigma \in (\{?, !\} \times L)^*$. Puisque nous nous plaçons dans le cas où les blocages de sortie sont explicitement modélisés, $\delta \in L$. Nous supposons que δ apparaît au plus une fois dans σ, sous la forme d'une émission, en dernière position de la séquence. Remarquons que c'est toujours le cas pour les séquences dont nous devons vérifier la présence ou l'absence dans les implantations. Nous définissons ci-dessous un testeur exécutable qui vérifie si la séquence σ fait partie des traces de l'implantation, sous une d'hypothèse d'équité bornée E_φ. Soit n la longueur de σ. Posons $\sigma = \mu_1 \ldots \mu_n$.

Définition 9.7 : soit L un ensemble d'interactions et $\sigma \in (\{?,!\} \times L)^*$. Le *testeur d'appartenance de* σ sous l'hypothèse E_φ est une IOSM $T_{\sigma,\varphi} = \langle S, L, T, s_0 \rangle$ telle que :

1. $S = ([1,n] \times [1,\varphi(n)]) \cup (\{inconc\} \times [1,p(n)]) \cup \{fail, pass\}$
2. L est l'ensemble d'interactions mentionné ci-dessus.
3. $s_0 = (1, 1)$
4. les transitions du testeur sont obtenues par les règles suivantes :
   a) pour $1 \le i \le n-1$ et $1 \le j \le \varphi(n)$, $((i,j), \bar{\mu}_i, (i+1,j)) \in T$ (transition attendue)
   b) pour $1 \le j \le \varphi(n)$, $((n,j), \bar{\mu}_n, pass) \in T$ (on a observé σ)
   c) pour $1 \le i \le n$, $1 \le j < \varphi(n)$ et $!\mathrm{a} \ne \mu_i$, $((i,j), ?\mathrm{a}, (inconc, j)) \in T$ (essai inconcluant)
   d) pour $1 \le j < \varphi(n)$, $((inconc, j), !\mathrm{r}, (1, j+1)) \in T$ (on recommence)
   e) pour $1 \le i \le n$ et $!\mathrm{a} \ne \mu_i$, $((i, \varphi(n)), ?\mathrm{a}, fail) \in T$ (trop d'essais inconcluants)

Le fonctionnement de ce testeur est très simple : on essaie d'observer la trace σ en envoyant les interactions correspondant aux réceptions de σ. Si on détecte un essai inconcluant, on envoie un reset et on recommence. Si au bout de $\varphi(n)$ essais inconcluants on n'a toujours pas observé σ, on conclut à l'échec. Nous allons maintenant étudier les propriétés de ce testeur , pour montrer qu'il fait bien ce que l'on attend de lui.

Définition 9.8 : soit $(I, X(I))$ un automate exécutable. Nous appelons *exécutions* de $T_{\sigma,\varphi} \| I$ les traces de $T_{\sigma,\varphi} \| I$ qui correspondent à des exécutions de I.

Propriété 9.1 : soit $(I, X(I))$ un automate exécutable. T outes les traces de $T_{\sigma,\varphi} \| I$ sont finies, de longueur inférieure à $(n+1)\varphi(n) + 2$.

Démonstration : on note $<$ la plus petite relation d'ordre sur l'ensemble des états S de $T_{\sigma,\varphi}$ qui satisfait :

pour $i < i'$, $1 \le i < n$ et $1 \le j \le \varphi(n)$: $(i,j) < (i',j)$

pour $1 \le j < \varphi(n)$: $(i,j) < (inconc, j)$ et $(inconc, j) < (1, j+1)$

pour $1 \le j \le \varphi(n)$ et $1 \le i \le n$: $(i,j) < fail$ et $(i,j) < pass$

Pour toute trace de $T_{\sigma,\varphi} \| I$, les états successifs atteints par $T_{\sigma,\varphi}$ sont strictement croissants. C'est une conséquence de la définition de la composition parallèle sur les IOSM, et du fait que cette propriété est vraie pour chacune des règles a) à e) de la définition 9.7. Comme l'ensemble des états S est fini, on conclut que toutes les traces de $T_{\sigma,\varphi} \| I$ sont finies, de longueur bornée par le cardinal de S, soit au plus $(n+1)\varphi(n)+2$. ❑

Propriété 9.2 : soit $(I, X(I))$ un automate exécutable. Sous l'hypothèse RI du reset bien implanté, si une exécution de $T_{\sigma,\varphi} \| I$ aboutit à un état $(pass, s)$, alors $\sigma \in X(I)$.

Démonstration : on raisonne sur les transitions qu'a dû exécuter le testeur pour aboutir dans l'état $pass$. La dernière transition du testeur est nécessairement obtenue par la règle b) de la définition 9.7. Soit $((n,j), \bar{\mu}_n, pass)$ cette transition. Si $k \neq 1$, pour aboutir dans l'état (k,j), le testeur a effectué une transition obtenue par la règle a), donc de la forme $((k-1,j), \bar{\mu}_{k-1}, (k,j))$. Si $k = 1$, soit le testeur est dans son état initial, et donc l'implantation aussi, car d'après la démonstration de la propriété précédente le testeur ne peut pas revenir deux fois dans le même état. Soit pour aboutir dans l'état $(1,j)$ le testeur a effectué une transition obtenue par la règle d) de la définition 9.7. Comme le reset est bien implanté, l'implantation est dans son état initial.

Donc $T_{\sigma,\varphi} \| I$ a effectué une suite de transitions internes menant de l'état $((1,j), s_{0I})$ à l'état $(pass, s)$ au cours de laquelle le testeur a exécuté $\bar{\sigma}$. D'après la définition de la composition parallèle, $\sigma \in Tr(I)$ au cours de l'exécution considérée de $T_{\sigma,\varphi} \| I$. Par définition des exécutions de $T_{\sigma,\varphi} \| I$, cela signifie que $\sigma \in X(I)$. ❑

Propriété 9.3 : soit $(I, X(I))$ un automate exécutable. Sous les hypothèses RI du reset bien implanté et $E(p)$ d'équité bornée, si une exécution de $T_{\sigma,\varphi} \| I$ aboutit à un état $(fail, s)$, alors $\sigma \notin Tr(I)$.

Démonstration : on raisonne sur les transitions qu'a dû effectuer le testeur pour aboutir dans l'état $fail$. La dernière transition du testeur est nécessairement obtenue par la règle e) de la définition 9.7. Soit $((i, \varphi(n)), ?\mathrm{a}, fail)$ cette transition. Pour aboutir dans un état $(k, \varphi(n))$, si $k \neq 1$ alors le testeur a exécuté $\bar{\mu}_{k-1}$ et si $k = 1$ alors l'implantation est dans son état initial (même démonstration que lors de la preuve précédente). On en déduit que $T_{\sigma,\varphi} \| I$ a effectué une suite de transitions internes menant de l'état $((1, \varphi(n)), s_{0I})$ à l'état $(fail, s)$ au cours de laquelle le testeur a exécuté $\bar{\mu}_1 \dots \bar{\mu}_{i-1}?\mathrm{a}$, donc l'implantation a exécuté $\mu_1 \dots \mu_{i-1}!\mathrm{a}$, qui est un essai inconcluant de σ (puisque d'après la règle e), $a \neq |\mu_i|$.

Si $\varphi(n) \neq 1$, pour aboutir dans l'état $(1, \varphi(n))$ le testeur a effectué une transition obtenue par la règle d) de la définition 9.7, soit $((inconc, \varphi(n)-1), !\mathrm{r}, (1, \varphi(n)))$. Pour aboutir à l'état $(inconc, \varphi(n)-1)$ le testeur a effectué une transition obtenue par la règle c) de la définition 9.7, soit $((i, \varphi(n)-1), ?\mathrm{a}, (inconc, \varphi(n)-1))$. Par le même raisonnement que ci-dessus, on montre donc que $T_{\sigma,\varphi} \| I$ a effectué une suite de transitions internes menant de l'état $((1, \varphi(n) - 1), s_{0I})$ à un état $((inconc, \varphi(n) - 1), s')$ au cours de laquelle le testeur a exécuté $\bar{\mu}_1 \ldots \bar{\mu}_{i-1}?a$, donc l'implantation a exécuté $\mu_1 \ldots \mu_{i-1}!a$, qui est un essai inconcluant de σ.

En itérant ce raisonnement $\varphi(n)$ fois, on déduit que pour aboutir finalement à l'état $(fail, s)$, $T_{\sigma,\varphi} \| I$ a effectué une suite de transitions internes menant de l'état $((1, 1), s_{0I})$ à l'état $(fail, s)$ au cours de laquelle le testeur a exécuté la séquence $\bar{\sigma}_1 !r \bar{\sigma}_2 !r \ldots \bar{\sigma}_{p(n)}$ où les $\bar{\sigma}_i$ sont des essais inconcluants de σ. D'après l'hypothèse d'équité bornée, on conclut que $\sigma \notin Tr(I)$. ❑

Propriété 9.4 : soit $(I, X(I))$ un automate exécutable ayant un reset bien implanté. Les seuls états puits des exécutions de $T_{\sigma,\varphi} \| I$ (ceux dont ne part aucune transition) sont de la forme $(pass, s)$ ou $(fail, s)$.

Démonstration : considérons un état de $T_{\sigma,\varphi} \| I$ de la forme $((inconc, j), s)$. Dans cet état le testeur peut envoyer un reset, et par hypothèse «exécution 1» l'implantation peut accepter ce reset. Donc cet état n'est pas un état puits.

Considérons maintenant un état de la forme $((i, j), s)$. Comme les blocages de sortie sont explicitement modélisés dans l'implantation, tous les états de l'implantation peuvent faire au moins une émission (hypothèse «exécution 2»). Comme dans tous les états du testeur de la forme (i, j) les réceptions de toutes les interactions ont prévues (voir la définition 9.7), alors l'état $((i, j), s)$ n'est pas un état puits.

Seuls les états de la forme $(pass, s)$ ou $(fail, s)$ peuvent donc être des états puits. ❑

En vertu des propriétés 9.1 et 9.4, toute exécution de $T_{\sigma,\varphi} \| I$ aboutit donc au bout d'un nombre fini de transitions internes à un état $(pass, s)$ ou à un état $(fail, s)$. Comme par définition seules les exécutions peuvent être exécutées par l'implantation, il est naturel de choisir comme définition du verdict du testeur :

Définition 9.9 : soit $(I, X(I))$ un automate exécutable. On note :

$$Fail(T, (I, X(I))) \text{ ssi } (\exists \sigma \in X(I))\ ((T, \bar{\sigma}, fail))$$
$$Succ(T, (I, X(I))) \text{ ssi } (\exists \sigma \in X(I))\ ((T, \bar{\sigma}, pass))$$

De cette définition, et de la propriété 9.1, il découle que le testeur $T_{\sigma,\varphi}$ est exécutable au sens de la définition 9.1. Nous allons montrer maintenant qu'il est correct, c'est à dire qu'il teste bien l'appartenance de la séquence σ aux traces de l'implantation.

Propriété 9.5 : soit $(I, X(I))$ un automate exécutable. Sous les hypothèses RI du reset bien implanté et E_φ d'équité bornée :

$$Fail(T_{\sigma,\varphi}, (I, X(I))) \Leftrightarrow \sigma \notin Tr(I)$$

$$Succ(T_{\sigma,\varphi}, (I, X(I))) \Leftrightarrow \sigma \in Tr(I)$$

Démonstration : si $Fail(T, (I, X(I)))$ par définition de l'échec il existe $\sigma' \in X(I)$ telle que $(T, \bar{\sigma}', fail)$, donc d'après la propriété 9.3 $\sigma \notin Tr(I)$. Réciproquement si $Fail(T, (I, X(I)))$ est faux alors il n'existe pas de séquence $\sigma' \in X(I)$ telle que $(T, \bar{\sigma}', fail)$, donc d'après les propriétés 9.1 et 9.4 il existe une séquence $\sigma'' \in X(I)$ telle que $(T, \bar{\sigma}'', pass)$. Donc d'après la propriété 9.2 $\sigma \in X(I) \subseteq Tr(I)$.

Si $Succ(T, (I, X(I)))$ par définition du succès il existe une séquence $\sigma'' \in X(I)$ telle que $(T, \bar{\sigma}'', pass)$. Donc d'après la propriété 9.2 $\sigma \in X(I) \subseteq Tr(I)$. Réciproquement si $Succ(T, (I, X(I)))$ est faux alors il n'existe pas de séquence $\sigma'' \in X(I)$ telle que $(T, \bar{\sigma}'', pass)$, donc d'après les propriétés 9.1 et 9.4 il existe une séquence $\sigma' \in X(I)$ telle que $(T, \bar{\sigma}', fail)$, donc d'après la propriété 9.3 $\sigma \notin Tr(I)$. ❑

Pour terminer cette section, donnons un exemple de testeur d'appartenance. Soit $L = \{a, b, \delta\}$ et $\sigma = ?a!b$. Nous faisons l'hypothèse d'équité suivante : «pour une trace de longueur 2 le nombre maximal d'essais inconcluants consécutifs est 2» (i.e. $\varphi(2) = 3$). Alors le testeur correspondant est représenté sur la figure ci-dessous :

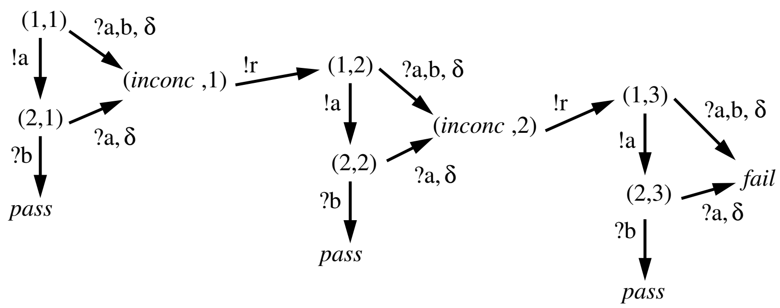


Figure 83 : testeur d'appartenance d'une séquence

4.2 T esteur de non appartenance d'une séquence

Nous définissons maintenant un testeur qui vérifie qu'une séquence n'appartient pas aux traces d'une implantation. Les notations sont les mêmes qu'à la section précédente.

Définition 9.10 : soit L un ensemble d'interactions et $\sigma \in (\{?, !\} \times L)^*$. Le *testeur de non-appartenance de* σ sous l'hypothèse E_φ est une IOSM $T_{\neg\sigma,\varphi} = \langle S, L, T, s_0 \rangle$ telle que :

1. $S = ([1, n] \times [1, \varphi(n)]) \cup (\{inconc\} \times [1, \varphi(n)]) \cup \{fail, pass\}$
2. L est l'ensemble d'interactions mentionné ci-dessus.

3. $s_0 = (1, 1)$
4. les transitions du testeur sont obtenues par les règles suivantes :
   a) pour $1 \le i \le n-1$ et $1 \le j \le \varphi(n)$, $((i, j), \bar{\mu}_i, (i+1, j)) \in T$ (transition attendue)
   b) pour $1 \le j \le \varphi(n)$, $((n, j), \bar{\mu}_n, fail) \in T$ (on a observé σ)
   c) pour $1 \le i \le n$, $1 \le j < \varphi(n)$ et $!a \ne \mu_i$, $((i, j), ?a, (inconc, j)) \in T$ (essai inconcluant)
   d) pour $1 \le j < \varphi(n)$, $((inconc, j), !r, (1, j+1)) \in T$ (on recommence)
   e) pour $1 \le i \le n$ et $!a \ne \mu_i$, $((i, \varphi(n)), ?a, pass) \in T$ (trop d'essais inconcluants)

Le fonctionnement de ce testeur est très simple : on essaie d'observer la trace σ en envoyant les interactions correspondant aux réceptions de σ. Si on détecte un essai inconcluant, on envoie un reset et on recommence. Si au bout de $p(n)$ essais inconcluants on n'a toujours pas observé σ, on conclut au succès (la séquence ne fait pas partie des traces de l'implantation). Si on observe la trace σ, on a évidemment un échec.

En comparant les définitions 9.7 et 9.7, on remarque que les testeurs sont identiques à la permutation près des états $pass$ et $fail$. Il en résulte que nous pouvons adopter pour le testeur $T_{\neg\sigma, \varphi}$ la même définition du succès que pour $T_{\sigma, \varphi}$ (la définition 9.9), que ce testeur est exécutable au sens de la définition 9.1, et qu'il vérifie la propriété suivante :

Propriété 9.6 : soit $(I, X(I))$ un automate exécutable. Sous les hypothèses RI du reset bien implanté et E_φ d'équité bornée :

$$Fail(T_{\neg\sigma, \varphi}, (I, X(I))) \Leftrightarrow \sigma \in Tr(I)$$

$$Succ(T_{\neg\sigma, \varphi}, (I, X(I))) \Leftrightarrow \sigma \notin Tr(I)$$

Le testeur représenté sur la figure ci-dessous teste la non-appartenance de ?a!b aux traces de l'implantation, sous les mêmes hypothèses de test qu'à la section précédente.

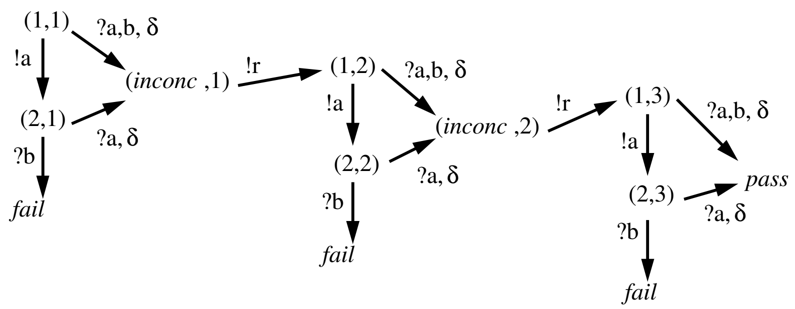


Figure 84 : testeur de non-appartenance d'une séquence

4.3 T esteur d'appartenance d'une séquence quotient

Nous avons montré au chapitre 7 que pour tester l'appartenance d'une séquence quotient σ_q à l'automate quotient résultant d'une congruence, il suffit de vérifier la propriété $R_{C(\sigma_q)}(I)$. Nous définissons ici un testeur pour vérifier cette relation.

Définition 9.1 1 : soit L un ensemble d'interactions, $\approx_L \subseteq L \times L$ une relation d'équivalence et $\sigma_q = \mu_{1q}\ldots\mu_{nq}$ une séquence d'actions quotient. Le *testeur d'appartenance de* σ_q sous l'hypothèse E_φ est une IOSM $T_{C(\sigma_q),\varphi} = \langle S, L, T, s_0 \rangle$ telle que :

1. $S = ([1, n] \times [1, \varphi(n)]) \cup (\{inconc\} \times [1, \varphi(n)]) \cup \{fail, pass\}$
2. L est l'ensemble d'interactions mentionné ci-dessus.
3. $s_0 = (1, 1)$
4. les transitions du testeur sont obtenues par les règles suivantes :
   a) pour $1 \le i \le n-1$ et $1 \le j \le \varphi(n)$:
   si $\mu_{iq} = ?a_{iq}$ alors $((i, j), !i(a_{iq}), (i+1, j)) \in T$
   et si $\mu_{iq} = !a_{iq}$ alors pour tout $a_i \in a_{iq}$ $((i, j), ?a_i, (i+1, j)) \in T$ (transition attendue)
   b) pour $1 \le j \le \varphi(n)$:
   si $\mu_{nq} = ?a_{nq}$ alors $((n, j), !i(a_{nq}), pass) \in T$ (on a observé σ_q !)
   et si $\mu_{nq} = !a_{nq}$ alors pour tout $a_n \in a_{nq}$ $((n, j), ?a_n, pass) \in T$ (on a observé σ_q!)
   c) pour $1 \le i \le n$, $1 \le j \le \varphi(n)$ et $!a_q \ne \mu_{iq}$, pour tout $a \in a_q$
   $((i, j), ?a, (inconc, j)) \in T$ (essai inconcluant)
   d) pour $1 \le j \le \varphi(n)$, $((inconc, j), !\mathrm{r}, (1, j+1)) \in T$ (on recommence)
   e) pour $1 \le i \le n$ et $!a_q \ne \mu_{iq}$, pour tout $a \in a_q$ $((i, \varphi(n)), ?\mathrm{a}, fail) \in T$ (trop d'essais inconcluants)

Le fonctionnement de ce testeur est le suivant : on essaie d'observer la séquence quotient σ_q. Pour cela, on envoie une instance d'une interaction correspondant aux réceptions de la séquence quotient (comme on a par hypothèse une congruence, n'importe quelle instance doit être acceptée). On se prépare à recevoir n'importe quelle instance de chaque émission de la séquence quotient. Si on détecte un essai inconcluant, on envoie un reset et on recommence. Si au bout de $\varphi(n)$ essais inconcluants on n'a toujours pas observé σ_q, on conclut à l'échec. On notera que les états de ce testeur sont exactement les mêmes que ceux du testeur $T_{\sigma_q, \varphi}$. La structure de ces deux testeurs est semblable, celui des séquences quotient se contentant d'avoir plus de transitions de réceptions, car chaque action quotient est remplacée par toutes ses instances possibles. On adopte pour le testeur $T_{C(\sigma), \varphi}$ la définition 9.9 du verdict.

La démonstration prouvant que ce testeur fait bien ce que l'on attend de lui, à savoir qu'il caractérise la propriété $R_{C(\sigma_q)}(I)$, ressemble beaucoup à celle déjà faite pour le testeur $T_{\sigma,\varphi}$. Nous nous contentons donc ici de formuler les propriétés jalonnant cette démonstration, en soulignant les différences par rapport aux démonstrations précédentes. Nous aurons besoin, ci-dessous, de formuler une hypothèse d'équité bornée sur un automate quotient. Pour cela, il nous faut une définition de ce qu'est une exécution de cet automate quotient :

Définition 9.12 : soit $(I, X(I))$ un automate exécutable et I_q l'automate quotient de I pour une congruence donnée. Une séquence quotient σ_q est une *exécution* de I_q ssi il existe $\sigma \in X(I)$ telle que $\tilde{\sigma} = \sigma_q$.

Propriété 9.7 : soit $(I, X(I))$ un automate exécutable. Toutes les traces de $T_{C(\sigma),\varphi} \| I$ sont finies, de longueur inférieure à $(n+1)\varphi(n)+2$.

Démonstration : identique à celle de la propriété 9.1, car les états sont les mêmes dans les deux testeurs. ❑

Propriété 9.8 : soit $(I, X(I))$ un automate exécutable. Sous l'hypothèse RI du reset bien implanté, si une exécution de $T_{C(\sigma),\varphi} \| I$ aboutit à un état $(pass, s)$, alors $(\exists \sigma \in C(\sigma_q))(\sigma \in X(I))$.

Démonstration : comme dans la démonstration de la propriété 9.2, on raisonne sur les transitions qu'a dû exécuter le testeur pour aboutir dans l'état $pass$. Par itération à partir de cet état final, on montre qu'il a exécuté une séquence $\bar{\sigma}$ telle que $\sigma \in C(\sigma_q)$.❑

Propriété 9.9 : soit $(I, X(I))$ un automate exécutable. Sous les hypothèses RI du reset bien implanté (appliquée à l'implantation) et $E(p)$ d'équité bornée (appliquée à l'automate quotient I_q de l'implantation), si une exécution de $T_{C(\sigma),\varphi} \| I$ aboutit à un état $(fail, s)$, alors $\sigma_q \notin Tr(I_q)$.

Démonstration : identique à celle de la propriété 9.3. On montre que le testeur a exécuté une séquence $\bar{\sigma}_1 !r \bar{\sigma}_2 !r \ldots \bar{\sigma}_{p(n)}$ où les séquences quotient $\tilde{\sigma}_i$ sont des essais inconcluants de σ_q. Ce qui permet de conclure sous l'hypothèse d'équité bornée appliquée à l'automate quotient. Comme nous l'avons montré au chapitre 7 (propriété 7.4), la conclusion $\sigma_q \notin Tr(I_q)$ équivaut à $(\forall \sigma \in C(\sigma_q))(\sigma \notin Tr(I))$. ❑

Propriété 9.10 : soit $(I, X(I))$ un automate exécutable ayant un reset bien implanté. Les seuls états puits des exécutions de $T_{C(\sigma),\varphi} \| I$ (ceux dont ne part aucune transition) sont de la forme $(pass, s)$ ou $(fail, s)$.

Démonstration : identique à celle de la propriété 9.4. Le point clé est que dans tous les états de la forme (i, j), les réceptions de toutes les interactions sont prévues, par construction du testeur (c'est pourquoi il faut être prêt à recevoir toutes les instances de toutes les interactions quotient). ❏

En vertu de ces propriétés, le testeur $T_{C(\sigma), \varphi}$ est un testeur exécutable. De plus, on aboutit naturellement à la propriété suivante, qui montre que le testeur rend bien le service que l'on attend de lui :

Propriété 9.1 1 : soit $(I, X(I))$ un automate exécutable. Sous les hypothèses RI du reset bien implanté (appliqué à l'implantation) et E_φ d'équité bornée (appliquée au quotient) :

$$Fail(T_{C(\sigma), \varphi}, (I, X(I))) \Leftrightarrow \neg R_{C(\sigma)}(I)$$

$$Succ(T_{C(\sigma), \varphi}, (I, X(I))) \Leftrightarrow R_{C(\sigma)}(I)$$

Démonstration : semblable à celle de la propriété 9.5. ❏

Pour terminer cette section, donnons un exemple de testeur d'appartenance d'une séquence quotient. Soit $L = \{a_1, a_2, b_1, b_2, \delta\}$ et σ_q = ?a!b (les actions quotient sont notées a et b). On prend l'instanciation $i(a) = a_1$. Nous faisons l'hypothèse d'équité suivante : «pour une trace de longueur 2 le nombre maximal d'essais inconcluants consécutifs est 2» (i.e. $\varphi(2) = 3$). Alors le testeur correspondant est représenté sur la figure ci-dessous :

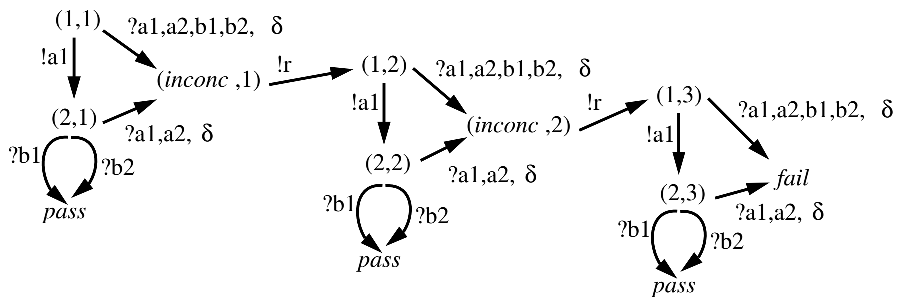


Figure 85 : testeur d'appartenance d'une séquence quotient

4.4 T esteur de non appartenance d'une séquence quotient

On définit un testeur pour vérifier la relation $R_{\neg C(\sigma_q)}(I)$ en inversant les états *pass* et *fail* dans la définition du testeur précédent. En notant $T_{\neg C(\sigma_q), \varphi}$ ce nouveau testeur , et puisque $R_{\neg C(\sigma_q)}(I) = \neg R_{C(\sigma_q)}(I)$, on obtient immédiatement que ce nouveau testeur est exécutable, et vérifie :

Propriété 9.12 : soit $(I, X(I))$ un automate exécutable. Sous les hypothèses RI du reset bien implanté (appliqué à l'implantation) et E_φ d'équité bornée (appliquée au quotient) :

$$Fail(T_{\neg C(\sigma),\varphi}, (I, X(I))) \Leftrightarrow \neg R_{\neg C(\sigma)}(I)$$
$$Succ(T_{\neg C(\sigma),\varphi}, (I, X(I))) \Leftrightarrow R_{\neg C(\sigma)}(I)$$

4.5 T esteurs de séquences double quotient

Au chapitre 10 nous étudierons des automates sur lesquels on définit deux niveaux de congruences (une deuxième congruence est définie sur l'automate quotient résultant d'une première congruence) et nous éprouverons le besoin de vérifier sur les implantations les propriétés suivantes :

$$R_{CC(\sigma_{qq})}(I) \text{ ssi } (\exists \sigma_q \in C(\sigma_{qq}))(\exists \sigma \in C(\sigma_q))(\sigma \in Tr(I))$$
$$R_{\neg CC(\sigma_{qq})}(I) \text{ ssi } (\forall \sigma_q \in C(\sigma_{qq}))(\forall \sigma \in C(\sigma_q))(\sigma \notin Tr(I))$$

On remarque qu'en définissant l'ensemble

$$CC(\sigma_{qq}) = \bigcup_{\sigma_q \in C(\sigma_{qq})} C(\sigma_q)$$

on peut mettre ces propriétés sous la forme

$$R_{CC(\sigma_{qq})}(I) \text{ ssi } (\exists \sigma \in CC(\sigma_{qq}))(\sigma \in Tr(I))$$
$$R_{\neg CC(\sigma_{qq})}(I) \text{ ssi } (\forall \sigma \in CC(\sigma_{qq}))(\sigma \notin Tr(I))$$

Ces propriétés sont donc de la même forme que les propriétés précédemment étudiées ne faisant appel qu'à un seul niveau de quotient. Seul change l'ensemble $C(\sigma_q)$ qui est ici remplacé par $CC(\sigma_{qq})$. Il est donc possible de les tester en utilisant les testeurs précédemment définis, dans lesquels on répercute les changements liés au changement d'ensembles de traces à prendre en compte, soit :

1. changer «pour tout $a \in a_q$» en «pour tout $a \in \bigcup_{a_q \in a_{qq}} a_q$» pour les réceptions définies par les règles a), b) et e) de la définition 9.7.
2. changer « $i(a_q)$ » en « $i(i(a_{qq}))$ » pour les émissions définies par les règles a), b), c) et e) de la définition 9.7.

En effectuant ces transformations, nous obtenons donc deux testeurs exécutables, $T_{CC(\sigma_{qq}),\varphi}$ et $T_{\neg CC(\sigma_{qq}),\varphi}$, qui caractérisent respectivement les propriétés $R_{CC(\sigma_{qq})}(I)$ et $R_{\neg CC(\sigma_{qq})}(I)$.

4.6 Composition de testeurs

Comme indiqué dans l'introduction de ce chapitre, nous savons ramener , sous des hypothèses de test adéquates, toute vérification de conformité à celle d'une composition finie, mettant en œuvre des conjonctions et des disjonctions, de propriétés élémentaires. Nous avons, au cours des sections précédentes, montré comment définir des testeurs exécutables qui vérifient ces propriétés élémentaires. Nous étudions maintenant comment composer ces testeurs afin de vérifier les conjonctions et disjonctions de telles propriétés.

Nous appelons *testeur élémentaire* un quelconque des testeurs exécutables définis dans les sections précédentes. Les *testeurs* sont définis à partir des deux règles suivantes :

1. un testeur élémentaire est un testeur .
2. si T_1 et T_2 sont des testeurs, $T_1 \otimes T_2$ est un testeur , où la composition $\otimes$ désigne l'un des deux opérateurs $\wedge$ ou $\vee$ définis ci-dessous.

Définition 9.13 : soient $T_1 = \langle S_1, L, T_1, s_{01} \rangle$ et $T_2 = \langle S_2, L, T_2, s_{02} \rangle$ deux testeurs. Le *testeur conjonction* $T = T_1 \wedge T_2 = \langle S, L, T, s_0 \rangle$ est une IOSM telle que :

1. $S = (S_1 \times \{1\}) \cup (S_2 \times \{2\}) \cup \{pass, fail\}$. Nous noterons pour simplifier s_1 un état $(s, 1)$ où $s \in S_1$ (et idem pour S_2). Les états $fail_1$ et $fail_2$ sont renommés en $fail$ et l'état $pass_2$ en $pass$.
2. L est l'ensemble d'interactions commun à T_1 et T_2.
3. $s_0 = (s_{01}, 1)$ que nous notons pour simplifier s_{01}.
4. les transitions du testeur composé sont obtenues par les règles suivantes :
   a) si $(s, \mu, s') \in T_1$ alors $(s_1, \mu, s'_1) \in T$
   b) si $(s, \mu, s') \in T_2$ alors $(s_2, \mu, s'_2) \in T$
   c) $(pass_1, !\mathrm{r}, s_{02}) \in T$

Définition 9.14 : soient $T_1 = \langle S_1, L, T_1, s_{01} \rangle$ et $T_2 = \langle S_2, L, T_2, s_{02} \rangle$ deux testeurs. Le *testeur disjonction* $T = T_1 \vee T_2 = \langle S, L, T, s_0 \rangle$ est une IOSM telle que :

1. $S = S_1 \times \{1\} \cup S_2 \times \{2\} \cup \{pass, fail\}$. Nous noterons pour simplifier s_1 un état $(s, 1)$ où $s \in S_1$ (et idem pour S_2). Les états $pass_1$ et $pass_2$ sont renommés en $pass$ et l'état $fail_2$ en $fail$.
2. L est l'ensemble d'interactions commun à T_1 et T_2.
3. $s_0 = (s_{01}, 1)$ que nous notons pour simplifier s_{01}.
4. les transitions du testeur composé sont obtenues par les règles suivantes :
   a) si $(s, \mu, s') \in T_1$ alors $(s_1, \mu, s'_1) \in T$
   b) si $(s, \mu, s') \in T_2$ alors $(s_2, \mu, s'_2) \in T$

c) $(fail_1, !\mathrm{r}, s_{02}) \in T$

Notons que dans chaque cas le testeur composé possède des états nommés s_{01}, $pass$ et $fail$, et qu'il est donc possible, en accord avec les définitions ci-dessus, de le composer à nouveau avec un testeur . Ceci garantit une certaine cohérence à la définition ci-dessus. La définition du verdict est calquée sur celle des testeurs élémentaires.

Définition 9.15 : soit $(I, X(I))$ un automate exécutable et T un testeur . On note :

$$Fail(T, (I, X(I))) \text{ ssi } (\exists\sigma \in X(I))\,((T, \bar{\sigma}, fail))$$
$$Succ(T, (I, X(I))) \text{ ssi } (\exists\sigma \in X(I))\,((T, \bar{\sigma}, pass))$$

Dans le cas d'une conjonction, ceci signifie que le succès du testeur composé est obtenu lorsque l'on atteint le succès du deuxième testeur , ce qui sous-entend que l'on a également atteint le succès du premier testeur auparavant, à cause de la structure du testeur composé. Par contre, l'échec du testeur composé résulte de l'échec de l'un ou l'autre des testeurs composants. Dans le cas d'une disjonction au contraire, le succès du testeur composé est obtenu soit dès que le succès du premier testeur est atteint, soit, en cas d'échec de ce premier testeur , si le succès du deuxième testeur est atteint. Ces propriétés seront formalisées ci-dessous.

Propriété 9.13 : soit T un testeur et $(I, X(I))$ un automate exécutable. T outes les traces de $T \| I$ sont bornées, de longueur inférieure à la somme des bornes des longueurs des testeurs élémentaires composant T (exprimées par la propriété 9.1)

Démonstration : par induction sur le nombre de testeurs élémentaires composés pour obtenir le testeur T. La propriété est vraie pour $n = 1$ par application de la propriété 9.1. Supposons qu'elle est vraie pour tout testeur T_1 composé de n testeurs élémentaires au plus, et considérons un testeur $T = T_1 \otimes T_2$. Supposons que l'on a défini un ordre < sur l'ensemble S_1 des états de T_1 tel que toute transition de T_1 fait strictement croître les états (même chose pour T_2). Nous définissons un ordre sur l'ensemble S des états de T :

pour $s \in S_1$, $s \neq fail$ (conjonction) ou $s \neq pass$ (disjonction) et $s' \in S_2$, : $s_1 < s'_2$

Alors pour toute trace de $T \| I$ les états successifs atteints par le testeur sont strictement croissants. C'est une conséquence du fait que par hypothèse d'induction ceci est vrai pour les transitions de T_1 vis à vis de l'ordre défini sur S_1, pour les transitions de T_2 vis à vis de l'ordre défini sur S_2, et des règles a) b) c) des définitions 9.14 et 9.14 qui toutes préservent cette propriété. Comme l'ensemble S est fini, de cardinal égal à $card(S_1) + card(S_2) - 1$, on en déduit la propriété. ❑

Propriété 9.14 : soit T un testeur et $(I, X(I))$ un automate exécutable ayant un reset bien implanté. Les seuls états puits des exécutions de $T \| I$ sont de la forme $(pass, s)$ ou $(fail, s)$.

Démonstration : par induction sur le nombre de testeurs élémentaires composés pour obtenir le testeur T. La propriété est vraie pour $n = 1$ par application de la propriété 9.4. Supposons qu'elle est vraie pour tout testeur T_1 composé de n testeurs élémentaires au plus, et considérons un testeur $T = T_1 \otimes T_2$ (dans lequel T_2 est lui aussi composé au plus de n testeurs élémentaires). En reprenant les arguments avancés lors de la démonstration de la propriété 9.4, et sous l'hypothèse d'induction, les états de T autres que $pass_1$, $pass_2$, $fail_1$ et $fail_2$ ne peuvent pas être des états puits. De plus :

1. si T est une conjonction, il possède une transition $(pass_1, !\mathrm{r}, s_{02})$, et par hypothèse «exécution 1» l'implantation peut accepter ce reset. Donc l'état $pass_1$ n'est pas un état puits du testeur T. Les autres états sont ceux renommés en $pass$ ou $fail$.
2. si T est une disjonction, il possède une transition $(fail_1, !\mathrm{r}, s_{02})$, et par hypothèse «exécution 1» l'implantation peut accepter ce reset. Donc l'état $fail_1$ n'est pas un état puits du testeur T. Les autres états sont ceux renommés en $pass$ ou $fail$. ❑

La définition du verdict de test et les propriétés 9.13 et 9.14 montrent que tout testeur est exécutable au sens de la définition 9.1. Nous étudions maintenant la relation d'implantation qui est caractérisée par ces testeurs.

Propriété 9.15 : soit $T = T_1 \wedge T_2$ un testeur conjonction et $(I, X(I))$ un automate exécutable ayant un reset bien implanté. Alors :

$$Succ(T_1 \wedge T_2, I) \Leftrightarrow Succ(T_1, I) \wedge Succ(T_2, I)$$

$$Fail(T_1 \wedge T_2, I) \Leftrightarrow Fail(T_1, I) \vee Fail(T_2, I)$$

Démonstration : par définition du succès du testeur T, $Succ(T_1 \wedge T_2, I)$ signifie $(\exists \sigma \in X(I))\ ((T, \bar{\sigma}, pass))$. Donc il existe une séquence de transitions de T telle que $(s_{01}, \bar{\sigma}, pass_2)$. Cette séquence de transitions commence dans un état de la forme s_1 et se termine dans un état de la forme s_2. D'après la définition du testeur composé, seule une transition résultant de l'application de la règle c) de la définition 9.14 permet une transition entre ces deux sous-ensembles d'états. Donc il existe deux séquences d'actions σ' et σ'' telles que $(s_{01}, \bar{\sigma}', pass_1)$, $(pass_1, !\mathrm{r}, s_{02})$ et $(s_{02}, \bar{\sigma}'', pass_2)$ et qui vérifient $\sigma = \sigma' ?\mathrm{r} \sigma''$. Par définition du testeur composé, ceci entraîne $(T_1, \bar{\sigma}', pass)$ et $(T_2, \sigma'', pass)$. De plus, comme $\sigma \in X(I)$ et que le reset est bien implanté, $\sigma' \in X(I)$ et $\sigma'' \in X(I)$. Nous en déduisons $Succ(T_1, I)$ et $Succ(T_2, I)$. Nous avons donc montré : $Succ(T_1 \wedge T_2, I) \Rightarrow Succ(T_1, I) \wedge Succ(T_2, I)$.

Par définition de l'échec de T, $Fail(T_1 \wedge T_2, I)$ signifie $(\exists \sigma \in X(I))\ ((T, \bar{\sigma}, fail))$. Donc il existe une séquence de transitions de T telle que $(s_{01}, \bar{\sigma}, fail_1)$ ou $(s_{01}, \bar{\sigma}, fail_2)$. Dans le premier cas, par définition du testeur composé, on a $(T_1, \bar{\sigma}, fail)$, donc $Fail(T_1, I)$. Dans le deuxième cas, nous sommes en présence d'une séquence de transitions qui commence dans un état de la forme s_1 et se termine dans un état de la forme s_2. D'après la définition du testeur composé, seule une transition résultant de l'application de la règle c) de la définition 9.14 permet une transition entre ces deux sous-ensembles d'états. Donc il existe deux séquences d'actions σ' et σ'' telles que $(s_{01}, \bar{\sigma}', pass_1)$, $(pass_1, !r, s_{02})$ et $(s_{02}, \bar{\sigma}'', fail_2)$ et qui vérifient $\sigma = \sigma'?r\sigma''$. Par définition du testeur composé, ceci entraîne $(T_2, \bar{\sigma}'', fail)$. De plus, comme $\sigma \in X(I)$ et que le reset est bien implanté $\sigma'' \in X(I)$. Ce qui signifie $Fail(T_2, I)$. Nous avons donc montré : $Fail(T_1 \wedge T_2, I) \Rightarrow Fail(T_1, I) \vee Fail(T_2, I)$.

Enfin, les seuls états terminaux du testeur étant de la forme $pass$ ou $fail$, et toute exécution étant finie, il découle de la définition du verdict que $\neg Succ(T, I) = Fail(T, I)$. D'où les équivalences. ❑

Propriété 9.16 : soit $T = T_1 \vee T_2$ un testeur disjonction et $(I, X(I))$ un automate exécutable ayant un reset bien implanté. Alors :

$$Succ(T_1 \vee T_2, I) \Leftrightarrow Succ(T_1, I) \vee Succ(T_2, I)$$

$$Fail(T_1 \vee T_2, I) \Leftrightarrow Fail(T_1, I) \wedge Fail(T_2, I)$$

Démonstration : obtenue de façon immédiate en s'inspirant de celle de la propriété précédente, compte-tenu de la structure du testeur disjonction, qui conclut au succès dès que l'état $pass$ du premier testeur est atteint, et enchaîne sur le second testeur si au contraire l'état $fail$ du premier est atteint. ❑

Nous savons donc construire des testeurs exécutables nous permettant de caractériser toute propriété composée de conjonctions ou disjonctions de propriétés élémentaires. Pour terminer en beauté cette section, nous ne résisterons pas au plaisir d'illustrer ce résultat par un petit exemple. Soit $L = \{a, b, \delta\}$. La figure 86 montre le testeur exécutable correspondant à la vérification de la conjonction $R_{?a!b}(I) \wedge R_{\neg ?a!\delta}(I)$, sous l'hypothèse d'équité où $p(2) = 3$.

Cet exemple montre bien que le testeur exécutable ainsi défini n'est pas optimisé, c'est à dire n'est pas le plus efficace pour vérifier la propriété choisie en un temps minimum. Ceci est dû au caractère systématique de sa construction, par concaténation de testeurs élémentaires qui testent chacun une propriété élémentaire. Dans le cas qui nous intéresse, la partie supérieure du testeur pourrait être optimisée, en produisant un verdict $fail$ dès que la trace incorrecte ?a! δ est détectée. Nous n'étudierons pas plus de façon théorique comment optimiser les testeurs, mais nous indiquerons au chapitre 10 que cela est fait pour la mise en œuvre pratique de TVEDA.

5 Algorithme de test

Pour conclure ce chapitre, et convaincre le lecteur encore sceptique de la proximité indiscutable entre les testeurs définis ci-dessus et des algorithmes de test, nous indiquons ci-dessous, dans une notation pseudo-formelle, comment transformer ces

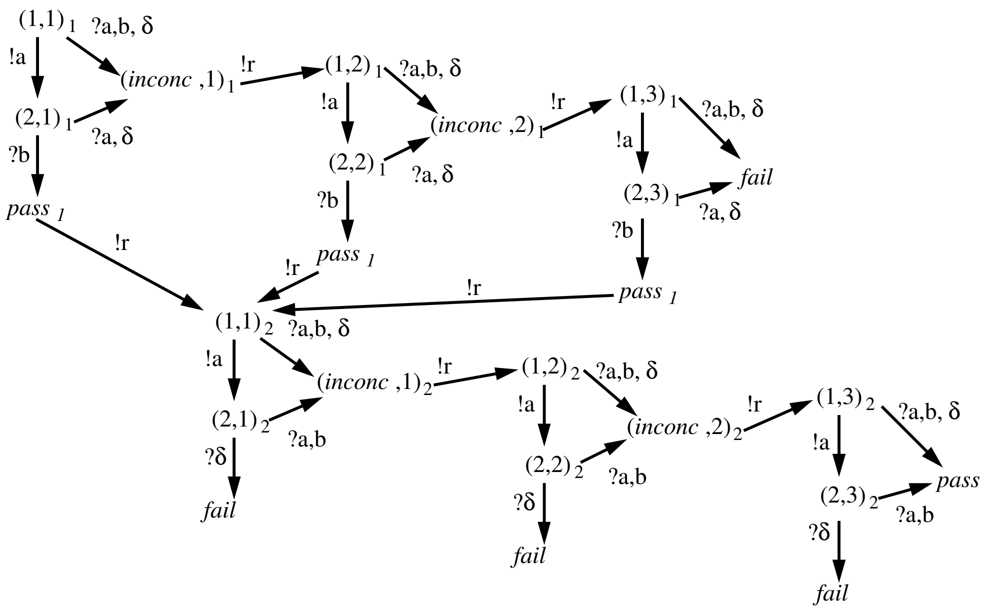


Figure 86 : testeur conjonction de testeur élémentaires

testeurs en du simili-TTCN. Le seul point délicat est la façon particulière de gérer l'action δ, qui modélise l'observation des blocages de sortie. Nous désignons par $S(s, \mu)$ l'état atteint par le testeur à partir de l'état s sur exécution de le transition étiquetée par μ (unique car le testeur est déterministe). L'arbre TTCN est $Arbre(s_0)$, où $Arbre(s)$ est une fonction de construction d'arbres définie par :

1. si $s \neq pass$, $s \neq fail$ et si $?\delta$ n'est pas possible dans l'état s :

Arbre(s) := !a
Arbre(S(s,!a)) → si une émission est prévue
?b1
Arbre(S(s,?b1))
→ pour toutes les réceptions prévues
?bn
Arbre(S(s,?bn))

2. si $s \neq pass$, $s \neq fail$ et que $?\delta$ est possible dans l'état s :

   Arbre(s) := start timer

   !a
   Arbre(S(s,!a)) — si une émission est prévue

   ?b1
   Arbre(S(s,?b1))

   — pour toutes les réceptions prévues

   ?bn
   Arbre(S(s,?bn))

   ?timeout
   Arbre(S(s,? δ))

3. $Arbre\,(fail)$:= verdict fail

4. $Arbre\,(pass)$:= verdict pass

CHAPITRE 10

Le calcul pratique du test de conformité

1 Introduction

L'objectif de cette thèse, tel que nous l'avons formulé dans le premier chapitre, est de fournir un cadre mathématique dans lequel on puisse interpréter les algorithmes développés empiriquement pour le logiciel TVEDA. Au bout de neuf chapitres, cent quatre vingt dix huit pages, quatre vingt six figures, soixante dix huit définitions et cinquante trois propriétés, nous disposons d'un arsenal conséquent de concepts et de résultats théoriques. Il nous reste, dans ce dernier chapitre, à convaincre le lecteur de la pertinence de ce travail pour remplir les buts que nous nous sommes fixés.

Dans un premier temps, nous montrons comment lier entre eux les ingrédients disparates introduits dans les différents chapitres, afin de constituer une méthode cohérente de génération automatique de tests pour des automates à entrées et sorties. Cette récapitulation nous fournit l'occasion d'examiner avec un œil critique les résultats théoriques; en particulier , nous tentons d'apprécier leur adéquation pour rendre compte des procédures effectivement suivies par les concepteurs humains de suites de tests et leur importance relative dans la génération de tests de tous les jours.

Puis nous décrivons, succinctement, les lignes directrices de la mise en œuvre de cette méthode dans le logiciel TVEDA : depuis la description d'automates finis, étendus et de congruences au moyen des techniques de description formelles Estelle ou LDS, jusqu'à la production des tests dans les langages TTCN ou Menuet [Langlois 89] [Deslaugiers 92]. Nous nous attachons à mettre en évidence les difficultés qui se posent pour changer en algorithmes efficaces ou en règles empiriques les formules obtenues jusqu'ici. Nous terminons par la présentation de l'outil lui-même, présent et à venir , et par un bilan des principales applications qui ont été traitées à ce jour .

Pour conclure, nous examinons quelques prolongements possibles au travail présenté dans cette thèse.

2 Une méthode de génération de tests

Nous commençons par décrire les différentes étapes d'une méthode de génération de tests. Cette méthode est assez complète en ce sens qu'elle aboutit à la définition précise d'une expérience de test à effectuer pour tester une spécification dans un environnement de test donné. Nous utilisons, au cours de la présentation ci-dessous, les concepts et le vocabulaire qui ont été définis au cours des chapitres précédents. Cependant, on ne trouvera pas ici de formulation théorique précise (fini les équations !).

La raison est que pour permettre les développements théoriques des chapitres précédents en limitant la complexité desdits chapitres, nous avons dû séparer les problèmes, morceler la difficulté : chaque chapitre faisant le tour d'une question de manière plus ou moins indépendante des autres. Cependant, lorsqu'il s'agit de mettre tout cela simultanément en œuvre, il faut bien lier certains concepts : les questions de pouvoir d'observation du test à travers l'architecture de test interagissent avec les mécanismes de sélection des tests, les hypothèses de test doivent être combinées entre elles, etc.

L'étude théorique doit donc être considérée comme un moyen d'analyser les problèmes posés par le test et de définir les concepts pertinents, plus que comme une formalisation complète de la méthode de génération de tests décrite ici. Cependant, dans la mesure du possible, nous indiquons comment formuler précisément avec les outils théoriques développés les interactions entre les différents ingrédients de la génération de tests, et pourquoi les solutions trouvées tout au long des chapitres précédents justifient la méthode globale.

Celle-ci comporte quatre étapes, décrites ci-dessous.

2.1 Déterminer ce qu'on veut tester

La première chose à faire est de définir le système pour lequel on va engendrer les tests et le type de conformité que l'on veut vérifier .

L'objectif final d'un test est de vérifier la conformité d'un système à sa spécification. Mais comme nous l'avons vu au chapitre 6, la taille de certains systèmes est telle qu'il n'est pas possible de les tester en une seule fois. On est donc amené dans certains cas à découper la spécification complète en morceaux plus petits, qui vont constituer autant de petites spécifications à tester séparément. Pour effectuer ce découpage, suivant les enseignements du chapitre 6, il vaut mieux tenter de déceler des fonctions indépendantes dans la spécification : nous formulerons cela précisément un peu plus loin.

Il faut ensuite définir précisément l'architecture de test qui sera mise en œuvre : en particulier les accès, c'est à dire les points de contrôle et d'observation, dont on dispose sur l'implantation. Ce sont eux qui définissent l'environnement de test, l'interface entre l'outil de test et le système à tester . Dans le cas où la spécification globale a été découpée en sous-systèmes, il faut savoir si l'implantation permet un accès aux interfaces internes à ces sous-systèmes. Dans le cas contraire, pour le test de l'un des composants, on doit considérer les autres comme faisant partie de l'environnement de test. Ces deux situations sont illustrées sur la figure 87.

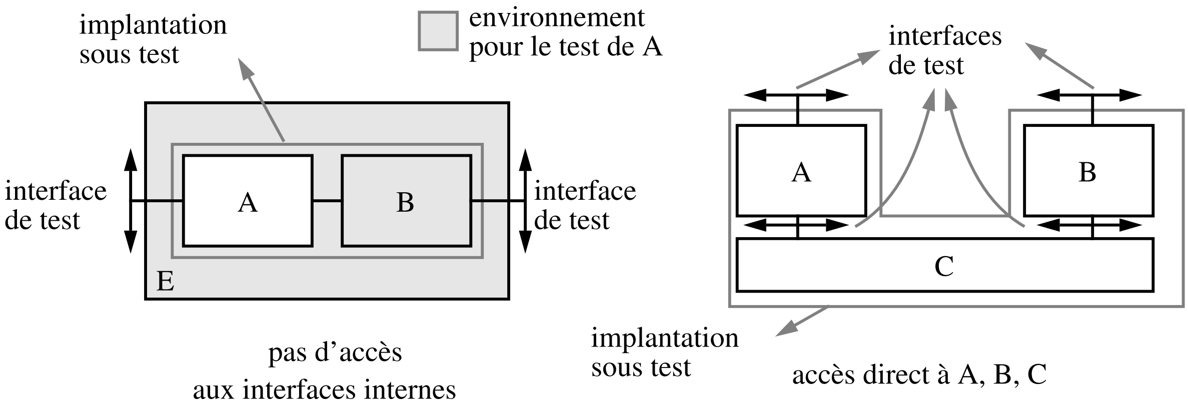


Figure 87 : systèmes composés et interfaces de test

Dans tous les cas, à l'issue de ces choix (morceau de spécification à tester, architecture de test), nous avons implicitement défini la spécification S que nous voulons tester et l'environnement de test E qui doit être pris en compte. Cette définition est implicite en ce sens que ces deux composants sont précisément identifiés, mais nous ne cherchons pas à construire explicitement leur représentation sous forme d'automates !

Le chapitre 4 nous a appris qu'il n'est pas possible de vérifier dans ces conditions n'importe quelle conformité : le pouvoir d'observation du test est limité à l'équivalence de traces modulo l'environnement de test, soit :

$$R_t(E)\ (I, S)\ \text{ ssi }\ Tr(E \parallel S)\ =\ Tr(E \parallel I)$$

Il nous faut donc choisir la relation d'implantation que nous allons tester en tenant compte de cette limite. Les chapitres 2, 3 et 4 nous enseignent que la relation de quasi-équivalence modulo l'environnement

$$R_5(E)\ (I, S)\ \text{ ssi }\ (\forall \sigma \in Tr(E \| S))\ (\sigma \in Tr(E \| I) \Rightarrow (O(\sigma, E \| S) = O(\sigma, E \| I)))$$

ou l'une des deux relations de préordre qui lui sont associées (en remplaçant l'égalité dans la formule ci-dessus par une inclusion, dans un sens ou dans l'autre), sont de bonnes candidates applicables aux systèmes modélisables par des automates à entrées et sorties non déterministes et incomplets, ce qui a toutes les chances d'être le cas du système que nous nous apprêtons à tester.

Dans la pratique, et modulo une découpe judicieuse de la spécification globale en parties élémentaires, il est fort possible que la relation d'implantation à vérifier varie selon les parties : égalité de comportement pour les parties qui doivent être reproduites à l'identique, inclusion pour les parties où on se satisfait d'une implantation de certaines fonctions seulement, inclusion inverse pour les parties où on ne veut surtout pas que l'implantation se permette des libertés non prévues par la spécification.

Nous sommes maintenant en mesure de revenir plus précisément sur le type d'hypothèse d'indépendance qui est faite en découpant la spécification en morceaux à tester séparément. Dans le cas où l'on a un accès direct aux interfaces internes (voir la figure

87, exemple de droite), nous sommes en mesure de tester $R_5\,(I_A, S_A)$, $R_5\,(I_B, S_B)$ et $R_5\,(I_C, S_C)$. Nous nous basons alors sur ces trois conformités pour conclure ou non à la conformité qui nous intéresse, à savoir $R_5\,(I_A \times I_B \times I_C, S_A \times S_B \times S_C)$. Ce faisant, nous faisons une hypothèse d'indépendance semblable à celle étudiée au chapitre 6.

Par contre, dans le cas où nous n'avons pas d'accès direct aux interfaces internes (exemple de gauche sur la figure 87), nous pouvons tester $R_5\,(E \| B)\;(I_A, S_A)$ (l'environnement de test pour le test de A est constitué de E et de B) et $R_5\,(E \| A)\;(I_B, S_B)$. Pour aboutir à la conclusion qui nous intéresse, à savoir $R_5\,(E)\;(I_A \times I_B, S_A \times S_B)$, nous devons formuler une hypothèse que nous appelons aussi hypothèse d'indépendance, mais qui est différente de celle étudiée au chapitre 6.

Considérons l'exemple du protocole INRES [Hogrefe 92], déjà introduit au chapitre 6. Ce protocole de transfert de données orienté connexion sert de cas d'étude, dans le cadre du groupe de travail ISO/ITU *Formal Methods in Conformance Testing* [ISO 93], pour illustrer diverses techniques formelles de test car, quoique simple, il comporte un certain nombre de mécanismes représentatifs des protocoles réels. Nous l'utiliserons donc, tout au long de cette section, comme exemple pour notre propre méthode de génération de tests. La figure 88 montre l'architecture de test retenue : il s'agit de l'architecture de test distant [ISO 91]. Dans ce cas, on dispose de deux points de contrôle et d'observation (PCO) sur l'équipement sous test : l'un immédiatement à l'interface supérieure de l'équipement (❶) pour le testeur supérieur, l'autre accédant à l'interface inférieure à travers un canal de communication (testeur inférieur, ❹). Les interfaces ❷ et ❸ ne sont pas accessibles à travers cette architecture de test.

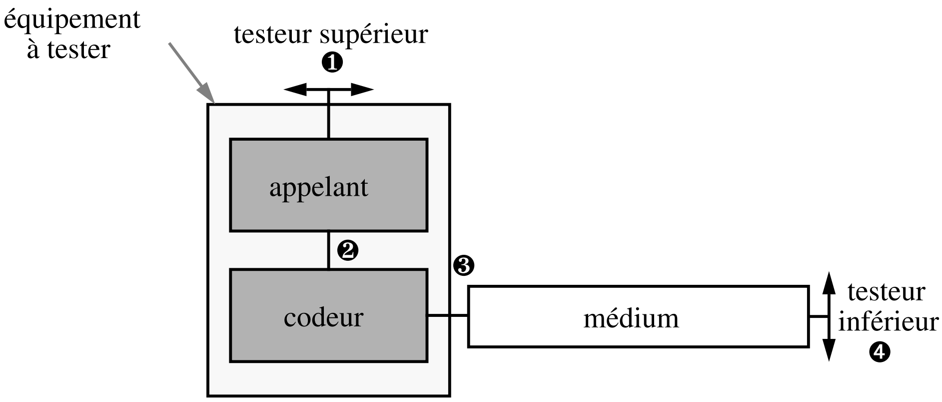


Figure 88 : architecture de test pour le protocole INRES

L'implantation sous test est constituée d'un équipement qui réalise la partie appelant du protocole INRES, incluant le codeur. Nous avons expliqué au chapitre 6 pourquoi l'hypothèse d'indépendance entre les deux parties *appelant* et *codeur* est justifiée : nous pouvons donc tester ces parties séparément. Pour le test de l'appelant par exemple, l'environnement de test est constitué du codeur et du médium.

Pour les raisons de complexité évoquées au chapitre 4, nous ne pouvons pas en règle générale modéliser explicitement l'environnement de test sous forme d'un automate à entrées et sorties. Nous adoptons donc la méthode approchée, fondée sur l'utilisation de règles empiriques, pour tenir compte de l'environnement de test. Au chapitre 4 nous avons montré sur quelques exemples que pour déterminer le testeur canonique asynchrone, on peut remplacer le calcul exact $TCA(E \| S)$ par un calcul approché $TCA(R(S))$ où $R(S)$ désigne la spécification à tester transformée par les règles empiriques convenables.

Nous nous inspirons ici des résultats ci-dessus, mais nous les appliquons dans un contexte différent puisque ce n'est pas le testeur canonique asynchrone que nous cherchons à calculer. Les règles empiriques modélisent l'effet de l'environnement de test sur le contrôle et l'observation du système à tester : donc vue de l'outil de test, l'implantation sous test I se comporte comme sa transformée $R(I)$. La méthode approchée consiste donc ici à remplacer le test de la relation de quasi-équivalence modulo l'environnement ($R_5(E)(I, S)$) par celui de la quasi-équivalence entre les transformées: $R_5(R(I), R(S))$. Cette dernière formule définit ce que nous devons tester. Nous désignons ci-dessous par $S' = R(S)$ l'automate à partir duquel les tests doivent être calculés.

Dans le cas du test du module *appelant* de l'exemple INRES, les effets conjugués du codeur et du canal de transmission sont modélisés par trois règles :

1. un renommage des actions prend en compte l'encodage effectué par le codeur : les PDU du protocole INRES (comme CR, CC, DT) sont transmis au médium après avoir été encapsulés dans des primitives de service de transfert de données du niveau inférieur ($MDA\ Treq$).
2. un autre renommage, qui se superpose au précédent, prend en compte le fait que le canal de transmission transforme les $MDA\ Treq$ en $MDA\ Tind$.
3. enfin on utilise une désynchronisation (voir le chapitre 4, section 4.5) pour représenter le fait que des messages en transit dans le médium peuvent se croiser. Dans l'outil TVEDA (présenté à la section 3 de ce chapitre), cette désynchronisation est implantée par le calcul du *non-déterminisme événement* [Phalippou 91].

2.2 Déterminer la spécification formelle du système à tester

Par spécification formelle, nous entendons ici un modèle formel à partir duquel nous allons calculer les tests. Ce modèle est constitué d'un automate à entrées et sorties et d'une structure sur cet automate, qui combine une congruence avec une structure d'automate étendu. C'est cette spécification formelle qui sera représentée concrètement pas une description en Estelle ou LDS (voir la section 3 décrivant l'outil TVEDA) : à l'étape précédente nous avons déterminé la *sémantique* de cette description, c'est à dire son modèle sous forme d'un automate fini; maintenant nous déterminons la *syntaxe* qui devra être utilisée, c'est à dire comment cet automate est décrit au moyen de transitions étendues.

La structure dont nous disposons sur l'automate à tester résulte de deux choix de modélisation :

1. trouver des ensembles d'actions équivalentes, qui vont donc déterminer une congruence sur l'automate S'. En pratique, comme nous l'avons montré au chapitre 7, les classes d'équivalence regroupent des interactions invalides, ou des interactions qui ne diffèrent que par des données transmises de façon transparente. Mais on peut aussi parfois détecter d'autres ensembles d'actions équivalentes que ceux-là. Nous notons S'_q l'automate quotient issu de ces équivalences.
2. se donner une structure d'automate étendu X' sur cet automate quotient S'_q. Comme tout bon automate étendu qui se respecte, cette structure peut être du plus bel arbitraire. Cependant, en pratique, il n'est pas interdit d'y glisser une certaine logique : par exemple, s'en servir pour refléter la structure de la spécification informelle du système à tester. Ce peut être par exemple le regroupement des états selon les états principaux et certaines variables (qui apparaissent dans la norme) et le regroupement des interactions en fonction des noms de PDU et des paramètres.

En pratique, cette structure à deux niveaux est représentée en Estelle ou LDS de la façon suivante : chaque classe d'équivalence d'actions (celles qui déterminent la congruence) est représentée par un message (interaction Estelle ou signal LDS), tandis que la structure d'automate étendu est celle qui résulte des transitions syntaxiques Estelle ou LDS, comme expliqué au chapitre 8. Avec ce choix de modélisation, le graphe d'accessibilité de la spécification Estelle ou LDS représente directement l'automate quotient S'_q. Bien entendu, la structure obtenue dépend directement du style de programmation en Estelle ou LDS utilisé, car on peut plus ou moins regrouper ou séparer le comportement en transitions étendues. C'est pourquoi l'outil TVEDA offre une certaine souplesse pour influer sur cette structure d'automate étendu.

Dans l'étape ci-dessus, les structures sont définies sur l'automate S' obtenu à partir de la spécification S en incorporant l'influence de l'environnement de test. La spécification Estelle ou LDS fournie en entrée de l'outil de test définit seulement un automate quotient S_q sur S (le graphe d'accessibilité) et une structure d'automate étendu X sur S_q. Pour prendre en compte l'environnement de test, nous utilisons des règles empiriques qui s'appliquent directement à la spécification Estelle ou LDS. De cette façon, nous obtenons une spécification Estelle ou LDS transformée, qui définit l'automate quotient $R(S_q) = S'_q$ (son graphe d'accessibilité) et l'automate étendu X' sur S'_q (sa structure syntaxique). Pour disposer de toute l'information nécessaire à la construction de l'automate S', nous avons encore besoin de savoir comment associer à chaque action quotient de S'_q les actions élémentaires qui lui correspondent : c'est ce que nous appelons les *règles de codage* des interactions Estelle ou LDS. Elles sont d'habitude fournies en supplément de la spécification Estelle ou LDS, par exemple sous la forme d'une description en ASN.1 [ISO 92a].

Bien entendu, la double structure ci-dessus décrit le cas le plus complexe. Si la taille de la spécification le permet, on peut se contenter d'un seul niveau de structure, ce qui simplifie en conséquence la génération de tests.

Illustrons tout cela sur notre exemple INRES. La figure 89 montre une partie de la description en Estelle du protocole.

```
state disconnected, wait, connected, sending
var olddata : ...
      counter : 0..4
      number : 0..1

initialize to disconnected begin end;

from disconnected to wait
❶  when user .iconreq
     begin counter := 0 output pdu_access.cr end;

from wait to connected
   when pdu_access.cc
❷    begin number := 1; counter := 0;
     output user .iconconf end;

from wait to same delay(5)
   provided counter<4
❸    begin output pdu_access.cr;
     counter := counter+1 end;

❹  provided otherwise to disconnected
     begin output user .idisind end;

from connected to sending
   when user .idatreq(isdu)
❺    begin output pdu_access.dt(number ,isdu)
     olddata := isdu end;

❻ trans when user .FCS_error
   begin end;
```

Figure 89 : description en Estelle du protocole INRES (extrait)

Les interactions Estelle représentent directement des classes d'équivalence d'actions élémentaires, comme indiqué au point 1 ci-dessus. Par exemple, *FCS_error* représente toutes les interactions réelles ayant une erreur dans la séquence de contrôle de trame (classe d'équivalence d'interactions invalides). De même, l'interaction *IDA Treq* possède un paramètre *isdu* de type Estelle *indéfini* (type «...»). Ce type à une seule valeur représente toutes les valeurs possibles pour le champ données véhiculé par le message, donc l'interaction *IDA Treq* représente la classe d'équivalence de tous les messages transportant des données.

Le graphe d'accessibilité de la spécification Estelle représente donc bien, comme il est dit plus haut, un automate quotient. Pour connaître l'automate complet, et en particulier les actions élémentaires (qui seront celles utilisées dans les testeurs concrets), il nous faut utiliser des *règles de codage* . Nous appelons ainsi les règles qui permettent de calculer à partir des interactions Estelle les trames (séquences de bits) échangées par les implantations du protocole ou du système. Celles-ci peuvent prendre plusieurs formes :

1. une valeur . Par exemple, «le PDU *CR* est codé 10000001».
2. un ensemble de valeurs possibles. Par exemple «les PDU *DT* sont de la forme 11zxxxxy, où z représente le paramètre *number* , xxxx sont des bits quelconques représentant le paramètre *isdu* et où y est un bit de parité de contrôle».
3. une règle de construction. Par exemple «les trames *FCS_error* correspondent à tous les octets dont le dernier bit est un bit de parité incorrect».

Le deuxième niveau de structure est constitué des transitions syntaxiques Estelle, qui peuvent regrouper plusieurs transitions de l'automate fini (graphe d'accessibilité) sous-jacent. Par exemple, la transition ❸ de la figure 89 ci-dessus regroupe quatre transitions élémentaires, correspondant respectivement aux valeurs 0, 1, 2 et 3 de la variable *counter* . On remarque sur l'extrait ci-dessus que la structure syntaxique Estelle correspond à une certaine découpe logique du fonctionnement du protocole : la transition ❸ représente le mécanisme de réémission normal sur échéance temporisateur , tandis que la transition ❹, par exemple, représente l'abandon de la tentative de connexion après quatre essais infructueux.

2.3 Calculer les propriétés à vérifier sur l'implantation

Pour vérifier la conformité $R_5(I', S')$ (où $I' = R(I)$ représente l'implantation vue à travers l'environnement de test) nous appliquons successivement deux hypothèses d'uniformité correspondant aux deux niveaux de structure, quotient et automate étendu, définis sur S'.

Pour tester $R_5(I', S')$ on applique une hypothèse d'uniformité utilisant la congruence définie sur les actions (voir le chapitre 7). Nous avons alors le choix entre appliquer :

1. une hypothèse d'uniformité forte. Soit Σ l'instanciation d'un ensemble de séquences couvrant les transitions de l'automate quotient S'_q : on teste $R_\Sigma(I', S')$.
2. une hypothèse d'uniformité faible. Soit Σ l'instanciation d'un ensemble de séquences couvrant les transitions de l'automate quotient S'_q : on teste $R_\Sigma(I', S')$ et $R_5(I'_q, S'_q)$.

En pratique, l'ensemble Σ est calculé à partir des séquences d' actions quotient en utilisant les règles de codage des interactions Estelle ou LDS. Cependant, lorsque ce niveau de structure est utilisé, l'automate quotient S'_q est souvent très gros. Il n'est donc pas toujours possible de tester $R_\Sigma(I', S')$ qui risque de prendre un temps prohibitif. Dans ce cas, on se contentera d'un nouveau type d'hypothèse d'uniformité, qui consiste à tester seulement $R_5(I'_q, S'_q)$ pour conclure à $R_5(I', S')$.

Dans le cas du protocole INRES, le graphe d'accessibilité de la spécification Estelle est trop gros pour qu'une séquence couvrant les transitions de cet automate soit réellement utilisable. Nous aurons donc recours au nouveau type d'hypothèse d'uniformité mentionné ci-dessus.

Si la taille de l'automate S'_q s'y prête, on peut vérifier directement la conformité $R_5(I'_q, S'_q)$ en utilisant la méthode décrite au chapitre 7. C'est à dire que l'on formule une hypothèse de régularité convenable (voir le chapitre 5) sous laquelle ce test est ramené à celui d'une relation d'implantation finie $R_{\Sigma_q}(I'_q, S'_q)$. Pour vérifier cette relation d'implantation finie entre des automates quotients, on applique la méthode de construction d'un test concret développée au chapitre 7, section 3.2. Celle-ci définit les vérifications à effectuer sur l'implantation concrète I', qui consistent à tester des propriétés $R_{c(\Sigma)}(I')$ et $R_{\neg c(\Sigma')}(I')$ pour des ensembles Σ et Σ' convenables. Là encore, le calcul des instanciations nécessaires se fait à l'aide des règles de codage des interactions Estelle ou LDS.

Dans le cas contraire, pour tester $R_5(I'_q, S'_q)$, on applique une hypothèse d'uniformité étendue utilisant la structure d'automate étendu X' définie sur S'_q (voir le chapitre 8). nous avons le choix entre appliquer :

1. une hypothèse d'uniformité étendue forte. Soit Σ_q un ensemble de séquences couvrant les transitions étendues de X' : on teste $R_{\Sigma_q}(I'_q, S'_q)$. Pour vérifier cette relation d'implantation finie entre des automates quotients, on applique ici aussi la méthode de construction d'un test concret développée au chapitre 7, section 3.2, comme indiqué ci-dessus.
2. une hypothèse d'uniformité étendue faible. Soit Σ_q un ensemble de séquences couvrant les transitions étendues de X' et S''_{qq} l'automate quotient correspondant à une approximation supérieure S''_q de l'EFSM X' sur S'_q (ça se complique, non ?). Alors on cherche à tester $R_{\Sigma_q}(I'_q, S'_q)$ et $R_5(I'_{qq}, S''_{qq})$ où I'_{qq} est une pure vue de l'esprit mais cependant fort utile pour écrire la formule précédente.

La première de ces deux relations est testée comme indiqué au point 1. Pour vérifier la deuxième, nous utilisons la méthode approchée décrite à la section 4.3 du chapitre 8. On formule dans un premier temps une hypothèse de régularité convenable sur l'automate I'_{qq}, sous laquelle la conformité $R_5(I'_{qq}, S''_{qq})$ est ramenée à une conformité finie $R_{\Sigma_{qq}}(I''_q, S''_q)$. Pour vérifier alors cette dernière relation, la méthode approchée prévoit de vérifier des propriétés $R_{c(\Sigma)}(I'_q)$ et $R_{\neg c(\Sigma')}(I'_q)$ pour des ensembles Σ et Σ' convenables. Mais ici, ces propriétés portent encore sur l'automate quotient I'_q. On les ramène à des vérifications sur l'implantation réelle I' en remarquant que pour vérifier

$$(\exists \sigma_q \in C(\sigma_{qq}))\ (\sigma_q \in Tr(I'_q))$$

il faut tester

$$(\exists \sigma_q \in C(\sigma_{qq}))\ (\exists \sigma \in C(\sigma_q))\ (\sigma \in Tr(I'))$$

ce qui correspond à la propriété $R_{CC(\sigma_{qq})}(I')$ sur l'implantation I', que nous avons définie au chapitre 9. De même, vérifier $R_{\neg c(\Sigma')}(I'_q)$ se ramène à la vérification de $R_{\neg CC(\Sigma)}(I')$. Le calcul des instanciations du premier niveau se fait à partir de la description Estelle ou LDS, celui du deuxième niveau en utilisant les règles de codage.

Ici encore, dans le cas du protocole INRES, une utilisation directe de la régularité sur le graphe d'accessibilité Estelle n'est pas envisageable pour des raisons de taille. Nous appliquons donc une hypothèse d'uniformité étendue utilisant la structure du programme Estelle. Si nous optons pour l'hypothèse d'uniformité étendue forte, nous devons vérifier la conformité à un ensemble de séquences couvrant les transitions étendues, ce que nous appelons la méthode du *tour de transitions étendues*. Si au contraire nous optons pour une hypothèse d'uniformité étendue faible, nous devons vérifier en sus une certaine équivalence de structure entre une approximation supérieure du graphe d'accessibilité Estelle et l'implantation sous test.

L'approximation supérieure choisie peut être le *graphe de contrôle*, comme indiqué au chapitre 8. Dans le cas d'Estelle, cet automate s'obtient à partir de la spécification en ne gardant que l'information sur l'état principal dans les transitions (clauses *from* et *to*) et en faisant abstraction des valeurs des paramètres des interactions. Dans l'outil TVEDA (voir la section 3) un mécanisme basé sur le *contrôle* ou non des variables locales du *body* testé permet de faire varier finement le degré d'approximation supérieure utilisé pour calculer les propriétés à tester .

Une séquence double quotient appartenant à Σ_{qq} est donc constituée d'événements de l'automate de contrôle (comme $!DT$ par exemple), donc sans les paramètres des interactions. Une instanciation de premier niveau s'obtient en choisissant des valeurs de paramètres conformément à ce qui est prévu par la spécification Estelle (par exemple $!DT(1, ...)$); puis une instanciation du deuxième niveau est obtenue par les règles de codage (ce qui donne finalement 1 1100001, la séquence de bits envoyée par l'implantation réelle).

2.4 Calculer le test à mettre en œuvre

À l'issue de l'étape précédente, nous avons défini un certain nombre de propriétés à vérifier sur l'implantation, qui constituent ce que l'on doit tester . Ces propriétés peuvent avoir les formes suivantes :

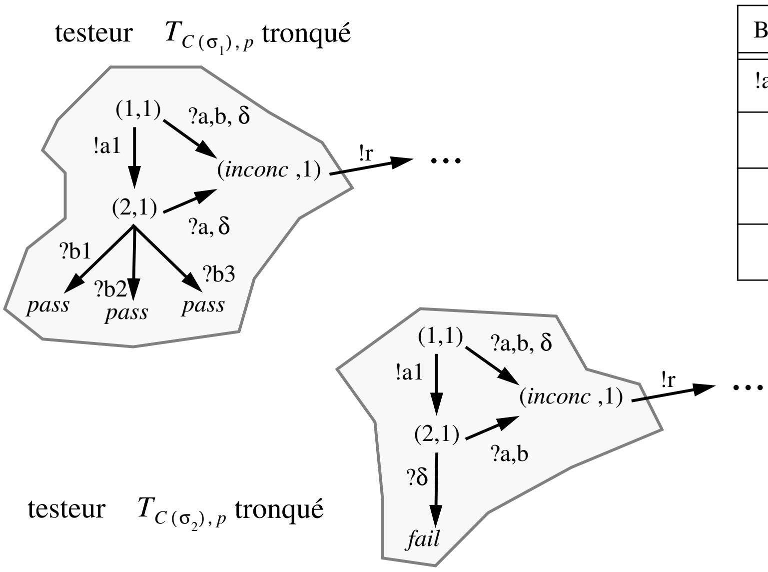
1. $R_{\Sigma}(I', S')$ (appartenance ou non appartenance de séquences à l'implantation)
2. $R_{C(\Sigma)}(I')$ ou $R_{\neg C(\Sigma')}(I')$ (appartenance on non appartenance d'une séquence quotient à l'implantation)
3. $R_{CC(\Sigma)}(I')$ ou $R_{\neg CC(\Sigma)}(I')$ (appartenance ou non appartenance d'une séquence double quotient à l'implantation)

Nous avons étudié au chapitre 9 comment construire des testeurs exécutables de taille raisonnable pour vérifier ces différentes propriétés, sous une hypothèse d'équité bornée convenable. Un testeur exécutable décrit complètement l'expérience de test qui doit être menée, et aboutit à un verdict (*pass* ou *fail*) sur la conformité. Il modélise donc l'exécution d'une suite de tests, et ne se place donc pas au même niveau que la description de la suite de tests abstraite dans un langage de description des tests, TTCN ou Menuet par exemple.

Il est cependant possible d'effectuer une traduction directe entre le testeur exécutable et le langage de test, comme indiqué au chapitre 9, section 5. Cependant, une telle traduction définit des tests (TTCN par exemple) n'aboutissant qu'aux verdicts succès ou échec, car ils incluent les tentatives successives pour observer des comportements, qui, du fait du non-déterminisme des implantations, sont parfois difficiles à observer .

La coutume de l'usage du TTCN veut au contraire que le test décrive seulement la procédure à suivre pour tenter d'observer une fois un comportement de l'implantation. En cas d'échec dans cette tentative du au non-déterminisme de l'implantation, un verdict *inconcluant* est prononcé. Le nombre de tentatives à faire avant de fournir un verdict définitif n'est pas précisé dans la suite abstraite TTCN : cette décision est laissée à l'appréciation des gens qui mettent en œuvre la suite de tests. Pour retrouver une fiche de test proprement dite à partir du testeur exécutable, il faut donc tronquer le graphe représentant le comportement de ce testeur dès qu'un état $fail$, $pass$ ou $(inconc, i)$ est rencontré.

Enfin, les langages de description des tests offrent en général des possibilités de structuration du test (comme Estelle ou LDS permettent de structurer des spécifications). Par exemple, en TTCN, on peut définir une structure sur les interactions, comportant des noms de messages et des paramètres dont les valeurs particulières pour un test donné sont précisées par des tables de contraintes (cette structure est semblable à celle des interactions dans les automates étendus Estelle ou LDS). On l'utilise pour rendre compacte la description syntaxique du test. De même, on aura intérêt à réunir en un même test TTCN plusieurs tests élémentaires présentant des parties initiales communes, au lieu de les concaténer comme indiqué sur la figure 79 (chapitre 9). Tout ceci est illustré sur la figure 90.



Behaviour	C	V
!a start T	a1	
?b	*	pass
?a	*	inconc
?timeout T	*	fail

Test TTCN

Figure 90 : calcul du test à mettre en œuvre

Sur cet exemple, les actions $a = \{a_1, a_2\}$ et $b = \{b_1, b_2, b_3\}$ sont des actions quotient. La colonne **C** du tableau TTCN indique les contraintes, la colonne **V** les verdicts. Le symbole * utilisé dans une contrainte signifie que l'on doit accepter n'importe quelle valeur de l'action quotient. La contrainte a_1 pour l'émission de l'action quotient a signifie que c'est cette valeur particulière qui est émise.

2.5 Quelques considérations sur le résultat obtenu

Il peut être intéressant de commenter brièvement certaines caractéristiques de la méthode de génération de tests présentée ci-dessus. En particulier, nous insistons sur les points qui peuvent prêter à discussion :

1. la méthode décrite ci-dessus n'est pas plus «syntaxique» qu'autre chose. D'ailleurs, traiter une méthode de «syntaxique» n'a jamais eu un sens extrêmement précis. Bien que ce terme n'ait pas obligatoirement une connotation péjorative et bien que non, pas du tout voyons, je n'ai aucune raison de m'énerver à force de l'entendre à longueur de journée à propos de TVEDA, j'insiste encore une fois sur le fait que la syntaxe (de la spécification Estelle ou LDS) est seulement un moyen empirique pour donner une structure à la spécification (étape 2, section 2.2), structure qui est ensuite exploitée pour sélectionner les propriétés nécessaires et suffisantes à vérifier sur l'implantation (étape 3, section 2.3, qui n'a vraiment rien de syntaxique).
2. l'importance relative des ingrédients théoriques dans la recette globale de génération de tests ne reflète pas forcément les préoccupations des théoriciens du domaine. L'hypothèse d'indépendance, et son influence couplée avec celle de l'environnement de test, ont un effet primordial sur la définition du test à faire : or ces deux points ne sont quasiment pas abordés dans l'état de l'art. Pour la sélection des tests proprement dite, l'hypothèse d'uniformité, sous ses diverses formes, joue un rôle central, reléguant l'hypothèse de régularité à un rôle d'appoint : or mis à part pour le test des logiciels séquentiels, l'uniformité n'est presque pas étudiée par les chercheurs du domaine, qui consacrent au contraire tant d'efforts à améliorer encore et toujours les résultats classiques exploitant la régularité. Quant à la gestion du non-déterminisme des implantations lors de l'exécution des tests, pour laquelle nous proposons ici d'utiliser l'équité bornée, c'est un des sujets superbement ignorés par la recherche.
3. la méthode décrite ne donne pas une solution unique au problème de la génération des tests, mais identifie les étapes à respecter et les questions à se poser à chaque étape. Pour chacune de ces étapes, un choix d'options est proposé. Les étapes 1 et 2 sont très ouvertes, puisqu'il s'agit de faire des choix de modélisation. L'étape 3 propose une assez large panoplie d'hypothèses entre lesquelles faire son choix. C'est cette indispensable souplesse qui selon nous rend notre méthode apte à s'adapter à des situations assez variées.
4. l'efficacité des suites de test obtenues, en terme d'erreurs trouvées par rapport à l'effort consacré au test, dépend bien sûr de la pertinence des hypothèses de test choisies pour les construire. Celles que nous avons étudiées sont celles faites par les concepteurs humains de suites de test. Parmi les prolongements possibles de cette thèse (voir la section 4) figurent l'analyse des fautes réellement trouvées au cours des campagnes de test, afin de mesurer la qualité des hypothèses, et le cas échéant la définition de nouvelles hypothèses de test plus conformes à la réalité.

Au risque de décevoir le lecteur pressé de transformer ses spécifications en tests exécutables, il faut ajouter que le résultat obtenu, même s'il définit précisément les opérations à réaliser pour obtenir des suites de tests, est loin de constituer un cahier des charges précis pour la fabrication d'un outil de génération (ce serait trop facile !). En effet, il manque deux informations clés :

1. la définition d'algorithmes efficaces, c'est à dire applicables à des protocoles ou systèmes réels, pour calculer les séquences et les tests. Par exemple : implantation des différentes méthodes classiques de test basées sur la régularité, calcul de

séquences couvrant les transitions étendues d'un automate étendu, procédé pour réunir dans un même test TTCN plusieurs tests de séquences élémentaires différentes, etc.

2. le savoir faire empirique pour faire les choix entre toutes les options de la méthode et pour réaliser au mieux les phases de conception : quels sous-systèmes traiter de façon indépendante ? Comment définir les règles de transformation empirique qui modélisent l'effet de l'environnement de test ? Quand utiliser telle ou telle variante de l'uniformité ? Et bien d'autres choix encore...

C'est la raison pour laquelle le lecteur sceptique risque de faire remarquer que cette thèse est bien jolie, mais qu'elle ne donne même pas l'information nécessaire pour reconstituer TVEDA, et qu'a fortiori on ne voit pas bien ce que tout cet effort théorique peut apporter de plus à des algorithmes définis depuis belle lurette par de bonnes vielles recettes empiriques. À ce contradicteur peu perspicace nous opposerons donc les mérites suivants de notre belle théorie :

1. elle permet de *comprendre* les mécanismes mis en jeu par le test, et donc de répondre à plusieurs des questions que nous nous sommes posées dans l'introduction de cette thèse : qu'est-ce que le test peut prouver ? Que garantit le succès du test ? Comment formuler la couverture atteinte avec une suite de test donnée ? De plus, l'étude détaillée des différentes hypothèses de test aide à comprendre les raisons pour lesquelles elles peuvent se trouver invalidées, donc aide à choisir les hypothèses pertinentes dans chaque situation.
2. elle permet aussi d'*améliorer* l'outil existant : inclure dans TVEDA les dernières avancées théoriques, par exemple sur les méthodes classiques de test; incorporer en un tout cohérent différentes approches du test (en particulier les mécanismes de sélection des tests variés décrits dans la section 2.3) : la méthode globale décrite dans cette section fournit la base des spécifications de la future version V3 de l'outil TVEDA, qui introduira justement beaucoup plus de souplesse dans la gestion des stratégies de test.
3. enfin, à plus long terme, elle définit un *cadre* et des *concepts* ouverts à des recherches futures : autres hypothèses de test, autres méthodes de sélection, autres relations d'implantation, autres modèles de systèmes.

3 Présentation de l'outil TVEDA

Notre ambition n'est pas, dans cette section, de décrire en détail le logiciel de génération de tests TVEDA : ce serait beaucoup trop long, ce n'est pas l'objet de cette thèse, et le lecteur intéressé peut se reporter aux documents techniques CNET disponibles : description de la première version de la méthode et de l'outil [Phalippou 91], description de la version actuelle de l'outil [Offredo 91] [Simonneau 91] [Harou 92] [Risser 93a] [Risser 93b] [Risser 93c] [Fauchier 93]. Pour ceux qui n'ont pas accès aux documents internes CNET , des articles résument les principales évolutions de l'outil [Phalippou 90a] [Phalippou 90b] [Rouger 92] [Boullier 93].

Nous nous contentons donc ici de décrire en quoi TVEDA est relié à la théorie présentée jusqu'ici, les principales fonctions qui composent TVEDA à l'heure actuelle et leurs évolutions programmées dans un avenir proche, et les applications de génération de tests les plus significatives qui ont été traitées jusqu'à présent : nous voulons persuader le lecteur de cette thèse que non seulement les idées développées jusqu'ici sont applicables, mais que ce jugement s'appuie sur la réalisation concrète d'un outil opérationnel et sur l'étude au moyen de cet outil d'un nombre respectable d'exemples significatifs.

3.1 TVEDA et la théorie

La démarche même qui a abouti à cette thèse, exposée dans le chapitre d'introduction, explique les liens étroits qui existent entre l'outil TVEDA et la formalisation théorique que nous venons d'exposer. C'est en effet pour expliquer et formaliser les algorithmes définis de façon empirique et implantés dans TVEDA que ce travail a été entrepris. Satisfaits du résultat obtenu, nous affirmons donc que TVEDA est une implantation de la méthode de génération de tests issue de la théorie. Toutefois, le recouvrement entre l'outil et la formalisation n'est que partiel, chacun contenant des éléments que l'autre ne prend pas en compte.

En effet, la théorie ne formalise qu'une partie des mécanismes mis en œuvre dans l'outil : parce que nous n'avons pas développé la théorie suffisamment pour pouvoir tout prendre en compte, et aussi parce que la théorie a pour but de comprendre les problèmes liés à la génération de tests, ce qui laisse de côté :

1. les problèmes liés au traitement des langages Estelle et LDS. Une (grosse) partie de l'outil constitue l'interface avec les langages de spécification : à partir d'une spécification en langage formel, comment construire une représentation abstraite d'un automate étendu générique. Dans cette partie, les construction syntaxiques propres à chaque langage de spécification sont analysées et traduites vers un modèle commun (voir la figure 91 à la section suivante). Grâce à cette traduction préliminaire, le reste de l'outil TVEDA est indépendant du langage de spécification choisi. C'est également dans cette partie de l'outil que sont définis les sous-ensembles de chaque langage qui peuvent être traités, ce qui a une influence directe sur l'écriture des spécifications formelles destinées à TVEDA.
2. les problèmes liés au style du TTCN produit par TVEDA. La théorie se borne à définir la sémantique des tests (le testeur exécutable du chapitre 9) et quelques éléments de syntaxe (comment écrire en TTCN les parties *behaviour description*, *constraints*, *verdicts*); mais le TTCN est un langage conçu non seulement pour être compilé, mais également pour être lu par des humains, et il comprend donc un nombre important d'informations informelles : formulation en langage naturel des buts des tests, commentaires, etc. Les conventions de nommage des différents identificateurs manipulés constituent également des repères importants pour un lecteur humain. Enfin, la structure d'une suite de tests, organisée en une hiérarchie de groupes et sous-groupes de tests, a une grande influence sur l'appréhension globale de sa logique interne. Reproduire automatiquement tous ces éléments de façon satisfaisante a été l'une des préoccupations majeures lors du développement initial de TVEDA, dont les résultats étaient avant tout destinés à

une appréciation humaine. Dans la première version V1, ces effets de style représentaient même plus de la moitié du code du logiciel !

En contrepartie, l'outil, dans sa version actuelle, n'implante qu'une partie de la méthode de génération issue de la théorie, telle que nous l'avons décrite dans la section 2 de ce chapitre. Les restrictions portent principalement sur l'étape 3 (section 2.3) qui définit les mécanismes de sélection des tests. À l'heure actuelle, la version V2 de TVEDA suppose que l'on vérifie seulement la conformité entre les automates quotients de premier niveau au moyen d'une hypothèse d'uniformité étendue faible utilisant la structure d'automate étendu : on ne dispose que d'un seul mécanisme de sélection des tests.

De plus, la méthode approchée (section 4.3 du chapitre 8) nécessaire pour tirer parti de l'hypothèse de régularité n'est pas implantée pour le moment, pas plus qu'un moyen de calculer des séquences couvrant les transitions étendues de l'EFSM. Donc il n'est pas possible de calculer complètement les propriétés à vérifier sur l'implantation. C'est pourquoi la version V2 de TVEDA ne produit pas des suites de tests complètes, mais des *squelettes* de suites de tests : on énumère bien des propriétés à vérifier sur l'implantation, mais chacune de ces propriétés est décrite de façon incomplète. À partir de ces descriptions incomplètes, l'étape 4 produit des tests eux aussi incomplets, que nous appelons des squelettes de tests (il manque les parties *préambules* et *séquences de vérification* [Phalippou 91]).

Par chance, si les mécanismes qui ne sont pas implantés constituent le cœur de l'algorithme de génération, les parties manquantes qui en résultent ne forment qu'un faible pourcentage des pages de la suite de test attendue (d'où l'on peut déduire que le langage TTCN ne répartit pas la complexité des tests de façon uniforme). Ceci justifie l'utilisation de TVEDA V2 en tant que méthode de génération de tests *assistée par ordinateur* : on produit automatiquement des suites de tests incomplètes, qui sont ensuite complétées manuellement pour obtenir des suites opérationnelles. Une évaluation récente (octobre 1993) de cette méthode assistée sur des exemples significatifs (protocole D niveaux 2 et 3) a montré qu'un gain important peut être espéré : ramener le temps de développement de la suite de tests d'un homme.an par la méthode totalement manuelle à six hommes.mois par la méthode assistée par ordinateur, temps d'écriture de la spécification formelle compris.

3.2 Composants de l'outil TVEDA

Les principaux composants qui existent à l'heure actuelle constituent la version V2 de TVEDA. Ce sont :

1. les interfaces avec les langages Estelle [Offredo 91] et LDS [Harou 92], qui traitent un certain nombre de constructions syntaxiques de ces langages, et produisent un automate étendu abstrait à partir d'un module Estelle (ou d'un bloc LDS). L 'outil TVEDA travaille à partir de la représentation interne sous forme arborescente représentant la syntaxe de la spécification formelle. Cet arbre est lui-même produit à partir de la description formelle sous forme textuelle par un environnement Concerto dédié au langage considéré. Notons que si l'environnement LDS est un produit commercial, l'environnement Estelle a été développé dans le cadre de la réalisation de TVEDA [Simonneau 91].

2. un module de calcul des tests implantant une stratégie particulière de sélection des tests, qui à partir d'un automate étendu abstrait produit un squelette de suite de tests en TTCN [Phalippou 91]. Le TTCN est directement produit sous le format interne requis par l'archiviste de l'environnement TTCN de Concerto.
3. un mécanisme de liens hypertextes, reliant seulement pour l'instant une suite de tests TTCN à la spécification Estelle qui a servi à la produire. Ce mécanisme permet de relier chaque test individuel à la transition ou aux transitions Estelle que ce test vise à vérifier [Fauchier 93].

La figure 91 présente l'architecture cible de l'outil TVEDA envisagée pour la prochaine

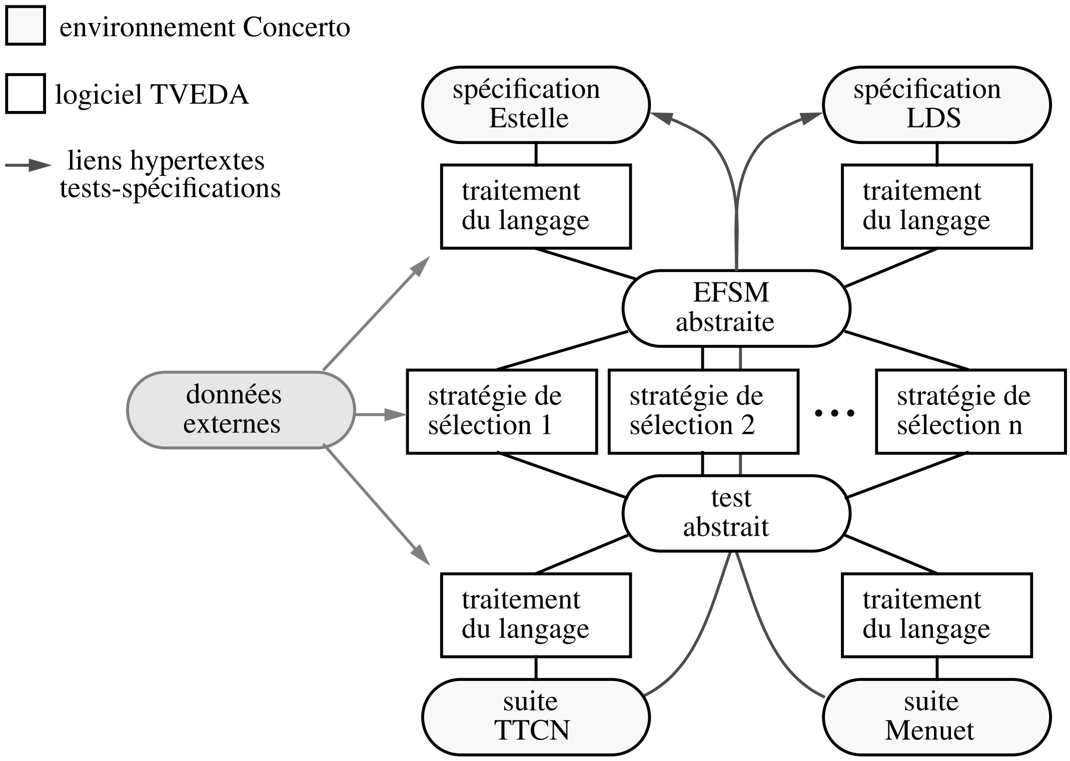


Figure 91 : architecture de TVEDA V3

version (V3) de l'outil, qui doit bénéficier des derniers développements présentés dans cette thèse : les améliorations, par rapport à la version V2 dont nous avons parlé à la section précédente, portent principalement sur :

1. l'introduction de plusieurs mécanismes de sélection des tests, présentés à la section 2.3 de ce chapitre, ainsi que d'une interface utilisateur offrant toute la souplesse voulue pour jongler avec ces mécanismes.
2. la production automatique de tests complets et non plus de squelettes, grâce à l'implantation de procédures calculant complètement les séquences définies dans la section 2.3 déjà mentionnée.

3. l'introduction d'un modèle générique de représentation des tests, permettant de s'abstraire d'une syntaxe particulière, et donc de décrire ensuite les tests dans différents langages. Menuet [Langlois 89] a été choisi comme deuxième langage pour cette version de l'outil.

Comme indiqué sur la figure, un fichier de *données externes* sert à paramétrer les différentes phases de la production des tests. Ce fichier regroupe les informations nécessaires au calcul des tests qui ne peuvent pas être déduites de la spécification formelle du protocole à tester . Par exemple :

1. comment établir une correspondance entre les messages Estelle et les PDU du protocole (un même message Estelle générique peut en effet servir à coder tous les PDU, ou bien on peut trouver un message Estelle pour chaque PDU, etc.).
2. quelles règles empiriques appliquer à l'automate étendu pour prendre en compte l'effet de l'environnement de test.
3. quelle stratégie de sélection des tests adopter .
4. comment structurer la suite de tests en groupes et sous-groupes de tests.
5. quel style adopter pour les parties informelles des suites TTCN à produire.

L'ensemble de l'outil est développé sous l'atelier de génie logiciel Concerto [André 88] et représente actuellement environ 12000 lignes de code en Ulysse (un langage à objets au-dessus de Le_Lisp). Cette taille devrait sérieusement augmenter (au moins doubler) lorsque toutes les nouvelles fonctions offertes par la V3 auront été implantées.

3.3 Principales applications traitées

La version V2 de TVEDA a été appliquée aux systèmes mentionnés ci-dessous. Dans le tableau la taille indiquée pour la spécification est la taille du module à partir duquel les tests sont produits, augmentée de la taille des déclarations nécessaires (les chiffres sont approximatifs). Il est évident que cette taille est indicative seulement, car elle dépend bien sûr du style d'écriture dans la technique formelle considérée. Le nombre de transitions indiqué correspond aux corps de transition (*begin-end*) Estelle et aux constructions *input* LDS.

Domaine	Protocole	Langage	Taille de la spécification		Nombre de tests
			lignes	transitions	
exemple	INRES	Estelle	260	21	41
exemple	INRES	LDS	220	9	11
RNIS	LAPD	Estelle	9000	800	765
RNIS	D niv3 (usager)	Estelle	3500	250	681
RNIS	D niv3 (réseau)	Estelle	9700	750	2321

Domaine	Protocole	Langage	Taille de la spécification		Nombre de tests
			lignes	transitions	
OSI	RTSE	Estelle	5400	200	320
OSI	P1	Estelle	5000	115	
GSM	MAP V2	LDS	900	50	140

L'étude de ces exemples nous a montré que l'outil TVEDA a les performances nécessaires pour engendrer des suites de tests pour des applications réelles. Les sous-ensembles des langages Estelle ou LDS acceptés par l'outil sont suffisants pour que l'on n'ait pas de difficulté à écrire les spécifications mentionnées ci-dessus. Nous avons abouti aux conclusions suivantes quant à la qualité et à la pertinence des suites de tests engendrées :

1. les squelettes de tests produits pour les protocoles de couches basses (INRES, protocole D niveaux 2 et 3) sont satisfaisants. Sur les trois exemples RNIS, une comparaison détaillée et exhaustive entre les suites produites automatiquement par TVEDA et les suites officielles normalisées à l'ETSI a confirmé un recouvrement de plus de 90% : la stratégie de sélection des tests implantée dans TVEDA V2 est satisfaisante pour ce type de protocoles. De plus, il semble possible de compléter à un coût raisonnable les suites produites pour les rendre opérationnelles : la qualité des résultats de TVEDA est donc bonne.
2. par contre, les squelettes de tests produits pour les protocole applicatifs (RTSE, P1, MAP) ne sont pas satisfaisants. Pour de tels protocoles, seuls des tests mettant en œuvre des exemples de fonctionnement de bout en bout semblent intéressants : c'est donc la stratégie de sélection des tests de TVEDA V2 qui est inadaptée dans ce cas.

Ces résultats confirment les analyses faites lors des divers chapitres traitant des méthodes de sélection des tests. Ils justifient le besoin d'enrichir la palette de stratégies de sélection des tests offerte par TVEDA, en appliquant les résultats obtenus dans cette thèse. C'est le principal objectif de la nouvelle version en cours de développement.

4 Quelques prolongements possibles de notre étude

Nous terminons ce chapitre, et la thèse, en indiquant quelques prolongements de notre étude qui nous paraissent indispensables ou intéressants. On ne trouvera pas ici d'idée très originale, ou de pistes radicalement différentes de celle que nous avons explorée : il est suffisamment difficile de construire un sujet de thèse cohérent pour ôter la tentation, après un marathon de deux cents pages, d'en imaginer un autre pour les suivants. Nous nous contenterons donc d'indiquer les compléments naturels sans lesquels notre étude théorique resterait inachevée, ainsi que quelques développements au voisinage immédiat de nos recherches auxquels, faute de temps, de place et probablement de volonté, nous avons renoncé ici.

Tout d'abord, la *mise en œuvre complète* de la méthode de génération de tests exposée dans ce chapitre est indispensable pour valider complètement le travail qui a été fait. Comme indiqué à la section précédente, la version actuelle de TVEDA implante une partie seulement des résultats théoriques obtenus. Les bons résultats de cette version, ainsi que les analyses détaillées ayant servi de guide à la conception des modèles théoriques, laissent espérer que les stratégies de sélection des tests étudiées sont réellement applicables. Mais nous ne considérerons ce résultat comme acquis qu'après avoir fait l'expérience. Celle-ci requiert encore un important travail de mise au point d'algorithmes et de règles empiriques efficaces, travail qui est actuellement en cours.

Ensuite, pour *valider les hypothèses de test* étudiées dans cette thèse, il est très important de les confronter avec les fautes réellement détectées lors des campagnes de test dans les laboratoires. Un analyse des résultats obtenus dans les centres de tests doit permettre de construire des modèles de fautes réalistes et des statistiques sur les fréquences d'apparition des diverses fautes. Ceci peut éventuellement conduire à la formulation de nouvelles hypothèses de test et à leur étude théorique, complétant ainsi la panoplie déjà disponible.

Afin de mettre au point a priori des suites de tests faisant appel aux hypothèses de test adéquates, il serait intéressant d'étudier comment *prévoir les types de fautes* qu'il faut attendre, par l'analyse de la spécification des systèmes ou de leur code source (ou de toute autre information disponible sur le produit à tester). L'intérêt évident d'une telle démarche, qui s'apparente aux prévisions sur la fiabilité des systèmes, est de permettre directement la conception de suites de tests optimisées, sans attendre le retour d'expérience décrit au paragraphe précédent, qui de plus n'est pas toujours disponible pour des systèmes que l'on ne teste qu'une fois.

Parmi les prolongements immédiats de nos recherches, on peut citer l'étude du *pouvoir d'observation du test distribué* : en effet, notre modèle de testeur unique rend compte uniquement d'une situation où le test peut observer toutes les propriétés à l'interface définie par l'environnement de test. En pratique, l'existence de plusieurs interfaces physiquement distantes rend impossible la réalisation rigoureuse d'un tel test centralisé. Comprendre comment le pouvoir d'observation du test est affaibli par son éclatement en plusieurs testeurs permettrait de mettre au point les nécessaires procédures de synchronisation entre ces testeurs, ainsi que les langages de description des tests adaptés.

On peut également envisager de prolonger les réflexions sommaires faites dans cette thèse sur la *mesure de la couverture* des tests. Pour cela, l'approche algébrique adoptée dans cette thèse (preuve de propriétés sur les implantations testées sous les hypothèses de test convenables) pourrait être complétée par une approche probabiliste (confiance dans les propriétés supposées sur les implantations en fonction des probabilités pour que les hypothèses de test soient vraies). Pour être plus réaliste, il faudrait même aller plus loin, car il parait improbable que les hypothèses de tests soient totalement vérifiées, et que l'implantation ait réellement les propriétés attendues : les propriétés de l'implantation sont à peu près vérifiées si les hypothèses de test sont presque vraies, point de vue que nous qualifierons de topologique, étant donné qu'il fait appel à des notions de voisinage et de continuité.

Et pour terminer , n'oublions pas que les automates, finis ou étendus, ne sont pas forcément le modèle ultime et universel, et qu'il doit être possible, tout en gardant les principales idées sur le test utilisées dans cette thèse, les relations d'implantation, les testeurs, les hypothèses de test, d'appliquer celles-ci à *d'autres formalismes* aptes à représenter ou à structurer certaines parties des systèmes que nous avons dû ignorer ici faute de pouvoir les décrire, comme par exemple la spécification de leurs performances, pour laquelle il est impératif de décrire leur évolution dans le temps.

Bibliographie

[Abramsky 87]

S. Abramsky, *Observation equivalence as a testing equivalence*, Theoretical computer science, vol. 53, pp. 225-241, 1987.

[André 88]

E. André, A. Conchon, *Un atelier Concerto pour la gestion des NEF*, L'Écho des Recherches, n. 134, pp. 5-14, 1988.

[Arakawa 92]

N. Arakawa, M. Phalippou, N. Risser, T. Soneoka, *Combination of conformance and interoperability testing*, proceedings of FORTE 92, Perros-Guirec, October 1992.

[Arnold 92]

A. Arnold, *Systèmes de transitions finis et sémantique des processus communicants*, Masson, 1992.

[Bernot 91]

G. Bernot, M.C. Gaudel, B. Marre, *Software testing based on formal specifications : a theory and a tool*, Software Engineering Journal, November 1991, pp. 387-405.

[Bochmann 89]

G. v. Bochmann, R. Dssouli, J. Zhao, *Trace analysis for conformance and arbitration testing*, IEEE Transactions on Software Engineering, n.15, vol.11, pp.1347-1356, 1989.

[Bochmann 91]

G. v. Bochmann, A. Das, R. Dssouli, M. Dubuc, A. Ghedamsi, G. Luo, *Fault models in testing*, proceedings of the 4th International Workshop on Protocol Test Systems, Leidschendam, October 1991.

[Boullier 93]

L. Boullier, M. Phalippou, A. Rouger, *Experimenting test selection strategies*, SDL '93 Using Objects, pp. 267-278, O. Færgemand et A. Sarma editors, North-Holland, 1993.

[Brinksma 88]

E. Brinksma, *A theory for the Derivation of Tests*, proceedings of Eighth International Conference on Protocol Specification, Testing and Verification, Elseiver Science Publishers, 1988.

[Brinksma 91]

E. Brinksma, J. Tretmans, L. Verhaard, *A framework for test selection*, proceedings of Protocol Specification, Testing and Verification XI, Orlando, June 1991.

[Brinksma 93]

E. Brinksma, *Sur la couverture des validations partielles*, actes du 3ème Colloque Francophone sur l'Ingéniérie des Protocoles, Montréal, septembre 1993.

[Castanet 86]

R. Castanet, R. Sijelmassi, *Preparing distributed testing*, proceedings of Protocol Specification, Testing and Verification VI, Montréal, June 1986.

[Castanet 93]

R. Castanet, O. Koné, *Deriving coordinated testers for interoperability*, proceedings of the 6th International Workshop on Protocol Test Systems, Pau, September 1993.

[Cavalli 92]

A. Cavalli, Sung Un Kim, *Automated protocol conformance test generation based on formal methods for LOTOS specifications*, proceedings of the 5th International Workshop on Protocol Test Systems, Montréal, September 1992.

[Cavalli 93a]

A. Cavalli, J. P. Favreau, M. Phalippou, *Formal Methods in Conformance Testing : results and perspectives*, proceedings of the 6th International Workshop on Protocol Test Systems, Pau, September 1993.

[Cavalli 93b]

A. Cavalli, Sung Un Kim, P. Maigron, *Improving conformance testing for LOTOS*, proceedings of FORTE'93, Boston, October 1993.

[CCITT 84]

CCITT , *Modèle de référence pour l'interconnexion des systèmes ouverts pour les applications CCITT* , Recommandation X.200, 1984.

[CCITT 88a]

CCITT , *Specification and Description Language (SDL)* , Recommendation Z.100, 1988.

[CCITT 88b]

CCITT , *Spécification de la couche liaison de données de l'interface usager-réseau RNIS* , Recommandation Q.921, 1988.

[CCITT 88c]

CCITT , *Spécification de la couche 3 de l'interface usager-réseau RNIS* , Recommandation Q.931, 1988.

[CCITT 88d]

CCITT , *Data communication networks : message handling systems* , Recommendation X.411, 1988.

[CCITT 88e]

CCITT , *Mobile Application Part* , Recommendation Q.1051, 1988.

[Chow 78]

T. Chow , *Testing Software Design Modelled by Finite-State Machines* , IEEE Transactions on Software Engineering, vol. SE-4, n. 3, may 1978.

[Chun 90]

W. Chun, P. Amer , *Test Case Generation for Protocols Specified in Estelle* , proceedings of FORTE'90, Madrid, november 1990.

[Combes 93]

P. Combes, M. Michel, B. Renard, *Formal verification of telecommunication service interactions using SDL methods and tools* , SDL '93 Using Objects, pp. 267-278, O. Færgemand et A. Sarma editors, North-Holland, 1993.

[De Nicola 84]

R. De Nicola, M. Hennessy , *Testing equivalences for processes* , Theoretical computer science, vol. 34, pp. 83-133, 1984.

[Deslaugiers 92]

M. Deslaugiers, *Projet ARA VIS, Environnement de production de tests MENUET , manuel d'utilisation de l'éditeur de fiches de test E.F.T version 4 de MENUET* , Note technique NT/PAA/TSA/TLR/3028, CNET , mai 1992.

[Diehl 93]

C. Diehl, C. Jard, J-X. Rampon, *Reachability Analysis on Distributed Executions*, Lecture Notes in Computer Science 668, pp. 629-643, 1993.

[Drira 92]

K. Drira, *Transformation et composition de graphes de refus*, Thèse de l'Université Paul Sabatier, Toulouse, 1992.

[Dssouli 86]

R. Dssouli, *Étude des méthodes de test pour les implantations de protocoles de communication basées sur les spécifications formelles*, Thèse de Ph. D., Université de Montréal, décembre 1986.

[Ehrig 85]

H. Ehrig, B. Mahr, *Fundamentals of Algebraic Specifications, Equations and Initial Semantics*, Spriger-Verlag, 1985.

[Evtushenko 89]

N. Evtushenko, A. Petrenko, *Fault-detection capability of multiple experiments*, Automatic Control and Computer Science, Allerton Press Inc., New-York, vol. 23, n. 3, 1989.

[Fauchier 93]

F. A. Fauchier, *Génération automatique de liens hypertextes pour l'outil TVEDA sous Concerto*, Rapport de stage ESIAL, juillet 1993.

[Fujiwara 91a]

S. Fujiwara, G. Bochmann, F. Khendek, M. Amalou, A. Ghedamsi, *Test Selection Based on Finite State Models*, IEEE Transactions on Software Engineering, vol. 17, n. 6, juin 1991.

[Fujiwara 91b]

S. Fujiwara, G. Bochmann, *Testing non-deterministic state machines with fault coverage*, proceedings of the 4th International Workshop on Protocol Test Systems, The Hague, October 1991.

[Gaudel 91]

M.C. Gaudel, *Algebraic specifications*, chapitre 22 du Software Engineer's Reference Book, John McDermid ed., Butterworth, 1991.

[Gaudel 92]

M.C. Gaudel, *Test Selection Based on ADT Specifications*, proceedings of the 5th International Workshop on Protocol Test Systems, Montreal, September 1992.

[Gill 62]

A. Gill, *Introduction to the theory of finite-state machines*, Mc Graw-Hill, New-York, 1962.

[Gonenc 70]

G. Gonenc, *A method for the design of fault-detection experiments*, IEEE Trans. Comput. vol. C-19, june 1970.

[Guitton 84]

P. Guitton, *Description, validation et test de conformité des protocoles de communication*, Thèse de l'Université de Bordeaux I, 1984.

[Harou 92]

R. Harou, *Conception et réalisation d'une interface avec LDS pour le logiciel de génération de tests TVEDA*, Rapport de stage ENSSAT, juillet 1992.

[Hennessy 80]

M. Hennessy, R. Milner, *Observing nondeterminism and concurrency*, proceedings of the 7th ICALP, Lecture Notes in Computer Science 85, 1980.

[Hennie 64]

F. Hennie, *Fault Detecting Experiments for Sequential Circuits*, proceedings of the 5th Annual Symposium on Switching Circuits Theory and Logical Design, November 1964.

[Hoare 85]

C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[Hogrefe 88]

D. Hogrefe, *Automatic generation of test cases from SDL specifications*, SDL newsletter, n.12, June 1988.

[Hogrefe 92]

D. Hogrefe, *OSI formal specification case study : the INRES protocol and service, revised*, Universität Bern, report IAM-91-012, update 1992.

[Hopcroft 79]

J. Hopcroft, J. Ullman, *Introduction to automata theory, languages and computation*, Addison-Wesley publishing company, 1979.

[Hsieh 71]

E. Hsieh, *Checking Experiments for Sequential Machines*, IEEE transactions on Computers, vol C-20, n. 10, pp. 1152-1166, October 1971.

[ISO 89a]

ISO, *Information Processing Systems, Open Systems Interconnection, Estelle - A Formal Description Technique based on an Extended State Transition Model*, International Standard IS-9074, 1989.

[ISO 89b]

ISO, *Information Processing Systems, Open Systems Interconnection, LOTOS - A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*, International Standard IS-8807, 1989.

[ISO 91]

ISO, *Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework*, International Standard IS-9646, 1991.

[ISO 92a]

ISO, *Information Processing Systems, Open Systems Interconnection, Specification of Abstract Syntax Notation One (ASN.1)*, Draft International Standard DIS-8824, 1992.

[ISO 92b]

ISO, *Information Processing Systems, Open Systems Interconnection, Specification of ASN.1 Encoding Rules*, Draft International Standard DIS-8824, 1992.

[ISO 93]

ISO SC 21 P.54 / ITU TS SG 10 Q.8, *Formal Methods in Conformance Testing*, Working document - approved Geneva output, October 1993.

[Kloosterman 92]

H. Kloosterman, *Test derivation from non-deterministic finite state machines*, proceedings of the 5th International Workshop on Protocol Test Systems, Montréal, September 1992.

[Kristoffersen 92]

F. Kristoffersen, L. Verhaard, M. Zeeberg, *Test derivation for SDL based on ACTs*, actes de FORTE 92, Lannion, 1992.

[Langlois 89]

C. Langlois, E. Paul, *De la spécification formelle à la validation de systèmes*, actes du deuxième colloque annuel du club FIABEX, INT-Évry, 22-23 novembre 1989.

[Leduc 90]

G. Leduc, *On the role of implementation relations in the design of distributed systems using LOTOS*, Thèse d'Agrégation de l'enseignement supérieur, Université de Liège, 1990.

[Luo 93]

G. Luo, A. Petrenko, G.v. Bochmann, *Selecting test sequences for partially-specified nondeterministic finite state machines*, Publication n. 864, Université de Montréal, février 1993.

[Lynch 87]

N. Lynch, M. Tuttle, *Hierarchical correctness proofs for distributed algorithms*, Proceedings of the 6th annual ACM Symposium on Principles of Distributed Computing, pp. 137-151, Vancouver, August 1987.

[Milner 89]

R. Milner, *Communication and Concurrency*, Prentice Hall, 1989.

[Moore 56]

E. Moore, *Gedanken-experiments on sequential machines*, Automata studies, pp. 129-153, 1956.

[Myers 79]

G. Myers, *The art of software testing*, John Wiley & Sons Inc., 1979.

[Naito 81]

S. Naito, M. Tsunoyama, *Fault detection for sequential machines by transition tours*, proceedings of IEEE Fault Tolerant Computer Systems, 1981.

[Offredo 91]

G. Offredo, *Implantation sous Concerto d'un algorithme de génération de tests pour la validation de protocoles*, Rapport de stage ENSSAT, juillet 1991

[Petrenko 91]

A. Petrenko, *Checking experiments with protocol machines*, proceedings of the 4th International Workshop on Protocol Test Systems, The Hague, October 1991.

[Petrenko 92]

A. Petrenko, N. Yevtushenko, *Test suite generation for a FSM with a given type of implementation errors*, proceedings of Protocol Specification, Testing and Verification XII, Orlando, June 1992.

[Petrenko 93]

A. Petrenko, G. v. Bochmann, R. Dssouli, *Conformance relations and test derivation*, proceedings of the 6th International Workshop on Protocol Test Systems, Pau, September 1993.

[Phalippou 88]

M. Phalippou, R. Groz, *Using Estelle for verification : an experience with the T.70 teletex transport protocol*, proceedings of FORTE 88, Stirling, September 1988.

[Phalippou 90a]

M. Phalippou, R. Groz, *Evaluation of an empirical approach for computer-aided test cases generation*, proceedings of the 3rd International Workshop on Protocol Test Systems, Washington, October 1990.

[Phalippou 90b]

M. Phalippou, R. Groz, *From Estelle specifications to industrial test suites, using an empirical approach*, proceedings of FORTE 90, Madrid, November 1990.

[Phalippou 91]

M. Phalippou, *TVEDA : an experiment in computer-aided test case generation from formal specifications of protocols*, Note Technique NT/LAA/SLC/347, CNET, janvier 1991.

[Phalippou 92]

M. Phalippou, *The limited power of testing*, proceedings of the 5th International Workshop on Protocol Test Systems, Montréal, September 1992.

[Phalippou 93]

M. Phalippou, *Executable testers*, proceedings of the 6th International Workshop on Protocol Test Systems, Pau, September 1993.

[Phalippou 94]

M. Phalippou, *Abstract testing and concrete testers*, proceedings of the 14th International Symposium on Protocol Specification, Testing and Verification, Vancouver, June 1994.

[Rafiq 85]

O. Rafiq, R. Castanet, C. Chraïbi, J.P. Goursaud, J. Haddad, X. Perdu, *Towards an environment for testing OSI protocols*, proceedings of Protocol Specification, Testing and Verification V, Toulouse, June 1985.

[Rafiq 90]

O. Rafiq, R. Castanet, *From conformance testing to interoperability testing*, proceedings of the 3rd International Worshop on Protocol Test Systems, Washington, October 1990.

[Rafiq 91]

O. Rafiq, *Le test de conformité des protocoles*, Réseaux et Informatique Répartie, vol. 1, n. 1, pp. 101-142, 1991.

[Rapps 85]

S. Rapps, E. Weyuker, *Selecting Software Test Data Using Data Flow Information*, IEEE Transactions on Software Engineering, vol. 11, n. 4, April 1985.

[Risser 93a]

N. Risser, *Manuel d'utilisation de TVEDA V2*, Document technique FT.CNET/DT/LAA/SLC/EVP/5, mars 1993.

[Risser 93b]

N. Risser, *Manuel de maintenance de TVEDA V2*, Document technique FT.CNET/DT/LAA/SLC/EVP/6, mars 1993.

[Risser 93c]

N. Risser, *Manuel d'installation de TVEDA V2*, Document technique FT.CNET/DT/LAA/SLC/EVP/4, mars 1993.

[Rouger 89]

A. Rouger, P. Combes, *Exhaustive validation and test generation in ELVIS*, proceedings of the 4th SDL forum, SDL '89 The Language at Work, North-Holland, 1989.

[Rouger 92]

A. Rouger, M. Phalippou, *Test cases generation from formal specifications*, proceedings of the XIV International Switching Symposium, Yokohama, October 1992.

[Sabnani 88]

K. Sabnani, A. Dahbura, *A Protocol Testing Procedure*, Computer Networks and ISDN Systems, vol. 15, n. 4, 1988.

[Sarikaya 87]

B. Sarikaya, G. v. Bochmann, E. Cerny, *A test design methodology for protocol testing*, IEEE Transactions on Software Engineering, vol. 13, n. 5, May 1987.

[Sarikaya 89]

B. Sarikaya, *Conformance Testing : Architectures and Test Sequences*, Computer Networks and ISDN Systems, vol. 17, pp. 111-126, 1989.

[Sidhu 89]

D. Sidhu, T. Leung, *Formal Methods for Protocol Testing : A Detailed Study*, IEEE transactions on software engineering, vol. 15, n. 4, avril 1989.

[Silva 92]

M. Silva, T. Murata, *B-Fairness and Structural B-Fairness in Petri Net Models of Concurrent Systems*, Journal of Computer and System Sciences, n. 44, pp. 447-477, 1992.

[Simonneau 91]

F. Simonneau, *Un environnement Estelle pour l'atelier de génie logiciel Concerto*, Rapport de stage ENSSAT, juillet 1991.

[SPECS 91]

SPECS-Testing, *Prototype of Methods and Tools for Testing*, document D8.14, projet RACE SPECS, 1991.

[Tretmans 91]

J. Tretmans, P. Kars, E. Brinksma, *Protocol conformance testing : a formal perspective on ISO IS-9646*, Memoranda Informatica 91-32, University of Twente, 1991.

[Tretmans 92a]

J. Tretmans, L. Verhaard, *A queue model relating synchronous and asynchronous communication*, proceedings of Protocol Specification, Testing and Verification XII, 1992.

[Tretmans 92b]

J. Tretmans, *A formal approach to conformance testing*, PhD. thesis, Twente university, 1992.

[Tretmans 93]

J. Tretmans, *A formal approach to conformance testing*, proceedings of the 6th International Workshop on Protocol Test Systems, Pau, September 1993.

[Tripathy 92]

P. Tripathy, K. Naik, *Generation of adaptative test cases from nondeterministic finite state models*, proceedings of the 5th International Workshop on Protocol Test Systems, Montréal, September 1992.

[Vasilevsky 73]

M. Vasilevsky, *Failure diagnosis for automata*, Kibernetika (Transl.), n. 4, july 1973.

[Verhaard 92]

L. Verhaard, J. Tretmans, P. Kars, E. Brinksma, *On asynchronous testing*, proceedings of the 5th International Workshop on Protocol Test Systems, Montréal, September 1992.

[Vuong 89]

S. Vuong, W. Chan, M. Ito, *The UIOv-Method for Protocol Test Sequence Generation*, proceedings of the second International Workshop on Protocol Test Systems, Berlin, october 1989.

[Weyuker 91]

E. Weyuker, B. Jeng, *Analysing Partition Testing Strategies*, IEEE Transactions on Software Engineering, vol. 17, n. 7, July 1991.

[White 87]

L. White, *Software testing and verification*, Advances in Computers, vol. 26, Academic Press, 1987.

[Yao 93]

M. Yao, A. Petrenko, G. v. Bochmann, *Conformance Testing of Protocol Machines without Reset*, Publication n. 861, Département d'informatique et de recherche opérationnelle, Université de Montréal, février 1993.

Sigles et notations

ASN.1	Abstract Syntax Notation 1 : langage de spécification normalisé par l'ISO.
CCITT	Comité Consultatif International pour le Télégraphe et le Téléphone. Organisme de normalisation international en matière de télécommunications.
CNET	Centre National d'Études des Télécommunications.
Concerto	Atelier Logiciel conçu au CNET et commercialisé par Sema Group.
EFSM	Extended Finite State Machine (automate à états finis étendu).
Estelle	Langage normalisé par l'ISO pour la spécification des protocoles.
ETSI	European T elecommunication Standards Institute. Organisme de normalisation européen.
FSM	Finite State Machine (machine à états finis complète et déterministe).
GSM	Global System for Mobile communication (ou Groupe Spécial Mobiles, selon d'autres sources). Système européen de téléphonie mobile (Itinéris de France Télécom par exemple).
HSI	Harmonized State Identification. Méthode de test basée sur les automates finis.
INRES	INitiator -RESponder . Exemple fameux de protocole d'école.
IOSM	Input Output State Machine (automates à entrées et sorties).
ISO	International Standards Organisation. Organisme de normalisation international.

ITU-TS	International Telecommunication Union - Telecommunications Sector : nom moderne du CCITT.
LAPD	Link Access Protocol on channel D. Protocole de niveau 2 de l'interface usager-réseau du RNIS.
LDS	Langage de Description et de Spécification : normalisé par le CCITT.
LTS	Labelled Transition Systems (systèmes de transitions étiquetées).
MAP	Mobile Application Part. Un des protocoles du système GSM.
Menuet	Langage conçu au CNET pour la description des tests.
OPNFSM	PNFSM observable.
OSI	Open Systems Interconnexion : modèle bien connu à sept couches.
PDU	Protocol Data Unit : messages échangés entre entités homologues de protocoles, dans le modèle OSI.
PNFSM	Machine à états finis partiellement spécifiée et non déterministe.
RNIS	Réseau Numérique à Intégration de Services (Numéris en France).
RTSE	Reliable Transfer Service Element. Un protocole de la couche 7.
TT	Tour de transitions.
TTCN	Tree and Tabular Combined Notation : langage normalisé par l'ISO et le CCITT pour la description des tests.
TVEDA	Outil de génération de tests développé au CNET Lannion.
UIO	Séquence d'uniques entrées-sorties.
Ulysse	Héros grec. Également extension du langage Le_Lisp par des concepts objets.
$\varnothing$	Ensemble vide.
$\wp(E)$	Ensemble des parties de l'ensemble E.
$S = \langle S_S, L_S, T_S, s_{0s} \rangle$	Automate à entrées et sorties S, ayant un ensemble d'états S_S, d'interactions L_S, de transitions T_S et un état initial s_{0s}.
S, S', S'', S_1, S_i etc.	Spécifications.
I, I', I'', I_1, I_i etc.	Implantations.
T, T', T'', T_1, T_i etc.	Testeurs.
E, E' etc.	Environnement de test.
$!a$	Événement émission de l'interaction a.

$?a$	Événement réception de l'interaction a.
μ	Événement générique (émission ou réception).
τ	Événement inobservable (étiquette d'une transition interne).
$!\delta$	Événement modélisant un blocage de sortie.
$\lvert\mu\rvert$	Interaction associée à l'événement μ.
$\bar{\mu}$	Événement obtenu à partir de l'événement μ en changeant une entrée en sortie ou une sortie en entrée.
$\sigma = \mu_1 \ldots \mu_n$	Séquence d'événements.
ε	La séquence vide.
$\sigma \cdot \sigma'$	Concaténation des séquences σ et σ'.
Σ	Ensemble de séquences.
$\bar{\sigma}$	Séquence obtenue à partir de la séquence σ en changeant les entrées en sorties et les sorties en entrées.
(s, μ, s')	Transition de l'état s à l'état s' étiquetée par l'événement μ.
(s_0, σ, s_n)	$(\exists (s_i)_{1 \le i < n} \in S_s^n)\ (\forall i, 1 \le i \le n)\ ((s_{i-1}, \mu_i, s_i) \in T_s)$
(S, σ, s_n)	(s_0, σ, s_n)
(s_0, ε, s_1)	$s_0 = s_1$ ou $(\exists n \ge 1)\ (s_0, \tau^n, s_1)$
$(s_0, \vec{\mu}, s_1)$	$(\exists s_2, s_3 \in S_s)\ ((s_0, \varepsilon, s_2) \wedge (s_2, \mu, s_3) \wedge (s_3, \varepsilon, s_1))$
$(s_0, \vec{\sigma}, s_n)$	$(\exists (s_i)_{1 \le i < n} \in S_s^n)\ (\forall i, 1 \le i \le n)\ ((s_{i-1}, \vec{\mu}_i, s_i) \in T_s)$
$O(\sigma, S)$	$\{a \in L \mid \sigma!a \in Tr(S)\}$. Sorties autorisées pour l'automate S après la trace σ.
$Tr(S)$	Ensemble des traces de l'IOSM (ou du LTS) S.
$TM(S)$	Automate de traces de l'IOSM (ou du LTS) S.
$S \parallel S'$	Composition parallèle des IOSM (ou des LTS) S et S'.
$\hat{S}$	IOSM complété en rajoutant toutes les réceptions non prévues dans S.
$R_\sigma(I)$	$\sigma \in Tr(I)$ (une propriété de l'IOSM I).
$R_{\neg\sigma}(I)$	$\sigma \notin Tr(I)$ (une propriété de l'IOSM I).
$R(I, S)$	Relation d'implantation quelconque entre l'implantation I et la spécification S.

$R \leq R'$	La relation d'implantation R est moins forte que la relation R'.
$R_1(I, S)$	$(\forall\sigma \in Tr(S))\ (\sigma \in Tr(I) \Rightarrow O(\sigma, I) \subseteq O(\sigma, S))$
$R_{1bis}(I, S)$	$Tr(I) \subseteq Tr(\hat{S})$
$R_2(I, S)$	$(\forall\sigma \in Tr(S))(\sigma \in Tr(I) \Rightarrow$ $\{O(\sigma, I) \subseteq O(\sigma, S) \wedge (O(\sigma, I) = \emptyset) \Leftrightarrow (O(\sigma, S) = \emptyset)\})$
$R_{2bis}(I, S)$	$Tr(I) \subseteq Tr(\hat{S}) \wedge$ $(\forall\sigma \in Tr(\hat{S}))\ (\sigma \in Tr(I) \Rightarrow (O(\sigma, I) = \emptyset) \Leftrightarrow (O(\sigma, S) = \emptyset))$
$R_3(I, S)$	$Tr(S) \subseteq Tr(I)$
$R_4(I, S)$	$Tr(S) \subseteq Tr(I) \wedge (\forall\sigma \in Tr(S))\ ((O(\sigma, I) = \emptyset) \Leftrightarrow (O(\sigma, S) = \emptyset))$
$R_5(I, S)$	$(\forall\sigma \in Tr(S))\ (\sigma \in Tr(I) \wedge (O(\sigma, S) = O(\sigma, I)))$
$R_{5bis}(I, S)$	$Tr(I) = Tr(\hat{S})$
$R_t(I, S)$	$Tr(I) = Tr(S)$ (équivalence de traces).
$R_\Sigma(I, S)$	$(\forall\sigma \in \Sigma)\ (\sigma \in Tr(I) \cap Tr(S) \Rightarrow (O(\sigma, S) = O(\sigma, I)))$ (relation d'implantation finie).
$R_{\Sigma, A}(I, S)$	$(\forall\sigma \in \Sigma)\ (\sigma \in Tr(I) \cap Tr(S) \Rightarrow (O(\sigma, S) \cap A = O(\sigma, I) \cap A))$ (autre relation d'implantation finie).
$Succ(T, I)$	Succès du testeur T appliqué à l'implantation I.
$Fail(T, I)$	Échec du testeur T appliqué à l'implantation I.
$TCA(S)$	Testeur canonique asynchrone de l'IOSM S.
$S = \langle S_S, L_S, T_S, s_{0s} \rangle$	Système de transitions étiquetées S, ayant un ensemble d'états S_S, d'interactions L_S, de transitions T_S et un état initial s_{0s}.
$out(s)$	$\{\mu \in L \mid \exists s' \in S,\ (s, \vec{\mu}, s')\}$ (pour un LTS).
$s\ ref\ A$	L'ensemble d'actions A est refusé dans l'état s.
$I\ after\ \sigma$	$\{s \in S \mid (I, \sigma, s)\}$
$Ref(\sigma, I)$	$\{A \subseteq L \mid I\ after\ \sigma\ ref\ A\}$
$I\ conf\ S$	$(\forall\sigma \in Tr(S))\ (\sigma \in Tr(I) \Rightarrow Ref(\sigma, I) \subseteq Ref(\sigma, S))$ (relation d'implantation utilisée dans le cadre synchrone).
$I \approx_{te} S$	$Tr(I) = Tr(S) \wedge (\forall\sigma \in Tr(I))\ (Ref(\sigma, I) = Ref(\sigma, S))$ (équivalence de refus, ou de test).
$RG(S)$	Graphe de refus du LTS S.

$\tilde{F}$	Transformation du graphe de refus F définie au chapitre 3.
$TCS(S)$	Testeur canonique synchrone du LTS S.
$AR(F)$	Automate asynchrone recepteur du graphe de refus F.
$AS(F)$	Automate asynchrone émetteur du graphe de refus F.
$SY(T)$	Automate synchrone de l'automate de traces T.
$R_t(E)\ (I, S)$	$Tr(E \| I) = Tr(E \| S)$ (équivalence de traces modulo l'environnement de test).
$R_1(E)\ (I, S)$	$(\forall\sigma \in Tr(S \| E))\ (\sigma \in Tr(I \| E) \Rightarrow O(\sigma, I \| E) \subseteq O(\sigma, S \| E))$ (relation d'implantation utilisée en présence d'un environnement de test).
$R(S)$	IOSM obtenue par application des règles empiriques R à l'IOSM S.
$R(N)$	Hypothèse de régularité consistant à supposer que l'implantation a au plus N états.
$S = \langle St, Li, Lo, h, S_0\rangle$	Machine à états finis partiellement spécifiée et non déterministe S ayant un ensemble d'états St, d'entrées Li, de sorties Lo, une fonction de comportement h et un état initial S_0.
L	$Li \times Lo$, ensemble de paires entrée/sortie. Un élément de L est noté u.
L^*	Ensemble des séquences d'éléments de L. Un élément de L^* est noté x.
$P - a/b \rightarrow Q$	Est la transition de l'état P à l'état Q étiquetée par a/b. C'est une autre notation pour $(Q, b) \in h(P, a)$.
$P = \varepsilon \Rightarrow Q$	$P = Q$
$P = a/b \Rightarrow Q$	$P - a/b \rightarrow Q$
$P = x \Rightarrow Q$	$(\exists P_0 \dots P_{k-1} \in St)\ (P = P_0 = u_1 \Rightarrow P_1 \dots P_{k-1} = u_k \Rightarrow P_k = Q)$ où $x = u_1 \dots u_k$
$P = x \Rightarrow$	$(\exists Q \in St)\ (P = x \Rightarrow Q)$
$Tr(P)$	$\{x \mid P = x \Rightarrow\ \}$ (les traces de la PNFSM P).
x^{in}	Pour $x \in L^*$, x^{in} est la séquence d'entrées obtenue en enlevant les sorties.
V^{in}	Pour $V \subseteq L^*$, $V^{in} = \{x^{in} \mid x \in V\}$

$Tr^{in}(P)$	$\{x^{in} \mid P = x \Rightarrow\ \}$ ($Tr^{in}(P) = Li^*$ pour tout automate complètement spécifié).
$Tr(S_i) \oplus Tr(Sj)$	$(Tr(S_i) \cup Tr(S_j)) - (Tr(S_i) \cap Tr(S_j))$
$S_i \not\equiv S_j$	$(\exists x \in Tr(S_i) \oplus Tr(Sj))\ (x^{in} \in Tr^{in}(S_i) \cap Tr^{in}(S_j))$ (états distingables).
$S \leq_{quasi} I$	$Tr(S) \subseteq Tr(I)$ et $(\forall x \in Tr(I))\ (x^{in} \in Tr^{in}(S) \Rightarrow x \in Tr(S))$ (relation d'implantation de quasi-equivalence, utilisée pour les PNFSM).
$S \leq_{\Pi} I$	$(V \cap Tr(S)) \subseteq Tr(I)$ où $V = \{x \mid x \in L^* \wedge x^{in} \in \Pi \cap Tr^{in}(S) \cap Tr^{in}(I)\}$ (inclusion de traces entre PNFSM).
$S =_{\Pi} I$	$S \leq_{\Pi} I \ \wedge I \leq_{\Pi} S$ (équivalence de traces entre PNFSM).
W	Ensemble de caractérisation (tel que $(\forall S_i, S_j \in St)\ (\exists x \in Tr(S_i) \oplus Tr(Sj))\ (x^{in} \in W)$).
W_i	Ensemble d'identification pour l'état S_i.
Z	$W \cup Li \cdot W \cup \ldots \cup Li^{m-n} \cdot W$ (ensemble de distinction).
P	Ensemble couvrant les états.
Q	Ensemble couvrant les transitions.
r	Reset (interaction spéciale ramenant un automate dans son état initial).
RI	Hypothèse de reset bien implanté.
$\phi(S)$	PNFSM obtenue par transformation de l'IOSM S.
Δ	Degré de flou d'une OPNFSM.
$S \times S'$	Produit indépendant des IOSM S et S'.
$Pr(S, L)$	IOSM obtenue par projection de l'IOSM S sur l'ensemble d'actions L.
IN	Hypothèse d'indépendance.
$(\approx_S, \approx_L)$	Congruence sur une IOSM.
$\tilde{s}, \tilde{a}$	Classes d'équivalence de l'état s et de l'action a.
$\tilde{\sigma}$	Séquence d'actions quotient de la séquence σ.
Σ_q	Ensemble de séquences d'actions quotient.

$S_q = S / (\approx_S, \approx_L)$	Automate quotient.
$S / \approx_L$	Automate quotient en cas d'ensembles d'actions équivalentes.
UF	Hypothèse d'uniformité forte.
Uf	Hypothèse d'uniformité faible.
$C(\sigma_q)$	Ensemble de séquences obtenu à partir de la séquence d'actions quotient σ_q.
$R_{C(\sigma_q)}(I)$	$(\exists\sigma \in C(\sigma_q))\ (\sigma \in Tr(I))$ (propriété de l'IOSM I).
$R_{\neg C(\sigma_q)}(I)$	$(\forall\sigma \in C(\sigma_q))\ (\sigma \notin Tr(I))$ (propriété de l'IOSM I).
$X = \{t_i\}_{1 \le i \le n}$	Automate étendu comportant les transitions étendues $\{t_i\}_{1 \le i \le n}$.
$E(X)$	Expansion de l'automate étendu X.
UEF	Hypothèse d'uniformité étendue forte.
UEf	Hypothèse d'uniformité étendue faible.
$X \subseteq Tr(S)$	Exécutions de l'IOSM S.
$(I, X(I))$	Automate exécutable.
E_φ	Hypothèse d'équité bornée par la fonction φ.
$T_{\sigma, p}$	Testeur d'appartenance de σ sous l'hypothèse $E(p)$.
$T_{\neg\sigma, p}$	Testeur de non-appartenance de σ sous l'hypothèse $E(p)$.
$T_{C(\sigma_q), p}$	Testeur d'appartenance de la séquence quotient σ_q sous l'hypothèse $E(p)$.
$T_{CC(\sigma_{qq}), p}$	Testeur d'appartenance de la séquence double quotient σ_{qq} sous l'hypothèse $E(p)$.
$T \wedge T'$	Testeur conjonction.
$T \vee T'$	Testeur disjonction.

Index